Sustav za praćenje i vizualizaciju karakteristika električnih bicikala

Bačani, Bernard

Master's thesis / Diplomski rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Electrical Engineering and Computing / Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:168:849518

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-22**

Repository / Repozitorij:

FER Repository - University of Zagreb Faculty of Electrical Engineering and Computing repozitory

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 282

SUSTAV ZA PRAĆENJE I VIZUALIZACIJU KARAKTERISTIKA ELEKTRIČNIH BICIKALA

Bernard Bačani

Zagreb, veljača 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 282

SUSTAV ZA PRAĆENJE I VIZUALIZACIJU KARAKTERISTIKA ELEKTRIČNIH BICIKALA

Bernard Bačani

Zagreb, veljača 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 2. listopada 2023.

DIPLOMSKI ZADATAK br. 282

Pristupnik:	**Bernard Bačani (0036515322)**
Studij:	Računarstvo
Profil:	Računalno inženjerstvo
Mentor:	izv. prof. dr. sc. Mirko Sužnjević

Zadatak: **Sustav za praćenje i vizualizaciju karakteristika električnih bicikala**

Opis zadatka:

Električni bicikli i romobili danas su postali vrlo popularna forma urbanog, ali i ruralnog prijevoza. Postoje različite usluge u okviru kojih se električni bicikli te romobili mogu iznajmiti, koristiti te nadopuniti električnom energijom. Za takve usluge potreban je sustav nadzora nad karakteristikama bicikala, primarno pozicijom i stanjem baterije, kao i lokacijama i stanjima mjesta na kojima se bicikl može napuniti. Vaš zadatak je proučiti tehnologije sustava za praćenje električnih bicikala, od Interneta stvari i senzora koji su potrebni za detekciju lokacije, stanja baterije te slanja podataka na poslužitelj, do poslužitelja za obradu podataka te mobilnih aplikacija za vizualizaciju podataka. Na temelju proučenog potrebno je osmisliti te implementirati sustav koji će na temelju snimljenih stvarnih podataka ili umjetno generiranih podataka o poziciji bicikala, stanju njihove baterije te lokaciji na kojima se baterija može dopuniti prikazati vizualizaciju navedenih podataka korisniku kroz digitalnu kartu te odgovarajuće oznake na njoj. Podatke je potrebno prikazati na mobilnoj aplikaciji."

Rok za predaju rada: 9. veljače 2024.

Zahvaljujem se mentoru izv. prof. dr. sc. Mirku Sužnjeviću na strpljenju, razumijevanju i vođenju kroz moj studij, te posebno u pružanju prilika koje su definitivno veoma pozitivno utjecale na moj život.

Zahvaljujem i svim svojim bližnjima, prijateljima i obitelji na potpori tijekom cijelog studiranja, ne bih došao do ovog trenutka bez njih.

Sadržaj

1. Uvod **3**

2. Specifikacija sustava **4**

2.1. Cilj sustava 4

2.2. Tehnički zahtjevi 4

3. Pregled tehnologija za praćenje električnih bicikala **6**

3.1. Pametni sustav praćenja e-bicikala 6

3.2. Sustav praćenja vozila u stvarnom vremenu pomoću Redis-a 8

3.3. Ostali sustavi za praćenje 9

4. Arhitektura sustava za praćenje i vizualizaciju **12**

4.1. Opis sustava 12

4.2. MQTT broker 15

4.3. MQTT pretplatnik 16

4.4. Poslužitelj baze podataka 17

4.5. Poslužitelj API-ja 17

4.6. Korišteni uređaj za objavljivanje poruka 18

4.7. Budući uređaj za objavljivanje poruka 19

5. Implementacija sustava i generiranje podataka **21**

5.1. MQTT broker konfiguracija 21

5.2. MQTT pretplatnik 21

5.3. Baza podataka 22

5.4. Poslužitelj API-ja 23

5.5. MQTT objavljivač 24

6. Postavljanje sustava na Fly.io **30**
6.1. Fly.io platforma 30
6.2. Troškovi mjesečnog korištenja Fly.io platforme 31
6.3. Odabir regije i konfiguracija sustava 31
6.4. Fly Postgres 32

7. Testiranje i evaluacija sustava **34**
7.1. Apache JMeter 34
7.2. Opis test plana 35
7.3. Rezultati testiranja MQTT brokera 35
7.3.1. Rezultati testiranja za 1, 10 i 100 konekcija 35
7.3.2. Rezultati testiranja za 1000 konekcija 36

8. Rezultati i rasprava **39**
8.1. Budući razvoj sustava 40

9. Zaključak **41**

Literatura **42**

Sažetak **45**

Abstract **46**

A: Kod **47**

1. Uvod

U suvremenom urbanom okruženju, električni bicikli postaju neizostavan dio svakodnevice, pružajući ekološki prihvatljivu alternativu konvencionalnim prijevoznim opcijama. Njihova sveprisutnost potvrđuje promjenu načina kako promatramo mobilnost u gradskim područjima.

S ciljem poboljšanja korisničkog iskustva i osiguranja optimalne upotrebe električnih bicikala, razvijeni su inovativni sustavi za praćenje i vizualizaciju ključnih karakteristika ovih modernih prijevoznih sredstava. Ovi sustavi omogućuju korisnicima da u potpunosti iskoriste prednosti električnih bicikala, prateći njihove performanse i prilagodbe u stvarnom vremenu.

Zadatak ovog rada je proučiti tehnologije sustava za praćenje električnih bicikala, uključujući Internet stvari i senzore za detekciju lokacije i stanja baterije. Proučavanje obuhvaća poslužitelje za prijenos, obradu i pohranu podataka. Na temelju istraživanja, cilj je osmisliti i implementirati sustav koji korisnicima omogućava vizualizaciju stvarnih ili umjetno generiranih podataka o položaju bicikala, stanju baterije te lokacijama za punjenje. Prikaz podataka na digitalnoj karti i odgovarajuće oznake izrađeni su od strane druge osobe ili tima te nisu dio ovog istraživačkog rada.

Rad je strukturiran na sljedeći način. Nakon uvoda, drugo poglavlje opisuje specifikaciju sustava s tehničkim zahtjevima. U trećem poglavlju je napravljen pregled postojećih tehnologija. Potom, četvrto i peto poglavlje opisuju arhitekturu i implementaciju sustava. Šesto poglavlje opisuje postavljanje sustava na Fly.io platformu te troškove sustava. U osmom poglavlju su analizirani rezultati, dajući prijedloge za razvoj u budućnosti, dok je deveto poglavlje zaključak. Na kraju slijede literatura i sažeci rada na hrvatskom i engleskom jeziku te razvijen kod.

2. Specifikacija sustava

2.1. Cilj sustava

Cilj sustava je pružiti potpunu vizualizaciju informacija o električnim biciklima korisnicima putem mobilne aplikacije. Sustav treba pratiti poziciju bicikala i stanje njihove baterije u stvarnom vremenu.

2.2. Tehnički zahtjevi

U ovom odjeljku će biti navedeni tehnički zahtjevi za sustav praćenja bicikala.

Senzori

Sustav za praćenje treba uključivati sljedeće senzore:

- GPS senzor za praćenje pozicije bicikla.
- Senzor stanja baterije za mjerenje kapaciteta baterije i ostalih parametara.

Komunikacija

Sustav treba podržavati komunikaciju između:

- Objavljivača i poslužitelja za prijenos podataka putem bežične mreže.
- Poslužitelja za prijenos podataka, poslužitelja za pohranu podataka i poslužitelja za obradu podataka.
- Mobilne aplikacije i poslužitelja za obradu podataka putem sigurne internetske veze.

Frekvencija prijenosa podataka

Podaci sa svakog bicikla trebaju se slati sustavu svakih 10 sekundi. Ova učestalost omogućava realno vrijeme praćenja i osigurava ažuriranje informacija na mobilnoj aplikaciji.

Izbjegavanje ovisnosti o dobavljaču

Kako bi se izbjegla ovisnost o dobavljaču, odnosno "vendor lock-in", sustav će koristiti isključivo alate otvorenog koda. Ovo će omogućiti fleksibilnost, interoperabilnost i smanjenje dugoročnih troškova održavanja. Svi dijelovi sustava, od objavljivača na biciklima do poslužitelja i mobilne aplikacije, bit će implementirani s pomoću otvorenih standarda i alata.

Skalabilnost

Sustav će biti dizajniran s ciljem podržavanja do 1000 bicikala koji istovremeno šalju podatke. Ovo uključuje optimizaciju mrežne infrastrukture, baze podataka i poslužiteljskih resursa kako bi se osigurala stabilna i učinkovita obrada podataka čak i u uvjetima visokog opterećenja. Skalabilnost će biti ključna za dugoročni uspjeh sustava, posebno u urbanim područjima gdje je veliki broj bicikala u kretanju istovremeno.

Enkripcija podataka tijekom prijenosa

Sustav će koristiti snažne algoritme enkripcije kako bi osigurao da se podaci šalju sigurno između bicikala, poslužitelja i mobilnih aplikacija. Korištenje protokola poput HTTPS pridonosi dodatnom sloju sigurnosti tijekom prijenosa.

Sigurnosne kopije i oporavak podataka

Redovito će se stvarati sigurnosne kopije podataka, a sustav će imati mehanizme za oporavak podataka u slučaju neočekivanih događaja ili gubitka podataka. Ovo će osigurati integritet i dostupnost podataka tijekom vremena.

3. Pregled tehnologija za praćenje električnih bicikala

3.1. Pametni sustav praćenja e-bicikala

U ovom radu [1], opisana je platforma za prikupljanje podataka o korištenju električno potpomognutih bicikala. Pojam "e-bicikl" odnosi se na bicikle opremljene malim motorom i baterijom gdje vozači uvijek moraju pedalirati, ali mogu uključiti električnu pomoć (obično s odabirom niske, srednje ili visoke postavke) po želji. Pomoć se isključuje kad pedaliranje prestane ili kada se postigne brzina od 15 m/h (25 km/h). Rad se ne bavi drugim vrstama e-bicikala gdje se pomoć može koristiti bez pedaliranja.

Skalabilnost, replikabilnost i modularnost bili su ključni faktori za dizajn sustava. Sustav je implementiran s otvorenim izvorom softvera i otvorenim hardverom kako bi omogućio rast vlastite istraživačke flote ili repliciranje sustava za druge flote e-bicikala na različitim lokacijama u Ujedinjenom Kraljevstvu ili drugim zemljama.

Ključni eksperimentalni zahtjev bio je da nema potrebe za korisničkim interakcijama (uključivanje i isključivanje sustava, punjenje, prijenos podataka). Sustav je dizajniran za rad s e-biciklima u niskom i srednjem cjenovnom rangu kako bi odražavao tipično iskustvo vožnje e-bicikla. Sustav je trebao pratiti lokaciju svakog bicikla, praćenje upotrebe pomoći na e-biciklu te pružiti informacije u stvarnom vremenu za korisnike i istraživače.

Hardver se sastoji od tri glavne komponente: Android telefona, otvorenog hardverskog sučelja i prilagođene ploče za napajanje. Sve su smještene u vodonepropusnoj kutiji ispod sjedala. Koristi Android telefon zajedno s IOIO pločom, jeftinim sučeljem putem USB-a koje omogućuje povezivanje s raznim senzorima. Android softverska biblioteka omogućuje komunikaciju između ploče i telefona, iskorištavajući prednosti Android API-ja i senzora poput GPS-a i akcelerometra.

Ključna prednost je mogućnost napajanja IOIO ploče iz baterije bicikla, koja istovremeno puni telefon, omogućujući kontinuirani rad bez intervencije. Sustav se povezuje s e-biciklom paralelno s baterijom, prije pogonskog sustava, omogućujući rad i kad je bicikl isključen. Upravljanje baterijom osigurava sklop za zaključavanje ispod napona, koji sprječava prekomjerno pražnjenje.

Za središnji uređaj testirano je nekoliko jeftinih Android telefona s niskom potrošnjom, sposobnih za automatsko pokretanje pri punjenju. Kriteriji su uključivali kvalitetu GPS signala, 3G povezivost i dugotrajnost baterije. Samsung Galaxy Ace 2 S6500, nakon prilagodbi, ispunjavao je te zahtjeve, omogućujući automatsko pokretanje nakon punjenja i stabilan rad u stvarnim uvjetima.

Android igra ključnu ulogu u optimizaciji potrošnje energije. Kao pozadinska usluga, osigurava minimalnu potrošnju energije tako da telefon ostaje u stanju mirovanja veći dio vremena. Sustav periodički budi telefon svakih 25 sekundi, provjeravajući akcelerometar tijekom 1.5 sekundi kako bi detektirao eventualni pokret bicikla. Ako se pokret detektira, aktiviraju se usluge GPS-a i praćenja, te se relevantni podaci bilježe. Nakon završetka vožnje, sve nadzorne usluge se isključuju, a telefon se vraća u stanje mirovanja.

Pohranjivanje GPS podataka lokalno na uređaju omogućuje smanjenje potrošnje energije, a ti podaci se šalju na poslužitelj svakih 25 sekundi u komprimiranom obliku. Telefon bilježi geografsku dužinu, širinu, visinu i točnost GPS-a, pružajući detaljne informacije o lokaciji. Dodatno, informacije o pomoći se šalju na poslužitelj svaki put kad dođe do promjene, osiguravajući ažuriranje u stvarnom vremenu.

Periodična prijava na poslužitelj svakih 3 sata pruža informacije o statusu baterije telefona i napajanju iz baterije bicikla. Ovi podaci koriste se za praćenje zdravlja flote bicikala te identificiranje mogućih problema. Baterija telefona može trajati oko 4 dana nakon što se isprazni baterija e-bicikla, pružajući kontinuirano praćenje čak i u slučaju gubitka napajanja.

Podaci e-bicikala šalju se na poslužitelj putem HTTP protokola, gdje se obrađuju i pohranjuju u MySQL bazu podataka. Na poslužitelju se izračunavaju ključne statističke informacije poput ukupne udaljenosti vožnje, ukupnog vremena vožnje i prosječne brzine. PHP skripte na Apache 2 poslužitelju izvršavaju obradu podataka.

Korisnici mogu pristupiti informacijama putem web sučelja koje prikazuje stvarne podatke o vožnji. Sučelje nudi informacije poput trenutne lokacije, brzine, prijeđene udaljenosti i drugih relevantnih parametara.

Sustav se pokazao pouzdanim stabilna platforma za prikupljanje, analizu i dijeljenje podataka o floti e-bicikala tijekom dvogodišnjih ispitivanja., pružajući visokokvalitetne podatke o lokaciji i pomoći. Trenutačni status sustava ispunjava postavljene ciljeve dizajna i omogućuje repliciranje od strane drugih istraživačkih projekata.

Kompromis između autonomije i kvalitete podataka utječe na životni vijek baterije e-bicikala, budući da sustav postupno troši bateriju tijekom vremena. Sustav ima dva načina potrošnje energije, štedeći kad je bicikl nepokretan, trošeći više tijekom vožnje.

3.2. Sustav praćenja vozila u stvarnom vremenu pomoću Redis-a

Sustav za praćenje vozila u stvarnom vremenu implementiran je korištenjem Redis-a i Golang-a. Razvijen je za praćenje autobusa u Helsinkiju, koristeći Redis PubSub, Streams, TimeSeries i Gears, te PostGIS i TileGen za podršku geoprostornim operacijama [2].

Arhitektura sustava smještena je na AWS instanci, a GoLang broker koristi se za procesiranje MQTT poruka i slanje podataka na različite lokacije. Redis PubSub koristi se za ažuriranja u stvarnom vremenu putem WebSocket-a, Streams za obradu događaja, a TimeSeries za bilježenje trenutačne brzine i lokacije autobusa.

Prikupljanje podataka započinje s MQTT dovodom Helsinške regionalne tranzitne agencije. Svi dijelovi sustava smješteni su na istoj AWS instanci, a resursi uključuju Redis komponente, PostGIS, TileGen, te različite API-je za pristup podacima.

Sustav se koristi za prijenos lokacija autobusa u stvarnom vremenu putem Redis PubSub kanala, dok se događaji obrađuju putem Streams i RedisGears. Trenutačna brzina i lokacija autobusa zabilježeni su u RedisTimeSeries s ciljem efikasnog agregiranja podataka i smanjenja potrošnje memorije.

