Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Anže Bertoncelj

Prikaz temperatur s hibridno mobilno aplikacijo

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Rok Rupnik

Ljubljana, 2018

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Besedilo teme diplomskega dela študent prepiše iz študijskega informacijskega sistema, kamor ga je vnesel mentor. V nekaj stavkih bo opisal, kaj pričakuje od kandidatovega diplomskega dela. Kaj so cilji, kakšne metode uporabiti, morda bo zapisal tudi ključno literaturo.

Na prvem mestu se zahvaljujem mojemu mentorju prof. Rupniku, za pomoč pri izdelavi diplomskega dela. Zahvaljujem se tudi podjetju Iskra Instrumenti d.d., ki mi je dovolilo uporabo izdelka v diplomski nalogi. Ne smem pa tudi pozabiti na vse družinske člane se posebej pa na starše, ki so mi tekom diplomske naloge dajali moralno podporo. Brez njih ta diplomska naloga ne bi bila končana. Hvala.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
CA	classification accuracy	klasifikacijska točnost
API	application programming interface	programski vmesnik
IoT	internet of things	internet stvari
JSON	javascript object notation	javascript objektna notacija
HTML	hyper text markup language	jezik za označevanje nadbesedila
MQTT	message queuing telemetry transport	daljinski transport sporočil z uporabo vrste
SMQTT	secure message queuing telemetry transport	varen daljinski transport sporočil z uporabo vrste
GPS	global positioning system	sistem globalnega pozicioniranja
CSS	cascading style sheets	kaskadne stilske podloge
JS	javascript	javascript
HTTP	hypertext transfer protocol	protokol za izmenjavo hiperteksta in večpredstavnostnih vsebin na spletu
HTTPS	hypertext transfer protocol secure	varen protokol za izmenjavo hiperteksta in večpredstavnostnih vsebin na spletu
TLS	transport layer security	varnost transportnih plasti
TS	typescript	typescript
ID	indetification	identifikacija

Povzetek

Naslov: Prikaz temperatur s hibridno mobilno aplikacijo

Avtor: Anže Bertoncelj

Cilj diplomske naloge je bil razviti hibridno mobilno aplikacijo, ki uporabniku omogoča pregled podatkov izmerjenih iz temperaturnih senzorjev postavljenih po objektu. Poleg aplikacije se je razvil tudi zaledni sistem, ki preko API vmesnika shranjuje meritve temperaturnih senzorjev. Aplikacija je bila razvita z uporabo ogrodja Ionic, ki uporablja Angular in Apache Cordova za izgradnjo hibridne aplikacije. Uporabnik lahko do podatkov temperaturnih senzorjev dostopa na več različnih načinov, in sicer preko že obstoječe lokalne baze, neposredno z uporabo MQTT protokola in preko našega zalednega sistema. Zaledni sistem uporablja Docker-Compose za povezavo komponent zapakiranih znotraj Docker vsebnikov. Vsaka komponenta je realizirana s svojo tehnologijo glede na vlogo, ki jo opravlja. Node.Js je bi uporabljen za implementacijo API-ja, Eclipse Mosquitto kot MQTT posrednik, phpPgAdmin za hiter vpogled do baze in PostgreSQL za bazo. Diplomska naloga vpisuje tudi različne načine razvoja hibridnih aplikacij, navede njihove prednosti in slabosti ter jih med seboj primerja.

Ključne besede: mobilna aplikacija, hibridna, ionic, docker.

Abstract

Title: Displaying Temperatures With Hybrid Mobile Application

Author: Anže Bertoncelj

The aim of the thesis was to develop a hybrid mobile application, which allows the user to monitor temperatures from sensors inside a building. In addition to the application, a back-end was developed. It is used for storing measurements sent from sensors via the API interface. The mobile application was built using the Ionic framework, which uses Angular and Apache Cordova to build hybrid mobile applications. It allows a user to access data from several databases, namely from a preexisting local database, direct connection to sensors using MQTT and from our own back-end database. The back-end is implemented with Docker-Compose, which connects all of the components packed up in Docker containers. Each container is implemented with different technology, depending on its role. Node.Js is used for API, Eclipse Mosquitto as an MQTT broker, phpPgAdmin for quick database access and PostgreSQL for a database. The thesis also describes different approaches to creating hybrid applications, shows their advantages and compares them.

Keywords: mobile application, hybrid, ionic, docker.

Poglavje 1

Uvod

1.1 Motivacija

Motivacija za diplomsko delo je bila razviti mobilno aplikacijo in zaledni sistem za podjetje Iskra Instrumenti d.d. Podjetje je za svoje potrebe razvilo temperaturne senzorje, ki so jih postaviti po objektu podjetja. Senzorji so izmerjene podatke nalagali na notranje podatkovne baze, vidne samo preko računalniške aplikacije. Podjetje je imelo željo razviti mobilno aplikacijo, ki bi podatke temperaturnih senzorjev pokazala tudi na mobilni napravi. Ker trenutna baza za shranjevanje podatkov ni bila primerna za shranjevanje podatkov temperaturnih senzorjev, je bilo zaželeno razviti tudi sistem shranjevanja izmerjenih vrednosti v bazo.

1.2 Predstavitev problema

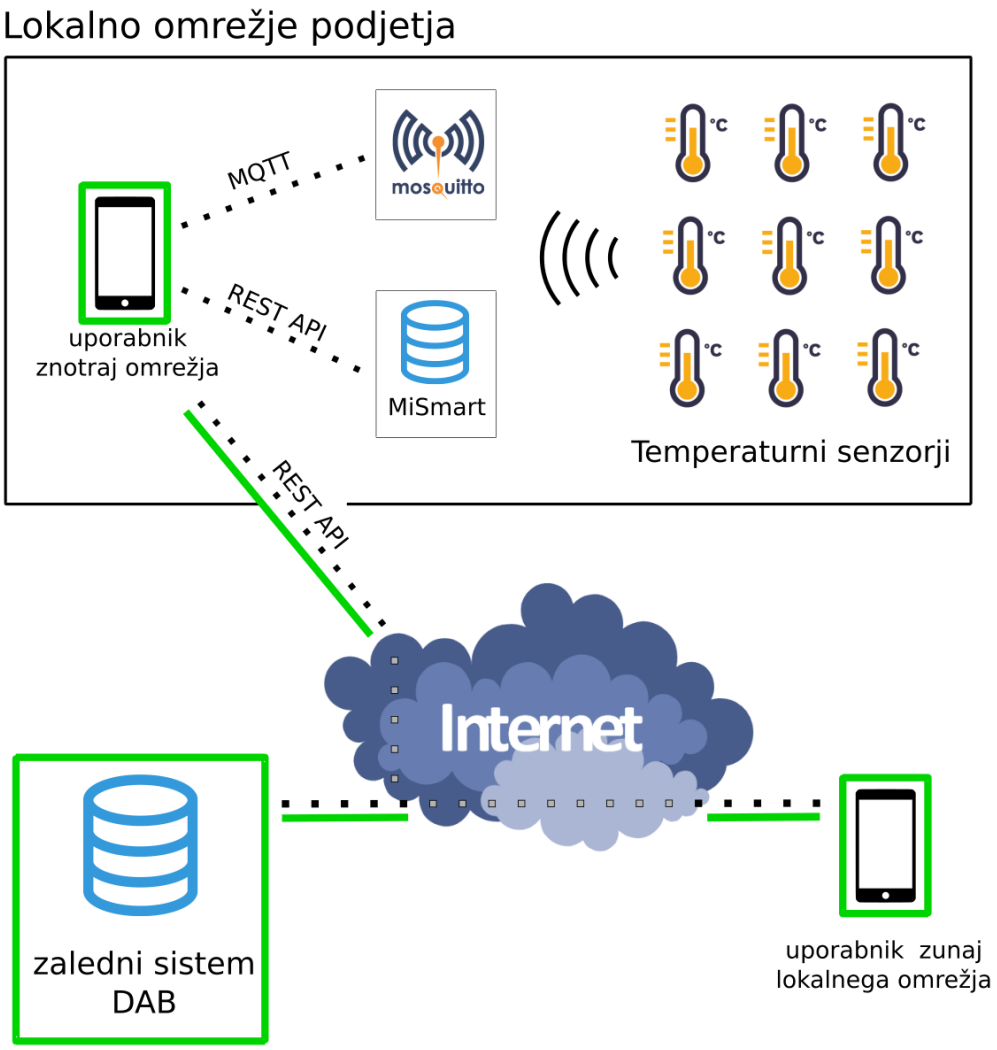
V podjetju imajo temperaturne senzorje, ki oddajajo svoje podatke preko MQTT na lokalno bazo poimenovano MiSmart. Iz lokalne baze MiSmart je možno preko klicev API-ja dostopati do podatkov.

Razviti je bilo potrebno mobilno aplikacijo, ki bo lahko preko protokola MQTT, lokalne baze MiSmart in preko svojega zalednega sistema DAB brala in prikazovala podatke in tako uporabniku dala vpogled nad temperaturnimi

v prostoru. Da bi lahko aplikacijo uporabljalo več zaposlenih v podjetju, še posebej pa vodstvo, je bila želja razviti hibridno aplikacijo in tako zagotoviti izvajanje na iOS platformah, kot tudi na platformah Android.

Poleg aplikacije je bilo potrebno razviti tudi nov zaledni sistem, na katerega bodo temperaturni senzorji preko protokola MQTT shranjevali podatke. Preko vmesnika APi, pa naj bi bili ti podatki dostopni mobilni aplikaciji.

Celotna arhitektura rešitve je prikazana na sliki 1.1. Z zeleno črto so označeni deli, ki sem jih razvil in predstavljeni v diplomski nalogi.



Slika 1.1: Arhitektura aplikacije. Z zeleno črto so označeni deli, ki so bili razviti v sklopu diplome. Uporabnik zunaj lokalnega omrežja lahko dostopa samo do zalednega sistema DAB, Uporabnik znotraj omrežja ima na voljo še bazo MiSmart in protokol MQTT.

1.3 Hibridna aplikacija

Za razvoj mobilne aplikacije imamo tri glavne možnosti. Domorodne aplikacije (ang. native applications), spletne aplikacije in hibridne aplikacije. Te se potem delijo na podskupine glede na to, katere tehnologije so uporabljene. Vsaka od možnosti prinaša določene prednosti in slabosti, zato je pred razvojem vredno pogledati vse.

