Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Kerry Mahne

Izdelava sistema za upravljanje sredstev

diplomsko delo

visokošolski strokovni študijski program
prve stopnje
računalništvo in informatika

Mentor: dr. Aleksander Sadikov

Ljubljana, 2018

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
AWS	Amazon Web Services	Amazonove spletne storitve
HTTP	Hypertext Transfer Protocol	protokol za prenos hiperteksta
REST	Representational State Transfer	predstavitveni prenos stanja
API	application programming interface	vmesnik za programiranje aplikacij
HTML	Hyper Text Markup Language	jezik za označevanje nadbesedila
CSS	Cascading Style Sheets	kaskadne stilske podloge
IAM	Identity and Access Management	nadzor identitet in dostopa
SES	Simple Email Service	enostavna e-poštna storitev
SNS	Simple Notification Service	enostavna obvestilna storitev
S3	Simple Storage Service	enostavna shrambna storitev
EC2	Elastic Compute Cloud 2	elastični računski oblak
AMI	Amazon Machine Image	Amazonova slika naprave
SMS	Short Message Service	storitev kratkih sporočil
USD	United States Dollar	ameriški dolar
GB	gigabyte	gigabajt
MB	megabyte	megabajt
SQL	Structured Query Language	strukturirani povpraševalni jezik
MVC	Model–view–controller	Model-pogled-krmilnik
DOM	Document Object Model	objektni model dokumenta

Povzetek

Naslov: Izdelava sistema za upravljanje sredstev

Avtor: Kerry Mahne

V sodelovanju s podjetjem Comtrade je bil za diplomsko delo izdelan sistem za upravljanje sredstvev in poslovnih pravil. Sistem omogoča opis in vnos kateregakoli tipa sredstev in izdelave pravil glede na vrednosti atributov sredstev. Pravilu dodelimo sredstva in ko so izpolnjeni njegovi pogoji, smo o tem obveščeni prek e-pošte. Sistem je sestavljen iz več gradnikov: tipov, atributov, dodeljenih atributov, sredstev, dodeljenih vrednosti atributov in pravil. Vsak gradnik ima svoje lastnosti in omejitve za operacije na sistemu. Za izdelavo sistema je bil uporabljen sodoben nabor tehnologij, glavne med njimi so Amazon Web Services za zaledni sistem v oblaku, Mongo Atlas za podatkovno bazo v oblaku in ReactJS za pročelni sistem. Po končanem testiranju je bila izdana pilotna različica sistema, ki je sinhronizirala podatke o testnih avtomobilih Avant2Go v sistem na vsake pol ure. Vsi avtomobili so bili dodeljeni pravilom, ki so na moj e-naslov pošiljala obvestila o uspešni evalvaciji. Sistem je bil analiziran in evalviran prek teh e-poštnih sporočil. V sklepnem poglavju so opisane izkušnje in pridobljena znanja pri izdelavi sistema.

Ključne besede: mobilnost, oblačno računalništvo, poslovna pravila, internet stvari, Javascript, Node.js, ReactJS, AWS, MongoDB.

Abstract

Title: Creation of an asset management system

Author: Kerry Mahne

This thesis presents the creation of an asset management system with business rules. It was done in cooperation with Comtrade. The system enables the input and description of any asset type and the creation of rules for asset's attribute values. It is composed of multiple components: types, attributes, assigned attributes, assets, assigned attribute values, and rules. Each component has its own properties and restrictions for system operations. The system was built with a modern collection of technologies, the key ones are Amazon Web Services for the backend cloud system, Mongo Atlas for the cloud-based database and ReactJS for the frontend system. After the testing, a pilot version was released that synchronized data for staging Avant2Go cars into the system on a half-hourly basis. All of the cars were assigned to rules, which sent notifications of successful evaluations to my email address. These emails were used for the analysis and evaluation of the system. The last chapter describes the experience and knowledge gained by building the system.