Rezultati sustava pokazuju sposobnost obrade oko 15 GB poruka dnevno, s performansama koje su zadovoljavajuće na relativno maloj skali. Iako je CPU inicijalno smatran glavnim izazovom, disk se pokazao kao ključno ograničenje u lokalnim testiranjima. Povećanjem resursa, poput prelaska na AWS gp3 EBS, sustav je postao funkcionalan, omogućujući pristup povijesnim podacima iz baze podataka.

3.3. Ostali sustavi za praćenje

U radu [3], opisan je sustav za praćenje električnih bicikala kako bi se spriječila krađa. Sustav koristi kombinaciju GPS i GSM tehnologija za precizno praćenje lokacije bicikala te prikupljanje podataka o ubrzanju i vibracijama. Postavljen je kao skalabilan sustav, omogućujući široku primjenu korisnicima.

Rad sustava odvija se sljedećim koracima:

1. Senzor prikuplja podatke o stanju bicikla.
2. GSM modul šalje podatke o lokaciji i stanju bicikla na web poslužitelj.
3. Web poslužitelj prikazuje podatke o lokaciji i stanju bicikla korisnicima.

U slučaju krađe, korisnik može koristiti web sučelje za praćenje lokacije bicikla i obavijestiti policiju. Sustav također može poslati SMS poruku korisniku ako se bicikl pomakne iz svoje uobičajene lokacije. Ovaj sustav pruža učinkovitu zaštitu električnih bicikala od krađe, istovremeno nudeći jednostavnost korištenja i niz korisnih značajki.

U drugom radu [4], opisuje se sustav praćenja vozila u stvarnom vremenu temeljen na GSM/GPS tehnologijama. Sustav omogućuje trenutne obavijesti o brzini i lokaciji vozila putem mobilnih telefona, uz mogućnost zaključavanja lokacije i upozorenja na prekoračenje postavljene brzine.

Sustav koristi GSM i GPS za praćenje lokacije i brzine vozila. GSM modul šalje tekstualne poruke vlasniku s informacijama o lokaciji i brzini, dok GPS modul određuje trenutnu lokaciju vozila.

Ključne značajke sustava uključuju:

- Obavijesti u stvarnom vremenu o brzini i lokaciji.
- Zaključavanje trenutne lokacije s obavijestima o premještanju.
- Zaključavanje brzine s upozorenjem na prekoračenje.

U radu [5], govori se o energetski učinkovitom sustavu u stvarnom vremenu praćenja vozila temeljenom na internet stvari (engl. *Internet of Things*, skraćeno IoT). Sustav, nazvan *Smart Vehicle System*, skraćeno SVS, sastoji se od tri ključne komponente: Tracking Unita, Clouda i Android aplikacije. SVS prikuplja podatke o temperaturi, brzini i lokaciji vozila, prenoseći ih u oblak putem GSM mreže. Android aplikacija omogućuje korisnicima pregledavanje podataka i postavljanje alarma.

Sustav SVS ističe se po sljedećim prednostima:

- Energetski učinkovit: Koristi komponente s niskom potrošnjom poput GSM modula i mikrokontrolera.
- Stvarno vrijeme: Pruža trenutne podatke o lokaciji, brzini i temperaturi vozila.
- Mobilni pristup: Dostupan putem Android aplikacije.
- Jednostavan za instalaciju: Lako se instalira i konfigurira.

U radu [6], opisuje se skalabilni sustav praćenja javnih autobusa koji koristi IoT tehnologije. Sustav rješava problem nesigurnosti javnog prijevoza primjenom GPS-a za praćenje autobusa i MQTT protokola za komunikaciju s poslužiteljem. Poslužitelj pruža korisnicima podatke o stvarnoj lokaciji autobusa putem REST API-ja.

Naglašavaju se prednosti korištenja IoT tehnologija u praćenju javnih autobusa, uključujući:

- Povećanje transparentnosti: Korisnici mogu pratiti lokaciju autobusa u stvarnom vremenu, povećavajući transparentnost usluge.
- Poboljšanje učinkovitosti: Praćenje autobusa u stvarnom vremenu omogućuje operatorima učinkovitije upravljanje prometom.

- Smanjenje troškova: IoT tehnologije smanjuju troškove praćenja autobusa u usporedbi s tradicionalnim metodama.

Sustav uključuje tri ključne komponente:

- Bežični senzori: Montiraju se na autobuse i prikupljaju podatke o lokaciji, brzini i smjeru.
- MQTT broker: Posreduje u komunikaciji između senzora i poslužitelja za obradu.
- Poslužitelj za obradu: Obrađuje podatke od senzora i pruža ih korisnicima putem REST API-ja.

Rad također raspravlja o izazovima implementacije IoT sustava praćenja javnih autobusa, uključujući sigurnost, pouzdanost i kompatibilnost s postojećom infrastrukturom javnog prijevoza. Ovaj sustav ima potencijal za transformaciju javnog prijevoza, pružajući bolju uslugu korisnicima, uz dodatne usluge poput planiranja putovanja, otkazivanja putovanja i informacija o prometu u stvarnom vremenu.

U radu [7], opisuje se sustav praćenja i detekcije krađe vozila koji koristi Google Cloud IoT Core i Firebase. Sustav koristi GPS module i senzore motora za prikupljanje podataka o lokaciji i statusu vozila, a korisnicima omogućuje praćenje putem web sučelja. Naglašavaju se prednosti korištenja Google Cloud IoT Core i Firebase, uključujući skalabilnost, pouzdanost i sigurnost od neovlaštenog pristupa.

Sustav se sastoji od sljedećih ključnih komponenti:

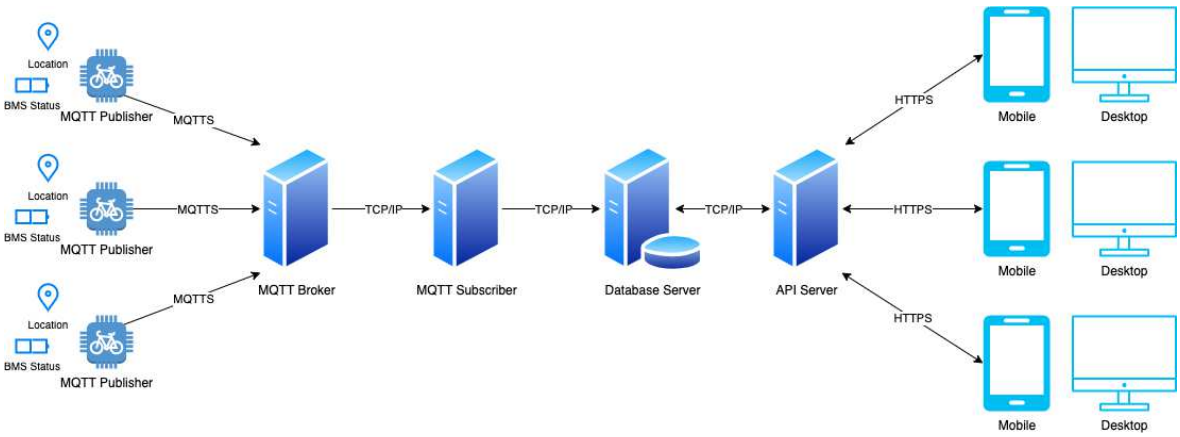
- GPS moduli: Prikupljaju podatke o lokaciji vozila.
- Senzori motora: Prikupljaju podatke o stanju motora, uključujući brzinu i temperaturu.
- Google Cloud IoT Core: Upravlja mrežom senzora i šalje podatke u Firebase.
- Firebase: Pohranjuje podatke i pruža web-sučelje za korisnike.

4. Arhitektura sustava za praćenje i vizualizaciju

4.1. Opis sustava

Za komunikaciju između bicikala, poslužitelja i drugih dijelova sustava, korišten je Message Queuing Telemetry Transport protokol, skraćeno MQTT. Sustav se sastoji od MQTT objavljivača, mobilne aplikacije i četiri poslužitelja smještena na platformi Fly.io (slika 4.1.):

- **MQTT broker** - smješten na instanci od 256 MB.
- **MQTT pretplatnik** - smješten na instanci od 256 MB.
- **Poslužitelj baze podataka** - smješten na instanci od 256 MB.
- **Poslužitelj API-ja** - smješten na instanci od 512 MB.



Slika 4.1. Arhitektura sustava

- **MQTT objavljivač** je komponenta sustava koja periodički objavljuje MQTT poruke o lokaciji i statusu baterije na MQTT broker.

- **MQTT broker** je komponenta sustava koja prima i prenosi MQTT poruke.
- **MQTT pretplatnik** je komponenta sustava koja je pretplaćena na MQTT broker te obrađuje MQTT poruke i sprema podatke u bazu podataka.
- **Poslužitelj baze podataka** je komponenta sustava koja trajno pohranjuje podatke.
- **Poslužitelj API-ja** je komponenta sustava koja prenosi podatke iz baze podataka do mobilne aplikacije.

Prednosti arhitekture s više poslužitelja

Povećanje kompleksnosti sustava arhitekturom s više poslužitelja nosi sa sobom niz prednosti koje poboljšavaju funkcionalnost, skalabilnost i pouzdanost sustava:

- **Podjela odgovornosti**: Razdvajanje funkcionalnosti sustava na različite poslužitelje omogućuje jasnu podjelu odgovornosti. MQTT broker brine se za komunikaciju između uređaja, MQTT pretplatnik prati promjene, poslužitelj baze podataka upravlja pohranom podataka, dok poslužitelj API-ja omogućuje pristup podacima putem sučelja.
- **Optimizacija resursa**: Svaki poslužitelj može biti optimiziran za specifičnu ulogu. Na primjer, poslužitelj baze podataka može imati posebne postavke prilagođene radu s velikim količinama podataka, dok poslužitelj API-ja može zahtijevati veću količinu memorije za bržu obradu zahtjeva.
- **Lakše skaliranje**: Različiti dijelovi sustava mogu zahtijevati različite razine resursa. Mogućnost povećanja ili smanjenja resursa na svakom poslužitelju omogućuje lakše skaliranje prema potrebama. Na primjer, ako se poveća opterećenje na MQTT brokeru, može se lako dodijeliti više resursa toj komponenti.
- **Povećana pouzdanost**: Razdvajanjem funkcionalnosti na više poslužitelja povećava se otpornost sustava na kvarove. Ako jedan poslužitelj doživi kvar, ostali dijelovi sustava mogu nastaviti s radom neometano. Ovo doprinosi općoj pouzdanosti sustava.
- **Lakše održavanje**: Različiti poslužitelji mogu imati različite komponente softvera i konfiguracije koje olakšavaju održavanje. Ažuriranje ili promjene u jednom dijelu

sustava ne moraju nužno utjecati na ostale dijelove.

- **Bolja sigurnost**: Razdvajanje funkcionalnosti može doprinijeti boljoj sigurnosti sustava. Na primjer, poslužitelj baze podataka može biti postavljen s posebnim sigurnosnim mjerama, a pristup API-ju može biti kontroliran kako bi se spriječio neovlašten pristup.

4.2. MQTT broker

MQTT broker ima ključnu ulogu u primanju, filtriranju i distribuciji poruka koje klijenti objavljuju. Kao rješenje za ovu svrhu koristi se Eclipse Mosquitto.

Eclipse Mosquitto [8] odabran je kao MQTT broker na temelju postavljenih zahtjeva za sigurnošću, obradom pogrešaka, nadzorom metrika osnovnog djelovanja, obradom podataka te integracijom u kontekstu MQTT poruka. Također, odabran je zbog skalabilnosti koja osigurava učinkovitost cijele usluge.

Izbor pravilnog MQTT brokera igra ključnu ulogu u implementaciji IoT sustava, a Eclipse Mosquitto se ističe zbog niza prednosti koje pruža [9]:

- **Open source:** Eclipse Mosquitto je open source softver, što omogućava slobodnu distribuciju, prilagodbu i poboljšanje prema specifičnim zahtjevima projekta.
- **Jednostavna konfiguracija i korištenje:** Mosquitto nudi jednostavnu konfiguraciju i upotrebu, što olakšava implementaciju MQTT komunikacije u sustavu.
- **Niski resursni zahtjevi:** Mosquitto ne zahtijeva značajne resurse sustava, što je posebno važno u okolini s ograničenim resursima.
- **Aktivna podrška zajednice:** Brojna zajednica korisnika i razvijatelja aktivno podržava Eclipse Mosquitto, pružajući rješenja i odgovore na pitanja.
- **Dovoljna skalabilnost:** Mosquitto ima dovoljne mogućnosti skaliranja, podržavajući do 100 000 mogućih veza, čime osigurava fleksibilnost u rastu sustava.

Kada razmatramo zašto je Mosquitto dobar odabir, trebamo obratiti pažnju na nekoliko ključnih čimbenika [9]:

- **Sigurnost:** Mosquitto pruža napredne sigurnosne značajke, uključujući podršku za SSL/TLS enkripciju i autentikaciju korisnika, čime osigurava pouzdanu i sigurnu komunikaciju.
- **Klasterizacija i automatsko skaliranje:** Mosquitto podržava klasterizaciju i automatsko skaliranje, olakšavajući upravljanje i proširivanje sustava prema potrebama.

- **Integracija podataka i pravila:** Mosquitto olakšava integraciju podataka i pravila, omogućujući učinkovitu obradu poruka i primjenu pravila unutar brokera.
- **Performanse:** S optimiziranim kodom, Mosquitto pruža visoke performanse, osiguravajući brzu i pouzdanu razmjenu poruka.
- **Rađeno za oblak:** Mosquitto je kompatibilan s okruženjima prilagođenima oblaku, što ga čini prikladnim izborom za implementaciju u oblaku.
- **Podrška za proširenja:** Fleksibilna arhitektura Mosquitto brokera omogućava podršku za različite proširenja, prilagodbe i dodatke prema potrebama projekta.
- **Troškovi:** Kao open source rješenje, Mosquitto smanjuje troškove implementacije i održavanja u usporedbi s komercijalnim alternativama.

4.3. MQTT pretplatnik

MQTT pretplatnik sprema sve primljene poruke u bazu podataka. Kao rješenje za ovu svrhu koristi se programski jezik Go.

Go [10], također poznat kao Golang, je programski jezik razvijen u Googleu. Razvijan je s fokusom na jednostavnost, produktivnost i performanse. Go je statički tipizirani jezik s automatskim upravljanjem smećem (engl. *garbage collection*), što ga čini jednostavnim za učenje i efikasnim za razvoj softvera.

Jedan od ključnih razloga za odabir Go jezika u ovom projektu bio je visoko rangiranje na benchmarku performansi [11]. Na tom benchmarku, Go se pozicionirao kao drugi najbrži jezik, odmah iza C-jezika. No, osim performansi, Go je odabran i zbog svoje jednostavnosti korištenja. Programeri ga često hvale zbog čistog i jednostavnog sintaksnog stila, brze kompilacije i ugrađenih alata za testiranje i upravljanje paketima.

Komunikacija s brokerom

- **Protokol:** MQTTS (TLS sigurna verzija MQTT priključka)
- **Priključak:** 8883 (standardni enkriptirani MQTT priključak)

4.4. Poslužitelj baze podataka

Poslužitelj baze podataka odgovoran je za pohranu podataka. Sastoji se od sljedećih komponenata:

- **PostgreSQL** [12]: Otvoreni sustav za upravljanje bazama podataka temeljen na objektno-relacijskom modelu. Poznat po svojoj pouzdanosti, podršci za kompleksne upite i proširenjima koja omogućuju napredne funkcionalnosti poput geoprostornih podataka.

- **PostGIS** [13]: Proširenje za PostgreSQL koje omogućuje podršku za geoprostorne podatke. Idealno za rad s prostornim informacijama poput geografskih podataka, kartografskih sustava i analiza prostornih podataka.

- **Flyway** [14]: Alat za upravljanje verzijama baza podataka koji omogućuje kontrolirano praćenje i primjenu promjena u shemi baze podataka tijekom vremena. Koristi se za olakšavanje procesa migracije baza podataka, održavanje konzistentnosti i upravljanje verzijama sheme baze podataka.