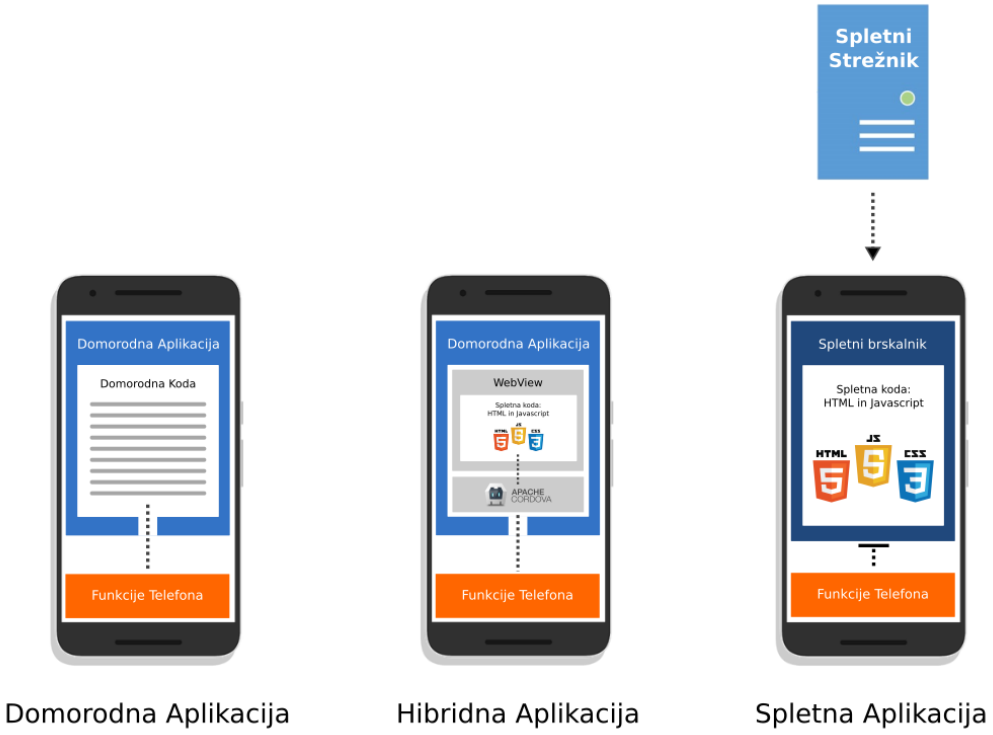
1.3.1 Domorodna aplikacija

Domorodna aplikacija (ang. native app) pomeni, da uporabimo programski jezik, ki se bo prevedel in izvajal neposredno na telefonu. Aplikacijo je potrebno razviti v jeziku, ki ga podpira platforma za katero razvijamo. Za telefone Android je to Java, za iOS Swift in Windows C#. S tem, da se aplikacija izvaja kar se da neposredno na telefonu, poskrbimo, da je aplikacija odzivna in hitra. Poleg hitrosti imamo preko vmesnega uporabniškega programa (ang. application programming interface ali API) na voljo tudi vse senzorje in funkcije telefona, kot so GPS in kamera. Največ aplikacij je domorodnih in imajo zato tudi največjo podpore, tako s strani skupnosti, kot tudi s strani platforme.

Na drugi strani pa izgubimo fleksibilnost. Za vsako platformo posebej moramo razviti popolnoma novo mobilno aplikacijo. V primeru, da aplikacijo želimo razviti za dveh najpopularnejši platformah Android in iOS, nam to praktično podvoji stroške razvoja.

1.3.2 Spletna aplikacija

Spletna aplikacija je običajna spletna stran, ki je prilagojena za izvajanje na mobilnih telefonih. To pomeni, da je oblika aplikacije narejena za manjše zaslone, uporablja primerno velike gradnike za zaslone na dotik in oblike gradnikov, ki oponašajo gradnike domorodnih aplikacij. Prednosti spletnih aplikacije so naslednje:



Slika 1.2: Primerjava med različnimi tipi mobilnih aplikacij.

- Relativno enostavne za razvoj.
- Izvajanje iste aplikacije na različnih platformah.
- Uporabnik lahko zažene aplikacijo, ne da bi jo moral prej naložiti na telefon.
- Aplikacija je zato tudi bolj dostopna.
- Uporabnik z vsakim dostopom do spletne strani dobi najnovejšo verzijo.

Razvijanje spletne aplikacije za seboj prinese tudi določene slabosti:

- Uporabnik potrebuje internetno povezavo za dostop do aplikacije.
- Zaganjanje aplikacije preko interneta je počasno.
- Aplikacija nima dostopa do senzorjev telefona in ostalih funkcij.
- Običajno počasnejša in manj odzivna od domorodnih aplikacij.
- Kljub temu, da naj bi vsi brskalniki prikazovali vsebino na enak način, temu na žalost ni tako. Še vedno se moramo prilagajati na vsak brskalnik posebej.
- Vzdrževati je potrebno strežnik, ki streže aplikacijo.

1.3.3 Hibridna aplikacija

Hibridna aplikacija je aplikacija napisana v kodi, ki ni domorodna platformi na kateri se izvaja. Glavni namen hibridne aplikacije je napisati kodo enkrat in to kodo potem poganjati na različnih platformah. Izvajanje ene kode na več platformah lahko dosežemo na različne načine, odvisno od izbire orodja.

Hibridna aplikacija strmi, da bi bila čim bolj podobna domorodni aplikaciji. Običajen uporabnik ne bi smel razločiti med hibridno in domorodno aplikacijo.

Vse hibridne aplikacije se na koncu vedno zavijejo v ovojnico domorodne aplikacije.

Glede na način izvajanja kode in prikazovanja vmesnika poznamo več tipov hibridnih aplikacij. V času razcveta hibridnih aplikacij okoli leta 2015 je bilo teh načinov veliko več, saj se je vsako večje podjetje poizkušalo izboriti mesto med hibridnimi aplikacijami. Večino orodji se je umaknilo v pozabo. Eno izmed takšnih orodji je bil Intel XDK.

Obdržala sta se dva glavna načina. Xamarin in React Native ter orodja, ki uporabljajo WebView, kot so Ionic, NativeScript in Framework7.

1.3.4 Hibridne aplikacije z uporabo WebViewa

WebView je orodje za prikazovanje spletnih strani na mobilnih napravah. Uporablja ga večino mobilnih brskalnikov. Aplikacije so napisane enako kot spletne strani z uporabo tehnologij HTML, CSS in JS. Da pa aplikacije ne izgledajo kot običajne spletne strani, uporabljajo ogrodja kot so Ionic in Framework7. Ta orodja poskrbijo za izgled gradnikov, ki imitirajo gradnike domorodnih aplikacij.

Poleg WebViewa se uporablja tudi PhoneGap, ki omogoča aplikacijam uporabo senzorjev telefona.

1.3.5 Hibridne aplikacije z uporabo ogrodja Xamarin

Xamarin je podjetje v lasti Microsofta in omogoča razvoj mobilnih aplikacij z uporabo jezika C#. Prednost razvoja je, da Xamarin namesto WebViewa uporablja domorodne elemente za prikazovanje vsebine in je zato občutno hitrejši [5]. Ker je koda napisana v jeziku C#, se zato brez problemov izvaja na Windows telefonih. Na telefonih iOS se C# koda najprej prevede v domorodno kodo, ki se lahko neposredno izvaja na ARM procesorjih. Na telefonih Android se ta prevede v posredni jezik in se izvaja v sprotnem prevajalniku (ang. Just-in-time compiler ali JIT compiler) [13].

Koda z uporabo Xamarin se izvaja hitreje, kot koda z uporabo WebViewa

in se lahko že primerja z hitrostmi domorodnih aplikacij [5].

1.3.6 Hibridne aplikacije z uporabo orodja React Native

Koda aplikacije je napisana z uporabo ogrodja React, ki je po delovanju podoben Angularju, predvsem v smislu, da oba uporabljata JavaScript in Model-Pogled-Krmilnik arhitekturo. Na prvi pogled je zelo podoben orodju Ionic, saj oba uporabljata kodo JavaScript, a se razlikujeta v ključni razliki. Namesto, da se koda izvaja v WebViewu, se podobno, kot z ogrodjem Xamarin, prevede v izvorno kodo in je tako končna aplikacija prava domorodna aplikacija in ne samo ovojnica za WebView.

Koda se piše na isti način kot React koda za spletne strani. Glavna razlika je pri izrisovanju elementov. Komponente se iz HTMl spremenijo v domorodne komponente. Načeloma bi lahko prevedli aplikacijo v Android in potem nadaljevali razvoj kot običajno domorodno aplikacijo.

1.3.7 Utemeljitev implementacije hibridne aplikacije

Naša ciljna aplikacija naj bi imela možnost izvajanja tako na telefonih Android, kot tudi na telefonih iOS. Če bi želeli razviti domorodno aplikacijo, bi morali razviti za vsako platformo svojo aplikacijo. Poiskati bi morali razvijalca, ki zna razvijati za obe platformi, ali pa plačevati dva razvijalca, vsakega za svojo platformo. Ta opcija zaradi povečanja stroškov ni bila primerna.

Druga možnost je bila razviti spletno aplikacijo. To lahko razvije en sam programer in bi se lahko izvajala na vseh platformah. Problem je nastal pri strežniku. Mobilna spletna aplikacija potrebuje strežnik, ki bo stregel aplikacijo uporabnikom. Že obstoječega strežnika ni bilo, zato bi poleg aplikacije morali postaviti in skrbeti še za nov strežnik.

Naslednja možnost je bila razviti hibridno aplikacijo. Odločiti smo se morali med uporabo tehnologij, ki uporabljajo WebView ali pa razvijati z orodjem Xamarin. Ker sam nimam veliko izkušen z razvojem v jeziku C#

in nisem velik podpornik stvari, ki jih razvije podjetje Microsoft, se za to možnost nismo odločili.

Na koncu smo morali izbrati eno izmed orodji, ki uporablja WebView. Z uporaba orodja Google Trends, s katerim lahko spremljamo popularnost iskalnega izida skozi čas, smo opazili, da je izmed vseh najbolj popularno ogrodje Ionic. To je bilo na koncu orodje v kateremu smo se odločili razviti aplikacijo.