Keywords: mobility, cloud computing, business rules, Internet of things, Javascript, Node.js, ReactJS, AWS, MongoDB.

Poglavje 1

Uvod in motivacija

Za diplomsko nalogo sem v sodelovanju s podjetjem Comtrade izdelal večuporabniški (angl. *multi-tenant*) zaledni sistem za upravljanje sredstev in delo s poslovnimi pravili. Poleg zalednega sistema smo izdelali pročelni sistem, ki služi kot nadzorna plošča za upravljanje sistema in funkcionalne teste, s katerimi se preverja ustrezno odzivanje sistema na HTTP-zahtevke. Je zelo abstrakten, zato je vanj teoretično mogoče vnesti in v njem opisati katerikoli tip sredstva. Tipu dodelimo atribute, ki jih bodo posamezna sredstva nato imela. Sredstvo je konkreten primer tipa, z določenimi vrednostmi atributov. Poleg operacij pisanja, posodabljanja, branja in brisanja nad vsemi gradniki sistema smo ustvarili še lasten podsistem za delo s poslovnimi pravili. Ta uporabnikom sistema omogoča ustvarjanje poljubno dolgo veriženih in globoko gnezdenih pravil, ki jim nato dodeli sredstva. Če sredstvo izpolni pogoje pravila, je uporabnik o tem obveščen prek želenega sredstva sporazumevanja. Več o posameznih gradnikih sistema v 2. poglavju.

Tehnologija se razvija vse hitreje. Vse več naprav ima internetno povezavo, zaradi česar se zadnja leta govori o tako imenovanem internetu stvari (angl. *Internet of Things*). Če ima naprava internetno povezavo, lahko brez težav izvemo veliko o njej. Podatkov ni treba več vnašati ročno, naprava ima senzorje in GPS, zato lahko le prebere vrednosti prek senzorjev in jih pošlje na želeno mesto v internetu. Meritve lahko spremljamo skoraj v realnem času

in imamo zelo dobro predstavo o stanju posamezne naprave. Nekatera stanja tudi zahtevajo ustrezen odziv, če na primer vemo, da je naprava na neugodni lokaciji, jo je treba premestiti. Z našim sistemom imajo uporabniki celovit pregled nad svojimi napravami oziroma sredstvi. Poleg pregleda in urejanja vrednosti naprav omogoča samodejno obveščanje o napravi glede na uporabnikove pogoje. Tako uporabniku ni treba stalno bdeti nad morebitnimi anomalijami, čudnimi podatki in neugodno lokacijo, temveč le ustvari pravila in čaka na obvestila. Sistem ima tudi širok spekter primerov uporabe, saj omogoča upravljanje najrazličnejših sredstev. Podjetju sistem pride prav, saj ima veliko različnih strank z različnimi potrebami, vse pa želijo imeti neko obliko sistema za upravljanje sredstev. Izdelava sistema za vsako sredstvo bi bila dolgotrajna in draga, zato mi je bila dana naloga, da izdelam večnamenski sistem, v katerem bo mogoče opisati vsa sredstva.

Veliko tipov sredstev lahko tudi ni pretirano kompleksnih, to so na primer pametne luči in boje. Za pametne luči je treba beležiti le lokacijo in napolnjenost baterije luči, če je slednja na primer nizka, je treba dobiti obvestilo, da bo baterijo treba zamenjati. Pri bojah je treba poznati le njihovo lokacijo in dobiti obvestilo, ko se boja oddalji za opredeljeno vrednost iz predpisane geolokacije. Za ta dva tipa sredstev ni vredno pisati namenskih sistemov, bolje je imeti en sistem, ki bo lahko obravnaval vsa sredstva. Omogočena je tudi podpora za zapletenejša sredstva, kot so avtomobili, saj so vsi atributi in pravila uporabniško definirani, tako lahko uporabnik v sistem vnese čisto vsak tip sredstva. Z izdelavo sistema se znižajo tudi stroški za nakup obstoječih rešitev, saj za deset tipov sredstev ni treba kupiti ali izdelati deset različnih sistemov.