4.5. Poslužitelj API-ja

Poslužitelj API-ja izlaže podatke iz baze putem prilagođenih REST API-ja. Napravljen je korištenjem sljedećih alata:

- **Spring Boot** [15]: Okvir za razvoj Java aplikacija koji pojednostavljuje proces izgradnje, konfiguracije i implementacije aplikacija. Omogućuje brzo postavljanje mikroservisa i podržava integraciju s različitim tehnologijama.

- **Java** [16]: Programski jezik i platforma za razvoj aplikacija. Java se koristi široko u svijetu softvera i poznata je po svojoj pouzdanosti, prenosivosti i širokoj primjeni u raznim domenama, uključujući web, mobilne i poslovne aplikacije.

- **Maven** [17]: Alat za upravljanje projektima i automatizaciju procesa izgradnje softvera u Javi. Maven omogućuje jednostavno upravljanje ovisnostima, izgradnju projekta, testiranje i distribuciju, čime olakšava upravljanje kompleksnim projektima.

- **Docker** [18]: Platforma za kontejnerizaciju koja omogućuje pakiranje, distribuciju i izvođenje aplikacija u kontejnerima. Kontejneri pružaju lagan, brz i dosljedan način izolacije aplikacija s njihovim ovisnostima, olakšavajući tako rad u različitim okruženjima i ubrzavajući razvoj, testiranje i implementaciju aplikacija.

Komunikacija s bazom podataka

- **Protokol:** TCP/IP (Standardni protokol za komunikaciju s PostgreSQL bazom)
- **Port:** 5432 (Standardni priključak za komunikaciju s PostgreSQL bazom)

Komunikacija s klijentom

- **Protokol:** HTTPS (TLS sigurna verzija HTTP protokola)
- **Port:** 443 (Standardni enkriptirani HTTP priključak)

4.6. Korišteni uređaj za objavljivanje poruka

U implementaciji sustava za praćenje i vizualizaciju koristi se uređaj TTGO T-Call ESP32 with SIM800L (slika 4.2.). Ovaj uređaj integrira ESP32 mikrokontroler s GSM/GPRS modulom (SIM800L), omogućujući uređaju komunikaciju putem mobilne mreže.

Specifikacije uređaja TTGO T-Call ESP32 with SIM800L:

- **Mikrokontroler:** ESP32
- **GSM/GPRS modul:** SIM800L
- **Komunikacijski priključci:** Micro-USB, SIM kartica, 3.5mm audio priključak
- **Integrirana antena:** Da
- **Baterija:** Li-Ion 18650 s podrškom za punjenje putem Micro-USB priključka
- **Senzori:** GPS modul

Funkcionalnosti uređaja:

- **Lokacija:** Uređaj koristi GPS modul za određivanje trenutačne lokacije uređaja.

- **Komunikacija:** Integrirani GSM/GPRS modul omogućuje uređaju slanje podataka putem mobilne mreže.

- **Energetska učinkovitost:** Mogućnost korištenja Li-Ion baterije s podrškom za punjenje olakšava mobilnost i produžava radno vrijeme uređaja.

Slika 4.2. TTGO T-Call ESP32 with SIM800L

4.7. Budući uređaj za objavljivanje poruka

Promjena uređaja provodi se kako bi se osigurala kontinuirana komunikacija putem mobilne mreže s obzirom na gašenje 2G mreža [19]. Uređaj LILYGO T-SIM7000G (slika 4.3.) pruža podršku za suvremenije 4G LTE mreže, čime se osigurava stabilnost i dugoročna upotrebljivost sustava za praćenje i vizualizaciju. Ovaj uređaj također integrira ESP32 mikrokontroler, ali s GSM/GPRS/4G modulom (SIM7000G).

Specifikacije senzora LILYGO T-SIM7000G:

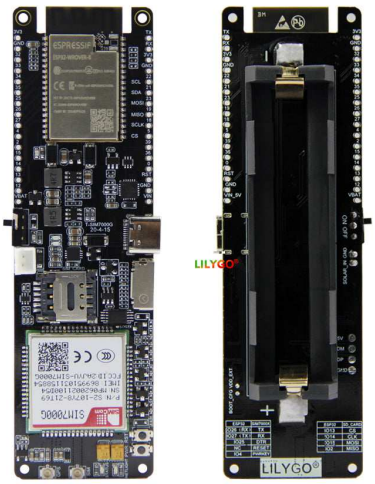
- **Mikrokontroler:** ESP32

- **GSM/GPRS/4G modul:** SIM7000G

- **Komunikacijski priključci:** Micro-USB, SIM kartica, 3.5mm audio priključak
- **Integrirana antena:** Da
- **Baterija:** Li-Ion 18650 s podrškom za punjenje putem Micro-USB priključka
- **Senzori:** GPS modul

Funkcionalnosti uređaja LILYGO T-SIM7000G:

- **Lokacija:** Uređaj koristi GPS modul za određivanje trenutačne lokacije uređaja.
- **Komunikacija:** Integrirani GSM/GPRS/4G modul omogućuje brzu i stabilnu komunikaciju putem mobilne mreže.
- **Energetska učinkovitost:** Mogućnost korištenja Li-Ion baterije s podrškom za punjenje olakšava mobilnost i produžava radno vrijeme uređaja.



Slika 4.3. T-SIM7000G [20]

5. Implementacija sustava i generiranje podataka

5.1. MQTT broker konfiguracija

U ovom odjeljku, opisana je konfiguracija MQTT brokera, koji omogućuje pouzdanu i efikasnu razmjenu poruka između različitih komponenata sustava.

- **Dockerfile:** definira kako će se izgraditi Docker slika koja će sadržavati MQTT broker. Ovdje se koristi slika `eclipse-mosquitto:2.0.18`, a zatim se postavljaju radni direktorij, kopiraju konfiguracijske datoteke i postavljaju potrebne dozvole. Također se izlaže port 1883, na kojem MQTT broker sluša.

- **Konfiguracijska datoteka (`mosquitto.conf`)**: postavlja različite parametre MQTT brokera, uključujući postavke perzistencije, vrstu i odredište zapisa dnevnika, dozvole, korisničke podatke i ostale relevantne konfiguracijske opcije.

- **Skripta za pokretanje (`entrypoint.sh`)**: koristi se prilikom pokretanja kontejnera kako bi postavila korisničke podatke (korisničko ime i lozinku) za MQTT broker. Ova skripta stvara datoteku s korisničkim podacima i pokreće MQTT brokera s navedenom konfiguracijom.

5.2. MQTT pretplatnik

Za primanje i pohranu podataka poslanih putem MQTT-a, implementiran je MQTT pretplatnik koristeći programski jezik Go. Ovaj pretplatnik služi kao most između MQTT brokera i baze podataka. Glavni programski elementi te najvažnije funkcije korištene u MQTT pretplatniku su:

- **Paketi:** Kôd koristi nekoliko vanjskih paketa, uključujući `paho.mqtt.golang` za MQTT komunikaciju i `pgxpool` za rad s PostgreSQL bazom podataka.

- **Tipovi podataka:** Definirani su tipovi `BatteryData` i `LocationData` koji predstavljaju strukture podataka koje se očekuju kao JSON podaci prilikom pretplate na određene MQTT teme.

- **Postavljanje:** Funkcije `setupPostgres` i `setupMQTTClient` koriste se za uspostavljanje veze s PostgreSQL bazom podataka i uspostavljanje veze s MQTT brokerom.

- **Rukovanje porukama:** Funkcije `handleBatteryMessage` i `handleLocationMessage` obrađuju poruke primljene putem MQTT-a i pohranjuju ih u PostgreSQL bazu podataka.

- **Glavna funkcija:** Glavna funkcija `main` postavlja sve potrebne veze, pretplaćuje se na određene MQTT teme, i zatim čeka signal za izlazak (`Ctrl+C`). Pri izlasku, program se odspaja od MQTT brokera i zatvara veze s bazom podataka.

Ovaj MQTT pretplatnik omogućuje učinkovitu obradu i pohranu podataka koje šalju električni bicikli putem MQTT-a.

5.3. Baza podataka

Za pohranu podataka o statusu baterija i lokacijama bicikala koristi se PostgreSQL baza podataka. Ove tablice i njihova međusobna povezanost omogućuju cjelovito pohranjivanje i organizaciju podataka o statusu baterija i lokacija električnih bicikala. U nastavku su prikazane SQL skripte koje definiraju shemu baze podataka i tablica:

- **Tablica** `batteries`: sadrži podatke o statusu baterija električnih bicikala. Svaka baterija ima jedinstveni identifikator (`battery_id`) te se bilježe informacije o stanju, mogućnosti punjenja, razini napunjenosti, temperaturi, te drugim relevantnim parametrima.

- **Tablica** `locations`: pohranjuje informacije o geografskim lokacijama električnih

bicikala. Svaka lokacija ima jedinstveni identifikator (`location_id`) te se čuvaju podaci o geografskoj širini, dužini i vremenskom žigu lokacije.

- **Tablica** `battery_reading_cell_voltages`: sadrži podatke o naponskim vrijednostima pojedinih ćelija baterija. Povezana je s glavnom tablicom `batteries` putem vanjskog ključa (`battery_reading_battery_id`).

- **Tablica** `battery_reading_bat_temps`: sadrži podatke o temperaturama baterija. Također je povezana s glavnom tablicom `batteries` putem vanjskog ključa (`battery_reading_battery_id`).

5.4. Poslužitelj API-ja

Za pružanje API-ja za praćenje i vizualizaciju podataka o električnim biciklima koristi se Spring Boot aplikacija napisana u programskom jeziku Java. Ove klase koriste anotacije poput `@Table`, `@Id`, `@GeneratedValue`, `@ElementCollection`, `@CollectionTable` kako bi definirale strukturu tablica i relacija u bazi podataka. U nastavku je opisana struktura koda i konfiguracija poslužitelja:

- **Klasa** `BatteryReading`: predstavlja entitet koji se koristi za pohranu podataka o statusu baterija električnih bicikala. Ova klasa je anotirana s `@Entity` kako bi se mapirala na odgovarajuću tablicu u bazi podataka.

- **Klasa** `LocationReading`: predstavlja entitet koji se koristi za pohranu podataka o geografskim lokacijama bicikala. Također je anotirana s `@Entity` kako bi se mapirala na odgovarajuću tablicu u bazi podataka.

- **Repozitoriji**: Spring Data JPA omogućava jednostavan pristup bazama podataka kroz repozitorije. Kroz sučelje `BatteryRepository` definirane su osnovne operacije nad entitetima tipa `BatteryReading`, dok je za entitete tipa `LocationReading` zadužen repozitorij `LocationRepository`.

- **Konfiguracija baze podataka**: odvija se putem `application.properties` datoteke koja sadrži informacije o URL-u, korisničkom imenu i lozinki za pristup PostgreSQL bazi podataka. Također, koristi se alat Flyway za migracije, što olakšava održavanje i ažuriranje strukture baze podataka tijekom vremena.

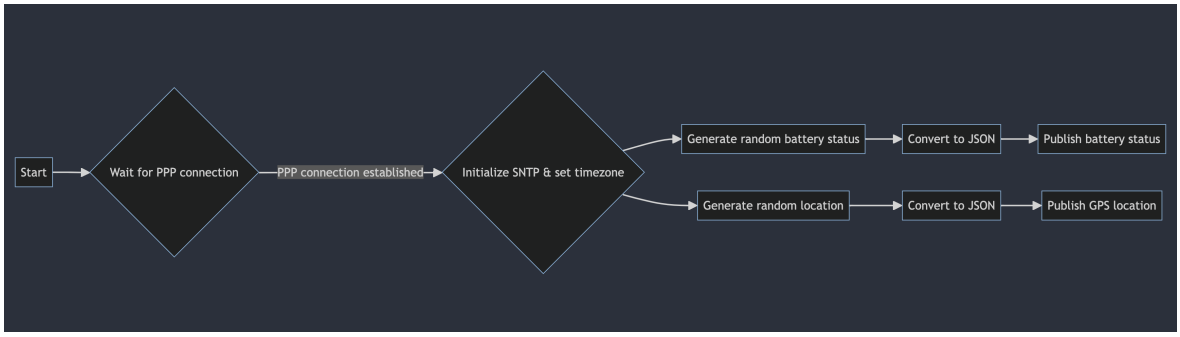
- **Konfiguracija Spring Boot aplikacije**: aplikacija je konfigurirana za pokretanje na portu 8080 (`server.port=8080`) te koristi paketiranje u Docker kontejner s JDK 17 (`FROM eclipse-temurin:17-jdk-alpine`). Pom.xml datoteka definira sve potrebne ovisnosti, uključujući Spring Boot Starter module, PostgreSQL driver, Lombok (za smanjenje boilerplate koda), Flyway za migracije baze podataka, te Springdoc za generiranje OpenAPI dokumentacije. U ovom slučaju, verzija Spring Boot Starter Parenta je 3.2.0, a koristi se JDK 17.

- **Docker kontejner**: aplikacija je pakirana kao Docker kontejner. Dockerfile definira proces izgradnje (build) aplikacije te pokretanje (runtime) kontejnera. Aplikacija se izgrađuje u dva koraka kako bi se smanjila veličina konačnog kontejnera.

5.5. MQTT objavljivač

Implementacija MQTT objavljivača koristi ESP-IDF okruženje i povezuje se s MQTT brokerom radi periodičkog slanja statusa baterije i GPS lokacije bicikla. Kôd je napisan u programskom jeziku C i koristi ESP32 mikrokontroler.

Dijagram toka objavljivača pojednostavljeno je prikazan na (slika 5.1.):



Slika 5.1. Dijagram toka

- **Inicijalizacija (slika 5.2.)**: inicijalizacija počinje s točkom A (Start). Zatim se čeka uspostava PPP veze (točka B). Kada se PPP veza uspostavi, prelazi se na inicijalizaciju SNTP-a i postavljanje vremenske zone (točka C).

- **Generiranje podataka (slika 5.3.)**: nakon inicijalizacije, generiraju se slučajni podatci o statusu baterije (točka D) i lokaciji (točka E).

- **Konverzija u JSON (slika 5.4.)**: generirani podaci o bateriji i lokaciji zatim se

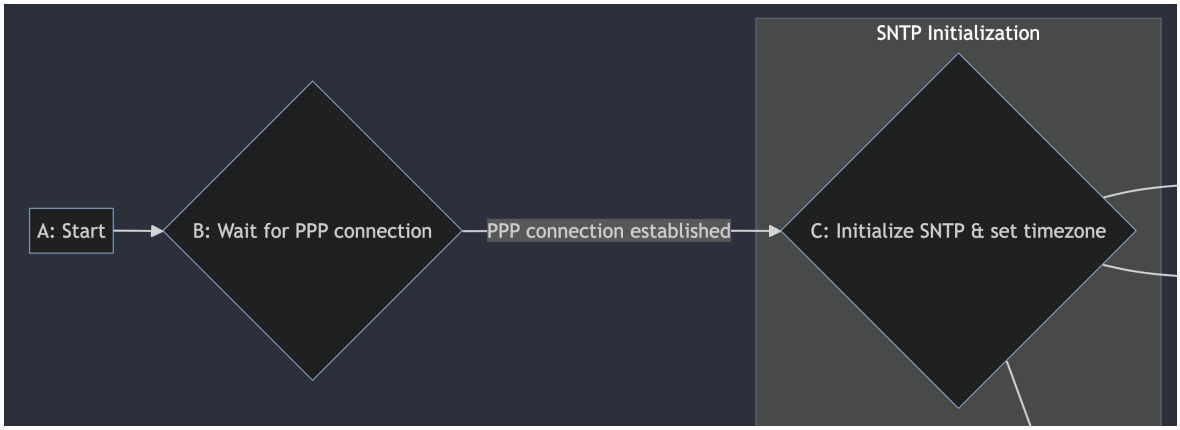
pretvaraju u JSON format (točke F i G).

- **MQTT objavljivanje (slika 5.5.)**: JSON podaci o statusu baterije i lokaciji objavljuju se putem MQTT protokola (točke H i I).