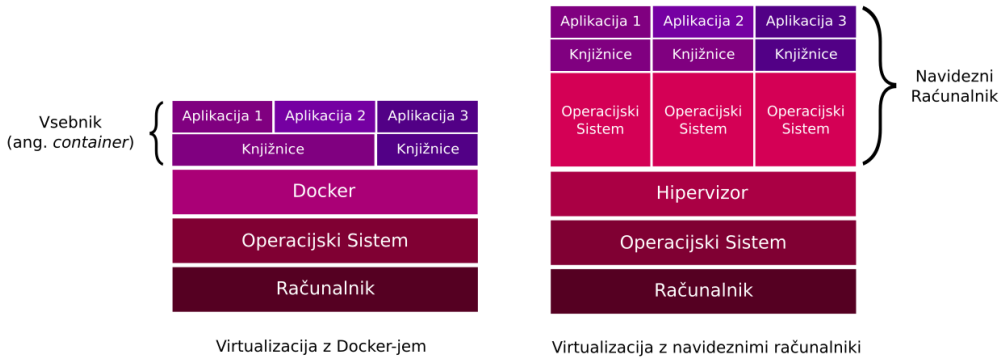
Poglavje 2

Uporabljene tehnologije

2.1 Docker

Docker je programsko orodje, ki programom zagotavlja virtualizacijo na nivoju operacijskega sistema. Takšna virtualizacija je pod drugim imenom znana pod angleškim imenom "*containerization*". Virtualizacija na nivoju operacijskega sistema je lastnost operacijskega sistema, ki dovoljuje izvajanje več izoliranim uporabniškim primerkom hkrati. Ti primerki razvijalcem zagotavljajo, da lahko zapakirajo aplikacijo z vsemi komponenti na katere se aplikacija navezuje, kot so knjižnice in ostale odvisnosti, in jih prikažejo kot eno rešitev.

Na nek način je Docker zelo podoben navideznemu računalniku (ang. Virtual Machine), vendar imata ključno razliko. Virtualni računalniki vsebujejo celotne operacijske sisteme, ki morajo ob vsakem zagonu poleg programa zagnati tudi operacijski sistem. Docker uporabi že obstoječi operacijski sistem in na njem zažene aplikacije. S tem je zagon veliko hitrejši in instance manjše [14].



Slika 2.1: Primerjava sestave Dockerja z navideznimi računalniki.

2.2 Ionic

Ionic je odprto-kodno programsko orodje za izdelavo hibridnih mobilnih aplikacij. Z orodjem Ionic lahko zgradimo aplikacije, ki se bodo lahko izvajale na telefonih Android, iOS in Windows [12].

Ionic razvijalcem nudi orodja in storitve za razvoj hibridnih mobilnih aplikacij z uporabo spletnih tehnologih, CSS, TS in HTML5. Z vtičnikom Apache Cordova Ionic programerjem dovoljuje dostop do funkcij telefona [3].

Ionic razvijalcem daje na voljo komponente, ki so posebej prilagojene vsaki ciljni platformi. Poleg komponent so na voljo tudi tipi pisave in animacije [4].

Ob samem razvoju si lahko programerji pomagajo z serviranjem aplikacije na brskalniku in tako privarčujejo čas prevajanja in nalaganja aplikacije, ki je potrebna, če želimo aplikacijo zaganjati na telefonu.

2.2.1 Apache Cordova

Apache Cordova (nekoč poznana pod imenom PhoneGap [1]) je ogrodje za izdelavo mobilnih spletnih aplikacij. Sama mobilna spletna aplikacija nima dostopa do vseh funkcionalnosti telefona, kot so dostop do kamere, datotek, obvestil in senzorjev (GPS, senzor za merjene pospeškov ipd.). Apache Cor-

dova omogoči razvijalcem možnost dostopa, do teh funkcij telefona in s tem pripelje hibridne aplikacije korak bližje domorodnim aplikacijam.

2.2.2 Angular

Angular je računalniško okolje za razvijanje čelnega dela (ang. front-end) spletnih aplikacij. Ena izmed ključnih prednosti, pred tradicionalnem interaktivnim spletnim stranem, ki uporabljajo HTML in JS, je povezava med pogledom in kodo. Sprememba pogleda se izraža, tudi kot sprememba v kodi. S tem Angular povezuje, prej ločena elementa spletnega pogleda in spletne kode, v eno celoto.

Namen Angularja je razvijanje spletnih aplikacij na eni strani (ang. single page aplication ali SPA) in je zato zelo uporaben kot okolje za razvoj hibridnih aplikacij, ki so na nek način prav tako spletne strani na eni strani.

2.3 MQTT

MQTT je internetni protokol, ki deluje na aplikacijski ravni z uporabo protokola TCP/IP. Sporočila izmenjuje na principu izdajatelja in naročnika (ang. publish-subscribe), kjer so izdajatelji naprave, ki ustvarjajo podatke in naročniki tisti, ki sprejemajo podatke. To pomeni, da lahko izdajatelj en podatek pošlje več naročnikom hkrati. MQTT je narejen za uporabo v okoljih, kjer je količina prenesenih podatkov v omrežju omejena, bodisi zaradi manj zmogljivih naprav ali manj zanesljivega omrežja [11].

Čeprav je bil protokol razvit že v letih 1999 je svojo popularnost dosegel šele v zadnji nekaj letih s prihodom interneta stvari (ang. internet of things ali IoT). Prav zaradi svoje majhne porabe internetnega pasu je idealen za uporabo v enostavnih napravah kot so naprave interneta stvari. Prav tako je zaradi tega razloga uporabljen kot en način komunikacije s temperaturnimi senzorji.

2.3.1 Eclipse Mosquitto

Naprave si sporočil ne morejo pošiljati neposredno drug drugemu. Vsa sporočila se najprej pošljejo posredniku (ang. broker), in ta potem posreduje sporočila vsem poslušalcem. Mosquitto je primer implementacije takšnega posrednika.

2.4 Postgres

Postgres, znan tudi pod imenom PostgreSQL, je objektno relacijska baza, ki ima več kot 20 [7] let aktivnega razvoja in zanesljive arhitekture, s katero si je prislužila ugled kot zanesljiva baza. Kljub temu, da ima PostgresSQL veliko funkcionalnosti, se v tem projektu uporablja zgolj kot običajna relacijska baza.

PostgreSQL je bil zasnovan za delovanje na Unix operacijskih sistemih, vendar mu njegova zasnova omogoča prenosljivost, tako da lahko deluje tudi na operacijskih sistemih macOS in Windows [8].

2.4.1 PhpPgAdmin

Za dostop do baze ponavadi potrebujemo nekega odjemalca (ang. client). Poleg primarnega odjemalca psql, ki temelji na vmesniku z ukazno vrstico, imamo množico naborov drugih odjemalcev, ki so uporabniku prijaznejši in ponujajo grafični vmesnik. Eden izmed njih je phpPgAdmin, ki je spletna različica računalniškega programa PgAdmin [10].

Največja prednost pred računalniško aplikacijo PgAdmin je njegova prenosljivost in enostavnost dostopa. Da lahko dostopamo do baze, nam ni potrebno nalagati nobenih novih programov. Imeti moramo samo internetno povezavo. S tem nam zelo olajša delo na terenu, ko nimamo dostopa do svoje programske opreme. Prednost se pokaže tudi v varnosti, saj nam ni potrebno imeti izpostavljenih vrat (ang. port) za neposredno povezavo do baze. Dovolj je že, da imamo dostop do spletne strani phpPgAdmin, in preko

nje nato upravljamo z bazo.

2.5 Git

Git je odprto-kodni sistem za obvladovanje verzij. Razvil ga je Linus Torvalds leta 2005 [9] za potrebe verzioniranja razvoja Linuxa. Narejen je, da pomaga programerjem pri delu na skupnem projektu, tako da beleži in arhivira zgodovino projekta. Hkrati omogoča sočasno delo na datotekah z načini reševanja konfliktov. Glavna značilnost Git sistema je, da je porazdeljen sistem za obvladovanje verzij (ang. distributed version control system ali DVCS). To pomeni, da ima uporabnik poleg centralnega strežnika svoj lokalni repozitorij, z vsemi podatki potrebnimi za delovanje brez povezave do centralnega repozitorija. V nasprotju z centraliziranim sistemom (ang. version control system ali VCS), kjer uporabnik brez povezava na strežnik ne more uveljaviti sprememb.

Git sistem podpira tudi veje razvoja. Vsaka veja predstavlja neodvisen razvoj. Vejitev pomeni odcepitev od glavne veje razvoja in nadaljevanje dela, ne da bi s tem vplivali na glavno vejo. Vejitve večjim ekipam omogočajo razvijalcem nemoteno dela na svoji veji. Uporabljajo se ga tudi lahko za sprotne popravke (ang. hotfix) in implementacije nove funkcionalnosti programa.

Git je bil pri tem projektu uporabljen zgolj kot način organiziranja in spremljanja poteka dela. Vsi deli projekta, kot tudi sama diplomska, so bili shranjeni v svoj git repozitorij.

2.6 Node.js

Node.js je odprto-kodni več platformni program, ki omogoča izvajanje JavaScript zunaj brskalnika. JavaScript se je razvil predvsem za potrebe spletnih strani, kjer se je uporabljal za izvajanje kode na strani odjemalca. Node.js omogoča razvijalcem izvajanje kode na strežniku in s tem izdelavo dinamičnih spletnih strani [6].

Prednost Node.js je podpora asinhronih vhodov in izhodov. Asinhrone operacije omogočajo ob klicu funkcije nadaljevanje izvajanje glavnega programa, medtem ko se koda funkcije izvaja vzporedno. Asinhrono izvajanje Node.js doseže z eno-nitnem delovanjem. V nasprotju z večnitnim delovanjem, kjer se vsakemu uporabniku dodeli svoja nit in sistemski viri, pri eno nitnem delovanju vsi uporabniki uporabljajo eno nit in enotne sistemske vire. Z deljenjem virov lahko privarčujemo na virih, povečamo hitrost spletne strani in povečamo število hkratnih povezav.