1.1 Zakaj namenski sistem namesto obstoječe rešitve?

Sistemov za upravljanje sredstev je veliko, tudi sistemov, ki omogočajo delo s poslovnimi pravili, je nekaj. Primer sistema z dobro definiranimi pravili

je Magento[12]. Njegovo domensko področje je sicer bolj spletno nakupovanje, vendar smo se pri izdelavi strukture lastnih pravil okvirno zgledovali po Magentovi strukturi pravil.

Večina sistemov za upravljanje sredstev je tudi bolj domensko omejenih, zaradi česar so lahko za svojo domeno močnejši od mojega, niso pa tako splošnonamenski. Podjetje poleg tega želi svojim strankam ponuditi sistem za upravljanje sredstev, stranke pa so zelo raznolike. Namesto nakupa več posameznih domenskih sistemov za upravljanje sredstev, na primer sistema za upravljanje avtomobilov, boj, baterij itd., je bolje ponuditi eno celovito rešitev, ki pokrije vsa domenska področja. Takšna rešitev v nasprotju z veliko drugimi sistemi nima omejitev glede atributov, tako na primer ni strogo vezana le na predefinirane atribute, kot je število vrat, sedežev itd., temveč si jih uporabnik poljubno ustvarja sam. V našem sistemu se tako lahko beležijo različni atributi glede na uporabnikove želje in potrebe.

Poglavje 2

Gradniki sistema

Sistem je sestavljen iz šestih osnovnih gradnikov, pet se jih uporablja za opisovanje sredstva, eden pa za definicijo poslovnih pravil. Tipi so najvišji nivo opisa sredstev, z imenom poimenujejo tip sredstva v sistemu. Atributi so namenjeni opisu vrednosti, ki jih sredstvo nekega tipa lahko ima. Tako ima posamezen poimenovan atribut določen tudi podatkovni tip, zaradi česar v sistemu en atribut pri različnih sredstvih ne more imeti različnih podatkovnih tipov. Ločen gradnik je tudi dodelitev atributov, ta beleži, kateri atribut je dodeljen kateremu tipu. Zaradi ločenosti omogoča razvijalcem in uporabnikom enoten in preprost pregled nad že dodeljenimi atributi za posamezen tip. Sredstva so poimenovane konkretne instance tipov, ki imajo lahko dodeljene vrednosti za atribute, dodeljene tipu. Dodeljene vrednosti sredstev so dejanske vrednosti atributov sredstva. Tako je mogoče abstraktno opisati vsak tip sredstva, za avtomobil bi bil proces na primer takšen:

1. Ustvarimo tip sredstva ‘avtomobil’.

2. Ustvarimo atributa ‘lokacija’ in ‘stanjeBaterije’.

3. Ta atributa dodelimo tipu avtomobil.

4. Ustvarimo poljubno število sredstev tipa ‘avtomobil‘, ki jim lahko dodelimo vrednosti za atributa ‘lokacija’ in ‘stanjeBaterije’.

Pravila so uporabniško definirani pogoji, ki – če so izpolnjeni – sprožijo želeni dogodek v sistemu, na primer pošiljanje obvestila po e-pošti. Pravila se definirajo glede na vrednosti posameznih atributov, nato se jim dodelijo sredstva, ki bi jih želeli spremljati, in sistem bo ob vsaki spremembi vrednosti sam evalviral pravilo glede na sredstvo z novimi vrednostmi in se ustrezno odzval. Tako bi v zgornjem primeru lahko opredelili pravilo, ki pošlje e-pošto, če ima avtomobil vrednost 'stanjeBaterije' manj kot 50, k pravilu bi dodelili vse avtomobile, in bi za vsakega dobili obvestilo, ko je njegovo 'stanjeBaterije' manj kot 50.

Več o posameznih gradnikih v pripadajočih podpoglavjih.