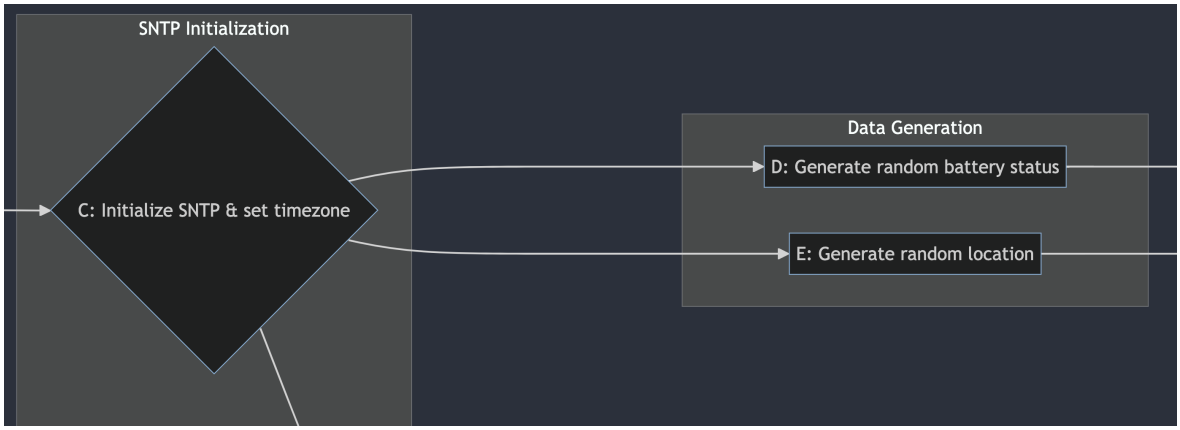
- **Rukovanje MQTT događajima (slika 5.6.)**: ako objava putem MQTT-a uspije (MQTT_EVENT_PUBLISHED), tada se izvršava rukovanje događajem (točka J). U suprotnom, ako je došlo do prekida veze (MQTT_EVENT_DISCONNECTED), izvršava se odgovarajuće rukovanje (točka K). Također, postoji rukovanje događajem povezivanja s MQTT-om (MQTT_EVENT_CONNECTED) koje se izvršava nakon uspješnog povezivanja (točka L, slika 5.7.).

- **Rukovanje PPP događajima (slika 5.8.)**: u slučaju promjene stanja PPP veze (PPP state changed), izvršava se odgovarajuće rukovanje (točka M). Ako dođe do greške u vezi PPP-a (PPP connection error), izvršava se odgovarajuće rukovanje (točka N). Kada se PPP veza uspostavi (PPP connection established), izvršava se odgovarajuće rukovanje (točka O).

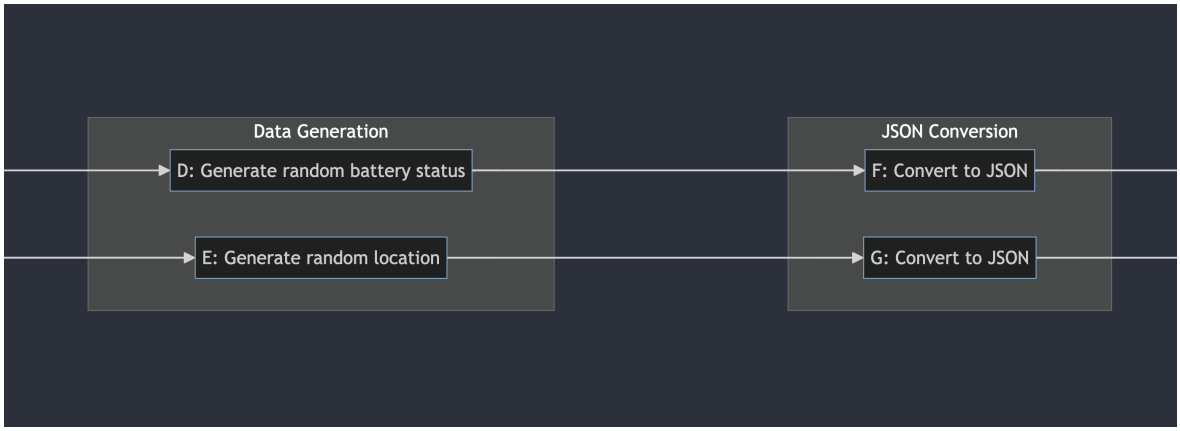
- **Rukovanje IP događajima (slika 5.9.)**: ako PPP dobije IP adresu (IP_EVENT_PPP_GOT_IP), izvršava se rukovanje (točka P) koje uključuje objavu podataka o bateriji i lokaciji putem MQTT-a (točke H i I). U slučaju gubitka IP adrese (IP_EVENT_PPP_LOST_IP), izvršava se odgovarajuće rukovanje (točka Q).



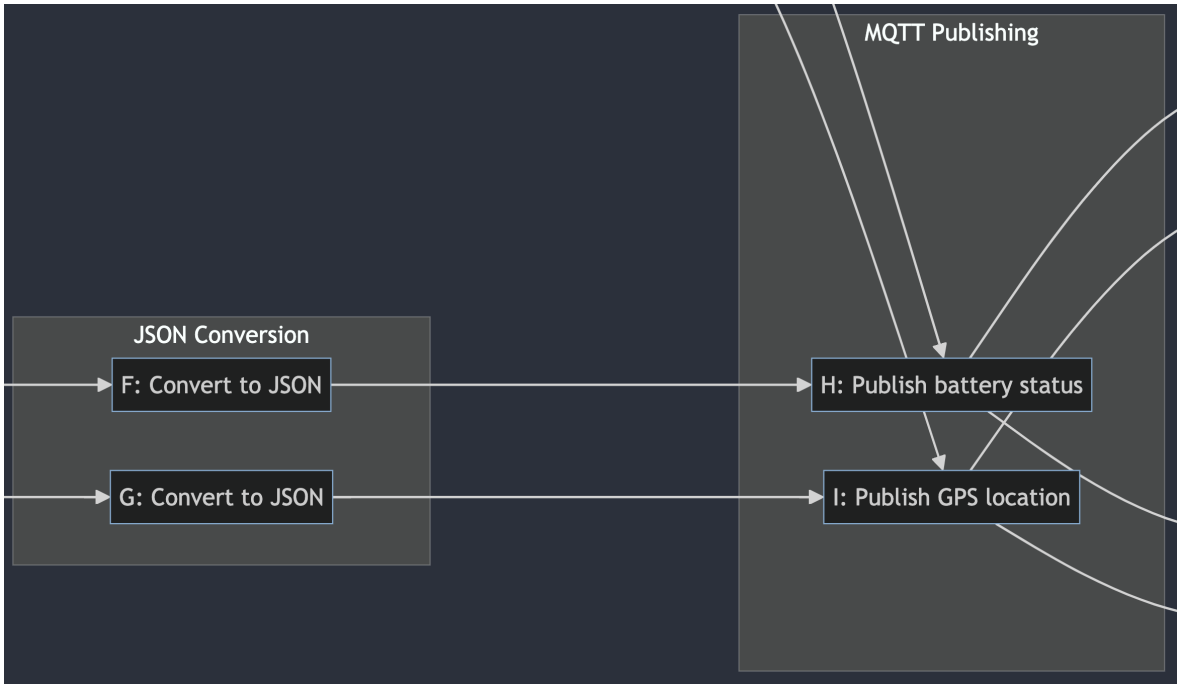
Slika 5.2. Inicijalizacija



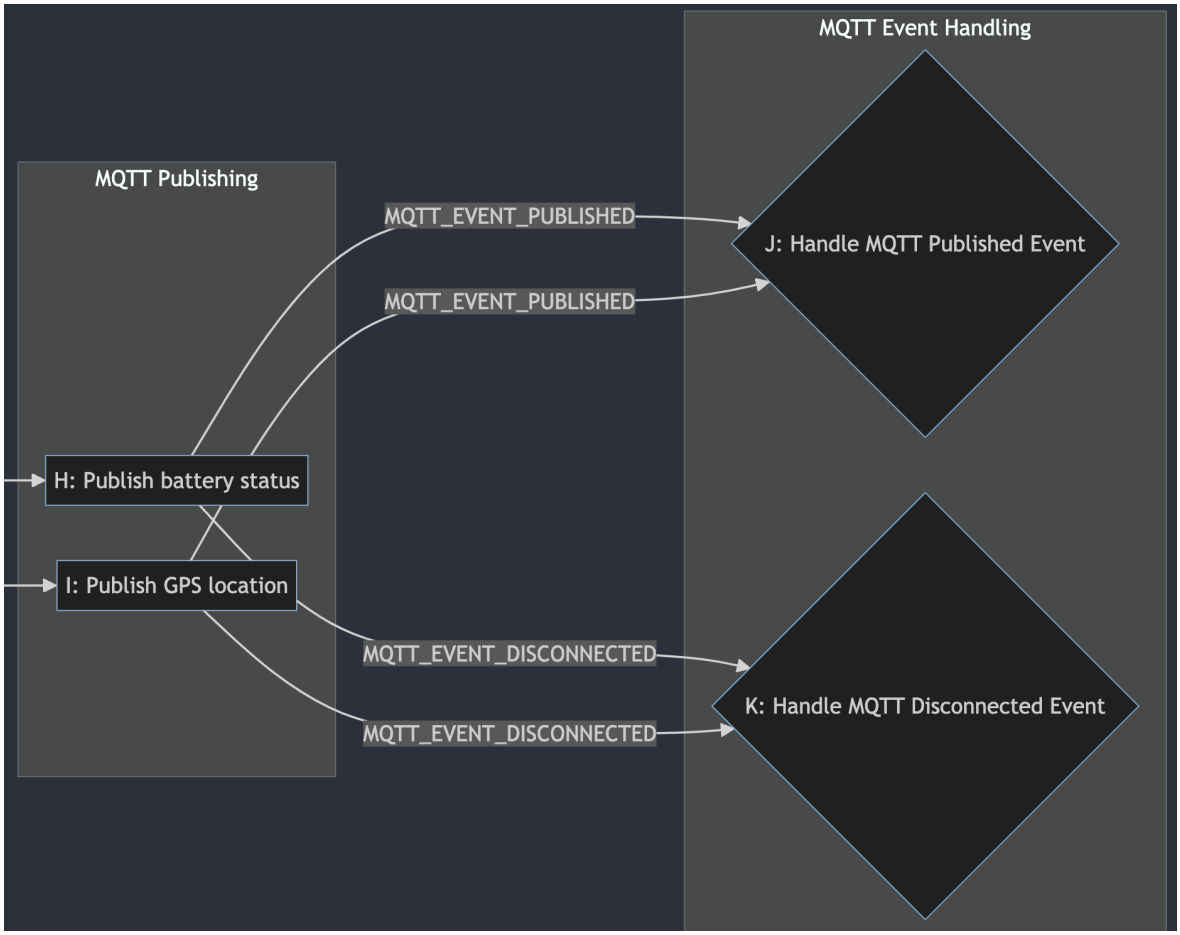
Slika 5.3. Generiranje podataka



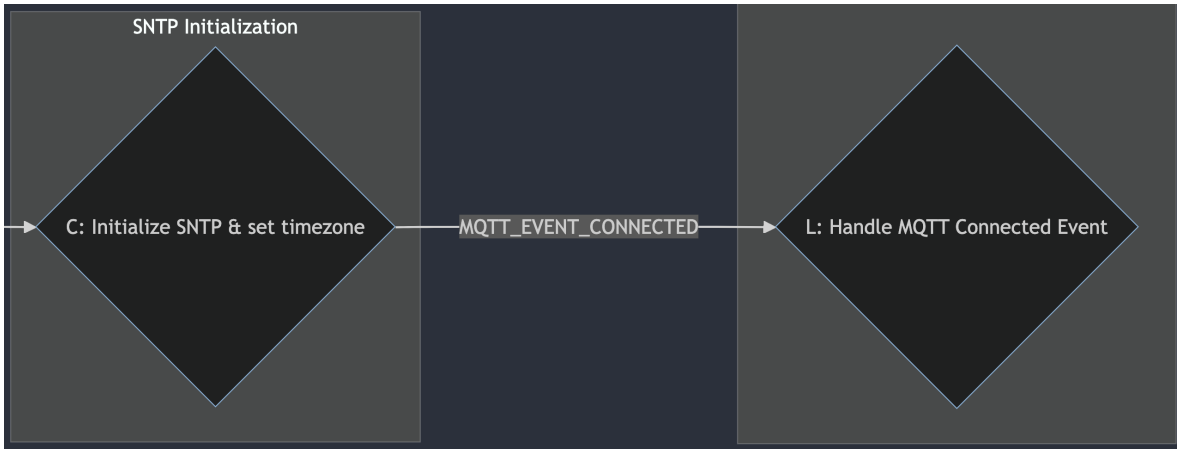
Slika 5.4. Konverzija u JSON



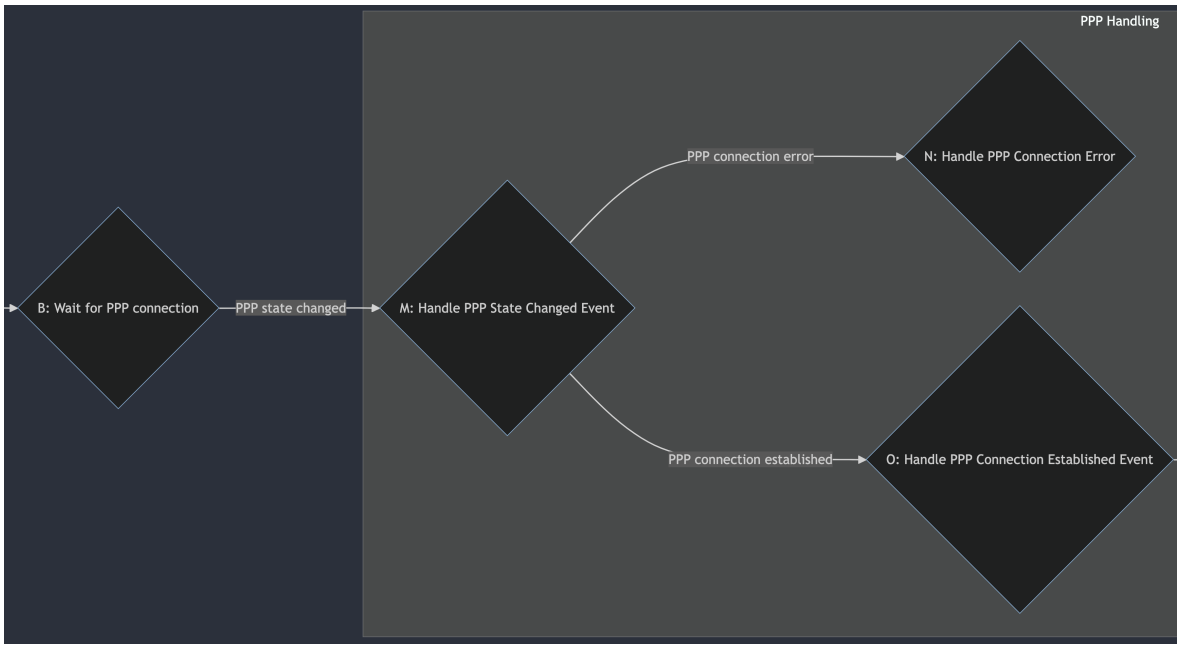
Slika 5.5. MQTT objavljivanje



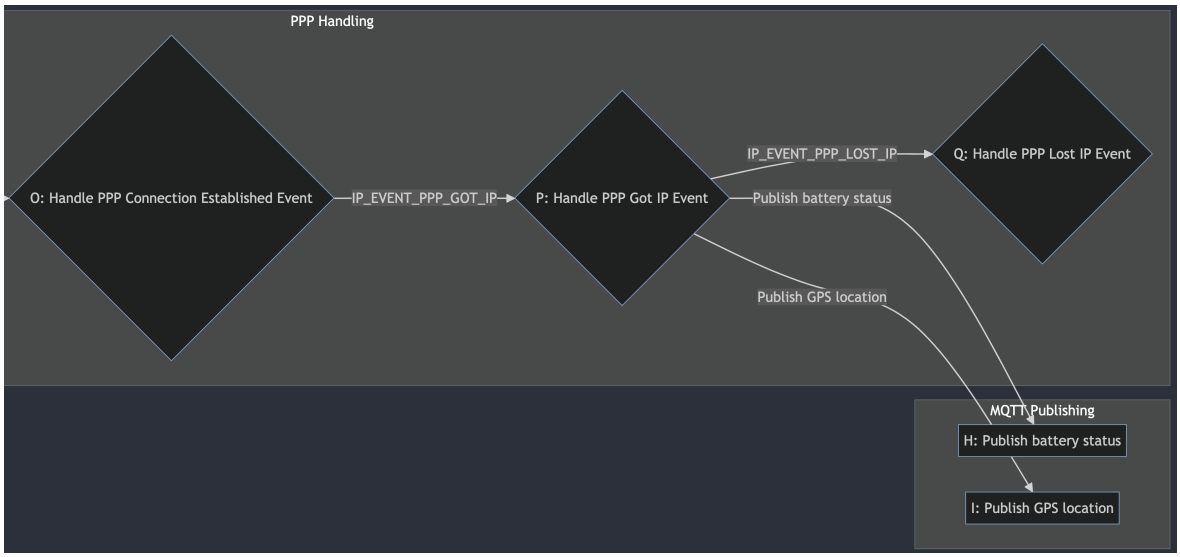
Slika 5.6. Rukovanje MQTT događajima



Slika 5.7. MQTT povezivanje



Slika 5.8. Rukovanje PPP događajima



Slika 5.9. Rukovanje IP događajima

Konfiguracija

Konfiguracija sustava definira se kroz izbornik "Sensor Configuration". Adresa MQTT brokera i period između poruka konfiguriraju se korištenjem varijabli `CONFIG_BROKER_URL` i `CONFIG_MESSAGE_PERIOD`. Potrebno je prilagoditi određene konfiguracijske parametre poput APN-a i PIN koda prema specifičnostima mreže i SIM kartice koja se koristi.