2.6.1 Express

Express je odprto-kodno spletno ogrodje za razvoj spletnih aplikacij v okolju Node.js. Programerjem omogoča enostavno implementacijo spletnega strežnika z uporabo že vnaprej napisanih funkcij s katerimi lahko poslušamo in se odzivamo na zahteve odjemalca. Poleg tega skrbi za uporabniške seje z uporabo obvladovanja uporabnikov, in s tem olajša implantacijo sej v HTTP protokolu, ki sam po sebi ne podpira stanj. Enostavno lahko tudi uredimo strežbo statičnih HTML strani z uporabo usmerjevalnika vhodnih zahtev.

2.6.2 Nodemon

Ko pišemo kodo v ogrodju Node.js in želimo videti spremembe na strežniku, moramo ob spremembi kode ponovno zagnati strežnik. Nodemon to naredi avtomatično namesto nas. Ob zagonu nenehno spremlja kodo in ob zaznavi spremembe ponovno zažene strežnik. Nodemon je še posebej priročen v okolju, kjer ponovni zagon strežnika vzame veliko časa.

Poglavje 3

Temperaturni senzorji

Temperaturne senzorje je razvilo podjetje za namene merjenja in nadziranja temperature v poslovnih stavbah. Razvoj senzorjev ni bil del moje diplomske naloge, vendar je eden od ključnih elementov celotnega sistema in ga bom zato na kratko predstavil. Sliko temperaturnega senzorja lahko vidimo na sliki3.1

Slika 3.1: Temperaturni senzor.

3.1 Predstavitev

Temperaturni senzorji so bili razviti z namenom učinkovitega upravljanja temperature v poslovnih prostorih s spremljanjem okoljskih pogojev v realnem času.

Napajajo se lahko preko baterije ali preko standardnega električnega omrežja s priklopom na vtičnico (230V). Baterijsko napajanje dopušča namestitev neodvisno od postavitve vtičnic in tudi kasnejše prestavljanje po prostoru. Vgrajena baterija omogoča do 12 mesecev samostojnega delovanja, ki pa se lahko še poveča v kolikor nastavimo manj pogosto pošiljanje podatkov.

Baterijski senzorji v primerjavi z napajanimi senzorji hodijo spat in zato izgubijo zgodovino meritev. Napajani senzorji lahko hranijo podatke za približno 2 dni.

Vsak senzor lahko meri eno ali več veličin, odvisno od tipa vsebovanega merilnega senzorja. Vsi merijo temperaturo, poleg tega pa lahko nekateri senzorji merijo tudi zračni pritisk in vlago. Skupaj z meritvami se pošiljata tudi moč signala in stanje baterije. V kolikor senzor ni baterijsko napajan se pošilja samo moč signala. Moč signala nam pomaga, da senzor postavimo na lokacijo, kjer bo v dosegu Wi-Fi omrežja in bo lahko brez težav pošiljal podatke.

3.2 Opis načina komunikacije

Temperaturni senzorji uporabljajo Wi-Fi omrežje za povezavo s sistemom zbiranja in prikaza podatkov. Poljubno so lahko nastavljeni, da preko omrežja pošiljajo podatke preko HTTP protokola ali pa preko MQTT protokola. Preko HTTP protokola pošiljajo podatke v obliki JSON formata. Preko MQTT pa lahko pošiljajo na dva načina. Prvi je, da vse podatke pošiljajo na eni temi v obliki JSON, drugi način pa je pošiljanje vsakega podatka posebej na svojo temo. Mobilna aplikacija podpira branje podatkov na vse tri načine.

Poglavje 4

Zaledni sistem DAB

4.1 Opis

Glavna naloga zalednega sistema je zbiranje podatkov, ki jih pošiljajo temperaturni senzorji in posredovanje podatkov mobilni aplikaciji, ki te podatke prikazuje. Zalednemu sistemu se je dodelila kratica DAB, kot okrajšava za angleško besedo *database*.

Pred postavitvijo lastnega zalednega sisteme je imela aplikacija že dva načina pridobivanja podatkov. Ena način je bil z uporabo že obstoječega baze MiSmart, kamor so temperaturni senzorji pošiljali podatke. Drug način pa preko MQTT protokola. Oba načina sta delovala brez problemov, a sta imela ključno omejitev. Uporabnik ni mogel dostopati do podatkov izven lokalnega omrežja. Ta zaledni sistem naj bi rešil ta problem, saj bi bil lociran zunaj lokalnega omrežja in tako omogočal uporabniku dostop do podatkov kjerkoli.

Celoten zaledni sistem je bil zavit v vsebnike (ang. containers) in implementiran v Dockerju. Implementacija v Dockerju se je izbrala zaradi naslednjih dveh razlogov:

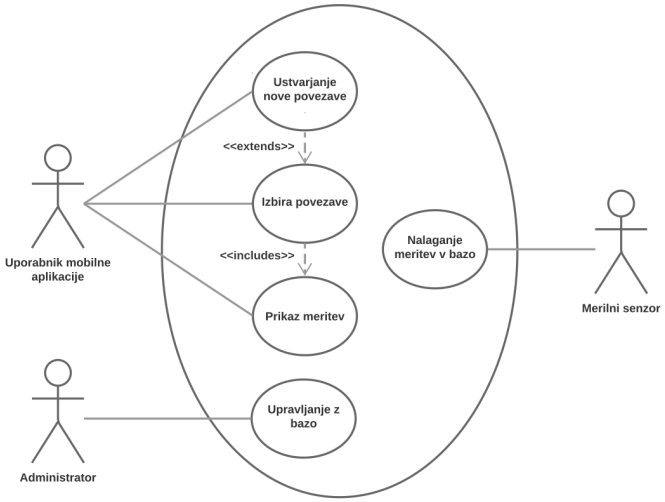
- Končna lokacija zalednega sistema ob začetku razvoja ni bila znana. Ustavitev in na novo postavitev sistema je v Dockerju enostavnejša, kot namestitev celotnega sistema od začetka. S tem je enostavnejša

tudi prenosljivost sistema.

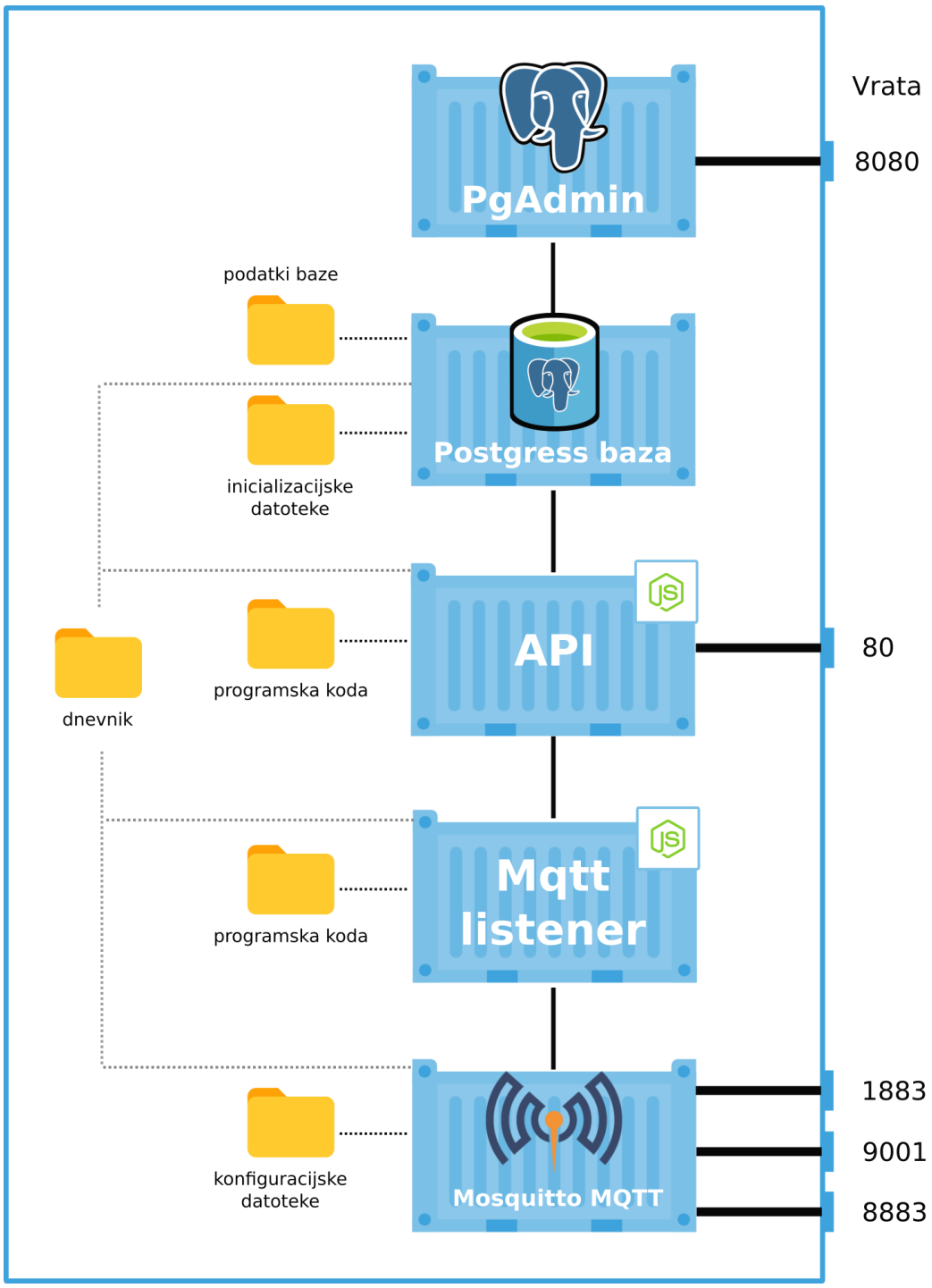
- Končna rešitev je sestavljena iz več manjših komponent. Orodje Docker-Compose nam omogoča enostaven zagon in povezljivost komponent med seboj.

4.2 Arhitektura

Zaledni sistem je implementiran v Dockerju in je zato sestavljen iz več komponent, ki so skupaj povezane z orodjem Docker-Compose. Vsaka komponenta se izvaja v svojem vsebniku (ang. container). Zaledni sistem je v celoti sestavljen iz 4 delov. Baze Postgres in API-ja, ki je povezan na bazo in preko katerega dostopamo do podatke baze. Potem imamo Mosquitto MQTT strežnik, na katerega naprave pošiljajo svoje meritve in *Mqtt listener*, ki posluša na Mosquitto MQTT strežniku in te podatke pošilja preko API-ja v bazo. Na koncu imamo še *PgAdmin*, ki je bila dodana zaradi lažjega dostopanja in opravljanja z bazo.



