Daljinski nadzor i upravljanje bežičnim čvorovima

Klarić, Tvrtko

Master's thesis / Diplomski rad

2016

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:623600

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-03-29**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

zir.nsk.hr

Image not found or type unknown

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij

DALJINSKI NADZOR I UPRAVLJANJE BEŽIČNIM ČVOROVIMA

Diplomski rad

Tvrtko Klarić

Osijek, 2016.

SADRŽAJ

1. UVOD 1
2. INTERNET OBJEKATA 2
3. POSTOJEĆA RJEŠENJA 4
3.1. Dweet 4
3.2. Freeboard 4
3.3. IFTTT 5
3.4. Cayenne 5
4. SUSTAV ZA NADZOR I UPRAVLJANJE ČVOROVIMA 6
4.1. Pregled korištenih tehnologija 6
4.1.1. ADO.NET 7
4.1.2. *WebSocket* 7
4.1.3. ASP.NET MVC 7
4.1.4. Knockout.js 8
4.1.5. Python 8
4.2. Baza podataka 9
4.3. Web aplikacija 11
4.3.1. Sloj za pristup podacima 13
4.3.2. Sloj poslovne logike 14
4.3.3. Prezentacijski sloj 15
4.4. Program na udaljenom čvoru 21
5. DEMONSTRACIJA SUSTAVA 22
5.1. Program na udaljenom čvoru 22
5.2. Web aplikacija 24
6. ZAKLJUČAK 27
LITERATURA 28
SAŽETAK 29
ABSTRACT 30
ŽIVOTOPIS 31
PRILOZI 32

1. UVOD

O pojmu Interneta objekata piše se i govori već nekoliko godina. Internet objekata je sustav međusobno povezanih računalnih, mehaničkih i digitalnih strojeva, predmeta ili ljudi koji imaju mogućnost međusobne komunikacije i prijenosa podataka putem računalne mreže bez potrebe za ljudskom interakcijom. Dok ga jedni najavljuju kao sljedeću tehnološku revoluciju, drugi upozoravaju na opasnosti koje će, ako ikada istinski zaživi, donijeti sa sobom. S postojećim sklopovljem, protokolima, programskim rješenjima te prednostima i problemima ove tehnologije imao sam se prilike upoznati prilikom pohađanja istoimenog kolegija na drugoj godini diplomskog studija. Usvojena znanja odlučio sam proširiti odabirom teme koja se većim dijelom tiče navedene problematike.

Zadatak je rada izrada sustava koji će korisniku omogućiti prikupljanje, pohranjivanje i prikaz vrijednosti izmjerenih na različitim čvorovima kao što su mikroupravljači opremljeni odgovarajućom senzorikom, jednokartična računala poput Raspberry Pi i sl. Osim prikupljanja podataka omogućeno je i slanje odgovarajućih naredbi na čvorove.

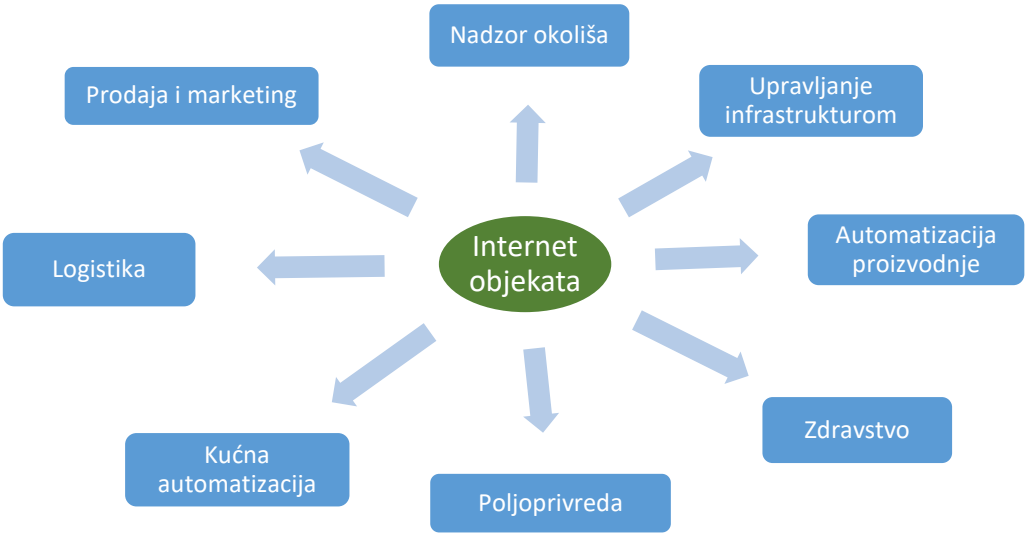
Rad je strukturiran na sljedeći način. U uvodnom je poglavlju ukratko objašnjeno što točno označava pojam „Internet objekata“. Zatim je istaknuto nekoliko aplikacija koje nude rješenja za udaljeni nadzor i upravljanje. Glavni dio rada bazira se na analizi napravljenog rješenja. Izvršen je pregled najznačajnijih korištenih tehnologija, objašnjen je korišteni arhitekturalni uzorak te je analiziran svaki pojedini sloj aplikacije. Sustav se sastoji od web aplikacije i udaljenih čvorova, stoga su navedeni i uvjeti koje mora ispunjavati program na udaljenom čvoru koji koristi aplikaciju. U posljednjem je poglavlju demonstrirana upotreba aplikacije i dan je jednostavan primjer mogućeg programa na čvoru.

2. INTERNET OBJEKATA

Razni izvori navode nekoliko imena za sustav koji ima jedan cilj – omogućiti povezanost fizičkih predmeta putem Interneta. Međutim, Internet objekata (engl. *Internet of Things*, u nastavku IoT), Internet stvari, Internet svega, Mreža stvari više je od toga. Prema [1], Internet objekata globalna je infrastruktura koja nudi napredne usluge međusobno povezujući objekte korištenjem postojeće i nove međusobno kompatibilne informacijske i komunikacijske tehnologije. U kontekstu IoT-a, objekt je dio fizičkog (fizički predmet) ili informacijskog (virtualni predmet) svijeta koji ima mogućnost biti identificiran i integriran u komunikacijske mreže. Uređaji uz obaveznu sposobnost komunikacije mogu obavljati funkcije mjerenja, upravljanja, pohrane i obrade podataka. Istovremeni napredak u razvoju ugradbenih i mikro-elektromehaničkih sustava i bežične komunikacije stvorio je osnovu koja je omogućila velike pomake u realizaciji IoT-a. Tradicionalni sustavi bežičnih senzorskih mreža, upravljački sustavi i sustavi za kućnu automatizaciju također su pridonijeli razvoju.

Mogućnosti primjene IoT-a možemo naći u raznim djelatnostima. Nadzor okoliša baziran na IoT tehnologijama – senzorske mreže za mjerenje kvalitete vode, zraka, atmosferskih prilika i sl. uvelike bi pomogle pri njegovoj zaštiti. Osim toga, takvi sustavi mogli bi biti integrirani u sustave koji bi ukazivali na opasnost od katastrofalnih prirodnih nepogoda. Kod upravljanja infrastrukturom (ceste, željeznice, mostovi itd.), IoT bi se mogao koristiti za nadzor strukturalnih promjena koje bi potencijalno mogle dovesti do rizika. Uz to, pomogao bi pri planiranju efikasnijeg rasporeda održavanja te potencijalno omogućio veću razinu automatiziranosti u samoj kontroli toka prometa. Automatizacija u proizvodnji svakako nije nov pojam, međutim, samo postojanje pojma *Industrial Internet of Things* ukazuje na mogućnost primjene IoT-a i u ovom području. Od upravljanja logistikom, kontrole proizvodnje, pa sve do automatizacije distribucije, omogućit će dinamičko odgovaranje zahtjevima tržišta i optimizaciju proizvodnog procesa. Uz alternativne izvore energije, veliku ulogu u njenom očuvanju može igrati i IoT. Očekuje se da će IoT uređaji biti ugrađeni u razne električne uređaje te će pružati korisne informacije koje će pomoći pri efektivnijem balansiranju proizvodnje i potrošnje energije. Uz to, mogli bi korisnicima omogućiti daljinski nadzor i upravljanje nad tim istim uređajima. U medicini bi služio udaljenom nadzoru kroničnih pacijenta, i kao automatski sustav obavijesti za hitne slučajeve. Još jedna moguća primjena bili bi uređaji koji bi na temelju izmjerenih vrijednosti mogli korisniku predlagali savjete koji bi poticali zdrav život. U nekima od navedenih primjera spominje se zahtjev za automatizacijom, odnosno nekom vrstom autonomnog, inteligentnog upravljanja sustavom.

Iako originalni koncept IoT-a nije podrazumijevao takvo ponašanje, sve je veći pomak u istraživanjima kojima je cilj integrirati i tu odliku.



Sl. 2.1. Moguće primjene Interneta objekata.

Kao i svaka druga tehnologija u razvoju, tako se i IoT susreće s mnogobrojnim poteškoćama i kritikama. Glavnina se odnosi na sigurnost, privatnost i utjecaj na okoliš. Često se izražava sumnja da se uz rapidni razvoj ne posvećuje dovoljno pažnje zadovoljavajućoj razini zaštite. Širenjem IoT-a hakerski napadi, osim virtualne, počinju predstavljati i fizičku prijetnju. Stoga je 23. rujna 2015. osnovana Zaklada za sigurnost IoT-a (engl. *Internet of Things Security Fundation*) s ciljem promoviranja sigurnosti promicanjem saznanja i najbolje prakse iz tog područja. Obećanja o povećanju kvalitete života, većoj efikasnosti i uštedi neki izvori vide kao prikriveni napad na privatnost ljudi u svrhu društvene i političke manipulacije. Prvi od problema koji se ističe jest pristanak korisnika. Korisnik bi trebao biti u potpunosti upućen u sve implikacije prikupljanja podataka te odluku donijeti na temelju tih informacija. Međutim, korisnici često nemaju dovoljno vremena ili tehničkog znanja. Uz to, zaštita privatnosti i standardi koji je se tiču trebali bi promovirati slobodu izbora. Te na posljetku, u budućnosti bi se trebalo obratiti više pažnje na anonimnost korisnika prilikom prijenosa podataka. Tehnologije zasnovane na poluvodičkim elementima, teški metali i toksične sintetičke kemikalije korištene u modernim elektroničkim uređajima čine IoT velikom prijetnjom za okoliš. Ugradnja dodatne elektronike u dosad relativno jednostavne uređaje skratila bi njihov vijek trajanja i dovela do još veće potrošnje sirovina. Uspije li nadvladati sve te probleme, Internet objekata zasigurno će dovesti do velikih promjena u načinu i kvaliteti ljudskog života.

3. POSTOJEĆA RJEŠENJA

S obzirom na to da se već nekoliko godina radi na razvoju IoT-a, ne čudi činjenica da se na Internetu može naći mnoštvo aplikacija koje nude usluge vizualizacije podataka i upravljanja uređajima. Sama vizualizacija podataka, za one koji žele napraviti vlastito rješenje, također je olakšana velikim brojem *JavaScript* biblioteka za izradu interaktivnih prikaza podataka (*d3.js, HighCharts.js, GoogleCharts, Chart.js* itd.) pa i gotovih servisa poput *Plotly* i sl. U nastavku je dan pregled samo nekih rješenja.