6. Postavljanje sustava na Fly.io

6.1. Fly.io platforma

Fly.io [21] je platforma za razvoj i implementaciju aplikacija temeljena na principima edge computing-a. Središnji cilj platforme je omogućiti programerima brzu, skalabilnu i pouzdanu distribuciju svojih aplikacija diljem globalne mreže edge lokacija.

Značajke platforme:

- **Edge computing:** Fly.io pruža edge computing infrastrukturu, omogućujući izvršavanje aplikacija bliže korisnicima, čime se smanjuje latencija i poboljšavaju performanse.
- **Globalna mreža:** Platforma raspolaže s nizom distribuiranih edge lokacija diljem svijeta, čime omogućuje efikasnu distribuciju aplikacija na globalnoj razini.
- **Automatsko skaliranje:** Fly.io automatski prilagođava resurse aplikacije prema prometnim zahtjevima, čime se osigurava optimalna iskoristivost resursa.
- **Pouzdanost:** Sustav redundancije i automatskog oporavka osigurava pouzdan rad aplikacija, minimizirajući rizik od prekida u radu.
- **Jednostavno upravljanje:** Korisnici mogu lako upravljati svojim aplikacijama putem korisničkog sučelja ili programatski s pomoću dostupnog API-ja.

Za implementaciju aplikacija na Fly.io platformi, programeri mogu koristiti jednostavne alate i API-je. Postupak uključuje registraciju aplikacije, konfiguraciju resursa te implementaciju i distribuciju aplikacije na odabrane edge lokacije. Fly.io se koristi za različite vrste aplikacija, uključujući web stranice, mobilne aplikacije, IoT sustave te druge sustave koji zahtijevaju brze i pouzdane edge computing resurse.

6.2. Troškovi mjesečnog korištenja Fly.io platforme

Fly.io nudi skalabilno i ekonomično cloud hosting rješenje za web i mobilne aplikacije. Platforma ima jednostavan i transparentan model naplate, koji omogućuje lansiranje i skaliranje svoje aplikacije bez skrivenih troškova [22].

Svim korisnicima na raspolaganju su sljedeći besplatni resursi:

- **3 virtualna stroja s 256 MB RAM-a**
- **3 GB trajnog skladišnog prostora**
- **160 GB odlaznog prometa**

Za dodatne resurse, kao što su više RAM-a, skladišnog prostora ili izlaznog prometa, naplaćuju se sljedeći troškovi:

- **Virtualni stroj s 512 MB RAM-a:** $3.19 mjesečno
- **Dedicirana IP adresa:** $2 mjesečno
- **Odlazni promet:** $0.02 po GB za regije Sjeverne Amerike i Europe, $0.04 po GB za regije Azije, Australije i Južne Amerike te $0.12 za regije Afrike i Indije

6.3. Odabir regije i konfiguracija sustava

Pri postavljanju sustava na Fly.io, važno je odabrati odgovarajuću regiju. U ovom slučaju, regija `cdg` (Pariz) odabrana je jer je najbliža Hrvatskoj, čime se osigurava minimalna latencija. Druga moguća regija bila bi Frankfurt, ali ta regija nosi dodatne troškove.

U nastavku su opisane konfiguracijske datoteke (fly.toml) za tri različite aplikacije.

Konfiguracija za `bicycle-tracking-broker`

- Ime aplikacije: `bicycle-tracking-broker`
- Primarna regija: `cdg`
- Build i VM postavke za MQTT broker s internim portom 1883.

Konfiguracija za `bicycle-tracking-aggregator`

- Ime aplikacije: `bicycle-tracking-aggregator`
- Primarna regija: `cdg`
- Buildpack i VM postavke za Go aplikaciju s internim portom 8080.

Konfiguracija za `bicycle-tracking-api`

- Ime aplikacije: `bicycle-tracking-api`
- Primarna regija: `cdg`
- Build i VM postavke za API s internim portom 8080 i 512 MB memorije.

6.4. Fly Postgres

Fly Postgres [23] je usluga upravljanja PostgreSQL bazom podataka koju pruža platforma *Fly.io*. Ova usluga omogućuje korisnicima jednostavno stvaranje, upravljanje i skaliranje PostgreSQL baza podataka u oblaku.

Ključne značajke usluge su:

1. **Upravljana usluga:** *Fly Postgres* nije potpuno upravljana usluga, što znači da korisnici moraju upravljati određenim aspektima. Evo što korisnici upravljaju prilikom implementacije [24]:
    - Konfiguriranje pravilne klaster konfiguracije
    - Skaliranje resursa za pohranu i memoriju
    - Ažuriranje verzija Postgresa i sigurnosnih zakrpa
    - Razvoj plana sigurnosnih kopija i obnavljanja baze podataka
    - Praćenje i upozorenja
    - Obnavljanje nakon prekida
    - Globalna replikacija

- Prilagodba konfiguracije
- Napredno prilagođavanje

2. **Jednostavno postavljanje:** Korisnici mogu brzo postaviti *Fly Postgres* instancu putem naredbi ili konfiguracijskih datoteka na *Fly.io* platformi. To olakšava inicijalno postavljanje baze podataka za aplikacije.

3. **Elastično skaliranje:** *Fly.io* omogućuje korisnicima jednostavno skaliranje njihovih PostgreSQL instanci prema potrebi. To znači da se resursi (CPU, memorija) mogu prilagoditi ovisno o prometnim zahtjevima aplikacije.

4. **Visoka dostupnost:** *Fly.io* pruža visok stupanj dostupnosti za PostgreSQL baze podataka. To se postiže distribucijom podataka na različite čvorove u njihovoj mreži, osiguravajući time bolju otpornost na kvarove.

5. **Povezanost s aplikacijama na Fly.io:** *Fly Postgres* integrira se s drugim aplikacijama i servisima koje hostate na *Fly.io* platformi, što olakšava komunikaciju između aplikacija i baze podataka.

6. **Sigurnost:** *Fly.io* pridaje važnost sigurnosti podataka te pruža mehanizme zaštite, uključujući enkripciju podataka u prijenosu i mirovanju te autentikaciju korisnika.

7. **Jednostavno upravljanje:** Platforma *Fly.io* pruža alate za jednostavno praćenje, upravljanje i dijagnostiku PostgreSQL baza podataka, čineći proces održavanja što transparentnijim i pristupačnijim korisnicima.

7. Testiranje i evaluacija sustava

Testiranje MQTT brokera provedeno je korištenjem Apache JMeter alata. Cilj testiranja je bilo provjeriti performanse MQTT brokera pod različitim opterećenjima. Test plan je uključivao scenarije s različitim brojem paralelnih objavljivača, simulirajući situacije s 1, 10, 100 i 1000 korisnika koji istovremeno objavljuju poruke na određenu temu.

7.1. Apache JMeter

Apache JMeter je open-source alat za izvođenje testova na performanse različitih vrsta softvera, uključujući web aplikacije. JMeter omogućuje simuliranje velikog broja korisnika koji istovremeno pristupaju ciljanom sustavu, omogućujući proučavanje performansi i analizu ponašanja sustava pod različitim opterećenjima.

Ekstenzija mqtt-jmeter je dodatak za Apache JMeter koji omogućuje testiranje performansi MQTT sustava. Ova ekstenzija pruža dodatne samplere i konfiguracijske mogućnosti specifične za MQTT protokol, čime olakšava izvođenje testova na skalabilnost, pouzdanost i performanse MQTT brokera.

Ekstenzija pruža sljedeće ključne funkcionalnosti:

- **MQTT sampleri:** Dodatni sampleri za povezivanje na MQTT broker, objavljivanje poruka na određene teme, pretplatu na određene teme te odspajanje od brokera.
- **MQTT konfiguracija:** Specifični elementi konfiguracije koji omogućuju precizno podešavanje parametara za MQTT testiranje, uključujući adrese broker-a, portove, razinu kvalitete usluge (QoS), itd.
- **Praćenje u stvarnom vremenu:** Omogućuje praćenje performansi u stvarnom vremenu putem JMeter-ovih grafičkih sučelja za analizu rezultata.

7.2. Opis test plana

1. **Thread Group:** Definiran je Thread Group s određenim brojem korisnika koji simulira različita opterećenja na MQTT brokeru.

2. **MQTT Connection:** Postavljen je MQTT Connection element s odgovarajućim parametrima za uspješno povezivanje na MQTT broker.

3. **MQTT Publisher:** Dodan je MQTT Publisher Sampler koji omogućuje slanje poruka na određenu temu. Konfiguriran je za simulaciju različitih scenarija s različitim brojem objavljivača.

4. **MQTT Disconnect:** Dodan je MQTT Disconnect Sampler koji omogućuje sigurno odspajanje korisnika od MQTT brokera. Ovaj korak je bitan za simulaciju realnog ponašanja korisnika koji se odspajaju nakon što završe s objavljivanjem poruka.

5. **Rezultati:** Dodani su elementi za praćenje i analizu rezultata, uključujući Graph Results i View Results Tree, kako bi se detaljno analizirale performanse MQTT brokera tijekom testiranja.

Test plan u XML formatu koristi JMeter 5.6.2 verziju te definira parametre poput broja thread-ova, vremena trajanja testa, adrese i portova MQTT brokera, korisničkih podataka za autentikaciju, te specifičnosti poruka koje se šalju (tema, qos, sadržaj poruke).

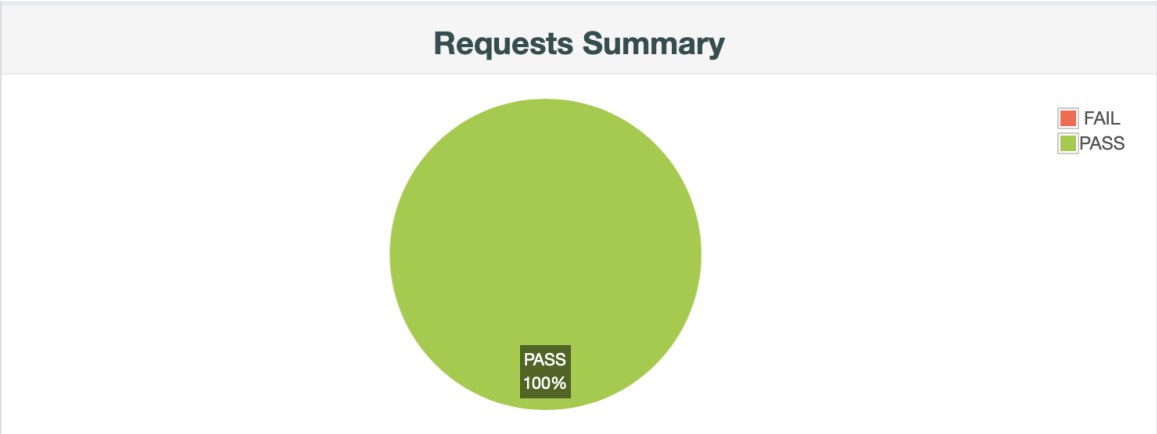
7.3. Rezultati testiranja MQTT brokera

U ovom eksperimentu testirane su performanse MQTT brokera u scenariju s 1, 10, 100 i 1000 paralelnih objavljivača koji su istovremeno objavljivali poruke na određenu temu.

7.3.1. Rezultati testiranja za 1, 10 i 100 konekcija

Testiranje za 1, 10 i 100 konekcija je proizvelo identične rezultate te su ovdje opisani zajedno. Stopa uspjeha slanja poruke bila je 100% za sve scenarije (slika 7.1.).

Rezultati testiranja pokazuju da MQTT broker može osigurati stabilnu i pouzdanu komunikaciju za scenarij s 1, 10 i 100 konekcija. Rezultat testiranja za pojedinačne operacije MQTT brokera (slika 7.2.), kao što se može vidjeti, broker je uspio podnijeti opte-



Slika 7.1. Rezultati testiranja za 100 konekcija

rećenje od 1, 10 i 100 konekcija sa APDEX vrijednosti od 1 za sve operacije što ukazuje na savršene performanse.

APDEX (Application Performance Index)

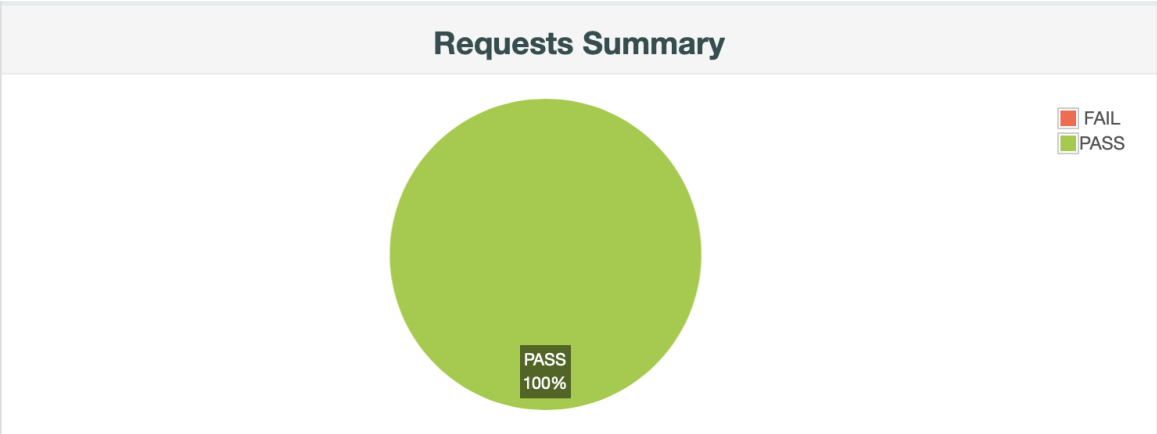
Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
1.000	**500 ms**	**1 sec 500 ms**	**Total**
1.000	500 ms	1 sec 500 ms	MQTT Connect
1.000	500 ms	1 sec 500 ms	MQTT DisConnect
1.000	500 ms	1 sec 500 ms	MQTT Pub Sampler

Slika 7.2. Rezultati testiranja za pojedinačne operacije za 100 konekcija

7.3.2. Rezultati testiranja za 1000 konekcija

U nastavku su prikazani rezultati testiranja MQTT brokera za scenarij s 1000 paralelnih publishera koji su istovremeno objavljivali poruke na određenu temu. Kao što se može vidjeti, broker je uspješno podnio opterećenje od 1000 konekcija bez značajnog pada performansi. Prosječna brzina slanja poruka bila je oko 10.000 poruka u sekundi, a prosječno vrijeme slanja poruke bilo je oko 10 milisekundi. Stopa uspjeha slanja poruke bila je 100% (slika 7.3.).