2.1 Tipi

Tipi se uporabljajo kot visokonivojska abstrakcija sredstev, opisana le z imenom. Uporabnik lahko ustvari poljubno število tipov in jim nato dodeljuje atribute, ki lahko skupaj popolnoma opišejo posamezno sredstvo. Zaradi visoke ravni abstraktnosti lahko definiramo katerikoli tip sredstva, tako avtomobile kot mize.

Omejitve

Ustvarjanje

Mogoče je ustvariti tip s kakršnimkoli imenom, edina omejitev je, da se to ne uporablja že za opis katerega drugega tipa.

Posodabljanje

Ker je tip dejansko le ime in obstaja možnost, da se uporabnik pri začetnem poimenovanju zmoti, se ime tipa lahko posodobi ne glede na preostale dele sistema. Edini pogoj je, da je novo ime niz znakov.

Brisanje

Tipa, ki ima že dodeljena sredstva in/ali atribute, ni mogoče izbrisati. S tem bi namreč spravili sistem v nekonsistenčno stanje, saj bi v njem obstajala sredstva neobstoječega tipa. Za izbris tipa je potreben izbris vseh sredstev tega tipa in njihovih dodeljenih vrednosti. Prav tako je treba odvzeti vse do tedaj dodeljene atribute.

2.2 Atributi

Atributi so kombinacija uporabniško določenega imena in podatkovnega tipa. Uporabnik jih lahko naredi poljubno mnogo in jih dodeljuje tipom. Tako je opis posameznega sredstva konkretnejši in uporabnejši kot le z imenom tipa. Poleg izbire imena in podatkovnega tipa atributa lahko uporabnik, če je za podatkovni tip izbral niz znakov ali število, izbere tudi vnaprej določene vrednosti tega atributa.
V sistemu so trenutno podprti naslednji podatkovni tipi:

1. število
2. niz znakov
3. geolokacija
4. datum (v formatu ISO 8601[25])
5. boolean

Omejitve

Ustvarjanje

Atributa z že obstoječim imenom ni mogoče ustvariti, tudi če je drugega podatkovnega tipa.

Posodabljanje

Mogoče je posodobiti ime atributa ali podatkovni tip. Ime se lahko posodobi, če je bilo novo ime podano kot niz znakov, ki ni že v uporabi kot ime drugega obstoječega atributa. Sprememba podatkovnega tipa atributa je dovoljena le, če ni sredstev, ki bi imela dodeljene vrednosti tega atributa, in če ni pogojev pravil, ki uporabljajo ta atribut. Če ne bi bilo te omejitve, bi v sistemu lahko obstajalo sredstvo z dodeljeno vrednostjo napačnega podatkovnega tipa, tako bi avtomobil na primer imel za atribut 'stanjeBaterije' dodeljeno celoštevilsko vrednost 80, nato bi podatkovni tip spremenili v niz znakov, dodeljena vrednost pa je še vedno celoštevilska.

Brisanje

Atributa ni mogoče izbrisati, če obstajajo sredstva z dodeljenimi vrednostmi tega atributa. V tem primeru bi v sistemu ostale dodeljene vrednosti neobstoječih atributov, kar bi privedlo do nekonsistenčnega stanja. Prav tako se ne da izbrisati atributa, ki je uporabljen v pogojih pravil, saj se sredstva, dodeljena pravilu, ne bi mogla evalvirati po neobstoječih atributih. Prav tako se ne da izbrisati atributa, ki je dodeljen enemu ali več tipom, pred tem je atribut treba odvzeti od tipa, sicer bi lahko ostale dodeljene vrednosti neobstoječih atributov.

2.3 Dodeljevanje atributov

Kot je bilo že nekajkrat navedeno, je za oprijemljivejši opis sredstev atribute mogoče dodeljevati posameznim tipom. Za večjo modularizacijo sistema so dodeljeni atributi hranjeni kot ločen gradnik, kar omogoča lažje poizvedovanje po tipih in njihovih dodeljenih atributih.