3.1. Dweet

Dweet (http://dweet.io) je servis koja omogućava vrlo jednostavno slanje poruka s jednog uređaja na sve uređaje koji su pretplaćeni na isti kanal. Poruke se šalju jednostavnim API pozivom, primjerice: https://dweet.io/dweet/for/**my-device-name**?status=on. Podebljanim slovima označeno je jedinstveno ime putem kojeg se uređaji mogu međusobno raspoznati. Nakon toga slijede parametri koji će se uz dodatne podatke odaslati svim pretplaćenim uređajima. Podaci se šalju u JSON formatu. Posljednji poslani podatak može se dobiti slanjem zahtjeva na https://dweet.io/get/latest/dweet/for/**my-device-name**. Prednost *Dweeta* je što ne zahtijeva nikakvu instalaciju, a na poprilično jednostavan način rješava problem komunikacije između uređaja. Na službenoj stranici mogu se preuzeti *JavaScript, Python* i *Ruby* biblioteke koje olakšavaju razvoj aplikacija koje bi koristile ovaj servis. Nedostatak je što je ipak riječ samo o rješenju koje omogućava komunikaciju između uređaja, a od korisnika se očekuje poprilična količina tehničkog znanja s obzirom da bi sam morao napraviti aplikaciju koja bi koristila ovu uslugu.

3.2. Freeboard

Freeboard (http://freeboard.io) je web aplikacija koja nudi mogućnost brze izrade interaktivnih panela za vizualizaciju podataka u stvarnom vremenu jednostavnim *drag and drop* mehanizmom. Aplikacija se može integrirati s gore spomenutim *Dweetom* ili podatke prikupljati pristupom bilo kakvom drugom web API-u. Najveća prednost *Freeboard* aplikacije njezina je jednostavnost. Omogućava izgradnju zanimljivih interaktivnih prikaza podataka u vrlo kratkom roku. Aplikacija je besplatna za korištenje, a naplaćuju se jedino privatni paneli ako korisnik ne želi da njegovi podaci budu vidljivi javnosti. Riječ je o *open source* projektu i sav je kod dostupan na Githubu. Nedostatak je aplikacije što nudi samo vizualizaciju podataka bez mogućnosti interakcije sa

uređajima. Kao i u prethodnom primjeru, od korisnika se zahtijeva određena količina tehničkog znanja kako bi podatke dostavio do aplikacije.

3.3. IFTTT

If This Then That (https://ifttt.com) aplikacija također nudi *publish-subscribe* uslugu uz mogućnost dodavanja okidača na događaje. Nekoliko okidača može se povezati u lanac događaja koji se nazivaju recepti. Ova aplikacija istinski se uključila u prostor IoT-a ponudivši integraciju s Belkin WeMo uređajima. Belkin WeMo je serija uređaja tvrtke Belkin International koji omogućavaju korisnicima udaljeno upravljanje kućnom elektronikom. Između ostalog, među proizvodima se nalaze prekidači, senzori pokreta, pametne LED žarulje, grijalice, kamere, aparati za kavu itd. Prednost aplikacije je velika količina gotovih recepata koje korisnik može koristiti, a svakako treba istaknuti i jednostavnost kojom se mogu stvarati novi recepti. Uz praktički nikakvo tehničko znanje mogu se napraviti poprilično kompleksni lanci događaja koji olakšavaju svakodnevne radnje ili automatskim prikupljanjem podataka daju dodatni uvid u svakodnevnicu. Nedostatak ovog sustava je što je za bilo kakvu interakciju s fizičkim svijetom potrebno nabaviti službene uređaje što u konačnici čini ovo rješenje poprilično skupim.

3.4. Cayenne

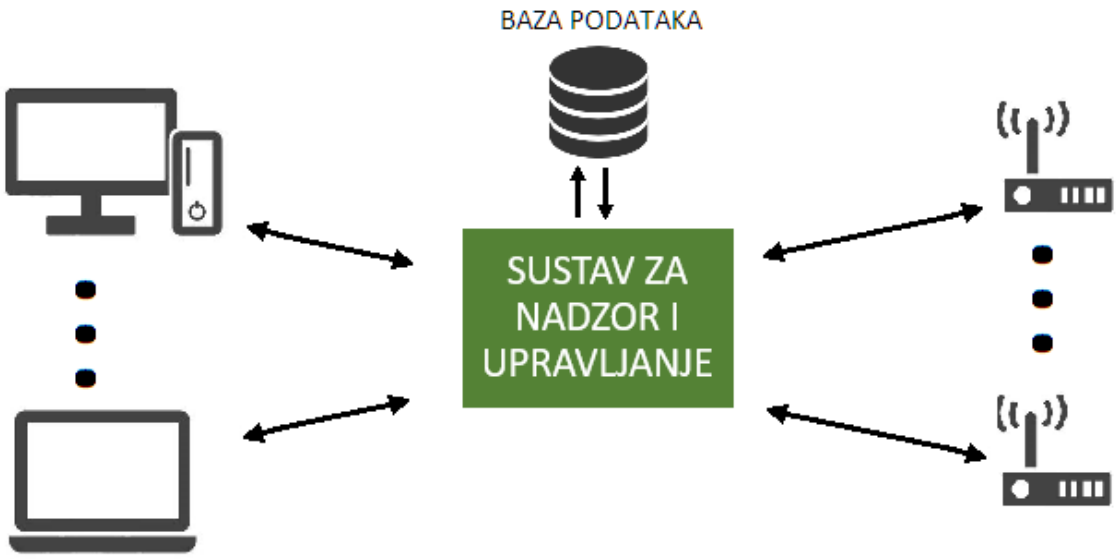
Cayenne (http://www.cayenne-mydevices.com) korisniku omogućava prikaz izmjerenih vrijednosti sa senzora spojenih na uređaj kao i kontrolu njegovih izlaza. U aplikaciji korisnik može birati između više vrsta senzora i aktuatora te vidjeti detaljne sheme njihovog spajanja na uređaj. Osim nadzora i kontrole, aplikacija nudi mogućnost postavljanja okidača, događaja i alarma koji ovise o nadziranim vrijednostima kao i mogućnost zadavanja planiranih događaja. Prednost ovog sustava je njegova mogućnost vizualizacije podataka i upravljanja aktuatorima na uređajima. Na službenoj stranici mogu se naći jednostavni vodiči za spajanje i konfiguraciju uređaja. Trenutno je *Cayenne* Android aplikacija, a najavljena je i iOS verzija. Na popisu podržanih uređaja nalaze se Raspberry Pi (modeli A i B prve generacije te Rasperry Zero) i Arduino (nekoliko modela). Osim uređaja, na stranici se može naći i popis podržanih senzora i aktuatora koji u ovom trenutku i nije toliko impresivan.

4. SUSTAV ZA NADZOR I UPRAVLJANJE ČVOROVIMA

Predloženi sustav se sastoji od web aplikacije i udaljenih uređaja koji mjere određene vrijednosti i čijim se izlazima može upravljati (sl. 4.1). Web aplikacija korisniku omogućava:

- registraciju i autentikaciju – na početnoj se stranici korisnik može registrirati te s registriranim korisničkim podacima pristupiti dijelu aplikacije putem kojega se obavlja pregled izmjerenih podataka i upravljanje,
- dodavanje uređaja – kako bi sustav mogao komunicirati s udaljenim uređajem, korisnik dodaje uređaj u aplikaciju, pri čemu se generira jedinstvena identifikacijska oznaka koja se koristi za komunikaciju između sustava i uređaja,
- dodavanje senzora i aktuatora – na svoje uređaje korisnik može dodavati senzore koji će provoditi mjerenje i aktuatore kojima će upravljati,
- pregled izmjerenih vrijednosti i stanja aktuatora – izmjereni podaci i prethodna stanja aktuatora grafički su prikazani u aplikaciji te
- kontroliranje aktuatora – korisnik može slati jednostavne uključi/isključi naredbe.

Program na udaljenom uređaju provodit će autentikaciju korisnika, slati mjerene podatke svakih N vremenskih trenutaka i primati naredbe korisnika.



Sl. 4.1. Fizička shema predloženog rješenja.

4.1. Pregled korištenih tehnologija

Prije osvrta na pojedine dijelove sustava dan je pregled značajnih tehnologija koje su korištene za izradu predloženog rješenja.

4.1.1. ADO.NET

ADO.NET je *Microsoftova* tehnologija za pristup podacima. Omogućava komunikaciju između relacijskih i nerelacijskih sustava. Koristi se za pristup i mijenjanje podataka pohranjenih u relacijske sustave podataka. Dvije bitne komponente ADO.NET-a su *Entity* okvir (engl. *Entity Framework*) i LINQ *to* SQL. *Entity* je okvir za objektno-relacijsko mapiranje (*engl. object-relational mapping*) koji mapira tablice i stupce baze podataka u objekte i svojstva objekata. Takva apstrakcija omogućila je razvojnim inženjerima rad s podacima bez pretjerane potrebe poznavanja baze u kojima su podaci pohranjeni. LINQ *to* SQL omogućava upite prema *Microsoft* SQL *Server* bazama podataka na način da se C# LINQ upiti pretvaraju u SQL naredbe.

4.1.2. *WebSocket*

WebSocket protokol definiran u [2] omogućuje dvosmjernu komunikaciju između klijenta i udaljenog poslužitelja. Prije postojanja *WebSocket* protokola bilo je potrebno slati višestruke HTTP zahtjeve što je dovodilo do raznih problema:

- poslužitelj je bio prisiljen koristiti nekoliko TCP veza za svakog klijenta: jednu za slanje podataka i novu vezu za svaku pristiglu poruku
- sa svakim HTTP zahtjevom s klijenta na poslužitelj moralo se slati zaglavlje koje je povećavalo količinu poslanih podataka,
- klijentska je skripta morala mapirati izlazne i ulazne veze kako bi pratila poruke.

Jednostavnije je rješenje koristiti jednu TCP vezu za promet u oba smjera, što *WebSocket* protokol i čini. Protokol se sastoji od dva dijela: rukovanja (engl. *handshake*) i prijenosa podataka. Rukovanje sliči HTTP zahtjevu kako bi poslužitelji mogli baratati *WebSocket* vezama na istom portu na kojem barataju i HTTP vezama. Nakon što se veza uspostavi, tj. nakon što klijent dobije odgovor s poslužitelja, podaci se prenose binarnim protokolom koji se ne pridržava konvencija HTTP protokola. Podaci se prenose u konceptualnim jedinicama, tzv. porukama, koje se mogu sastojati od više okvira. Takav pristup omogućava slanje poruka koje nemaju definiranu dužinu. Poruke se šalju okvir po okvir sve do posljednjeg označenog FIN bitom.

4.1.3. ASP.NET MVC

ASP.NET MVC je okvir za web aplikacije (engl. *web application framework*) koji implementira *Model-View-Controller* arhitekturalni uzorak. Zadaća je upravljača (engl. *controller*) primati korisnikove naredbe sa sučelja (engl. *view*) te ih prosljeđivati modelu (sloju poslovne logike) koji obrađuje podatke, vraća ih upravljaču koji ih zatim prikazuje na sučelju. Više informacija može se naći na [3].