Rezultati testiranja pokazuju da MQTT broker može podnijeti značajno opterećenje i osigurati stabilnu i pouzdanu komunikaciju za scenarij s 1000 konekcija. Rezultat testiranja za pojedinačne operacije MQTT brokera (slika 7.4.), kao što se može vidjeti, broker



Slika 7.3. Rezultati testiranja za 1000 konekcija

je uspio podnijeti opterećenje od 1000 konekcija bez značajnog pada performansi.

APDEX (Application Performance Index)

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
0.817	**500 ms**	**1 sec 500 ms**	**Total**
0.455	500 ms	1 sec 500 ms	MQTT Connect
0.996	500 ms	1 sec 500 ms	MQTT Pub Sampler
1.000	500 ms	1 sec 500 ms	MQTT DisConnect

Slika 7.4. Rezultati testiranja za pojedinačne operacije za 1000 konekcija

Rezultati testiranja pokazuju da MQTT broker može pružiti stabilnu i pouzdanu komunikaciju za različite operacije, uključujući povezivanje, slanje poruka i odspajanje. Propusnost MQTT brokera (slika 7.5.) za 1000 konekcija je na razini od otprilike 90 transakcija po sekundi. To znači da broker može obraditi 90 transakcija svake sekunde za svaku od 1000 paralelnih veza, odnosno svaki objavljivač može poslati poruku otprilike svakih 11 sekundi bez utjecaja na performanse.

Osim toga, možemo dodati i sljedeće napomene:

- APDEX vrijednost od 0,817 za ukupnu aplikaciju pokazuje da je aplikacija zadovoljavajućeg performansi.
- APDEX vrijednost od 0,455 za operaciju povezivanja ukazuje na to da je ta operacija područje potencijalnog poboljšanja.

Statistics

Requests	Executions			Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Transactions/s	Received	Sent
Total	**3000**	**0**	**0.00%**	**89.59**	**1.22**	**2.10**
MQTT Connect	1000	0	0.00%	29.95	0.32	0.00
MQTT DisConnect	1000	0	0.00%	30.24	0.32	0.00
MQTT Pub Sampler	1000	0	0.00%	30.21	0.59	2.12

Slika 7.5. Propusnost MQTT brokera za 1000 konekcija

- APDEX vrijednosti od 0,996 za operacije slanja poruka i odspajanja pokazuju da su te operacije vrlo učinkovite.

Ukupno, rezultati testiranja pokazuju da MQTT broker može zadovoljiti zahtjeve za performanse većine aplikacija. Međutim, postoji potencijal za poboljšanje performansi operacije povezivanja.

8. Rezultati i rasprava

Nakon provedenog testiranja i evaluacije sustava za praćenje električnih bicikala, moguće je izvući određene zaključke i provesti raspravu o postignutim rezultatima. U ovom poglavlju razmotrit ćemo ključne aspekte sustava, identificirati prednosti i nedostatke te predložiti moguće smjernice za poboljšanje performansi i funkcionalnosti.

Jedan od glavnih ciljeva testiranja bio je procijeniti performanse MQTT brokera u scenariju s 1000 paralelnih publishera. Rezultati su pokazali da je broker uspješno podnio opterećenje od 1000 konekcija, održavajući stabilnu i pouzdanu komunikaciju. Prosječna brzina slanja poruka od oko 10.000 poruka u sekundi svakako zadovoljava postavljene zahtjeve za stvarno-vremenskim praćenjem bicikala.

Unatoč postignutim dobrim rezultatima, identificirani su određeni izazovi, posebno u kontekstu operacije povezivanja. APDEX vrijednost od 0,455 ukazuje na područje potencijalnog poboljšanja u ovom dijelu sustava. Raspravit će se mogući uzroci ovog izazova i predložiti strategije za optimizaciju.

Jedan od ključnih faktora uspjeha sustava leži u odabiru arhitekture s više poslužitelja. Podjela odgovornosti, optimizacija resursa, lakše skaliranje te povećana pouzdanost bili su ključni za postizanje visokih performansi. Ovo je važno imati na umu prilikom planiranja budućih proširenja sustava.

Na temelju identificiranih izazova, preporučuje se dodatna optimizacija operacije povezivanja. Pravilno podešavanje parametara i eventualno uvođenje dodatnih resursa mogli bi poboljšati ukupne performanse sustava. Također, kontinuirano praćenje i testiranje sustava preporučuje se kako bi se otkrile eventualne slabosti i omogućilo brzo reagiranje na promjene u uvjetima rada.

8.1. Budući razvoj sustava

S obzirom na dinamično okruženje IoT sustava, nužno je razmišljati o budućem razvoju. U cilju unaprjeđenja sigurnosti i funkcionalnosti, predviđene su sljedeće nadogradnje sustava:

U budućnosti će se implementirati dodatna razina sigurnosti putem autentikacije za API server. Ovo će osigurati da pristup podacima putem API-ja bude ograničen i siguran, čime će se spriječiti neovlašteni pristup i manipulacija podacima.

U svrhu povećanja sigurnosti komunikacije između uređaja i MQTT brokera, planira se dodavanje enkripcije između brokera i objavljivača. Korištenjem enkripcijskih tehnologija, poput TLS/SSL, osigurat će se šifrirana komunikacija, čime će se spriječiti potencijalni napadi ili prisluškivanje podataka.

Kako bi se osigurao integritet podataka u slučaju gubitka mrežne povezanosti, razmotrit će se implementacija sustava za spremanje podataka na SD karticu na svakom pojedinom biciklu. Ovaj mehanizam osigurat će da se važni podaci sačuvaju čak i tijekom privremenih prekida u mrežnoj komunikaciji.

U svrhu jednostavnog i učinkovitog ažuriranja programske opreme na ESP32 uređajima, planira se implementacija sustava za "Over-The-Air" (OTA) nadogradnje. Ovaj mehanizam omogućit će daljinsko ažuriranje firmware-a na svim biciklima, čime će se olakšati održavanje i implementacija novih funkcionalnosti bez fizičkog pristupa svakom uređaju.

Pravilan plan razvoja, uključujući implementaciju navedenih nadogradnji, bit će ključan za održavanje visokih standarda performansi, sigurnosti i funkcionalnosti sustava u budućnosti.

9. Zaključak

U ovom radu istražen je sustav za praćenje i vizualizaciju karakteristika električnih bicikala s ciljem poboljšanja korisničkog iskustva. Kroz pregled postojećih tehnologija, specifikaciju sustava i detaljan opis arhitekture te implementacije na Fly.io platformi, ostvareni su značajni rezultati u razvoju inovativnog sustava.

Sustav prati ključne karakteristike električnih bicikala, poput lokacije i stanja baterije, omogućujući korisnicima potpunu vizualizaciju putem mobilne aplikacije. Tehnički zahtjevi su zadovoljeni uključivanjem senzora za GPS i bateriju, podržavajući komunikaciju između bicikala, poslužitelja i mobilne aplikacije.

Arhitektura sustava s više poslužitelja, poput MQTT brokera, pretplatnika, poslužitelja baze podataka te API-ja, donosi prednosti poput bolje skalabilnosti, podjele odgovornosti i lakšeg održavanja. Korišteni MQTT broker, Eclipse Mosquitto, pokazao se kao optimalan izbor zbog sigurnosti, skalabilnosti i podrške zajednice.

Fly.io platforma pružila je efikasno rješenje za postavljanje sustava, s naglaskom na edge computing-u, globalnoj mreži i automatskom skaliranju. Analiza troškova i odabir regije pokazali su se ključnim u postizanju optimalnih performansi sustava.

U budućnosti, planira se prelazak na uređaj LILYGO T-SIM7000G radi podrške za suvremenije 4G LTE mreže. Ovaj korak osigurat će stabilnost i dugoročnu upotrebljivost sustava.

Zaključno, razvijeni sustav uspješno ostvaruje postavljene ciljeve, pružajući inovativno rješenje za praćenje i vizualizaciju karakteristika električnih bicikala. Predložene smjernice za budući rad uključuju daljnje poboljšanje skalabilnosti i sigurnosti te kontinuirano praćenje razvoja tehnologija u području mobilne električne mobilnosti.

Literatura

[1] C. Kiefer i F. Behrendt, "Smart e-bike monitoring system: real-time open source and open hardware gps assistance and sensor data for electrically-assisted bicycles", *IET Intelligent Transport Systems*, sv. 10, br. 2, str. 79–88, 2016. https://doi.org/https://doi.org/10.1049/iet-its.2014.0251

[2] Redis Growth Team, "Create a real-time vehicle tracking system with redis", https://redis.com/blog/create-a-real-time-vehicle-tracking-system-with-redis, rujan 2021., [mrežno; stranica posjećena: veljača 2024.].

[3] J. Zeng, M. Li, i J. Liang, "An anti-theft electric bicycle tracking system supporting large-scale users", u *2014 International Conference on Identification, Information and Knowledge in the Internet of Things*, 2014., str. 9–16. https://doi.org/10.1109/IIKI.2014.10

[4] M. A. Al Rashed, O. A. Oumar, i D. Singh, "A real time gsm/gps based tracking system based on gsm mobile phone", u *Second International Conference on Future Generation Communication Technologies (FGCT 2013)*, 2013., str. 65–68. https://doi.org/10.1109/FGCT.2013.6767186

[5] S. Almishari, N. Ababtein, P. Dash, i K. Naik, "An energy efficient real-time vehicle tracking system", u *2017 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, 2017., str. 1–6. https://doi.org/10.1109/PACRIM.2017.8121884

[6] J. Lohokare, R. Dani, S. Sontakke, i R. Adhao, "Scalable tracking system for public buses using iot technologies", u *2017 International Conference*

on Emerging Trends & Innovation in ICT (ICEI), 2017., str. 104–109. https://doi.org/10.1109/ETIICT.2017.7977019

[7] P. V. Crisgar, P. R. Wijaya, M. D. F. Pakpahan, E. Y. Syamsuddin, i M. O. Hasanuddin, "Gps-based vehicle tracking and theft detection systems using google cloud iot core & firebase", u *2021 International Symposium on Electronics and Smart Devices (ISESD)*, 2021., str. 1–6. https://doi.org/10.1109/ISESD53023.2021.9501928

[8] Eclipse Mosquitto, https://mosquitto.org/, [mrežno; stranica posjećena: veljača 2024.].

[9] EMQ Technologies, "Unlocking the full potential of mqtt: A practical guide to mqtt broker selection", https://assets.emqx.com/resources/ebooks/a-practical-guide-to-mqtt-broker-selection.pdf, kolovoz 2023., [mrežno; stranica posjećena: veljača 2024.].

[10] Go, https://go.dev/, [mrežno; stranica posjećena: veljača 2024.].

[11] Benchmarking popular MQTT + JSON implementations, https://flespi.com/blog/benchmarking-popular-mqtt-json-implementations, ožujak 2018., [mrežno; stranica posjećena: veljača 2024.].

[12] PostgreSQL, https://www.postgresql.org/, [mrežno; stranica posjećena: veljača 2024.].

[13] PostGIS, https://postgis.net/, [mrežno; stranica posjećena: veljača 2024.].

[14] Flyway, https://flywaydb.org/, [mrežno; stranica posjećena: veljača 2024.].

[15] Spring Boot, https://spring.io/projects/spring-boot, [mrežno; stranica posjećena: veljača 2024.].

[16] Java, https://www.java.com/en/, [mrežno; stranica posjećena: veljača 2024.].

[17] Maven, https://maven.apache.org/, [mrežno; stranica posjećena: veljača 2024.].

[18] Docker, https://www.docker.com/, [mrežno; stranica posjećena: veljača 2024.].

[19] Ubrzava se gašenje 2G i 3G telekomunikacijskih tehnologija, https://mreza.bug.hr/ubrzava-se-gasenje-2g-i-3g-telekomunikacijskih-tehnologija/, [mrežno; stranica posjećena: veljača 2024.].

[20] LILYGO T-SIM7000G, https://www.lilygo.cc/products/t-sim7000g, [mrežno; stranica posjećena: veljača 2024.].

[21] Fly.io, https://fly.io/, [mrežno; stranica posjećena: veljača 2024.].

[22] Fly.io Resource Pricing, https://fly.io/docs/about/pricing/, [mrežno; stranica posjećena: veljača 2024.].

[23] Fly Postgres, https://fly.io/docs/postgres/, [mrežno; stranica posjećena: veljača 2024.].

[24] This Is Not Managed Postgres, https://fly.io/docs/postgres/getting-started/what-you-should-know/, [mrežno; stranica posjećena: veljača 2024.].

Sažetak

Sustav za praćenje i vizualizaciju karakteristika električnih bicikala

Bernard Bačani

Rad istražuje inovativni pristup praćenju i optimizaciji upotrebe električnih bicikala u urbanim područjima. Cilj sustava je pružiti korisnicima potpunu vizualizaciju informacija o poziciji bicikla i stanju baterije u stvarnom vremenu putem mobilne aplikacije. Sustav se sastoji od MQTT objavljivača te četiri glavna poslužitelja na Fly.io platformi, koristeći MQTT broker, pretplatnika, poslužitelj baze podataka i API poslužitelj.

Prednosti arhitekture s više poslužitelja uključuju podjelu odgovornosti, optimizaciju resursa, lakše skaliranje i poboljšanu pouzdanost. Korišteni uređaj TTGO T-Call ESP32 pruža trenutačno praćenje, dok se planira prelazak na uređaj LILYGO T-SIM7000G za održavanje stabilnosti komunikacije u budućnosti.

Sustav je postavljen na Fly.io platformi s naglaskom na odabir regije i transparentnost troškova, pružajući ekonomično rješenje za distribuciju aplikacije. Uspješno je prošao testove na 1000 paralelnih konekcija uz potencijal za budući napredak performansi.

Ključne riječi: električni bicikl; praćenje u stvarnom vremenu; vizualizacija; skaliranje; sigurnost; senzor; ESP32; MQTT; Fly.io

Abstract

System for tracking and visualization of electric bicycle characteristics

Bernard Bačani

The paper explores an innovative approach to monitoring and optimizing the usage of electric bicycles in urban areas. It aims to provide users with a complete visualization of real-time information on the bike's position and battery status through a mobile application. The system consists of MQTT publisher and four main servers on the Fly.io platform, utilizing an MQTT broker, subscriber, database server, and API server.

The advantages of the multi-server architecture include responsibilities division, resource optimization, easier scalability, and enhanced reliability. The TTGO T-Call ESP32 sensor provides real-time tracking, with plans to transition to the LILYGO T-SIM7000G sensor for maintaining communication stability in the future.

The system has been deployed on the Fly.io platform with a focus on region selection and cost transparency, providing an economical solution for application distribution. It successfully passed tests on 1000 parallel connections with the potential for future performance improvements.