Omejitve

Dodeljevanje

Posamezen atribut je lahko dodeljen enemu ali več tipom, ne more pa biti dodeljen enemu tipu večkrat.

Odvzemanje

Atributa ni mogoče odvzeti tipu, če obstajajo sredstva z dodeljenimi vrednostmi tega atributa. Tako bi v sistemu obstajala sredstva z dodeljenimi vrednostmi tega atributa, čeprav se glede na sistem sredstva sploh ne da opisati s tem atributom. Če se tipu odvzame atribut, se za vsa obstoječa in nadaljnja sredstva ta atribut ne uporablja več.

2.4 Sredstva

Zgradba sredstva je opisana prek tipa in njegovih dodeljenih atributov. Tako lahko hkrati obstaja več sredstev istega tipa, na primer več avtomobilov. Sredstva istega tipa se ločijo med seboj po imenu. Ko se ustvari sredstvo, ni treba, da se vrednosti atributov dodelijo takoj, to je mogoče narediti kadarkoli v življenjskem ciklu sredstva. Posamezno sredstvo ima lahko tako dodeljene vrednosti za *0* - n atributov njegovega tipa. Tako je sredstvo le poimenovana instanca nekega tipa, ki je lahko natančneje določena z dodeljevanjem vrednosti njegovim atributom.

Omejitve

Ustvarjanje

Ob ustvarjanju sredstva se vrednosti atributov še ne določijo, zato je treba podati želeni tip in ime, ki ne sme že biti v uporabi za sredstva želenega tipa.

Posodabljanje

Sredstvu je dovoljeno posodobiti ime, če to seveda ni že v uporabi. Tipa sredstva se ne da spremeniti, če je to dodeljeno kakšnemu pravilu ali če ima dodeljene vrednosti atributov. V tem primeru bi v sistemu obstajala sredstva z dodeljenimi vrednostmi za neki drug tip, kar bi porušilo evalvacijo pri pravilih. Sredstvo bi tudi imelo dodeljene vrednosti atributov glede na star tip, nov tip teh atributov lahko sploh ne bi imel dodeljenih.

Brisanje

Sredstva, ki je dodeljeno nekemu pravilu, se ne da izbrisati. V tem primeru bi se namreč evalvacija pravila poskušala izvesti nad neobstoječim sredstvom. Prav tako se ne da izbrisati sredstva, ki ima določene vrednosti atributov, saj bi v sistemu ostale dodeljene vrednosti neobstoječih sredstev.

2.5 Dodeljene vrednosti sredstev

Vsako sredstvo ima lahko glede na tip določenih *0 - n* vrednosti atributov. Tudi ti podatki se hranijo kot ločen modul zaradi lažjega pregleda že dodeljenih vrednosti. Vrednosti se obstoječemu sredstvu lahko dodelijo kadarkoli.

Omejitve

Dodeljevanje

Pri dodeljevanju vrednosti atributov je treba paziti na podatkovni tip atributa. Vrednosti se lahko dodelijo le za atribute, ki jih ima tip sredstva dodeljene. Posamezno sredstvo ima lahko le po eno dodeljeno vrednost za vsak atribut, da se izognemo konfliktom v sistemu. Sistem ne dovoli neujemanja podatkovnega tipa podane vrednosti in podatkovnega tipa atributa. Prav tako ni dovoljen napačen format podanega datuma, ta mora biti strogo v formatu ISO 8601. Dovoljen ni niti napačen format geolokacije, ki mora imeti le ključa geografska širina in geografska dolžina. Hkrati ni mogoče

dodeliti vrednosti za atribut, ki ga tip sredstva ne podpira. Če ima atribut vnaprej določene vrednosti, mora biti ena izmed njih tudi dodeljena vrednost, sicer bo dodelitev neuspešna.