4.1.4. Knockout.js

Knockout.js je *JavaScript* okvir za izradu *single-page* aplikacija. Kod ASP.NET MVC aplikacija, na zahtjev klijenta šalje se HTML dokument na kojemu su prikazani podaci. Na svaki dodatni zahtjev šalje se novi HTML dokument uz pripadajuće skripte, stilske dokumente, slike i ostale resurse potrebne za prikaz. *Single-page* aplikacije serviraju HTML i potrebne resurse pri prvom učitavanju stranice, a prilikom navigacije na stranici podaci se šalju u JSON obliku što bitno utječe na brzinu i performanse aplikacije. Knockout implementira MVVM (engl. *Model-View-ViewModel*) arhitekturalni obrazac. Kao i kod MVC-a, funkcija je sučelja omogućiti korisniku interakciju s aplikacijom. Model su podaci koje dohvaćamo *Ajax* pozivima. U kontekstu ove aplikacije, funkciju modela obavlja pozadinski sloj na poslužitelju. Model sučelja (engl. *ViewModel*) je veza između sučelja i modela, i zadaća mu je rukovanje logikom korisničkog sučelja.

Prema [4] najbitnije odlike Knockouta su:

- praćenje ovisnosti zasnovano na MVVM modelu – promjenom podataka automatski se ažurira pripadajući HTML,
- dvosmjerno deklarativno vezanje – jednostavan način povezivanja DOM elemenata s modelom sučelja,
- jednostavan i proširiv – okvir nudi određeni broj deklarativnih vezanja opće namjene (primjerice tekstualno vezanje, vezanje događaja na HTML elemente itd.), ali po potrebi korisnik može vrlo lako napraviti vlastita vezanja,
- neovisnost – osim jQuerya za neke napredne odlike, Knockout nije ovisan ni o jednoj drugoj vanjskoj biblioteci,
- čisti *JavaScript* – biblioteka koristi isključivo *JavaScript* koji radi sa svim klijentskim i poslužiteljskim tehnologijama i omogućuje kompatibilnost s većinom glavnih internetskih preglednika te
- mala veličina – biblioteka zauzima samo 46KB što omogućava brzo preuzimanje čak i na sporijim vezama.

S obzirom na to da se sadržaj prikazuje unutar jednog HTML dokumenta, ne postoje rute kojima se može pristupiti određenim sučeljima u aplikaciji. Taj problem riješen je *Sammy.js* bibliotekom koja korisniku daje privid ruta i omogućava dohvat podataka potrebnih za određeno sučelje.

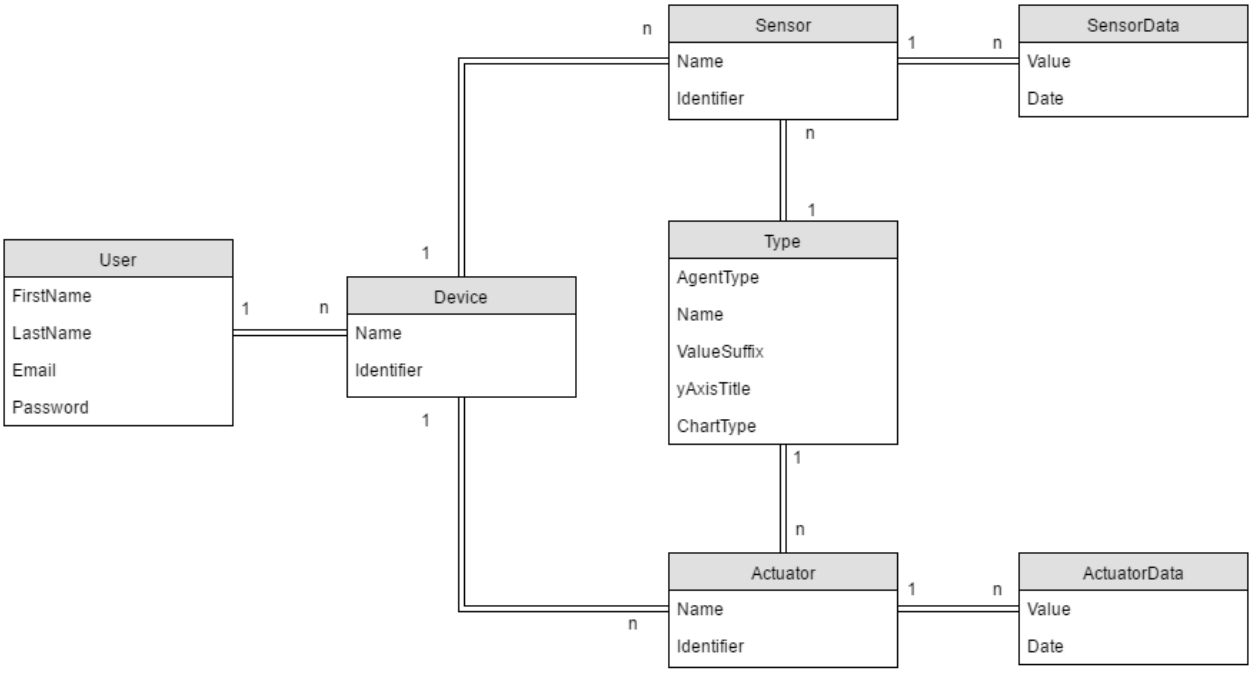
4.1.5. Python

Python je objektno orijentirani, interpreterski jezik visoke razine koji je 1990. godine razvio Guido van Rossum. Njegove značajke su:

- interpretacija međukoda – kod napisan u Pythonu kompilira se u prijenosni *bytecode* koji omogućava izvođenje na bilo kojoj platformi
- visoka razina – uz standardne tipove podataka (brojevi, znakovi, nizovi itd.) ima ugrađene tipove podataka visoke razine (liste, rječnici itd.)
- čista sintaksa – formatiranjem koda zamjenjuju se znakovi za definiranje blokova koda, pa je napisani program pregledan i jednostavan
- napredne značajke – nudi sve značajke koje se očekuju u modernom programskom jeziku: objektno orijentirano programiranje, dohvaćanje izuzetaka, redefiniranje standardnih operatora, pretpostavljene argumente itd.
- proširivost – pisan u modularnoj C arhitekturi, omogućuje lako proširivanje novim API-ima
- velika količina biblioteka – uz standardne koje dolaze uz instalaciju, moguće je preuzeti i dodatne biblioteke.

4.2. Baza podataka

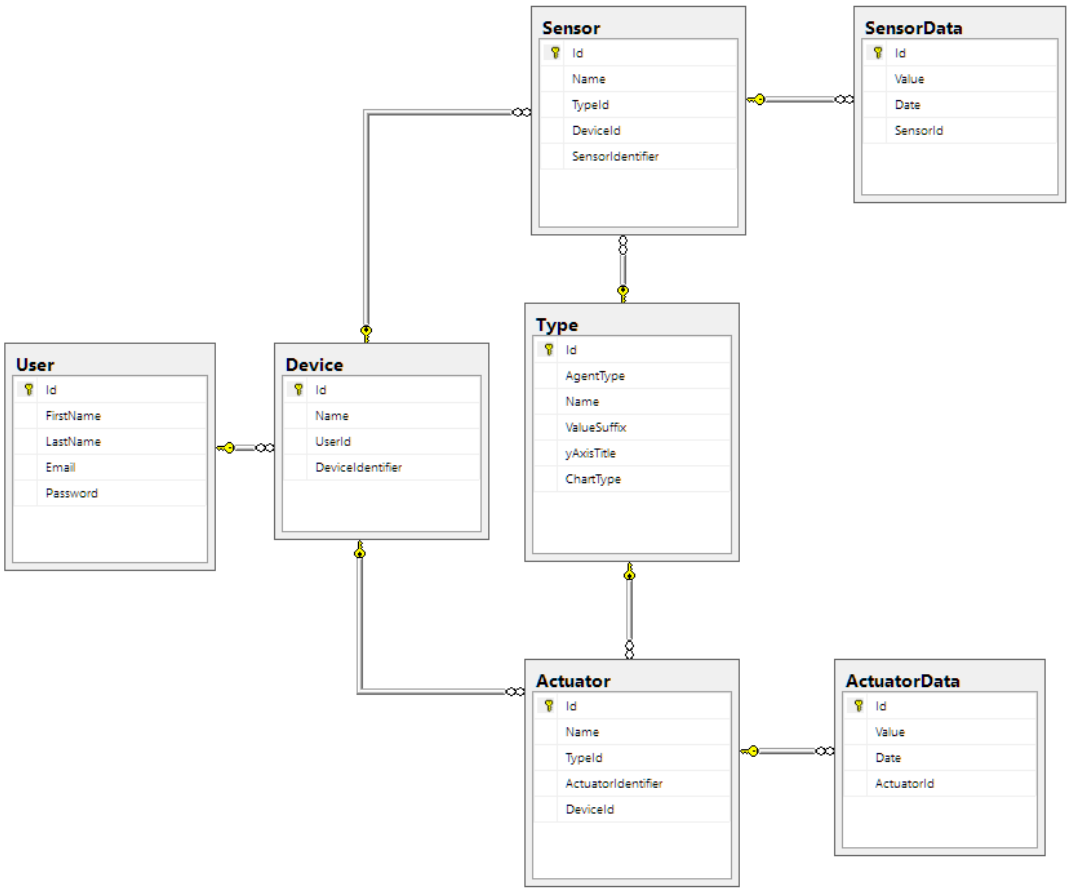
S obzirom na zahtjeve navedene na početku ovog poglavlja, baza podataka sustava ima tablice za pohranu podataka o: korisniku, korisnikovim uređajima, senzorima i aktuatorima koji pripadaju pojedinom uređaju, izmjerenim podacima senzora i prethodnim stanjima aktuatora. Uz navedene tablice, u bazi se nalazi i tablica u kojoj su pohranjeni podaci o nekoliko vrsta senzora i aktuatora. Podatke u posljednjoj tablici korisnici ne mogu mijenjati. Ona služi za popunjavanje padajućeg izbornika za izbor vrste senzora ili aktuatora prilikom dodavanja novog senzora ili aktuatora te za podešavanje izgleda prilikom grafičkog prikaza izmjerenih podataka. ER dijagram konceptualnog modela baze prikazan je na slici 4.2.



Sl. 4.2. ER dijagram konceptualnog modela.

Entitet „Korisnik“ (engl. *User*) može imati nekoliko „Uređaja“ (engl. *Device*). Svaki „Uređaj“ može imati više „Senzora“ (engl. *Sensor*) i „Aktuatora“ (engl. *Actuator*). „Senzor“ i „Aktuator“ povezani su s entitetom „Tip“ (engl. *Type*). Također svaki od njih može imati više entiteta s podacima (engl. *SensorData* i *ActuatorData*). U tablici „Korisnik“ spremat će se podaci potrebni za autentikaciju korisnika koja će se provoditi pomoću e-adrese i lozinke. „Uređaj“ osim imena ima i identifikaciju oznaku (engl. *identifier*) koja će služiti za raspoznavanje prilikom komunikacije aplikacije s uređajem i obratno. „Senzor“ uz ime ima i identifikacijsku oznaku koja će se koristiti prilikom slanja podataka s uređaja kako bi se podaci pohranili uz odgovarajući senzor. „Senzor“ ima i tip u kojem će biti pohranjeni podaci potrebni za iscrtavanje grafa za prikaz podataka. Korisnik neće imati mogućnost izmjene ili dodavanja podataka u tablicu tipova. „Podaci senzora“ imaju vrijednost i vrijeme kada su izmjereni. Identična je struktura i za „Aktuator“.