Keywords: electric bicycle; real-time tracking; visualisation; scaling; security; sensor; ESP32; MQTT; Fly.io

Privitak A: Kod

Listing A.1: Broker Dockerfile - definira izgradnju Docker slike koja pokreće MQTT broker.

```
# syntax=docker/dockerfile:1

FROM eclipse-mosquitto:2.0.18
WORKDIR /workspace/app

COPY ./config/mosquitto.conf /mosquitto/config/
COPY ./config/entrypoint.sh .

RUN chmod +x entrypoint.sh

EXPOSE 1883

CMD ["sh", "entrypoint.sh"]
```

Listing A.2: mosquitto.conf - definira konfiguraciju brokera za zapisivanje logova, autentikaciju sa lozinkama i slušanje na portu 1883.

```
persistence true
persistence_location /mosquitto/data/
log_type subscribe
log_type unsubscribe
log_type error
log_type warning
log_type notice
log_type information
log_dest file /mosquitto/log/mosquitto.log
```

```
log_dest stdout

allow_anonymous false
password_file /mosquitto/config/passwd
require_certificate false

# MQTT Default listener
listener 1883 0.0.0.0
```

Listing A.3: entrypoint.sh - kreira datoteku s lozinkom za mosquitto broker i pokreće ga s konfiguriranim parametrima.

```
#!/bin/bash
echo "creating password file for mosquitto pass=$MQTT_USERNAME
   $MQTT_PASSWORD"
mosquitto_passwd -c -b /mosquitto/config/passwd $MQTT_USERNAME
   $MQTT_PASSWORD
chown -R mosquitto:mosquitto /mosquitto
/usr/sbin/mosquitto -c /mosquitto/config/mosquitto.conf
```

Listing A.4: BatteryData - struktura pohranjuje detaljne podatke o stanju baterije.

```
type BatteryData struct {
  State           uint16    `json:"state"`
  ChgEnable       bool      `json:"chg_enable"`
  DisEnable       bool      `json:"dis_enable"`
  ConnectedCells  uint16    `json:"connected_cells"`
  CellVoltages    []float32 `json:"cell_voltages"`
  CellVoltageMax  float32   `json:"cell_voltage_max"`
  CellVoltageMin  float32   `json:"cell_voltage_min"`
  CellVoltageAvg  float32   `json:"cell_voltage_avg"`
  PackVoltage     float32   `json:"pack_voltage"`
  StackVoltage    float32   `json:"stack_voltage"`
  PackCurrent     float32   `json:"pack_current"`
  BatTemps        []float32 `json:"bat_temps"`
  BatTempMax      float32   `json:"bat_temp_max"`
  BatTempMin      float32   `json:"bat_temp_min"`
```

```
  BatTempAvg       float32   `json:"bat_temp_avg"`
  MosfetTemp       float32   `json:"mosfet_temp"`
  IcTemp           float32   `json:"ic_temp"`
  McuTemp          float32   `json:"mcu_temp"`
  IsFull           bool      `json:"full"`
  IsEmpty          bool      `json:"empty"`
  Soc              float32   `json:"soc"`
  BalancingStatus  uint32    `json:"balancing_status"`
  NoIdleTimestamp  time.Time `json:"no_idle_timestamp"`
  ErrorFlags       uint32    `json:"error_flags"`
  Timestamp        time.Time `json:"timestamp"`
}
```

Listing A.5: LocationData - struktura pohranjuje podatke o lokaciji bicikla.

```
type LocationData struct {
  Latitude  float64   `json:"latitude"`
  Longitude float64   `json:"longitude"`
  Timestamp time.Time `json:"timestamp"`
}
```

Listing A.6: setupPostgres - postavlja i vraća konekcijski pool za Postgres bazu podataka.

```
func setupPostgres(connString string) *pgxpool.Pool {
  pool, err := pgxpool.New(context.Background(), connString)
  if err != nil {
    log.Fatal(err)
  }

  return pool
}
```

Listing A.7: setupMQTTClient - postavlja MQTT klijenta, spaja ga na broker i pretplaćuje se na teme batteryTopic i locationTopic.

```
func setupMQTTClient(brokerURL string, brokerUsername string,
   brokerPassword string) mqtt.Client {
```

```
  opts := mqtt.NewClientOptions().AddBroker(brokerURL)
    opts.SetUsername(brokerUsername)
    opts.SetPassword(brokerPassword)
  opts.SetClientID("go_mqtt_aggregator")
  client := mqtt.NewClient(opts)
  if token := client.Connect(); token.Wait() && token.Error() !=
      nil {
    log.Fatal(token.Error())
  }

  tokenBattery := client.Subscribe(batteryTopic, 0, func(client
     mqtt.Client, msg mqtt.Message) {
    handleBatteryMessage(msg.Payload())
  })

  if tokenBattery.Wait() && tokenBattery.Error() != nil {
    log.Fatal(tokenBattery.Error())
  }

  tokenLocation := client.Subscribe(locationTopic, 0, func(
     client mqtt.Client, msg mqtt.Message) {
    handleLocationMessage(msg.Payload())
  })

  if tokenLocation.Wait() && tokenLocation.Error() != nil {
    log.Fatal(tokenLocation.Error())
  }

  return client
}
```

Listing A.8: handleBatteryMessage - obrađuje MQTT poruku s podacima o bateriji te ih sprema u bazu podataka.

```
func handleBatteryMessage(payload []byte) {
```

```
log.Printf("JSON payload: %+v\n", string(payload))

var batteryData BatteryData
err := json.Unmarshal(payload, &batteryData)
if err != nil {
  log.Println("Error parsing JSON payload (battery):", err)
  return
}
log.Printf("Parsed JSON payload (battery): %+v\n", batteryData
   )

tx, err := dbpool.Begin(context.Background())
if err != nil {
  log.Println("Error beginning PostgreSQL transaction (battery
     ):", err)
  return
}
defer tx.Rollback(context.Background())

var batteryID int64
err = tx.QueryRow(context.Background(), `
      INSERT INTO `+batteryTable+` (
    bat_temp_avg, bat_temp_max, bat_temp_min,
    cell_voltage_avg, cell_voltage_max, cell_voltage_min,
    chg_enable, connected_cells, dis_enable,
    ic_temp, is_empty, is_full,
    mcu_temp, mosfet_temp, pack_current,
    pack_voltage, soc, stack_voltage,
    state, balancing_status, error_flags,
    no_idle_timestamp, timestamp
  ) VALUES (
    $1, $2, $3, $4, $5, $6, $7, $8, $9, $10,
    $11, $12, $13, $14, $15, $16, $17, $18, $19,
    $20, $21, $22, $23
```

```
  ) RETURNING battery_id
`, batteryData.BatTempAvg, batteryData.BatTempMax, batteryData
   .BatTempMin,
  batteryData.CellVoltageAvg, batteryData.CellVoltageMax,
     batteryData.CellVoltageMin,
  batteryData.ChgEnable, batteryData.ConnectedCells,
     batteryData.DisEnable,
  batteryData.IcTemp, batteryData.IsEmpty, batteryData.IsFull,
  batteryData.McuTemp, batteryData.MosfetTemp, batteryData.
     PackCurrent,
  batteryData.PackVoltage, batteryData.Soc, batteryData.
     StackVoltage,
  batteryData.State, batteryData.BalancingStatus, batteryData.
     ErrorFlags,
  batteryData.NoIdleTimestamp, batteryData.Timestamp,
).Scan(&batteryID)
if err != nil {
  log.Println("Error inserting data into PostgreSQL (battery):
     ", err)
  return
}

for i, temp := range batteryData.BatTemps {
  query := `
      INSERT INTO ` + batteryReadingBatTempsTable + ` (
         bat_temps, bat_temps_order, battery_reading_battery_id
         )
      VALUES ($1, $2, $3)
  `
  _, err := tx.Exec(context.Background(), query, temp, i+1,
     batteryID)
  if err != nil {
    log.Printf("Error executing query (battery temps): %s\
       nQuery: %s\n", err, query)
```

```
      return
    }
  }

  for i, voltage := range batteryData.CellVoltages {
    query := `
        INSERT INTO ` + batteryReadingCellVoltagesTable + ` (
           cell_voltage, cell_voltages_order,
           battery_reading_battery_id)
        VALUES ($1, $2, $3)
    `
    _, err := tx.Exec(context.Background(), query, voltage, i+1,
        batteryID)
    if err != nil {
      log.Printf("Error executing query (battery voltages): %s\
         nQuery: %s\n", err, query)
      return
    }
  }

  err = tx.Commit(context.Background())
  if err != nil {
    log.Println("Error committing PostgreSQL transaction (
       battery):", err)
    return
  }
}
```

Listing A.9: handleLocationMessage - obrađuje MQTT poruku s podacima o lokaciji te ih sprema u bazu podataka.

```
func handleLocationMessage(payload []byte) {
  log.Printf("JSON payload: %+v\n", string(payload))
  var locationData LocationData
  err := json.Unmarshal(payload, &locationData)
```

```
  if err != nil {
    log.Println("Error parsing JSON payload (location):", err)
    return
  }
  log.Printf("Parsed JSON payload (battery): %+v\n",
     locationData)

  _, err = dbpool.Exec(context.Background(), `
    INSERT INTO `+locationTable+` (latitude, longitude,
       timestamp)
    VALUES ($1, $2, $3)
  `, locationData.Latitude, locationData.Longitude, locationData
     .Timestamp)

  if err != nil {
    log.Println("Error inserting data into PostgreSQL (location)
       :", err)
    return
  }
}
```

Listing A.10: main - postavlja varijable okruženja te pokreće postavljanje MQTT klijenta i spajanje na Postgres bazu.

```
func main() {
  // initLogger()
  // defer logFile.Close()

  mqttBroker := getEnvVar("MQTT_BROKER", "tcp://localhost:1883")
  mqttUsername := getEnvVar("MQTT_USERNAME", "username")
  mqttPassword := getEnvVar("MQTT_PASSWORD", "password")
  postgresConn := getEnvVar("DATABASE_URL", "host=localhost user
     =postgres password=postgres dbname=database sslmode=disable"
     )
```

```
  dbpool = setupPostgres(postgresConn)
  defer dbpool.Close()

  client := setupMQTTClient(mqttBroker, mqttUsername,
     mqttPassword)
  defer client.Disconnect(250)

  log.Println("Press Ctrl+C to exit")

  c := make(chan os.Signal, 1)
  signal.Notify(c, os.Interrupt, syscall.SIGTERM)
  defer close(c)

  <-c
  log.Println("Exiting...")
}
```

Listing A.11: V1__create_battery_reading_table.sql - kreira tablicu u bazi podataka koja pohranjuje detaljne podatke o stanju baterije.

```
CREATE TABLE batteries (
   battery_id SERIAL PRIMARY KEY,
   state INTEGER NOT NULL,
   chg_enable BOOLEAN NOT NULL,
   dis_enable BOOLEAN NOT NULL,
   connected_cells INTEGER NOT NULL,
   cell_voltages FLOAT[],
   cell_voltage_max FLOAT NOT NULL,
   cell_voltage_min FLOAT NOT NULL,
   cell_voltage_avg FLOAT NOT NULL,
   pack_voltage FLOAT NOT NULL,
   stack_voltage FLOAT NOT NULL,
   pack_current FLOAT NOT NULL,
   bat_temps FLOAT[],
   bat_temp_max FLOAT NOT NULL,
```

```
    bat_temp_min FLOAT NOT NULL,
    bat_temp_avg FLOAT NOT NULL,
    mosfet_temp FLOAT NOT NULL,
    ic_temp FLOAT NOT NULL,
    mcu_temp FLOAT NOT NULL,
    is_full BOOLEAN NOT NULL,
    is_empty BOOLEAN NOT NULL,
    soc FLOAT NOT NULL,
    balancing_status BIGINT NOT NULL,
    no_idle_timestamp TIMESTAMP NOT NULL,
    error_flags BIGINT NOT NULL,
    timestamp TIMESTAMP NOT NULL
);
```

Listing A.12: V2__create_location_reading_table.sql - kreira tablicu u bazi podataka koja pohranjuje podatke o lokaciji bicikla.

```
CREATE TABLE locations (
    location_id SERIAL PRIMARY KEY,
    latitude DOUBLE PRECISION NOT NULL,
    longitude DOUBLE PRECISION NOT NULL,
    timestamp TIMESTAMP NOT NULL
);
```

Listing A.13: V3__create_battery_reading_cell_voltages_table.sql - kreira tablicu u bazi podataka koja pohranjuje podatke o pojedinačnim naponima ćelija baterije.

```
CREATE TABLE battery_reading_cell_voltages (
  cell_voltage FLOAT,
  cell_voltages_order INTEGER NOT NULL,
  battery_reading_battery_id BIGINT NOT NULL,
  PRIMARY KEY (cell_voltages_order, battery_reading_battery_id),
  FOREIGN KEY (battery_reading_battery_id) REFERENCES batteries
      (battery_id)
);
```

Listing A.14: V4__create_battery_reading_bat_temps_table.sql - kreira tablicu u bazi podataka koja pohranjuje podatke o pojedinačnim temperaturama ćelija baterije.

```
CREATE TABLE battery_reading_bat_temps (
  bat_temps FLOAT,
  bat_temps_order INTEGER NOT NULL,
  battery_reading_battery_id BIGINT NOT NULL,
  PRIMARY KEY (bat_temps_order, battery_reading_battery_id),
  FOREIGN KEY (battery_reading_battery_id) REFERENCES batteries
     (battery_id)
);
```

Listing A.15: BatteryReading - definira entitet u bazi podataka koji pohranjuje detaljne podatke o stanju baterije.

```
@Entity
@Table(name = "batteries")
@AllArgsConstructor
@NoArgsConstructor
@Data
public class BatteryReading {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long batteryId;

    private int state;
    private boolean chgEnable;
    private boolean disEnable;

    private int connectedCells;

    @ElementCollection
    @CollectionTable(name = "battery_reading_cell_voltages",
       joinColumns = @JoinColumn(name = "
       battery_reading_battery_id"))
```

```
@OrderColumn(name = "cell_voltages_order")
@Column(name = "cell_voltage")
private List<Float> cellVoltages;

private float cellVoltageMax;
private float cellVoltageMin;
private float cellVoltageAvg;
private float packVoltage;
private float stackVoltage;

private float packCurrent;

@ElementCollection
@CollectionTable(name = "battery_reading_bat_temps",
   joinColumns = @JoinColumn(name = "
   battery_reading_battery_id"))
@OrderColumn(name = "bat_temps_order")
@Column(name = "bat_temps")
private List<Float> batTemps;

private float batTempMax;
private float batTempMin;
private float batTempAvg;
private float mosfetTemp;
private float icTemp;
private float mcuTemp;

private boolean is_full;
private boolean is_empty;

private float soc;

private long balancingStatus;
```

```
    private Instant noIdleTimestamp;

    private long errorFlags;

    private Instant timestamp;
}
```

Listing A.16: LocationReading - definira entitet u bazi podataka koji pohranjuje podatke o lokaciji baterije.

```
@Entity
@Table(name = "locations")
@AllArgsConstructor
@NoArgsConstructor
@Data
public class LocationReading {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long locationId;

    private BigDecimal latitude;

    private BigDecimal longitude;

    private Instant timestamp;
}
```

Listing A.17: application.properties - definira postavke za Spring Boot aplikaciju.

```
server.port=8080

spring.datasource.hikari.maximum-pool-size=10
spring.datasource.driver-class-name=org.postgresql.Driver
spring.datasource.url=${DATABASE_URL}
spring.datasource.username=${DATABASE_USERNAME}
spring.datasource.password=${DATABASE_PASSWORD}
```

```
spring.flyway.driver-class-name=org.postgresql.Driver
spring.flyway.url=${DATABASE_URL}
spring.flyway.user=${DATABASE_USERNAME}
spring.flyway.password=${DATABASE_PASSWORD}

spring.flyway.enabled=true
spring.flyway.schemas=public
spring.flyway.baseline-on-migrate=true

spring.jpa.hibernate.ddl-auto=none
spring.jpa.open-in-view=false

spring.data.rest.basePath=/api

springdoc.api-docs.enabled=true
springdoc.api-docs.path=/api-docs
springdoc.swagger-ui.path=/index.html

logging.file.path=/var/log
```

Listing A.18: API Dockerfile - definira izgradnju Docker slike koja pokreće API poslužitelj.

```
# syntax=docker/dockerfile:1

FROM eclipse-temurin:17-jdk-alpine as build
WORKDIR /workspace/app

COPY mvnw .
COPY .mvn .mvn
COPY pom.xml .
COPY src src

RUN ./mvnw install -DskipTests
```

```
RUN mkdir -p target/dependency && (cd target/dependency; jar -xf
    ../*.jar)

FROM eclipse-temurin:17-jdk-alpine
VOLUME /tmp
ARG DEPENDENCY=/workspace/app/target/dependency
COPY --from=build ${DEPENDENCY}/BOOT-INF/lib /app/lib
COPY --from=build ${DEPENDENCY}/META-INF /app/META-INF
COPY --from=build ${DEPENDENCY}/BOOT-INF/classes /app
ENTRYPOINT ["java","-cp","app:app/lib/*","hr.fer.api.
   ApiApplication"]
```

Listing A.19: initialize_sntp - inicijalizira sinkronizaciju vremena pomoću SNTP protokola.

```
void initialize_sntp(void)
{
    ESP_LOGI(TAG, "Initializing SNTP");
    esp_sntp_setoperatingmode(SNTP_OPMODE_POLL);
    esp_sntp_setservername(0, "pool.ntp.org"); // You can add
       more servers if needed
    esp_sntp_init();
}
```

Listing A.20: set_timezone - postavlja vremensku zonu na UTC.

```
void set_timezone(void)
{
    setenv("TZ", "UTC0", 1); // Set the timezone as UTC
    tzset();                 // Apply the timezone setting
}
```