Posodabljanje

Pri posodabljanju veljajo iste omejitve kot pri dodeljevanju. Prav tako se ne da posodobiti vrednosti, tako da bi zamenjali atribut vrednosti ali sredstvo, ki mu je vrednost dodeljena, dovoljeno je le posodabljanje dejanske dodeljene vrednosti.

Brisanje

Pri brisanju dodatnih omejitev ni, ker je najnižji nivo opisa sredstva v sistemu, se od dodeljene vrednosti naprej ne deduje nič. Evalvacija pravil bo sredstva, ki nimajo vseh potrebnih atributov, preskočila.

2.6 Pravila

Pravila so del sistema, ki skrbijo za evalvacijo sredstev glede na uporabniško določene pogoje.
Obstajata dva tipa pravil:

1. ob spremembi vrednosti

2. kronološka (k)

Trenutno so pogoji omejeni glede na tip atributa.

Atribut / Pogoj	Število	Niz znakov	Boolean	Geolokacija	Datum
==	✓	✓	✓		
≠	✓	✓	✓		
>	✓				
<	✓				
≥	✓				
≤	✓				
V radiju okrog geolokacije				✓	
Zunaj radija okrog geolokacije				✓	
V poligonu okrog geolokacije				✓	
Zunaj poligona okrog geolokacije				✓	
Pred datumom (k)					✓
Po datumu (k)					✓

Pogoji so med seboj lahko povezani na način ALL in ANY. Pri načinu ALL morajo biti izpolnjeni vsi navedeni pogoji, pri načinu ANY pa vsaj eden. Lahko so tudi poljubno globoko gnezdeni, primer kompleksnejšega pravila:

ANY:

- Pogoj 1
- Pogoj 2
- ALL:
  - Pogoj 3
  - ANY:
    - Pogoj 4
    - Pogoj 5

Pogoj 6

...

Pravilu so poleg pogojev dodeljena sredstva, nad katerimi izvaja evalvacijo. Za vsako dodeljeno sredstvo pravilo hrani zastavico, ali je bilo obvestilo o uspešni evalvaciji že poslano. Takoj ko je pogoj izpolnjen, se pošlje obvestilo in zastavica se nastavi na poslano. Naslednjič, ko je pogoj na sredstvu evalviran kot neizpolnjen, se zastavica nastavi nazaj na neposlano.

2.6.1 Pravila ob spremembi vrednosti

Za lažje razumevanje bomo obravnavali preprosto pravilo s pogojem, da je stanje baterije pod 20 odstotkov. Pravila se prožijo v treh primerih:

1. Ob posodobitvi dodeljene vrednosti nekega sredstva – tu se lahko na primer stanje baterije spremeni z 22 na 18, in bo ustrezala pogoju.

2. Ob posodobitvi pogojev – stanje baterije sredstva je na primer 25 odstotkov, nato posodobimo pogoj na stanje baterije pod 30 odstotkov. Sredstvo sedaj ustreza pogoju, a če v tem primeru ne bi izvedli evalvacije, tega uporabnik ne bi vedel.

3. Ob dodelitvi sredstva pravilu – pravilu namreč lahko dodelimo baterijo s stanjem napolnjenosti 15 odstotkov in takoj ustreza.

2.6.2 Kronološka pravila

Vezana so na datum, zato jih je treba periodično evalvirati. Uporabljajo se le z datumi, lahko se nastavijo za obveščanje pred želenim datumom in tudi po njem. To je na primer pravilo, ki pošlje obvestilo, da bo čez en teden potekla veljavnost registracije avtomobila, oziroma pravilo, ki pove, da je veljavnost kreditne kartice potekla pred desetimi dnevi.

Omejitve

Ustvarjanje

Omejitve so stroge pri pogojih, ker so popolnoma dinamični in uporabniško definirani. Podatkovni tip vrednosti pogoja se mora ujemati s podatkovnim tipom atributa, nad katerim je pogoj definiran. Geolokacija mora biti strogo poslana le kot geografska širina in dolžina, nič drugega. Pravila za obveščanje pred datumom in po njem morajo imeti poslane vrednosti v milisekundah.