Nakon provedene analize izrađena je odogovarajuća baza podataka. Za izradu je korišten Transact-SQL. Dijagram baze prikazan je na slici 4.3.



Sl. 4.3. Dijagram baze.

U odnosu na konceptualni model postoji nekoliko sitnih promjena. Svaka tablica ima primarni ključ koji jedinstveno označava svaki zapis u tablici. Strani ključ *UserId* povezuje tablice uređaja i korisnika. *DeviceId* povezuje senzor, odnosno aktuator, s uređajem, a *TypeId* ih povezuje s tipom. *SensorId* i *ActuatorId* povezuju podatke s pripadajućim senzorom, tj. aktuatorom.

4.3. Web aplikacija

Web aplikacija ima slojeviti arhitekturalni obrazac. Aplikacije koje implementiraju takav obrazac sastoje se od 4 sloja (sl. 4.4.). Prvi sloj je izvor podataka (engl. *data source*), u slučaju ove aplikacije, taj izvor je baza podataka. Iznad njega se nalazi sloj za pristup podacima (engl. *data access layer*) unutar kojega se nalazi kod koji pristupa bazi, dohvaća, briše i unosi podatke. Zadaća sloja poslovne logike (engl. *business logic layer*) jest priprema podataka za prikaz na prezentacijskom sloju (engl. *presentation layer*). Uz to, u njemu se nalazi *WebSocket* server i funkcije za komunikaciju s udaljenim uređajem. Prezentacijski sloj je sučelje prema korisniku preko kojega se prikazuju podaci i na kojemu se nalaze kontrole.



Sl. 4.4. Slojeviti arhitekturalni obrazac.

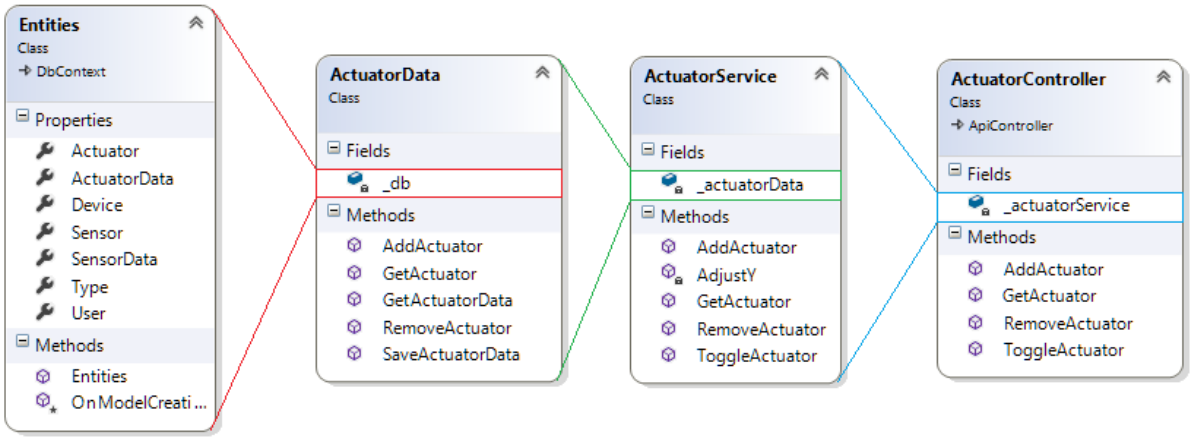
Komponente jednog sloja mogu komunicirati samo s komponentama unutar istog sloja ili s komponentama u sloju neposredno ispod (npr. komponente u prezentacijskom sloju nemaju mogućnost komunikacije s komponentama sloja za pristup podacima). Prema [5] glavne prednosti takvog pristupa su:

- apstrakcija – slojevi dopuštaju izmjene na apstraktnoj razini, a razina apstrakcije može se povećavati ili smanjivati ovisno o sloju,
- izolacija – omogućuje izoliranje tehnoloških nadogradnji po pojedinom sloju što smanjuje rizik i minimizira utjecaj na cjelokupni sustav. Primjerice, za pristup i rad s bazom korišten je *Entity* okvir, no primjenom ovog obrasca vrlo lako bi se mogao zamijeniti nekom drugom tehnologijom (npr. NPoco) uz minimalan utjecaj na sloj poslovne logike i cjelokupnu aplikaciju,
- upravljivost – odvajanje pomaže u identificiranju ovisnosti i organizira kod u više upravljivih dijelova,
- performanse – distribucija na više slojeva može poboljšati skalabilnost, toleranciju na kvarove i performanse,
- ponovno korištenje – omogućuje ponovno korištenje koda. Ova je aplikacija izrađena kao web aplikacija, ali prezentacijski sloj bi se lako mogao zamijeniti sučeljem za desktop aplikaciju te
- ispitljivost – testiranje ovakve aplikacije lakše je jer svaki sloj ima jasno definirano sučelje, kao i mogućnost lake izmjene implementacija pojedinog sloja.

Uz navedene slojeve u samom rješenju nalazi se još jedan odvojen projekt koji sadržava objekte za prijenos podataka (engl. *data transfer object*, u nastavku DTO). DTO-i se koriste za mapiranje objekata iz baze podataka u oblik i strukturu u kojima se prikazuju korisniku.

4.3.1. Sloj za pristup podacima

Na slici 4.5. prikazan je dijagram klasa koje čine dio *backenda* aplikacije. Konkretno na slici su prikazane sve metode za dohvat i manipulaciju podataka o aktuatorima. U sloju za pristup podacima nalaze se `Entities` i `ActuatorData` klase. `Entities` klasa generirana je *Entity* okvirom i njezina svojstva su objekti koji predstavljaju tablice u bazi podataka. `ActuatorData` klasa ima jedno privatno polje koje je zapravo instanca `Entities` klase. `ActuatorService` klasa dio je sloja poslovne logike, a njezino privatno polje je `ActuatorData` objekt. Na poslijetku se nalazi `ActuatorController` klasa koja nasljeđuje .NET-ovu ApiController klasu i ona čini dio prezentacijskog sloj aplikacije. Klase viših slojeva detaljnije će biti opisane kasnije u radu.



Sl. 4.5. Dijagram klasa.

Kako je spomenuto sloj za pristup podacima se sastoji od modela baze podataka i klasa s metodama koje obavljaju akcije nad podacima u bazi. U prilogu 1 je primjer `Actuator` klase koja je generirana iz *Actuator* tablice. Može se primijetiti da su stupci tablice mapirani u svojstva klase. U nastavku priloga je `ActuatorData` klasa s metodama za upite prema bazi. Prva je metoda `AddActuator` koja prima jedan parametar – objekt tipa `Actuator` koji sadrži podatke potrebne za stvaranje novog aktuatora na uređaju – *Id* uređaja kojemu pripada, ime, identifikacijsku oznaku (generirana u gornjem sloju) i *Id* tipa aktuatora. U metodi se vidi spomenuta odlika ADO.NET-a, a to je LINQ *to* SQL. Nad objektima baze možemo provoditi LINQ operacije koje se potom interpretiraju u T-SQL naredbe. Tako da će se kod u metodi `AddActuator` pretvoriti u sljedeću SQL naredbu:

```
INSERT INTO Actuator
```

```
VALUES (@DeviceId, @Name, @Identifier, @TypeId);
```

Uz navedenu metodu, u klasi se još nalaze metode:

- `GetActuator` – metoda za dohvaćanje aktuatora; prima *Id* aktuatora i vraća željeni aktuator,
- `GetActuatorData` – metoda koja dohvaća podatke za određeni aktuator u specificiranom vremenskom roku; prima *Id* aktuatora, početno i krajnje vrijeme i vraća listu vrijednosti,
- `RemoveActuator` - metoda za uklanjanje aktuatora, prima kao parametar *Id* aktuatora i vraća boolean vrijednost koja označava uspješnost operacije te
- `SaveActuatorData` – metoda za spremanje trenutnog stanja aktuatora, prima podatke i vraća uspješnost obavljenje akcije.

Slične klase napravljene su i za ostale entitete u bazi. Implementacija klasa sloja za pristup podacima dana je u sljedećoj točki koja detaljnije opisuje sloj poslovne logike.

4.3.2. Sloj poslovne logike

Pregled sloja poslovne logike započet ćemo analizom `DeviceService` klase. Kako je navedeno ranije, zadaća je komponenti u ovom sloju prilagodba podataka za prikaz, ali isto tako i prosljeđivanje metoda iz sloja za pristup podacima prezentacijskom sloju, koji inače ne bi imao pristup tim metodama (unos novih podataka, brisanje itd.). U prilogu 3 vidi se da se na početku `DeviceService` klase instancira objekt `DeviceData` klase. Prva metoda je `GetInitialData` koja prima korisnikovu e-adresu i vraća pripadajuće uređaje sa senzorima i aktuatorima koji se nalaze na uređaju, kao i posljednju izmjerenu vrijednost senzora i trenutno stanje aktuatora, ako uređaj ima senzore i/ili aktuatore. Unutar `Select` naredbe svojstva `Device` objekta mapiraju se u `DeviceDto` objekt. Ovdje se može vidjeti primjer prilagodbe podataka. `IndexSensorDto` ima svojstvo `LastDataEntry` koje se ne nalazi na `Sensor` objektu. Nadalje, prilikom mapiranja tog svojstva poziva se privatna funkcija `ConvertSensorValue` koja uzima zadnju vrijednost senzora i formatira ga u *string* oblika vrijednost + sufiks. Sufiks će ovisiti o odabranom tipu senzora - tako će primjerice zadnji zapis temperaturnog senzora izgledati „21°C“. Ako nema podataka na senzoru ispisat će se *„NO DATA“*. Slična se metoda poziva i za aktuatore gdje se umjesto boolean vrijednosti vraća *„ON“*, *„OFF“* ili *„NO DATA“*. Nakon pretvorbe uređaji se sortiraju silazno od uređaja s najvećim brojem senzora i aktuatora.

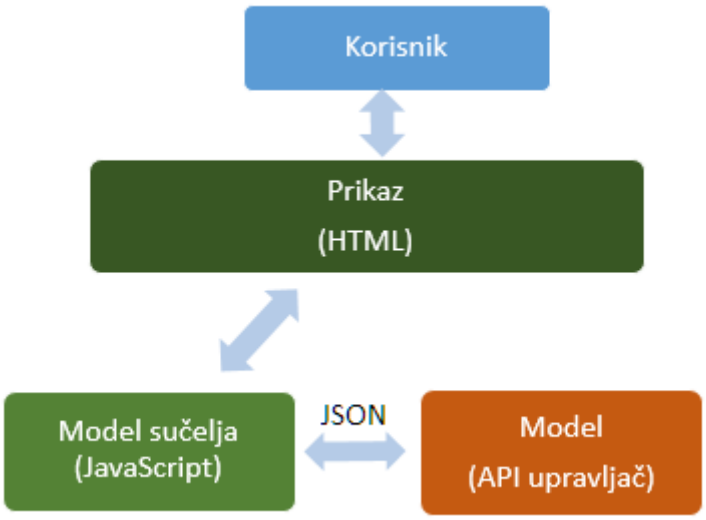
Sljedeća je metoda `AddDevice` koja prima ime uređaja i e-adresu korisnika koji ga je dodao i vraća podatke o stvorenom uređaju koji se na odgovarajući način koriste u prezentacijskom sloju aplikacije. U metodi se instancira novi objekt `Device` klase koji se potom, uz e-adresu, predaje metodi iz sloja za pristup podacima koja zatim pretražuje korisnike i uređaju dodjeljuje korisnikov

Id te ga potom sprema u bazu. Alternativno, dohvat korisnikovog *Id*-a mogao se obaviti u `DeviceService` prije spremanja novog uređaja pozivom metode koja bi primala e-adresu i vraćala *Id* tako da bi `AddDevice` metoda u `DeviceData` primala kao parametar samo objekt s podacima o novom uređaju. Na kraju klase nalazi se `RemoveDevice` metoda unutar koje se samo poziva odgovarajuća metoda iz donjeg sloja.