Slika 4.1: Diagram primerov uporabe.

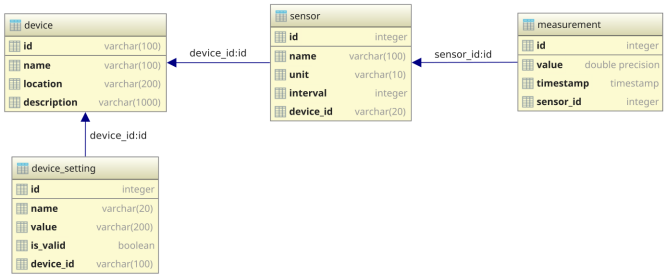


Slika 4.2: Sestava zalednega sistema DAB.

4.2.1 Baza

Postgres je glavna baza v katero se shranjujejo vsi podatki in meritve temperaturnih senzorjev. Do nje se lahko dostopa preko API-ja ali pa preko spletnega vmesnika *PgAdmin*. Neposrednega dostopa do baze ni.

Baza je sestavljena iz štirih entitet.



Slika 4.3: Entitetno-relacijski diagram.

Entiteta Device

Vsak temperaturni senzor je v smislu baze poimenovan kot naprava *"Device"*. Sem se shranjujejo osnovni podatki merilnega senzorja, ki ji potrebujemo, da senzor prepoznamo. To so ime senzorja, zapisan v atributu **"name"**, lokacija v atributu *"location"* in opis v atributu *"description"*. Vsi ostali podatki in nastavitve senzorja so shranjeni v relaciji *"device_settings"*.

Entiteta Device_Setting

Tu so shranjene nastavitve naprav. Vsaka naprava ima lahko poljubno veliko nastavitev in zato shranjevanje vseh nastavitev v entiteto ni bila smiselna. Entiteta je z entiteto *"Device"* povezana z odnosom ena proti mnogo (ang. one-to-many relationship). To pomeni da ima lahko ena naprava več nastavitev. Ampak vsaka nastavitev pripada samo eni napravi. V entiteti so zapisani podatki:

- **id**:. To je zaporedna identifikacijska številka, ki jo potrebujemo zgolj zato, da lahko vsako nastavitev med seboj ločimo.
- **name**: ime nastavitve
- **value**: vrednost nastavitve. Tip tega stolpca je *varchar*, saj lahko tako v njega shranimo tako številke, kot tudi nize.
- **date_set**: Datum kdaj je bila nastavitev shranjena v bazo. Ta parameter se potrebuje v primeru, če želimo vrniti samo najnovejše nastavitve.
- **is_valid**: Pove, ali je nastavitev še vedno aktualna. Med starimi in novimi nastavitvami lahko ločimo po podatku date, ampak v primeru, da želimo neko nastavitev odstraniti, lahko označimo v stolpcu *"is_valid"*, da nastavitev ni več aktualna. V nasprotnem primeru bi morali nastavitev odstraniti iz baze, kar bi pomenilo izgubo zgodovine nastavitev naprave.
- **device_id**: Kateri napravi pripada nastavitev.

Entiteta senzor

Entiteta *senzor* predstavlja vse senzorje, ki jih vsebuje neka naprava. To je potrebno ločiti, kaj senzor pomeni v kontekstu temperaturnih senzorjev. Temperaturni senzorji lahko merijo več parametrov, kot so: temperatura, vlaga, pritisk ... Vsak od teh tipov meritev je v bazi poimenovan kot senzor naprave. Kljub temu, da vse meritve, ki jih oddaja naprava, niso fizični senzorji znotraj naprave, so v tabeli vsi zapisani v tabelo senzor.

Entiteta *senzor* je povezana z entiteto *device* v odnosu ena proti mnogo. Vsaka naprava ima lahko več senzorjev, a vsak senzor ima lahko samo eno napravo.

Entiteta *senzor* vsebuje naslednje podatke:

- **id**: Zaporedna identifikacija številka, ki je namenjena zgolj za ločitev posameznega senzorja

- **unit**: Enoto s katero senzor meri. V primeru temperature je to lahko "°C"(stopinj Celzija) ali pa "°F"(stopinj Fahrenheit).
- **interval**: Kako pogosto pošilja meritve.
- **device_id**: Kateri napravi pripada senzor.

Entiteta meritve

Sem se shranjujejo meritve senzorjev.

- **id**: Zaporedna identifikacija številka namenjena zgolj za ločitev posamezne meritve.
- **value**: Vrednost meritve.
- **timestamp**: Čas zabeležene meritve. To je čas, ko je temperaturni senzor zabeležil meritev in ne, ko je bila meritev zapisana v bazo.
- **sentor_id**: Številka senzorja, ki je izmeril meritev.

Datoteke v Docker vsebnikih niso obstoječe, kar pomeni, da ob izbrisu slike vsebnika izgubimo vse podatke znotraj slike. To za potrebe baze ni ugodno, saj nam je v interesu, da se podatki ohranijo tudi ob izbrisu slike vsebnika. Zato ima Docker na voljo funkcionalnost nosilcev (ang. volumes), ki dovoljujejo shranjevanje podatkov zunaj slike vsebnika. To naredijo tako, da preslikajo vsebino datoteke znotraj vsebnika, v datoteko zunaj. Postgres vsebnik shranjuje svoje podatke v imenik znotraj vsebnika v */var/lib/postgresql/data*. Vse kar je bilo potrebno storiti je preslikati imenik v zunanji imenik. V našem primeru, je v datoteko docker-compose, bilo potrebno dodati vrstico - *./db/pgdata:/var/lib/postgresql/data*.

Poleg tega imenika smo preslikali tudi imenik z inicializacijsko datoteko. Ta se zažene samo v primeru, da je baza prazna in poskrbi za to, da se ustvarijo vse relacije in povezave med njimi in s tem pripravi bazo za vnos podatkov.

4.2.2 API

API je glavni del zalednega dela, saj je to edini del, ki je izpostavljen uporabnikom in omogoča tako nalaganje, kot tudi pridobivanje podatkov.

API je implementiran z orodjem Node.Js. Ta skupaj z ogrodjem Express omogoča hitro implementacijo API-jev.

Sama koda je razdeljena na dva dela. Prvi skrbi za povezavo z bazo, izvršuje poizvedbe in komunicira z bazo. Komunikacija je narejena s pomočjo knjižnica *ng-promise*. Ena od funkcionalnosti, ki jih podpira knjižnica je razširitev glavnega objekta z dodajanjem lastnih poljubnih repozitorijev. To naredimo v inicializacijski datoteki, kjer naštejemo katere objekte bi radi dodali. Te objekte lahko potem enostavno prenašamo in kličemo iz glavnega objekta.

Vse dostope do baze smo zapakirali v vmesnik objektov za podatkovni dostop (ang. data access object ali DAO), ki nam zakrije implementacijo vseh poizvedb do baze. S tem dobimo preglednejšo in lažje razumljivo kodo.

Drugi del so usmerjevalniki (ang. routers), ki sprejemajo uporabniške zahteve in z uporabo objektov za podatkovni dostop vračajo in nalagajo podatke v bazo.

Uporabnik lahko s spreminjanjem poti internetnega naslova povprašuje po različnih podatkih. Glavne poti do katerih lahko uporabnik dostopa so:

- **/devices** Podatki povezani z napravami.
- **/senzors** Podatki povezani z senzorji.
- **/senzors/{senzorId}/measuremens** Podatki povezani z meritvami.

Z dodajanjem identifikacijskih številk in z uporabo različnih HTTP klicev lahko uporabnik izvršuje specifične akcije nad podatki v bazi, kot so posodabljanje, pridobivanje in brisanje podatkov. Opis glavnih dostopov in akcij so prikazani v tabeli 1

API poskrbi, da lahko uporabnik na čim bolj enostaven in jasen način dostopa do podatkov. Namesto, da preslika relacije neposredno iz baze,

Vir	POST	GET	DELETE
/devices	Doda novo napravo	Vrne vse naprave	Odstranitev vseh strank
/devices/1	Napaka	Vrne napravo 1	odstrani napravo 1
/senzors/1/ measurments	Doda novo meritev	Vrne meritve	Napaka

Listing 1: Opisi glavnih dostopov do API-ja.

določene podatke združi skupaj in zakrije implementacijo v bazi. Če želi uporabnik poizvedovati po napravah, mu ni potrebno poznati da so senzorji in nastavitve shranjene v ločenih tabelah. Samo z eno poizvedbo API lahko uporabnik dostopa do vseh podatkov. Primer vrnjenih podatkov naprave je prikazan na sliki 2.

API uspešnost poizvedbe sporoča na dva načina. Vsaki poizvedbi glede na to, ali je prišlo do napake ali ne, določi HTTP kodo stanja. Koda "*200*"pomeni, da ni prišlo do napake, koda "*500*", da je bila napaka na strežniku in koda "*400*", da je napako naredil uporabnik. Poleg HTTP kode uporabniku vedno vrne parameter *success*, ki mu pove ali je bila poizvedba uspešna ali ne. Če je bila poizvedba uspešna so zraven pripeti podatki, v nasprotnem primeru pa je dodan parameter *"error"*, kjer je opisana napaka do katere je prišlo. Primer neuspešne povezave je prikazan na sliki 3.

```
{
"success": true,
"data": {
"id": "MCTEMP18",
"name": "Razvoj 2",
"location": null,
"description": null,
"sensors": [
{
"id": 6,
"name": "Temperature",
"unit": null,
"interval": null,
"device_id": "MCTEMP18"
}, ...
}
}
```

Listing 2: Primer vrnjenih podatkov na poizvedbo *GET /device/1*

```
{
"success": false,
"error": "Missing required parramaters.
Missing parameters: device_id"
}
```

Listing 3: Primer uporabniške napake na poizvedbo, kjer je uporabnik pozabil dodati parameter device_id

4.2.3 Mqtt listener

Mqtt listener je prav tako kot API, aplikacija napisana v Node.js z uporabo ogrodja Express. Pravzaprav sta si po sami izgradnji in uporabi knjižnic zelo podobna. Naloga *Mqtt listenerja* je, da posreduje podatke, ki so jih temperaturni senzorji poslali na MQTT strežnik in te preko API-ja vpiše v bazo.