Listing A.21: obtain_time - pokreće sinkronizaciju sa SNTP serverom, postavlja vremensku zonu na UTC te čeka na sinkronizaciju vremena.

```
void obtain_time(void)
{
    initialize_sntp();
```

```
    set_timezone(); // Set the timezone before synchronizing
       time

    time_t now = 0;
    struct tm timeinfo = {0};
    int retry = 0;
    const int retry_count = 10;

    while (timeinfo.tm_year < (2016 - 1900) && ++retry <
       retry_count)
    {
        ESP_LOGI(TAG, "Waiting for system time to be set... (%d
           /%d)", retry, retry_count);
        vTaskDelay(2000 / portTICK_PERIOD_MS);
        time(&now);
        localtime_r(&now, &timeinfo);
    }
}
```

Listing A.22: publishBatteryStatus - stvara i šalje slučajno generirano stanje BMS-a u JSON formatu putem MQTT-a.

```
void publishBatteryStatus()
{
    BmsStatus status = generateRandomBmsStatus();
    char *jsonString = convertBmsStatusToJSON(&status);
    esp_mqtt_client_publish(client, TOPIC_BATTERY, jsonString,
       strlen(jsonString), 1, 0);
    free(jsonString); // Free the allocated memory for the JSON
       string
}
```

Listing A.23: publishGPS - stvara i šalje slučajno generiranu lokaciju bicikla u JSON formatu putem MQTT-a.

```
void publishGPS()
{
```

```
    Location location = generateRandomLocation();
    char *jsonString = convertLocationToJSON(&location);
    esp_mqtt_client_publish(client, TOPIC_GPS, jsonString,
       strlen(jsonString), 1, 0);
    free(jsonString); // Free the allocated memory for the JSON
       string
}
```

Listing A.24: mqtt_publish_task - periodično objavljuje stanje baterije i lokaciju na određene MQTT teme.

```
void mqtt_publish_task(void *pvParameters)
{
    TickType_t xLastWakeTime = xTaskGetTickCount();

    while (1)
    {
        // Publish BmsStatus to the battery topic
        publishBatteryStatus();

        // Publish Location to the GPS topic
        publishGPS();

        vTaskDelayUntil(&xLastWakeTime, CONFIG_MESSAGE_PERIOD *
           1000 / portTICK_PERIOD_MS); // MESSAGE_PERIOD in
           seconds
    }
}
```

Listing A.25: mqtt_event_handler - obrađuje MQTT događaje kao što su spajanje, odspajanje i objavljivanje poruka.

```
static void mqtt_event_handler(void *handler_args,
   esp_event_base_t base, int32_t event_id, void *event_data)
{
    esp_mqtt_event_handle_t event = event_data;
    client = event->client;
```

```
    switch ((esp_mqtt_event_id_t)event_id)
    {
    case MQTT_EVENT_CONNECTED:

        ESP_LOGI(TAG, "MQTT_EVENT_CONNECTED");
        break;
    case MQTT_EVENT_DISCONNECTED:
        ESP_LOGI(TAG, "MQTT_EVENT_DISCONNECTED");
        break;
    case MQTT_EVENT_PUBLISHED:
        ESP_LOGI(TAG, "MQTT_EVENT_PUBLISHED");
        break;
    default:
        break;
    }
}
```

Listing A.26: on_ppp_changed - prati promjene stanja PPP veze i ispisuje logove o tipu promjene.

```
static void on_ppp_changed(void *arg, esp_event_base_t
   event_base, int32_t event_id, void *event_data)
{
    ESP_LOGI(TAG, "PPP state changed event %" PRIu32, event_id);
    if (event_id == NETIF_PPP_ERRORUSER)
    {
        /* User interrupted event from esp-netif */
        esp_netif_t **p_netif = event_data;
        ESP_LOGI(TAG, "User interrupted event from netif:%p", *
           p_netif);
    }
}
```

Listing A.27: on_ip_event - prati događaje vezane za dodjeljivanje IP adrese i ispisuje informacije o tim događajima.

```
static void on_ip_event(void *arg, esp_event_base_t event_base,
   int32_t event_id, void *event_data)
{
    ESP_LOGD(TAG, "IP event! %" PRIu32, event_id);
    if (event_id == IP_EVENT_PPP_GOT_IP)
    {
        esp_netif_dns_info_t dns_info;

        ip_event_got_ip_t *event = (ip_event_got_ip_t *)
           event_data;
        esp_netif_t *netif = event->esp_netif;

        ESP_LOGI(TAG, "Modem Connect to PPP Server");
        ESP_LOGI(TAG, "~~~~~~~~~~~~~~~");
        ESP_LOGI(TAG, "IP          : " IPSTR, IP2STR(&event->
           ip_info.ip));
        ESP_LOGI(TAG, "Netmask     : " IPSTR, IP2STR(&event->
           ip_info.netmask));
        ESP_LOGI(TAG, "Gateway     : " IPSTR, IP2STR(&event->
           ip_info.gw));
        esp_netif_get_dns_info(netif, 0, &dns_info);
        ESP_LOGI(TAG, "Name Server1: " IPSTR, IP2STR(&dns_info.
           ip.u_addr.ip4));
        esp_netif_get_dns_info(netif, 1, &dns_info);
        ESP_LOGI(TAG, "Name Server2: " IPSTR, IP2STR(&dns_info.
           ip.u_addr.ip4));
        ESP_LOGI(TAG, "~~~~~~~~~~~~~~~");
        xEventGroupSetBits(event_group, CONNECT_BIT);

        ESP_LOGI(TAG, "GOT ip event!!!");
    }
    else if (event_id == IP_EVENT_PPP_LOST_IP)
    {
        ESP_LOGI(TAG, "Modem Disconnect from PPP Server");
```

```
    }
    else if (event_id == IP_EVENT_GOT_IP6)
    {
        ESP_LOGI(TAG, "GOT IPv6 event!");

        ip_event_got_ip6_t *event = (ip_event_got_ip6_t *)
           event_data;
        ESP_LOGI(TAG, "Got IPv6 address " IPV6STR, IPV62STR(
           event->ip6_info.ip));
    }
}
```

Listing A.28: app_main - pokreće glavnu logiku programa, uključujući inicijalizaciju modema, uspostavljanje PPP veze te spajanje na MQTT broker i objavljivanje podataka.

```
void app_main(void)
{
    ESP_LOGI(TAG, "[APP] Startup..");
    ESP_LOGI(TAG, "[APP] Free memory: %" PRIu32 " bytes",
       esp_get_free_heap_size());
    ESP_LOGI(TAG, "[APP] IDF version: %s", esp_get_idf_version()
       );

    esp_log_level_set("*", ESP_LOG_INFO);
    esp_log_level_set("mqtt_client", ESP_LOG_VERBOSE);
    esp_log_level_set("MQTT_EXAMPLE", ESP_LOG_VERBOSE);
    esp_log_level_set("TRANSPORT_BASE", ESP_LOG_VERBOSE);
    esp_log_level_set("esp-tls", ESP_LOG_VERBOSE);
    esp_log_level_set("TRANSPORT", ESP_LOG_VERBOSE);
    esp_log_level_set("outbox", ESP_LOG_VERBOSE);

    /* Init and register system/core components */
    ESP_ERROR_CHECK(nvs_flash_init());
    ESP_ERROR_CHECK(esp_netif_init());
    ESP_ERROR_CHECK(esp_event_loop_create_default());
```

```
ESP_ERROR_CHECK(esp_event_handler_register(IP_EVENT,
   ESP_EVENT_ANY_ID, &on_ip_event, NULL));
ESP_ERROR_CHECK(esp_event_handler_register(NETIF_PPP_STATUS,
    ESP_EVENT_ANY_ID, &on_ppp_changed, NULL));

gpio_config_t io_conf;
io_conf.intr_type = GPIO_INTR_DISABLE;
io_conf.mode = GPIO_MODE_OUTPUT;
io_conf.pin_bit_mask = (1 << MODEM_RST) | (1 << MODEM_PWKEY)
    | (1 << MODEM_POWER_ON);
io_conf.pull_down_en = 0;
io_conf.pull_up_en = 0;
gpio_config(&io_conf);

gpio_set_level(MODEM_PWKEY, 0);
gpio_set_level(MODEM_RST, 1);
gpio_set_level(MODEM_POWER_ON, 1);

vTaskDelay(15000 / portTICK_PERIOD_MS);

// Synchronize time with NTP server
// obtain_time();

/* This helper function configures Wi-Fi or Ethernet, as
   selected in menuconfig.
 * Read "Establishing Wi-Fi or Ethernet Connection" section
    in
 * examples/protocols/README.md for more information about
    this function.
 */
// ESP_ERROR_CHECK(example_connect());

/* Configure the PPP netif */
esp_modem_dce_config_t dce_config =
```

```
        ESP_MODEM_DCE_DEFAULT_CONFIG(CONFIG_EXAMPLE_MODEM_PPP_APN)
        ;
    esp_netif_config_t netif_ppp_config = ESP_NETIF_DEFAULT_PPP
        ();
    esp_netif_t *esp_netif = esp_netif_new(&netif_ppp_config);
    assert(esp_netif);

    event_group = xEventGroupCreate();

    /* Configure the DTE */
#if defined(CONFIG_EXAMPLE_SERIAL_CONFIG_UART)
    esp_modem_dte_config_t dte_config =
        ESP_MODEM_DTE_DEFAULT_CONFIG();
    /* setup UART specific configuration based on kconfig
        options */
    dte_config.uart_config.tx_io_num =
        CONFIG_EXAMPLE_MODEM_UART_TX_PIN;
    dte_config.uart_config.rx_io_num =
        CONFIG_EXAMPLE_MODEM_UART_RX_PIN;
    dte_config.uart_config.rts_io_num =
        CONFIG_EXAMPLE_MODEM_UART_RTS_PIN;
    dte_config.uart_config.cts_io_num =
        CONFIG_EXAMPLE_MODEM_UART_CTS_PIN;
    dte_config.uart_config.flow_control = EXAMPLE_FLOW_CONTROL;
    dte_config.uart_config.rx_buffer_size =
        CONFIG_EXAMPLE_MODEM_UART_RX_BUFFER_SIZE;
    dte_config.uart_config.tx_buffer_size =
        CONFIG_EXAMPLE_MODEM_UART_TX_BUFFER_SIZE;
    dte_config.uart_config.event_queue_size =
        CONFIG_EXAMPLE_MODEM_UART_EVENT_QUEUE_SIZE;
    dte_config.task_stack_size =
        CONFIG_EXAMPLE_MODEM_UART_EVENT_TASK_STACK_SIZE;
    dte_config.task_priority =
        CONFIG_EXAMPLE_MODEM_UART_EVENT_TASK_PRIORITY;
```

```
    dte_config.dte_buffer_size =
       CONFIG_EXAMPLE_MODEM_UART_RX_BUFFER_SIZE / 2;

#if CONFIG_EXAMPLE_MODEM_DEVICE_BG96 == 1
    ESP_LOGI(TAG, "Initializing esp_modem for the BG96 module
       ...");
    esp_modem_dce_t *dce = esp_modem_new_dev(ESP_MODEM_DCE_BG96,
        &dte_config, &dce_config, esp_netif);
#elif CONFIG_EXAMPLE_MODEM_DEVICE_SIM800 == 1
    ESP_LOGI(TAG, "Initializing esp_modem for the SIM800 module
       ...");
    esp_modem_dce_t *dce = esp_modem_new_dev(
       ESP_MODEM_DCE_SIM800, &dte_config, &dce_config, esp_netif)
       ;
#elif CONFIG_EXAMPLE_MODEM_DEVICE_SIM7000 == 1
    ESP_LOGI(TAG, "Initializing esp_modem for the SIM7000 module
       ...");
    esp_modem_dce_t *dce = esp_modem_new_dev(
       ESP_MODEM_DCE_SIM7000, &dte_config, &dce_config, esp_netif
       );
#elif CONFIG_EXAMPLE_MODEM_DEVICE_SIM7070 == 1
    ESP_LOGI(TAG, "Initializing esp_modem for the SIM7070 module
       ...");
    esp_modem_dce_t *dce = esp_modem_new_dev(
       ESP_MODEM_DCE_SIM7070, &dte_config, &dce_config, esp_netif
       );
#elif CONFIG_EXAMPLE_MODEM_DEVICE_SIM7600 == 1
    ESP_LOGI(TAG, "Initializing esp_modem for the SIM7600 module
       ...");
    esp_modem_dce_t *dce = esp_modem_new_dev(
       ESP_MODEM_DCE_SIM7600, &dte_config, &dce_config, esp_netif
       );
#elif CONFIG_EXAMPLE_MODEM_DEVICE_CUSTOM == 1
    ESP_LOGI(TAG, "Initializing esp_modem with custom module
```

```
        ...");
    esp_modem_dce_t *dce = esp_modem_new_dev(
        ESP_MODEM_DCE_CUSTOM, &dte_config, &dce_config, esp_netif)
        ;
#else
    ESP_LOGI(TAG, "Initializing esp_modem for a generic module
        ...");
    esp_modem_dce_t *dce = esp_modem_new(&dte_config, &
        dce_config, esp_netif);
#endif
    assert(dce);
    if (dte_config.uart_config.flow_control ==
        ESP_MODEM_FLOW_CONTROL_HW)
    {
        esp_err_t err = esp_modem_set_flow_control(dce, 2, 2);
            // 2/2 means HW Flow Control.
        if (err != ESP_OK)
        {
            ESP_LOGE(TAG, "Failed to set the set_flow_control
                mode");
            return;
        }
        ESP_LOGI(TAG, "HW set_flow_control OK");
    }

#else
#error Invalid serial connection to modem.
#endif

    xEventGroupClearBits(event_group, CONNECT_BIT | GOT_DATA_BIT
        );

    /* Run the modem demo app */
#if CONFIG_EXAMPLE_NEED_SIM_PIN == 1
```

```
    // check if PIN needed
    bool pin_ok = false;
    if (esp_modem_read_pin(dce, &pin_ok) == ESP_OK && pin_ok ==
       false)
    {
        if (esp_modem_set_pin(dce, CONFIG_EXAMPLE_SIM_PIN) ==
           ESP_OK)
        {
            vTaskDelay(pdMS_TO_TICKS(1000));
        }
        else
        {
            abort();
        }
    }
#endif

    int rssi, ber;
    esp_err_t err = esp_modem_get_signal_quality(dce, &rssi, &
       ber);
    if (err != ESP_OK)
    {
        ESP_LOGE(TAG, "esp_modem_get_signal_quality failed with
           %d %s", err, esp_err_to_name(err));
        return;
    }
    ESP_LOGI(TAG, "Signal quality: rssi=%d, ber=%d", rssi, ber);

#ifdef CONFIG_EXAMPLE_MODEM_DEVICE_CUSTOM
    {
        char time[64];
        err = esp_modem_get_time(dce, time);
        if (err != ESP_OK)
        {
```

```
            ESP_LOGE(TAG, "esp_modem_get_time failed with %d %s
               ", err, esp_err_to_name(err));
            return;
        }
        ESP_LOGI(TAG, "esp_modem_get_time: %s", time);
    }
#endif

    err = esp_modem_set_mode(dce, ESP_MODEM_MODE_DATA);
    if (err != ESP_OK)
    {
        ESP_LOGE(TAG, "esp_modem_set_mode(ESP_MODEM_MODE_DATA)
           failed with %d", err);
        return;
    }

    /* Wait for IP address */
    ESP_LOGI(TAG, "Waiting for IP address");
    xEventGroupWaitBits(event_group, CONNECT_BIT, pdFALSE,
       pdFALSE, portMAX_DELAY);

    /* Start MQTT client */
    ESP_LOGI(TAG, "Starting MQTT client");
    mqtt_app_start();

    // Create the MQTT publish task
    xTaskCreate(&mqtt_publish_task, "mqtt_publish_task", 4096,
       NULL, 5, NULL);

    /* Wait for establishing connection */
    ESP_LOGI(TAG, "Waiting for establishing connection");
    xEventGroupWaitBits(event_group, GOT_DATA_BIT, pdFALSE,
       pdFALSE, portMAX_DELAY);
}
```