Unutar `ActuatorService` klase nalazi se implementacija komunikacije između sustava i udaljenog čvora preko *WebSocketa*. Za izradu servera korišten je *Alchemy-Websockets* čiji se kod i dokumentacija mogu naći na [6]. `ToggleActuator` metoda iz priloga 4 se poziva kada korisnik želi uključiti ili isključiti aktuator. Metoda kao parametar prima objekt tipa `ToggleActuatorDto` koji ima 4 svojstva: identifikacijsku oznaku uređaja kojem će se poslati poruka, željeno stanje, *Id* aktuatora i identifikacijsku oznaku aktuatora. Na početku se otvara *WebSocket* veza na koju je spojen i uređaj. Kako bi se osigurao prijenos podataka isključivo prema jednom uređaju koristi se njegova identifikacijska oznaka. Zatim se postavljaju rukovatelji događajima. Po primitku povratne poruke s uređaja veza će se zatvoriti. Ako se dogodi pogreška, veza se zatvara i korisnika se obavještava da se dogodila pogreška prilikom povezivanja s uređajem. Nakon toga se objekt mapira u JSON s dva svojstva – identifikatorom aktuatora i stanjem. JSON se potom pretvara u *string* i šalje uređaju (o obradi poslane poruke više u dijelu o klijentskoj aplikaciji na uređaju). Metoda zatim čeka odgovor uređaja ili, ako je vrijeme čekanja duže od 5 sekundi, zatvara vezu i obavještava korisnika o grešci. Ako odgovor stigne, stanje se sprema u bazu i vraća se podatak o novom stanju.

4.3.3. Prezentacijski sloj

ASP.NET MVC korišten je za prikaz početne stranice, registracije korisnika i prijavu korisnika u aplikaciju. Sama stranica aplikacije servira se kao prazan prikaz (engl. *view*), a sva funkcionalnost napravljena je *Knockoutom* i ASP.NET Web API 2 upravljačima. Kako je prikazano na slici 4.6. korisnikovom interakcijom sa sučeljem aplikacije tj. prikazom, pozivaju se odgovarajuće metode u modelu sučelja. Model sučelja zatim provodi interakciju sa podacima u modelu te obavlja potrebne promjene na sučelju.



Sl. 4.6. Blokovski dijagram prezentacijskog sloja.

Za razliku od MVC upravljača koji serviraju čitave HTML dokumente popunjene podacima, API upravljači vraćaju samo podatke u JSON obliku. Primjer metode API upravljača dan je u tablici 4.1. Metoda u primjeru poziva ranije spomenutu metodu za dohvat korisnikovih uređaja - `GetInitialData`.

Tab. 4.1. Metoda `GetUserDevices`.

IoTpanel/IoTpanel.Web/DeviceController.cs

```
[HttpGet]
[Route("api/device")]
public IHttpActionResult GetUserDevices()
{
    var mail = User.Identity.Name;

   try
    {
        var devices = _deviceService.GetInitialData(mail);
        return Ok(devices);
    }
    catch
    {
        return BadRequest();
    }
}
```

Prije same metode `HttpGet` atributom definirano je da će metoda odgovarati na HTTP zahtjev tipa GET. Sljedeći atribut definira rutu kojom će se moći pozvati metoda. U slučaju ispravnog dohvata podataka, uz zatražene podatke u poruci se šalje i statusni kod 200 koji označava uspješno obavljen zahtjev. Ako se prilikom izvođenja koda dogodi iznimka, odgovor će biti poslan s jednim od statusa pogreške (primjerice 500 – unutrašnja pogreška poslužitelja). Takvim se odgovorima zatim može lako rukovati unutar funkcije povratnog poziva greške *Ajax* zahtjeva koji je pozvao

metodu na poslužitelju. Ova metoda je izdvojena i iz razloga što ilustrira korištenje autorizacije u aplikaciji. Autorizacija i autentikacija je napravljena koristeći ASP.NET obrasce za autentikaciju (engl. *forms authentication*). U metodi za prijavu provjeravaju se unesena e-adresa i lozinka. Ako se korisnik uspješno prijavio, postavlja se autentikacijski *cookie* i u *User* objekt se spremaju podaci o prijavljenom korisniku kojima se zatim može lako pristupiti u ostalim metodama. Pristup upravljačima ili metodama unutar upravljača zatim se lako može ograničiti samo prijavljenim korisnicima upotrebom `Authorize` atributa.

Unutar modela sučelja nalaze se osmotrive (engl. *observable*) varijable koje će sadržavati tražene podatke i funkcije za pozivanje API metoda. Koncept osmotrivih varijabli pobliže je objašnjen na primjeru dodavanja novog uređaja. Unutar modela sučelja definiran je osmotrivi niz `devices` koji sadrži korisnikove uređaje. U HTML-u sučelja prolazi se kroz svaki objekt u nizu i provodi vezanje njegovih svojstava na sučelje, kako je prikazano u pojednostavljenom kodu u tablici 4.2.

Tab 4.2. Primjer *foreach* vezanja.

IoTpanel/IoTpanel.Web/App_Code/DeviceHelper.cshtml

```
<div data-bind="foreach: $root.devices">
       <h3 data-bind="text: name"></h3>
       <!-- vezanje pripadajućih senzora, aktuatora, itd.  -->
<div>
```

Dugme na sučelju povezano je klik događajem s funkcijom modela sučelja koja poziva API metodu za dodavanje novog uređaja.

Tab 4.3. Primjer klik vezanja.

IoTpanel/IoTpanel.Web/App_Code/SensorHelper.cshtml

```
<button type="button" data-bind="click: AddSensor">Add sensor</button>
```

Funkcija modela sučelja koja se poziva klikom prikazana je u tablici 4.4.

Tab 4.4. Funkcija za dodavanje uređaja.

IoTpanel/IoTpanel.Web/Scripts/app/main.js

```
self.AddDevice = function () {
            $.ajax({
                url: baseUrl + "/api/device",
                data: {
                      "name" : self.newDeviceName()
                 },
                type: "POST",
```

```
                success: function (response) {
                    self.newDevice(new DeviceObservable(response));
                    self.devices.push(self.newDevice());
                    $("#addDeviceModal").modal("hide");
                    toastr.success("Device successfully added", "Success")
                    self.newDeviceName("");
                },
                error: function () {
                    $("#addDeviceModal").modal("hide");
                    toastr.error("An error occured", "Error");
                    self.newDeviceName("");
                }
            });

        }
```

Unutar funkcije provodi se *Ajax* poziv kojemu predajemo *object literal* s parametrima:

- `url` – URL API metode koju pozivamo,
- `data` – podaci koje šaljemo HTTP zahtjevom,
- `type` – vrsta HTTP zahtjeva,
- `success` – povratna funkcija uspješnog zahtjeva te
- `error` – povratna funkcija neuspješnog zahtjeva.

Može se primijetiti da je ruta URL zahtjeva za stvaranje novog uređaja identična ruti za dohvat svih uređaja (api/device). Razlog tomu je konvencija pisanja REST (engl. *Representational State Transfer*) web servisa. API upravljač razlikuje te dvije rute po vrsti zahtjeva koji dolazi s klijenta. U prvom je slučaju riječ o GET zahtjevu, a za dodavanje se šalje zahtjev tipa POST. Metode za dohvaćanje, ažuriranje i brisanje jednog uređaja imale bi rutu „api/device/{id}“ koje bi se razlikovale po vrsti zahtjeva (GET, POST, DELETE). U zahtjevu se šalje samo ime novog uređaja jer se podaci o korisniku dohvaćaju u pozadinskom kodu, kako je prikazano u metodi za dohvat korisnikovih uređaja. U povratnoj funkciji uspješnog zahtjeva instancira se u objekt s podacima o dodanom uređaju (ime i identifikacijska oznaka dodanog uređaja). Objekt se zatim dodaje u gore spomenuti osmotrivi niz i korisnika se obavještava o uspješnosti obavljene akcije. *Knockout* detektira promjene na osmotrivoj varijabli i automatski ažurira sučelje. Ako se prilikom provođenja pozadinskih funkcija dogodi greška, korisnik se obavještava povratnom funkcijom neuspješnog zahtjeva.

Zadnja izmjerena vrijednost senzora i trenutno stanje aktuatora prikazani su na pregledu korisnikovih uređaja. Za prikaz svih izmjerenih vrijednosti senzora i stanja aktuatora tijekom nekog perioda korištene su još dvije *JavaScript* biblioteke – *HighCharts.js* i *Bootstrap-DateTimePicker.js*.

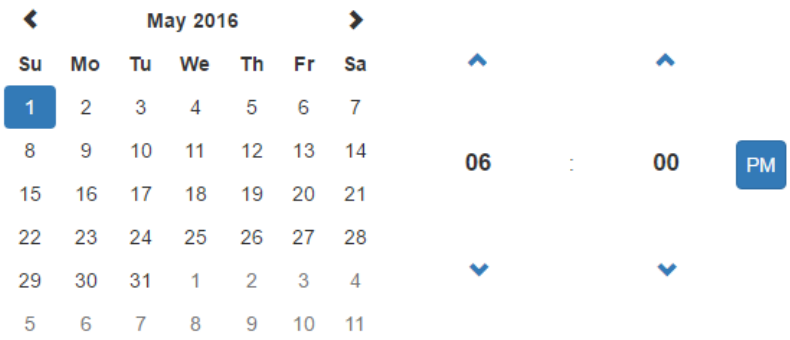
HighCharts.js biblioteka omogućuje interaktivni grafički prikaz skupova podataka. Grafove se može vrlo lako prilagoditi kako bi zadovoljili korisnikove potrebe. U kodu ispod prikazan je dio funkcije koja inicijalizira graf za prikaz podataka senzora. Funkcija prima objekt s postavkama potrebnim za inicijalizaciju (skupovi podataka, nazivi osi, vrsta grafa itd.). Podaci se koriste u *object literalu* koji se predaje `Chart` metodi koja konfigurira graf (tab. 4.5.). Konfiguracijske postavke ovise o vrsti senzora.

Tab 4.5. Inicijalizacija i konfiguracija *Highcharts* grafa.