Mqtt listener ob začetku delovanja začne poslušati na temo, (ang. topic) na katero temperaturni senzorji pošiljajo meritve. Za poslušanje MQTT komunikacije se uporablja knjižnica *async-mqtt*. Ta dovoljuje, da smo hkrati naročeni na več tem, a vse teme pošilja podatke na istega poslušalca. To pomeni, da moramo ob sprejetem paketu najprej razčleniti temo in ugotoviti kakšnega tipa je sporočilo in šele nato razčleniti podatke.

Temperaturni senzorji svoje podatke pošiljajo na več različnih tem, kot so */measurements* in */settings*. Da se lahko prijavimo na obe temi hkrati uporabimo nadomestni znak #.

Da je koda preglednejša, so vse komunikacije z API-jem shranjene na enem mestu v objektu *DabApi*. S tem je koda razdeljena na dva dela. Na objekt, ki pošilja API zahteve in objekt, ki sprejeta sporočila obdela in jih pošlje naprej.

4.2.4 Mosquitto mqtt

Mosquitto mqtt je strežnik, ki skrbi za MQTT povezavo med temperaturnimi senzorji in zalednim strežnikom. V MQTT protokolu nastopa kot posrednik (ang. broker) med poslušalci in izdajatelji. Vse kar izdajatelji oddajo na določeno temo posreduje vsem poslušalcem, ki poslušajo na to temo. V našem primeru so izdajatelji temperaturni senzorji, poslušatelj pa je samo *Mqtt listener*. V splošnem bi bilo lahko poslušalcev več, saj lahko začne poslušati kdorkoli z dostopom do strežnika.

Za varnost podatkov je poskrbljeno z protokolom SMQTT. Po delovanju je identičen protokolu MQTT, dodana sta samo koraka šifriranja in

dešifriranja. Za delovanje SMQTT-ja so zato potrebni certifikati. Ti se naložijo ob zagonu strežnika, zato je potrebno, da povežemo zunanjo datoteko z certifikati na ustrezno datoteko znotraj vsebnika.

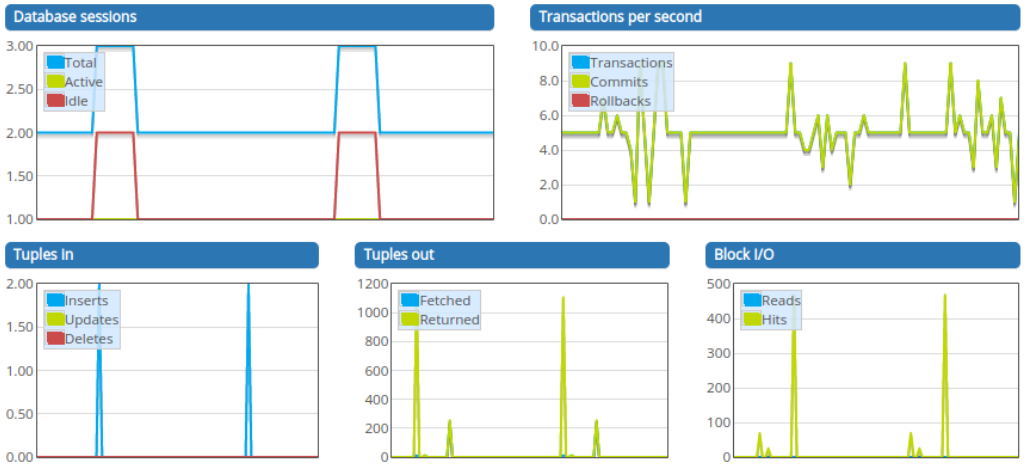
Prav tako moramo povezati tudi konfiguracijsko datoteko. Preko te datoteke nastavimo delovanje strežnika. V njej so opisane nastavitve strežnika in ostali parametri potrebni za delovanje, kot na primer omrežna vrata, največje dovoljeno število povezav, lokacija dnevnika ipd.

4.2.5 Spletni vmesnik PgAdmin

PgAdmin nam v celotnem sklopu služi bolj kot pripomoček in ne kot delovni element. Pomaga nam pri upravljanju z bazo. Nanjo se lahko povežemo preko spletnega vmesnika, kjer lahko pregledamo vsebino tabel in nad tabelami izvajamo poizvedbe.

Na pregledni plošči nam prikazuje tudi trenutne podatke, kot so število povezav na bazo, število transakcij, število vhodno/izhodnih operacij ipd. Pregledna plošča je prikazana na sliki 4.4.

Da preprečimo, nedovoljene dostope do baze, se mora uporabnik pred uporabo prijaviti z uporabniškim imenom in geslom. Ime in geslo se določita ob postavitvi. V našem primeru se določita v datoteki Docker-Compose in sicer z nastavitvijo okoljskih spremenljivk *POSTGRES_USER* in *POSTGRESS_PASSWORD*. z uporabo spletnega vmesnika PgAdmin, smo sistemu dodali dodatno plast zaščite. Pred povezavo je potrebno poznati najprej uporabniško ime in geslo spletnega vmesnika pgAdmin in nato še geslo za povezavo do baze.



Slika 4.4: Grafi na pregledni plošči spletnega portala PgAdmin.

4.3 Dnevnik

Končna postavitev zalednih sistemov je ponavadi na lokacijah do katerih nimamo neposrednega dostopa. V primeru napake zato ne moremo vedeti, zakaj je napaka nastala. Pomembno je, da se vse akcije, še posebej pa napake beležijo in nam tako dajo vpogled v delovanje strežnika in s tem možnost da odkrijemo napake.

Napake beležijo vsi vsebniki zalednega sistema razen PgAdmin. Ta za delovanje sistema ni kritičen. Baza postgres in strežnik Mosquitto MQTT že sama beležita svoj dnevnik in je zato potrebno dnevniške datoteke samo shraniti izven vsebnika, da se te ob izbrisu le tega ne izbrišejo. Komponenti *Mqtt listener* in API sta obe napisani z orodjem Node.js in zato v osnovni nimata nobenih dnevniških zapisov. Zapise moramo zato ustvarjati sami. Obe komponenti uporabljata knjižnico *logger* s pomočjo katere se shranjujejo izpisi med delovanjem programa v datoteke. S knjižnico *logger* tudi poskrbimo, da velikost datotek na naraste prekomerno. Ko dnevniške datoteke presežejo predpisano velikost se te stisnejo, odpre pa se nova prazna dnevniška datoteka.

Poglavje 5

Mobilna aplikacije iTemp

Mobilna aplikacija iTemp je aplikacija namenjena končnemu uporabniku. Uporabniku omogoča povezavo na strežnik s podatki in nato prikaz trenutnih meritev ter njihovo zgodovino.

5.1 Oblika vmesnika

Ker je aplikacija hibridna je bilo potrebo oblikovati dve različni aplikacij. Eno za Android in eno za iOS. Telefonom Windows nismo posvečali posebne pozornosti. Vsaka od operacijskih sistemov ima svoje značilnosti in smernice oblikovanje, ki se jih je potrebno držati, če želimo, da je aplikacija na telefonu uporabniku uporabna in na izgled podobna domorodnim aplikacijam. Razlike med oblikami vmesnika so prikazane na sliki 5.1.

- **Slog pisave**: iOS sistemi za svoj slog pisave uporabljajo pisavo Helvetica, medtem ko Android uporablja slog pisave Roboto.

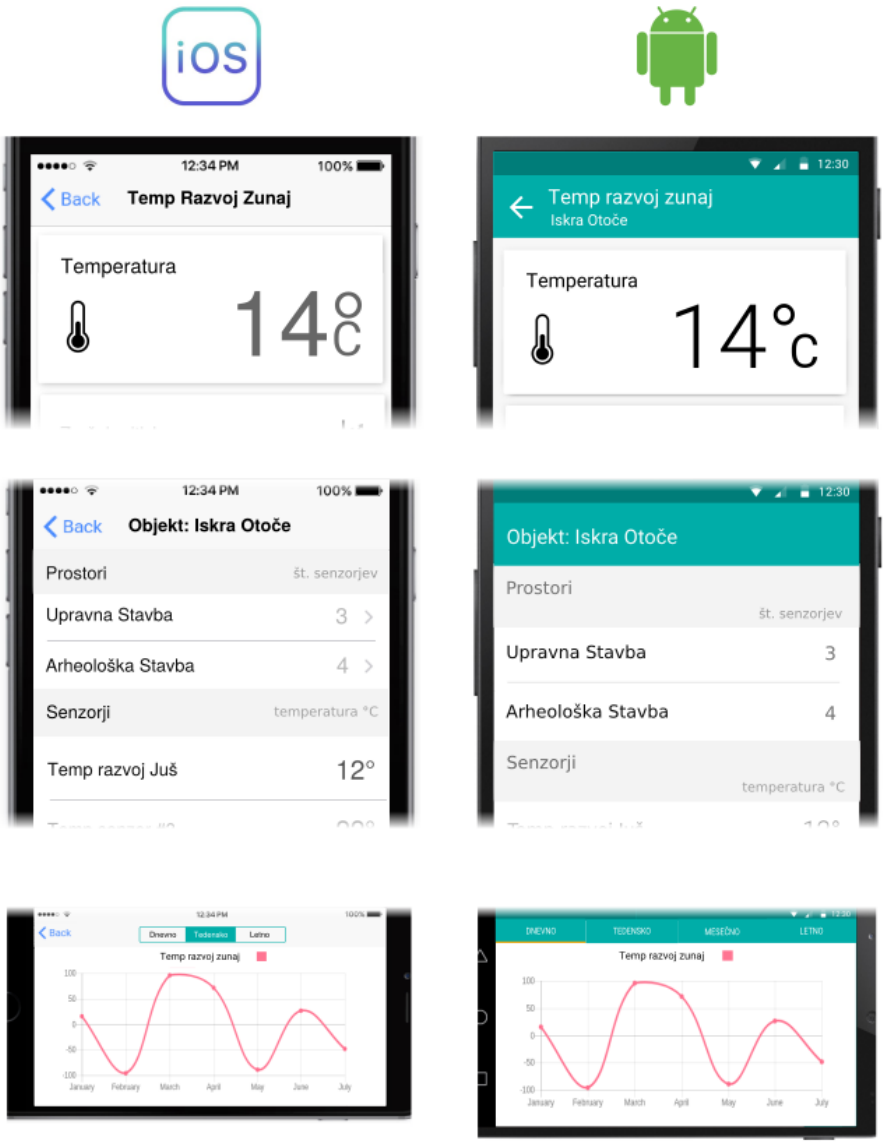
- **Gumb za nazaj**: iOS telefoni ne vsebujejo fizičnega gumba za vrnitev nazaj, zato je od samih aplikacij zahtevano, da le tega vključijo. Android telefoni že vsebujejo gumb za nazaj in ga zato ni potrebno dodajati.