IoTpanel/IoTpanel.Web/Scripts/app/main.js

```
function DrawSensorGraph(graphData) {
        sensorChart = new Highcharts.Chart({
            chart: {
                renderTo: 'chart',
                type: graphData.ChartType
            },
            title: {
                text: graphData.Name,
                x: -20
            },
            xAxis: {
                categories: graphData.xAxis,
                labels: {
                    enabled: false
                }
            },//...
```

Inicijalno se na pregledu prikazuje posljednjih 20 izmjerenih vrijednosti, stoga je korisniku omogućen izbor vremena za koje želi vidjeti podatke. Zbog velikog broja formata zapisa vremena implementirana je kontrola kojom se osigurava jedinstven način izbora. Na slici 4.7. prikazan je *Bootstap – DateTimePicker* kalendar i sučelje za odabir sata.



Sl. 4.7. *DateTimePicker* vremenski izbornik.

Kako je ranije spomenuto, Knockout omogućuje deklarativna vezanja za standardne HTML elemente, ali s obzirom na to da je korištena vanjska biblioteka, potrebno je napraviti vlastito vezanje prikazano u tablici 4.6.

Tab. 4.6. *DateTimePicker* rukovatelj vezanjem (engl. *binding handler*).

IoTpanel/IoTpanel.Web/Scripts/app/main.js

```
ko.bindingHandlers.dateTimePicker = {
        init: function (element, valueAccessor, allBindingsAccessor) {
            var options = allBindingsAccessor().dateTimePickerOptions || {};
            $(element).datetimepicker(options);

            ko.utils.registerEventHandler(element, "dp.change", function (event) {
                var value = valueAccessor();
                if (ko.isObservable(value)) {
                    if (event.date != null && !(event.date instanceof Date)) {
                        value(event.date.toDate());
                    } else {
                        value(event.date);
                    }
                }
            });

            ko.utils.domNodeDisposal.addDisposeCallback(element, function () {
                var picker = $(element).data("DateTimePicker");
                if (picker) {
                    picker.destroy();
                }
            });
        }
    };
```

Vezanje je prvo potrebno registrirati kako bi se moglo koristiti unutar HTML `data-bind` atributa. Funkcija koja registrira vezanje ima dvije povratne funkcije `init` i `update`. Povratna funkcija `update` poziva se ako vrijednost osmotrive varijable na koju je element vezan ovisi, primjerice, o funkciji koja se obavlja interakcijom s nekim drugim elementom sučelja ili o nekoj drugoj osmotrivoj varijabli. S obzirom na to da za primjenu aplikacije to nije potrebno, implementirana je samo `init` funkcija. Ona prima tri parametra:

- `element` – DOM element na kojem se provodi vezanje
- `valueAccessor` – *JavaScript* funkcija kojom se dohvaća svojstvo modela sučelja koje je veže na element; pozivom ove funkcije dobiva se trenutna vrijednost svojstva
- `allBindingsAccessor` – objekt kojim se može pristupiti ostalim svojstvima modela sučelja koji su vezani za isti element.

U funkciji se na elementu inicijalizira kontrola za odabir vremena kojoj se preko `allBindingsAccessora` može proslijediti *object literal* za konfiguraciju. U rukovatelju događajima (engl. *event handler*) promjenom vrijednosti na kalendaru novoizabrana vrijednost dodjeljuje se osmotrivoj varijabli vezanoj za element. S obzirom na to da se u *Knockout* aplikacijama DOM elementi dinamički dodaju i uklanjaju, na kraju je unutar

`domNodeDisposal.addDisposeCallback` funkcije potrebno ukloniti element kako bi se prilikom navigacije na neko drugo sučelje „počistile“ sve varijable koje su bile korištene u vezanju.

4.4. Program na udaljenom čvoru

Program na udaljenom čvoru koji bi koristio sustav trebao bi se sastojati od:

1. Funkcije koja bi pozivala API metodu za prijavu korisnika. Ruta metode je „api/user/login“ te kao parametre prima objekt sa svojstvima `Username` i `Password`.
2. Funkcije koja bi osluškivala *WebSocket* port definiran identifikatorom uređaja. Poruke koje dolaze tim putem su JSON objekti s `actuatorId` (identifikacijska oznaka aktuatora) i `status` svojstvima (1 označava uključivanje aktuatora, a 0 isključivanje). Ova funkcionalnost je opcionalna.
3. Funkcije koja bi očitavala vrijednosti sa senzora, formatirala ih u niz JSON objekata s `Identifier` (identifikacijska oznaka senzora) i `Value` svojstvima i slala ih POST zahtjevom na „api/sensor/data“. Ova je funkcionalnost također opcionalna.

Primjer i detaljnije objašnjenje dani su u sljedećem poglavlju.

5. DEMONSTRACIJA SUSTAVA

5.1. Program na udaljenom čvoru

Kako bi se ilustriralo korištenje sustava, dan je primjer jednostavnog programa napisanog u Pythonu koji očitava vrijednost temperaturnog senzora i upravlja dvjema svjetlećim diodama spojenim na Raspberry Pi 3.

Raspberry Pi je računalo na jednoj ploči (engl. *single board computer*) koje je proizvela Raspberry Pi zaklada. Prvi model pušten je u prodaju u veljači 2012. i od onda je izašlo još šest modela: Pi 1 model B, modeli A+ i B+, Pi 2 model B, Raspberry Zero i Pi 3 model B. Računala na jednoj ploči imaju sve dijelove kao i standardna stolna računala – mikroprocesor, memoriju, ulazno – izlazne jedinice. Osim uobičajene računalne periferije poput miša, tipkovnice i monitora, na Raspberry je putem ulazno – izlaznih jedinica opće namjene (engl. *general purpose input - output*) moguće spojiti različite senzore i aktuatore. Prednosti Rasberryja u odnosu na standardna računala su:

- cijena – ovisno o modelu, u trenutku pisanja ovog rada cijena novog uređaja kreće se između 20 i 35 dolara; međutim, treba uzeti u obzir i cijenu perifernih uređaja potrebnih za rad čija cijena nije uključena u tom iznosu (izvor napaja, SD kartica, tipkovnica, monitor, HDMI kabel)
- mala potrošnja – uređaju je za rad potrebno napajanje od 5V, i, ovisno o modelu i količini spojene periferije, troši između 0.7 i 2 A
- velika zajednica – uz službenu stranicu, na Internetu se može naći ogromna količina projekata i dokumentacije.

Prednost Raspberryja 3 u odnosu na starije modele (osim povećanih hardverskih specifikacija) jest ugrađen Wi-Fi adapter koji omogućuje spajanje na bežične mreže. Stariji su modeli imali mogućnost spajanja na Internet putem Ethernet porta, a za spajanje na bežične mreže bio je potreban dodatni USB Wi-Fi adapter. Nedostaci Raspberryja su:

- nemogućnost nadogradnje – sklopovlje na ploči teže je zamijeniti nego kod stolnih ili prijenosnih računala
- ograničena računalna moć – unatoč popriličnom napretku, Raspberry po svojim hardverskim specifikacijama zaostaje za standardnim računalima.

Program na uređaju sastoji se od tri dijela – funkcije za autorizaciju korisnika, funkcije za mjerenje i slanje podataka sa senzora i funkcije za upravljanje izlazima. Nakon uspješne autorizacije korisnika, u dvije paralelne niti istovremeno se izvode preostale dvije funkcije. U tablici 5.1.

prikazan je pojednostavljeni kod funkcije koja osluškuje *WebSocket* port i, ovisno o naredbi, uključuje ili isključuje neku od svjetlećih dioda.

Tab. 5.1. Funkcije za upravljanje izlazima i osluškivanje *WebSocket* porta.

Raspberry/actuator.py

```
ws = create_connection("ws://" + serverUrl + "/" + deviceIdentifier)

def SwitchActuatorStatus(pin, status):
    if status == 1:
        GPIO.output(pin, GPIO.HIGH)
    else:
        GPIO.output(pin, GPIO.LOW)

def ToggleActuator(actuatorId, status):
    if actuatorId == firstActuatorId:
        SwitchActuatorStatus(firstActuatorPin, status)
    elif actuatorId == secondActuatorId:
        SwitchActuatorStatus(secondActuatorPin, status)

def ListenToSocket():
    while 1:
        message =  ws.recv()
        if message != "":
            request = json.loads(message, object_hook = as_payload)
            ws.send("1")
            ws.close()
            actuatorId = request.ActuatorIdentifier
            status = request.Status
            ToggleActuator(actuatorId, status)
            message = ""
            ws.connect("ws://" + serverUrl + ":8080/" + deviceIdentifier)
```

Program stvara vezu prema *WebSocket* serveru i u `ListenToSocket` funkciji očekuje poruku. Poruka se po primitku parsira u JSON i vraća se povratna poruka kako bi pozadinski kod na poslužitelju mogao nastaviti s provođenjem. Veza se zatim zatvara i poziva se `ToggleActuator` funkcija kojoj se iz parsirane poruke prosljeđuju identifikacijska oznaka aktuatora i njegovo stanje. `ToggleActuator` prema oznaci nalazi željeni aktuator te poziva `SwitchActuatorStatus` funkciju koja kao parametre prima GPIO konektor na koji je spojen aktuator i njegovo novo stanje. Nakon toga ponovno se otvara veza i čeka se sljedeća poruka.

Očitavanje vrijednosti i njezino spremanje obavlja se periodički u intervalu definiranom varijablom `sendInterval`, kako je prikazano u tablici 5.2.

Tab. 5.2. Pojednostavljeni prikaz funkcija za mjerenje i slanje izmjerenih podataka.

Raspberry/sensor.py

```
def ReadTemperature():
    #očitavanje vrijednosti
```

```
def SendSensorData(temperature, session):
    dataArray = []
    temperatureData = "{'Identifier':'"+sensorId+"','Value':'"+str(temperature)+"'}"
    dataArray.append(temperatureData)
    r = session.post(url, data = {'data': dataArray})
    if r.text != "1":
        error_log = open("error_log.txt", "w")
        error_log.write(r.text)
        error_log.close()
        session = LogIn()
        ReadAndSendSensorData(session)

def ReadAndSendSensorData(session):
    temperature = ReadTemperature()
    SendSensorData(temperature, session)
    threading.Timer(sendInterval, ReadAndSendSensorData, [session]).start()
```

`ReadAndSendData` prvo očitava i vraća vrijednost senzora. Izmjerena vrijednost zatim se sa sjednicom (engl. *session*) predaje `SendSensorData` funkciji. Sjednica sadrži autorizacijske podatke dobivene iz funkcije za prijavu korisnika koji su potrebni kako bi se onemogućio pristup neautoriziranim korisnicima. Izmjerena vrijednost se s identifikacijskom oznakom senzora formatira u *string* JSON oblika i dodaje nizu podataka koji će se poslati API metodi aplikacije. Metoda prima niz *stringova* ako bi korisnik htio istovremeno poslati vrijednosti s nekoliko senzora. Primljeni podaci se parsiraju u C# objekte i pohranjuju u bazu. U slučaju neželjenog odgovora on se zapisuje i program ponovno pokušava pristupiti aplikaciji.

5.2. Web aplikacija

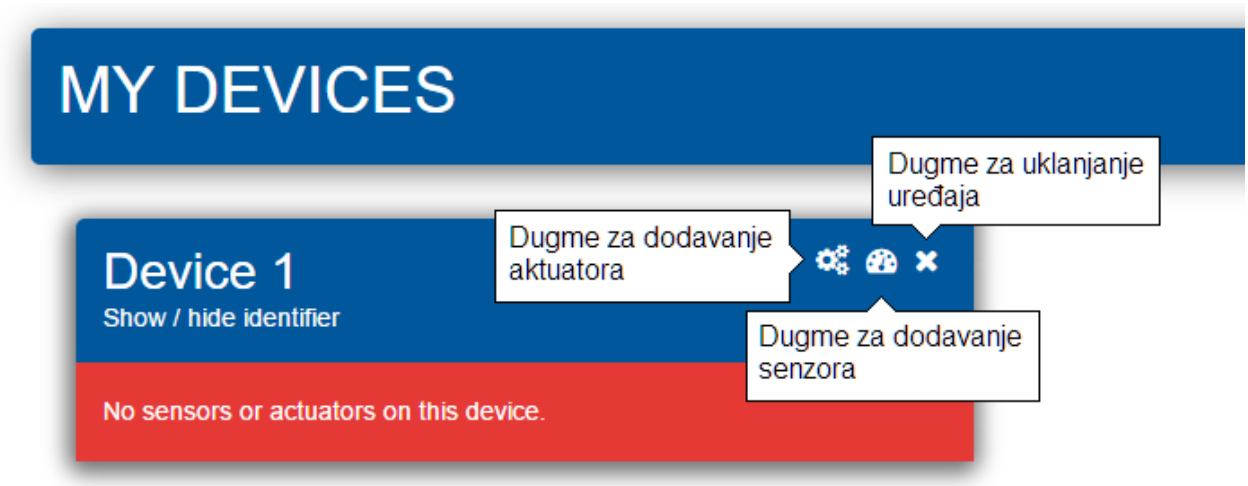
Nakon uspješne registracije i pristupa aplikaciji, korisnika dočekuje sučelje prikazano na slici 5.1.



Sl. 5.1. Početno sučelje aplikacije.

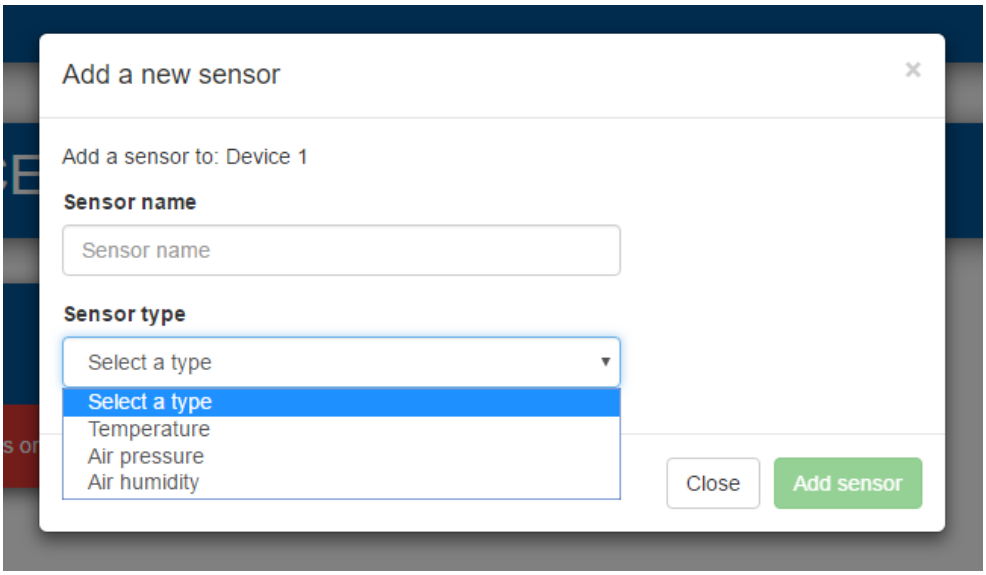
Klikom na dugme „Add new“ prikazuje se jednostavni modalni prozor u kojemu se nalazi polje za unos imena uređaja. Nakon dodavanja, na sučelju se pojavljuje novi uređaj kojemu korisnik

zatim može dodavati senzore i aktuatore. Klikom na „Show/hide identifier“ korisnik može pogledati identifikacijsku oznaku uređaja (slika 5.2.).



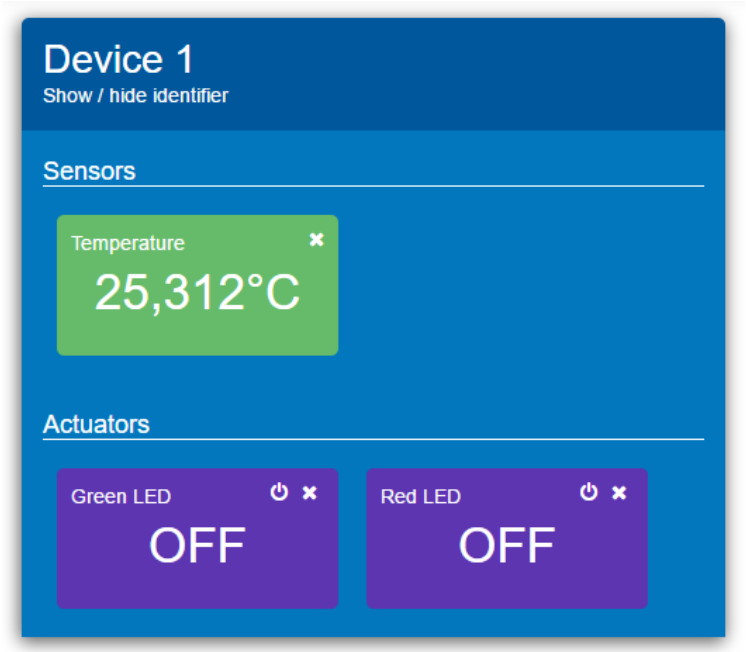
Sl. 5.2. Novi uređaj.

Korisnik prilikom dodavanja novog senzora ili aktuatora mora upisati njegovo ime i izabrati jednu od vrsta ponuđenih u padajućem izborniku, kako je prikazano na slici 5.3.



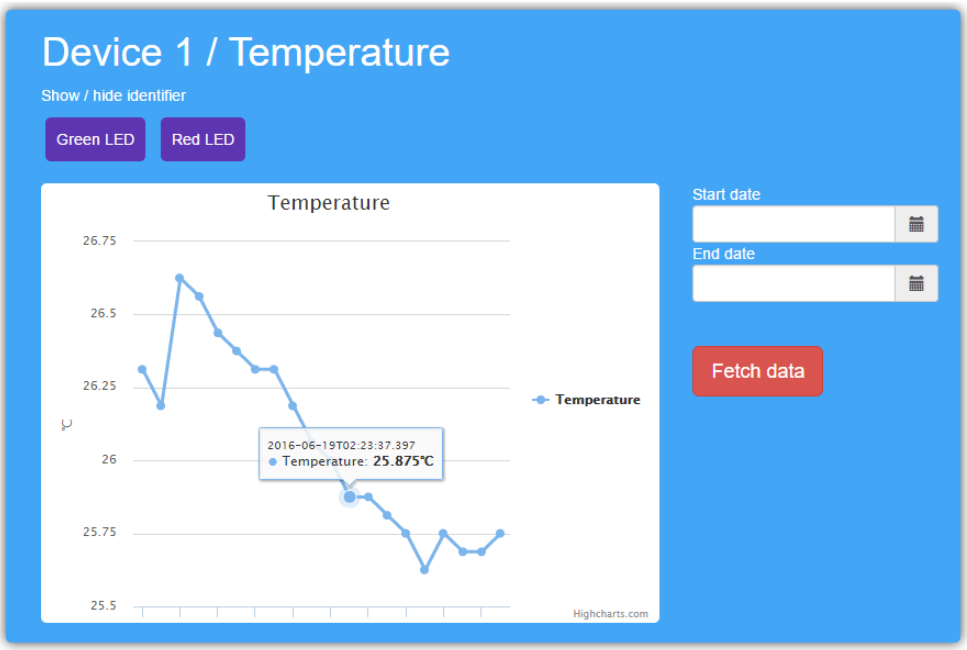
Sl. 5.3. Modalni prozor za dodavanje novog senzora.

Senzore i aktuatore na uređaju, njihovu posljednju izmjerenu vrijednost i trenutno stanje korisnik može vidjeti na početnom sučelju (slika 5.4.). Ovdje također ima mogućnost uklanjanja senzora i aktuatora s uređaja klikom na dugme u gornjem desnom kutu pojedinog senzora ili aktuatora. Na aktuatorima se nalazi i dugme za uključivanje i isključivanje aktuatora. Za pregled starijih podataka potrebno je kliknuti na senzor ili aktuator.



Sl. 5.4. Uređaj sa senzorom i aktuatorima.

Sučelje za prikaz svih izmjerenih vrijednosti dano je na slici 5.5. pri prvom učitavanju korisniku prikazuje 20 zadnjih izmjerenih vrijednosti. Klikom na „Show/hide identifier“ ispod imena senzora i uređaja kojemu pripada, korisnik može pogledati identifikacijsku oznaku senzora. Ispod toga se nalaze poveznice na ostale senzore i uređaje koji se nalaze na tom uređaju. Kako bi pregledao starije vrijednosti, korisnik bira početni i krajnji datum u izbornicima desno od grafa. Klikom na „Fetch data“ odgovarajući podaci prikazat će se na grafu. Prikaz starijih stanja aktuatora identičan je ovome, uz razliku što se umjesto linijskog koristi stupčasti graf.



Sl. 5.5. Sučelje za prikaz izmjerenih vrijednosti senzora.

6. ZAKLJUČAK

Zadatak rada bio je izrada sustava za nadzor i upravljanje bežičnim čvorovima. U radu je najprije dan kratki osvrt na Internet objekata i pregled nekoliko postojećih rješenja za nadzor i upravljanje. Predloženo rješenje ima slojeviti arhitekturalni obrazac. Za pohranu podataka korištena je relacijska baza podataka. Zadaća najnižeg sloja aplikacije jest interakcija s bazom. Podaci dohvaćeni na korisnikov zahtjev prosljeđuju se u sloj poslovne logike ili se podaci proslijeđeni iz gornjih slojeva spremaju u bazu. U sloju poslovne logike podaci dohvaćeni iz baze pripremaju se za prikaz korisniku. U slučaju pohrane podataka, podaci pristigli sa sučelja pripremaju se za pohranu u bazu. Prezentacijski sloj omogućuje korisniku interakciju sa sustavom – prikazuje podatke i prima korisnikove naredbe. Komunikacija na relaciji web aplikacija – udaljeni čvor ostvarena je putem HTTP i *WebSocket* protokola.