- **Spremembe v obliki** Android in iOS uporabljata različno obliko grafičnih elementov in je zato potrebno poskrbeti, da so ti kar se da podobni elementom domorodnih aplikacij.

5.1.1 Izbira barve

Za določitev barve aplikacije je potrebno določiti primarno barvo, ki je uporabljena največ in je prepoznavna kot barva aplikacije in sekundarno barvo, ki je kontrastna primarni barvi in obarva elemente, ki želimo da izstopajo. Primarna barva se je določila kot perzijsko zelena ■ in se tako ujema z osnovno barvo podjetja. Za sekundarno barvo pa se je izbrala spletno oranžna ■, ki je bila tudi določena po navdihu barve podjetja.

Poleg primarne in sekundarne barve je bilo potrebno določiti tudi temne in svetle odtenke obeh barv. Te se uporablja predvsem pri stiku dveh elementov, kjer nočemo spreminjati osnovne barve, ampak še vedno želimo barvno ločiti elementa med seboj. Primer uporabe je med orodno vrstico in vrstico z obvestili. Razlika je prikazana na sliki 5.1.
Primarni temni in svetli odtenek: ■ ■ ■
Sekundarni temni in svetli odtenek: ■ ■ ■



Slika 5.1: Primerjava vmesnika med telefonoma z operacijskim sistemom iOS in Android.

5.2 Opisi posameznih delov aplikacije

Aplikacija je napisana v ogrodju Angular. Angular že sam po sebi spodbuja in podpira modularno sestavo aplikacije. Aplikacija smo zato razdelili najprej na module, le ti pa se nato delijo na komponente, storitve, direktive in usmerjevalnike.

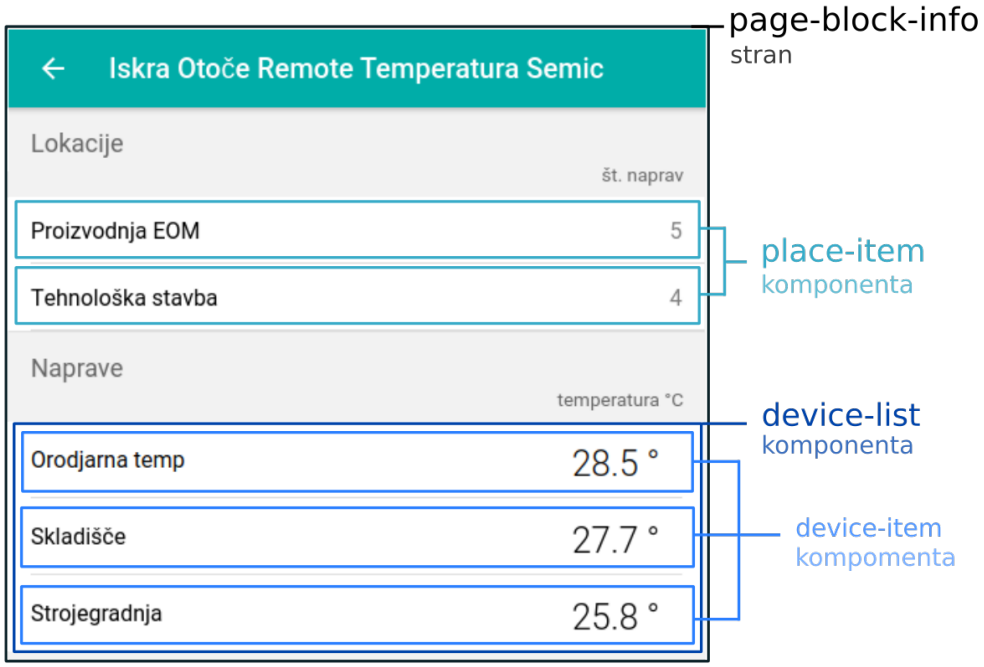
Ker je aplikacija majhna je modul en sam in vsebuje celotno aplikacijo.

Komponente so del modula. Vsaka opravlja s svojim koščkom zaslona. Vsaka komponenta je sestavljena iz pogleda, ki določa izgled in upravljalnika, ki opravlja z njim.

V Inoicu imamo poleg komponent tudi posebne vrste komponent poimenovane strani (ang. pages). Po strukturi so identične navadnim komponentam, a se razlikujejo v tem, da vsaka stran upravlja s celim oknom aplikacije in v tem, da imajo že v naprej vgrajene funkcije za navigacijo. Vsakemu oknu pripada svojo stran, ki je potem sestavljena iz manjših komponent. Razdelitev strani na komponente je prikazana na sliki 5.2.

Razdelitev na komponente je pomembna saj nam razdeli stran na več manjših delov. Namesto ene velike strani, imamo več manjših in s tem preglednejših komponent. Prednost komponent je tudi v ponovni uporabni. Eno komponento lahko uporabimo na več straneh.

Poleg komponent nam ogrodje Angular ponuja tudi storitve. Storitve so zelo podobne komponentam, le da so brez pogleda. Komponente naj ne bi same pridobivale, še manj pa shranjevale podatke. Namenjene so samo predstavitvi podatkov, pridobivanje pa prepustijo storitvam [2]. V naši aplikaciji sta prisotni dve storitvi: *block-service*, ki skrbi za shranjevanje povezav in *connection-handler*, ki skrbi za povezavo do vira podatkov.



Slika 5.2: Razdelitev okna na posamezne komponente.

5.3 Tipi povezav

Aplikacija podpira povezavo na različne tipe strežnikov. Podprti strežniki so: MQTT strežnik, MiSmart strežnik in naš zaledni sistem DAB. Vsak od njih ima različen način komunikacije. Da bi lahko vsi objekti dostopali do podatkov, brez da bi morali poznati implementacijo posamezne komunikacije, vsi objekti dostopajo do podatkov preko identičnih zahtev.

Zahteve delujejo na princip opazovalcev (ang. observables). Običajne funkcije ob izhodu vrnejo podatke. To pomeni, da mora celotna koda počakati, da se funkcija izvede in vrne podatke. Veliko programskega časa je zato zelo slabo izkoriščenega. Funkcija zato namesto podatkov, brez zakasnitve vrne objekt, na katerega se lahko uporabnik prijavi (ang. subscribe). Ko podatki prispejo, se kliče funkcija *observer.next()*, ki posreduje podatek naročniku. Prednost tega načina je, da lahko naročniku večkrat posredujemo podatke.

Da samim objektom ni potrebno skrbeti na kakšen način dostopajo do podatkov, so se funkcije za poizvedovanje o podatkih zavile v svoj vmesnik *connection-handler*, ki določuje samo imena funkcij, vsebina, pa je implementirana v vsakem krmilniku posebej.

5.3.1 Krmilnik z povezavo MQTT

Vsi senzorji pošiljajo svoje meritve preko MQTT protokola na MQTT strežnik, na katerega se lahko mobilna naprava prijavi, kjer posluša in sprejema podatke.

Namen MQTT strežnika je posredovanje podatkov, prispelih od senzorjev, naročnikom. Tu se pojavi problem, da ne moremo poizvedovati po podatkih, kadarkoli bi želeli. Če aplikacija želi pokazati meritev nekega senzorja ne more poslati zahteve MQTT strežniku, ki bi vrnila katerokoli meritve. Drugi problem je, da se podatki pošiljajo v več manjših paketih. Namesto, da bi vsi podatki nekega senzorja prišli hkrati se pošljejo razdrobljeno. Problema rešimo tako da podatke skladiščimo in čakamo na prihod vseh podatkov, predem jih lahko pokažemo.

Probleme smo rešil z implementacijo storitve *Mqtt database*. Storitev ob začetku delovanja inicializira prazne objekte, kamor se shranjujejo naprave, senzorji in njihove meritve. Hkrati se tudi priklopi na vse teme MQTT strežnika in začne poslušati. Ko prispejo podatki, te podatke razčleni in jih shrani na ustrezna mesta znotraj objektov. Stare podatke osveži, v primeru novih, pa ustvari nove objekte. Ob vsaki spremembi podatkov preveri, ali so prispeli vsi podatki in če je potrebno obvestiti naročnike o novih sprejetih podatkih.

Problem nastane tudi, ko želimo pokazati zgodovino podatkov. Preko MQTT se pošiljajo samo trenutne meritve. Če želimo dostopati do zgodovine, moramo poslati poseben ukaz na temo */device/deviceId/gethistory* in s tem sporočiti napravi, da naj pošlje svojo zgodovino. Zgodovina obsega obdobje dveh dni, saj za daljše obdobje ni dovolj prostora v pomnilniku.

Problem je nastal tudi z identifikacijskimi številkami senzorja. Pričakuje se da ima vsak senzor unikatno identifikacijsko številko po kateri ga lahko ločimo od ostalih senzorjev. Od MQTT strežnika dobimo samo ime senzorja. Zato je bilo potrebno ustvariti identifikacijsko številko z združitvijo imena senzorja in ID naprave. Oba podatka sta se zapisala v JSON objekt, ki se je potem pretvoril v niz. Niz se je uporabljal kot identifikacijske številke senzorja.

5.3.2 Krmilnik z bazo MiSmart

MiSmart je baza razvita za potreba shranjevanja meritev merilnih inštrumentov, ki jih proizvaja podjetje. Ob razvoju temperaturnih senzorjev je bila logična odločitev shraniti tudi te meritve v že obstoječo bazo.