Prilikom izrade aplikacije nisam naišao ni na kakve veće probleme, a izazov je svakako bio osmišljavanje izgleda korisničkog sučelja i odabir tehnologije kojom bih ga napravio. Iako je inicijalno bilo u planu napraviti u potpunosti MVC aplikaciju, pokazalo se da je *Knockout* u kombinaciji sa API kontrolerima pruža puno bolji korisnički doživljaj. Glavna je prednost aplikacije relativna jednostavnost njezinog korištenja, a najveća je mana čitavog sustava program na udaljenom uređaju. Potencijalni korisnik aplikacije morao bi imati osnovno znanje o programiranju kako bi uspio povezati sve dijelove sustava. Stoga bi gotov program na uređaju uvelike olakšao korištenje sustava. Relacijska baza podataka bila je sasvim dovoljna u okviru ovog rada, no u produkcijskom okruženju, s potencijalno milijardama zapisa, performanse bi aplikacije iznimno brzo opale. Taj bi se problem mogao riješiti implementacijom platforme koja nudi *big data* rješenja (Google Cloud, Azure, Hadoop, i sl.). Čitav se model sučelja aplikacije nalazi u jednoj datoteci od nekoliko stotina linija koda. Iako to nije previše, eventualno dodavanje novih funkcionalnosti nikako ne bi popravilo situaciju. *Knockout.js* u kombinaciji s nekom od biblioteka za učitavanje datoteka i modula kao što su *Require.js* ili *System.js* omogućava razdvajanje modela sučelja u više manjih modela. Te komponente, osim što pružaju veću preglednost koda, omogućuju i brži rad aplikacije jer nije potrebno servirati čitav model sučelja odjednom nego samo zatraženi dio. Posljednje poboljšanje koje bi potencijalno povećalo mogućnost pronalaska greški u aplikaciji i ubrzalo budući razvoj, ali koje bi imalo neznatan utjecaj na rad same aplikacije, jest zamjena *JavaScript* koda *TypeScriptom*. *TypeScript* je nadskup *JavaScripta* i transkompilira se u njega. Njegove velike prednosti u odnosu na *JavaScript* jesu postojanje tipova podataka, klasa, modula, anonimnih funkcija, sučelja itd. Zbog činjenice da je to u pozadini ipak „samo“ obični *JavaScript* kod, podržava ga većina Internetskih preglednika.

LITERATURA

[1] International Telecommucation Union, Overview of the Internet of Things, 06.2012.

[2] The WebSocket protocol, https://tools.ietf.org/html/rfc6455, pristup 22. lipnja 2016.

[3] ASP.NET MVC overview, https://msdn.microsoft.com/en-us/library/dd381412(v=vs.108).aspx, pristup 22. lipnja 2016.

[4] A. Akinshin, Getting started with Knockout.js for .NET Developers, Packt Publishing, Birmingham, 2015.

[5] Chapter 3: Architectural patterns and styles, https://msdn.microsoft.com/en-us/library/ee658117.aspx, pristup 20. lipnja 2016.

[6] Alchemy WebSockets, https://github.com/Olivine-Labs/Alchemy-Websockets, pristup 15. svibnja 2016.

SAŽETAK

Tema diplomskog rada je izrada sustava za daljinski nadzor i upravljanje bežičnim čvorovima koji mogu imati različite senzore i aktuatore. U radu je napravljen kratki pregled Interneta objekata i postojećih rješenja za nadzor i upravljanje. Osim analize predloženog rješenja, opisane su i tehnologije korištene pri izradi. Web aplikacija izrađena je u slojevitom arhitekturalnom uzorku korištenjem *Microsoftov* .NET okvir i *Knockout.js* okvir za *single-page* aplikacije. Komunikacija na relaciji web aplikacija – udaljeni čvor napravljena je implementacijom HTTP i *WebSocket* protokola. U posljednjem je poglavlju na jednostavnom primjeru demonstrirano korištenje sustava.

Ključne riječi: Internet objekata, bežični čvorovi, slojeviti arhitekturalni obrazac, ADO.NET, *WebSocket* protokol, ASP.NET Web API 2, *Knockout.js*, Python, Raspberry Pi

ABSTRACT

Title: Remote monitoring and control of wireless nodes

This thesis presents the process of developing a remote monitoring and control system for wireless nodes that can have a variety of sensors and actuators. The paper offers a brief overview of the Internet of Things, followed by a presentation of several existing solutions for monitoring and control. After analysing the provided solution, the paper additionally provides a description of the technologies used to create the application. The web application was developed in a layered design pattern using Microsoft's .NET framework and Knockout.js single-page application framework. The communication between the web application and the node was accomplished by implementing the HTTP and WebSocket protocols. The paper concludes with an exemplification of the system's functioning using a simple example.

Keywords: Internet of Things, wireless nodes, layered design patter, ADO.NET, WebSocket protocol, ASP.NET Web API 2, *Knockout.js*, Python, Raspberry Pi

ŽIVOTOPIS

Tvrtko Klarić rođen je u Vinkovcima 28. veljače 1991. Od rođenja živi u Vođincima gdje je pohađao osnovnu školu „Vođinci“. Nakon završetka osnovne škole, 2005. godine upisuje 1. razred opće gimnazije „M. A. Reljković“ u Vinkovcima. 2009. godine završio je srednju školu i upisuje Elektrotehnički fakultet u Osijeku, sveučilišni preddiplomski studij računarstva. Po završetku studija, 2013. godine upisuje sveučilišni diplomski studij, smjer procesno računarstvo.

Vlastoručni potpis

_____

PRILOZI

Prilog 1: `Actuator` klasa (IoTpanel/IoTpanel.Data/Database/Actuator.cs)

```
public partial class Actuator
    {
        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",
"CA2214:DoNotCallOverridableMethodsInConstructors")]
        public Actuator()
        {
            this.ActuatorData = new HashSet<ActuatorData>();
        }

        public int Id { get; set; }
        public string Name { get; set; }
        public int TypeId { get; set; }
        public System.Guid ActuatorIdentifier { get; set; }
        public int DeviceId { get; set; }

        public virtual Device Device { get; set; }
        public virtual Type Type { get; set; }
        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",
"CA2227:CollectionPropertiesShouldBeReadOnly")]
        public virtual ICollection<ActuatorData> ActuatorData { get; set; }
    }
```

Prilog 2: `ActuatorData` klasa (IoTpanel/IoTpanel.Data/ActuatorData.cs)

```
public class ActuatorData
    {
        Entities _db = new Entities();

        public Actuator AddActuator(Actuator model)
        {
            var addedActuator = _db.Actuator.Add(model);
            _db.SaveChanges();

            return addedActuator;
        }


        public bool RemoveActuator(int actuatorId)
        {
            var actuator = _db.Actuator.FirstOrDefault(a => a.Id == actuatorId);
            _db.Actuator.Remove(actuator);

            return _db.SaveChanges() > 0;

        }

        public Actuator GetActuator(int actuatorId)
        {
            return _db.Actuator.FirstOrDefault(a => a.Id == actuatorId);
        }
```

```
        public bool SaveActuatorData(Database.ActuatorData model)
        {
            _db.ActuatorData.Add(model);
            return _db.SaveChanges() > 0;
        }

public InstrumentDataDto GetActuatorData(int actuatorId, DateTime StartDate, DateTime
EndDate)
        {
            var data = new InstrumentDataDto();
            var values = _db.Actuator.FirstOrDefault(s => s.Id ==
actuatorId).ActuatorData.Where(sd => sd.Date >= StartDate && sd.Date <= EndDate);
            data.yAxis = values.Select(v => v.Value ? 1 : 0.1).ToList();
            data.xAxis = values.Select(v => v.Date).ToList();

            return data;
        }
```

Prilog 3: `DeviceService` klasa (IoTpanel/IoTpanel.Service/DeviceService.cs)

```
public class DeviceService
{
    DeviceData deviceData = new DeviceData();

    public AppIndexDto GetInitialData(string email)
    {
        var data = new AppIndexDto();

        try
        {
            var devices = deviceData.GetUserDevices(email);

            data.Devices = devices.Select(d => new DeviceDto()
            {
                Id = d.Id,
                Name = d.Name,
                DeviceIdentifier = d.DeviceIdentifier,
                Sensors = d.Sensor.Where(s => s.SensorData != null)
                .Select(s => new IndexSensorDto()
                {
                    DeviceId = s.DeviceId,
                    Id = s.Id,
                    Name = s.Name,
                    Type = s.Type.Id,
                    LastDataEntry = ConvertSensorValue(s)
                }).ToList(),
                Actuators = d.Actuator.Select(a => new IndexActuatorDto()
                {
                    DeviceId = a.DeviceId,
                    DeviceIdentifier = d.DeviceIdentifier,
                    Id = a.Id,
                    Identifier = a.ActuatorIdentifier,
                    Name = a.Name,
                    Type = a.Type.Id,
                    LastDataEntry = ConverActuatorvalue(a)
                }).ToList()
            }).ToList()
                .OrderByDescending(d => (d.Sensors.Count + d.Actuators.Count))
                .ToList();

            TypeData typeData = new TypeData();
            data.SensorTypes = typeData.GetSensorTypes();
            data.ActuatorTypes = typeData.GetActuatorTypes();

            return data;
        }
        catch (Exception)
        {
            throw;
        }
    }
```

```
private string ConvertSensorValue(Sensor sensor)
        {
            var data = sensor.SensorData.OrderByDescending(s => s.Date)
                                .FirstOrDefault();
            if (data != null)
            {
                return data.Value.ToString() + sensor.Type.ValueSuffix;
            }
            else
            {
                return "NO DATA";
            }
        }

private string ConverActuatorValue(Actuator actuator)
        {
            var data = actuator.ActuatorData.OrderByDescending(a => a.Date)
                                .FirstOrDefault();
            if(data != null)
            {
                if(data.Value)
                {
                    return "ON";
                }
                else
                {
                    return "OFF";
                }
            }
            else
            {
                return "NO DATA";
            }
        }

      public DeviceDto AddDevice(string name, string email)
        {
            try
            {
                var newDevice = new Device()
                {
                    Name = name,
                    DeviceIdentifier = Guid.NewGuid()
                };

                var addedDevice = deviceData.AddDevice(newDevice, email);

                if (addedDevice == null)
                {
                    return new DeviceDto();
                }

                return new DeviceDto()
                {
                    Id = addedDevice.Id,
                    Name = addedDevice.Name,
                    DeviceIdentifier = addedDevice.DeviceIdentifier
                };
            }
```

```
            catch (Exception)
            {
                throw;
            }
        }

        public bool RemoveDevice(int deviceId)
        {
            try
            {
                return deviceData.RemoveDevice(deviceId);
            }
            catch (Exception)
            {
                throw;
            }
        }
    }
```

Prilog 4: `ToggleActuator` metoda (IoTpanel/IoTpanel.Service/ActuatorService.cs)

```
public ActuatorResponse ToggleActuator(ToggleActuatorDto data)
        {
            var url = ConfigurationManager.AppSettings["baseUrl"];
            var ws = new WebSocket($"ws://{url}/{data.DeviceIdentifier}");
            var response = new ActuatorResponse();
            ws.OnMessage += (sender, e) =>
            {
                if (e.Data == "1")
                {
                    ws.Close();
                }
            };
            ws.OnError += (sender, e) => { response.Success = false; };

            var actuatorRequest = new
            {
                ActuatorIdentifier = data.ActuatorIdentifier,
                Status = data.Status
            };
            var message = new JavaScriptSerializer().Serialize(actuatorRequest);

            ws.Connect();
            ws.Send(message);

            var timeStarted = DateTime.Now;
            var span = TimeSpan.FromSeconds(5);

            while (ws.IsAlive)
            {
                var current = DateTime.Now;
                if (DateTime.Now - timeStarted > span)
                {
                    response.Success = false;
                    ws.Close();
                }
                else
                {
                    response.Success = true;
                }
            }
```

```
            if (response.Success)
            {
                var model = new Data.Database.ActuatorData()
                {
                    ActuatorId = data.ActuatorId,
                    Date = DateTime.Now,
                    Value = data.Status == 1 ? true : false
                };

                _actuatorData.SaveActuatorData(model);
                response.Status = data.Status;
            }

            return response;
        }
```