Do baze se dostopa preko spletnega REST API-ja in je zato implementacija krmilnika preprosta. Ko aplikacija potrebuje podatke, se pošlje zahteva na spletni API. Edini problem je, da API klici niso bili prilagojeni potrebam mobilne aplikacije in moramo za pridobitev nekaterih podatkov poslati več zahtev.

5.3.3 Krmilnik zalednega sistema DAB

Ker je bil zaledni sistem razvit skupaj z mobilno aplikacijo, so API klici prilagojeni potrebam mobilne aplikacije. Implementacija zato ni bila zahtevna. Razdelitev krmilnika na dva dela je izboljšal preglednost kode. Prvi del skrbi za povezavo do baze, drugi pa za obdelavo in preslikavo podatkov.

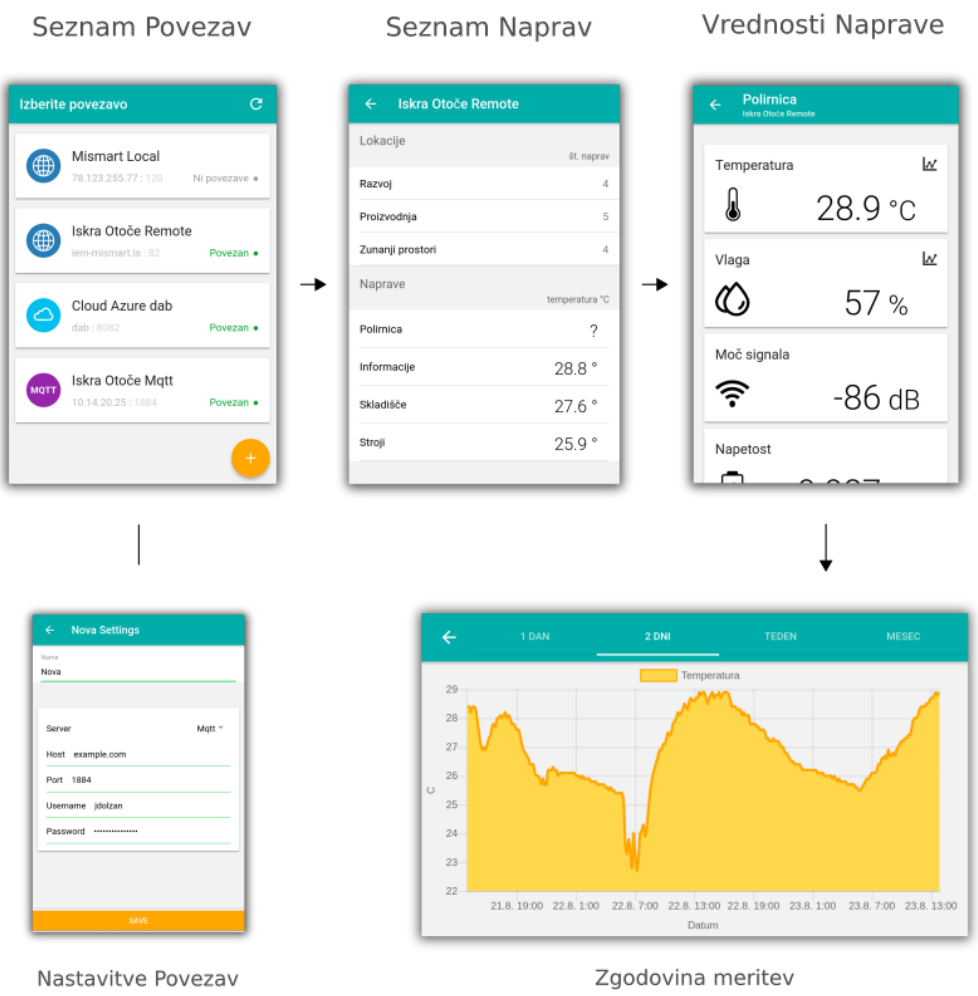
5.4 Predstavitev delovanja

Na prvem zaslonu ima uporabnik možnost opravljanja z povezavami do strežnikov. Vsak strežnik je označen s svojo ikono, imenom, naslovom in statusom povezave.

S klikom na gumb "+", ki je v spodnjem desnem kotu, lahko uporabnik doda novo povezavo na strežnik. Odpre se mu okno, kjer lahko uredi povezavo.

Z izbiro strežnika se izpišejo vsi senzorji in prostori, kjer se senzorji nahajajo. Z navigacijo po prostorih lahko uporabnik pride do vsakega merilnega mesta. Ob vsakem senzorju se izpiše tudi zadnja izmerjena temperatura. Ob primeru napake se namesto temperature izpiše "?".

Ob kliku na senzor se prikažejo vse izmerjene veličine naprave. Te se od naprave do naprave razlikujejo. Ponavadi so to vlažnosti, zračni pritisk in pa lastnosti naprave, kot so napetost baterije in moč Wi-Fi signala. Vsaka od meritev ima svojo ikono. S klikom na izmerjeno veličino se odpre novo okno, kjer je omogočen vpogled v enodnevno, dvodnevno, tedensko in mesečno zgodovino.



Slika 5.3: Okna aplikacije in povezave med njimi.

Poglavje 6

Zaključek

V diplomski nalogi je bil predstavljen razvoj mobilne aplikacije in zalednega sistema za shranjevanje podatkov. Vključena je bila tudi predstavitev uporabljenih orodji, primerjava med različnimi načini izdelave hibridne aplikacije, predstavitev merilnih senzorjev, opis zalednega sistema in opis mobilne aplikacije. Rezultat je bila mobilna aplikacija, ki uporabniku nudi pregleden in enostaven pregled temperatur in ostalih izmerjenih veličin. Realiziral se je tudi zaledni sistem, ki skrbi za shranjevanje in posredovanje podatkov mobilni aplikaciji.

6.1 Ideje za nadaljnji razvoj

Kot taka je aplikacije zaključena celota, ki podpira vse potrebne funkcije za delovanje. Še vedno pa bi jo lahko razširili z dodatnimi funkcionalnosti glede na uporabniške zahteve.

6.1.1 Uporabniški vmesnik za nadzor baze

Manipulacija baze je trenutno možna samo preko API klicev in ročnih SQL poizvedb. Takšna načina komunikacija z bazo sta primerna za programsko opremo a neprimerna za uporabnike. Dodal bi se lahko uporabniški vmesnik,

ki bi omogočal enostavno urejanje in spreminjaje podatkov v bazi s pomočjo grafičnega vmesnika.

6.1.2 Varnost

Za varnost je bilo v celotnem sistemu že nekaj poskrbljeno, vendar je to še možno izboljšati. Komunikacija z bazo se izvaja z nezavarovanim protokolom HTTP. Ta protokol bi lahko nagradili z uporabno protokola HTTPS, ki omogoča varen prenos podatkov z uporabo TLS kanalov.

Dostop do podatkov tudi ni primerno omejen. Trenutni imajo vsi uporabniki dostop do vseh podatkov. Bazo bi lahko razširili, da bi vsebovala seznam uporabnikov ter njihove pravice. Z uvedbo uporabnikov, bi bilo potrebno spremeniti tudi mobilno aplikacijo. Dodati bi bilo potrebno novo okno za prijavo in registracijo uporabnika.

6.1.3 Dodane funkcionalnosti aplikacije

Za izboljšanje uporabniške izkušnje, bi lahko aplikacijo nadgradili z funkcionalnosti kot so:

- **Določanje priljubljenih naprav**. Uporabnik bi lahko z določanjem priljubljenih naprav, te lažje razvrščal in imel preglednejši nadzor nad izmerjenimi vrednosti, še posebej če je teh zelo veliko.
- **Graf več izmerjenih vrednosti hkrati**. En graf sam po sebi ni tako zanimiv. Z možnostjo prikazovanja več veličin na enem grafu, bi lahko uporabnik primerjal izmerjene vrednosti in med njim iskal soodvisnosti.
- **Shranjevanje meritev v načinu brez povezave**. Namesto, da bi aplikacija prikazovala podatke smo takrat, ko je povezana na internet, bi s shranjevanjem podatkov, uporabnik še vedno imel možnost pregledati podatke shranjene na telefonu.

Literatura

[1] Adobe announces agreement to acquire nitobi, creator of phonegap. Dosegljivo: https://news.adobe.com/press-release/adobe-creative-cloud-dps/adobe-announces-agreement-acquire-nitobi-creator-phonegap. [Dostopano: 20. 8. 2018].

[2] Angular - why services. Dosegljivo: https://angular.io/tutorial/toh-pt4. [Dostopano: 28. 8. 2018].

[3] Ionic docs - ionic native. Dosegljivo: https://beta.ionicframework.com/docs/native/. [Dostopano: 1. 10. 2016].

[4] Ionic ui components. Dosegljivo: https://ionicframework.com/docs/components/. [Dostopano: 21. 8. 2018].

[5] Mobile app performance redux. Dosegljivo: https://medium.com/@harrycheung/mobile-app-performance-redux-e512be94f976. [Dostopano: 28. 8. 2018].

[6] Node git repository. Dosegljivo: https://github.com/nodejs/node. [Dostopano: 26. 8. 2018].

[7] Postgress release date. Dosegljivo: https://www.postgresql.org/about/news/978/. [Dostopano: 12. 8. 2018].

[8] Postgress suported platforms. Dosegljivo: https://www.postgresql.org/docs/9.1/static/supported-platforms.html. [Dostopano: 27. 8. 2018].

[9] Initial revision of "git", commit. Dosegljivo: https://github.com/git/git/commit/e83c5163316f89bfbde7d9ab23ca2e25604af290, 2005. [Dostopano: 26. 8. 2018].

[10] What is phppgadmin? Dosegljivo: http://phppgadmin.sourceforge.net/doku.php, 2013. [Dostopano: 26. 8. 2018].

[11] Gaston C. Hillar. *MQTT Essentials - A Lightweight IoT Protocol*. Packt Publishing, 2017.

[12] Max Lynch. Announcing windows support in ionic 2. Dosegljivo: https://blog.ionicframework.com/announcing-windows-support-in-ionic-2. [Dostopano: 20. 8. 2018].

[13] Jonathan Peppers. *Xamarin 4.x Cross-Platform Application Development - Third Edition*. Packt Publishing, 2016.

[14] Sudhi Seshachala. Docker vs vms. Dosegljivo: https://devops.com/docker-vs-vms/. [Dostopano: 15. 8. 2018].