Posodabljanje

Ni dovoljeno spremeniti tipa pravila, saj bi to pomenilo spremembo načina evalvacije. V tem primeru je to že drugo pravilo, zato je pravilo bolje zbrisati in ga ustvariti na novo z novimi pogoji.

Brisanje

Pravila se lahko izbrišejo v vsakem primeru.

Dodeljevanje in odvzemanje sredstev

Edina omejitev pri tem je, da se ne da dodati/odvzeti neobstoječih sredstev.

Poglavje 3

Uporabljene tehnologije

Za dejansko izdelavo sistema smo uporabili sodoben nabor tehnologij. Vsaka bo podrobneje opisana v pripadajočih podpoglavjih.

Za zaledni sistem smo namesto klasičnega pristopa ubrali rešitev v oblaku, in sicer gostovanje na Amazonovi spletni rešitvi Amazon Web Services (AWS) z uporabo programskega jezika Javascript, natančneje Node.js. Na njem se ne zgradi celoten sistem z enotno vstopno točko, ampak se za vsako krajišče na želeni HTTP-zahtevek požene ustrezna funkcija lambda. Ker sistem gostuje na AWS, smo lahko za upravljanje registriranih uporabnikov uporabili Cognito in IAM. To sta rešitvi AWS za registracijo, prijavo in dodeljevanje pravic posameznim uporabnikom. Za pošiljanje e-pošte smo uporabili Amazonov Simple Email Service (SES), za pošiljanje obvestil in izvajanje funkcij publish/subscribe smo uporabili Simple Notification Service (SNS). Datoteke in nadzorno ploščo smo gostovali na Simple Storage Service (S3).

Za podatkovno bazo smo tudi izbrali oblačno rešitev, in sicer Mongo Atlas, kar je v bistvu MongoDB v oblaku. Ima tudi brezplačno obliko, pri kateri je velikost diska in pomnilnika omejena, vendar je bila ravno pravšnja za testiranje in izdelavo pilotne različice sistema.

Nadzorno ploščo smo izdelali z uporabo React.js in redux ter oblikovne knjižnice react-bootstrap.

3.1 Zaledni sistem

3.1.1 Javascript in Node.js

Javascript je dinamičen, šibkotipen objektni skriptni programski jezik, ki ga je razvilo podjetje NetScape[23]. Programerjem omogoča ustvarjanje dinamičnih in interaktivnih spletnih strani. Poleg jezikov CSS in HTML je eden izmed treh glavnih sestavnih delov svetovnega spleta. Za njegovo izvajanje ni treba imeti dodatkov, je samodejno podprt v vseh sodobnih spletnih brskalnikih. Njegovo dejansko ime je ECMAScript, krajše kar ES. Trenutno je zunaj že verzija ES7, vendar se uporabljata predvsem ES6 in ES5. Maja 2009 je Ryan Dahl ustvaril različico Javascripta, ki se lahko uporablja kot strežnik. Ta knjižnica se imenuje Node.js, ima asinhron vhod/izhod in dogodkovno arhitekturo[26]. Asinhronost pomeni, da se koda ne izvaja nujno v zaporedju vrstica za vrstico, saj obstajajo tako imenovane ovirajoče operacije. To so operacije, kakršne so na primer HTTP-zahteve na spletni strani – dokler ni nazaj rezultatov, se ustavi celoten potek izvajanja kode. Z asinhronostjo poskrbimo, da se drugi deli kode izvajajo, medtem ko eni deli kode čakajo na rezultate za obdelavo. Rezultate največkrat obdelamo tako, da definiramo funkcijo **callback**, ki se proži, ko pride odziv HTTP-zahtevka. Medtem se po kodi lahko izvajajo druge operacije. Funkcija callback ima lahko enega ali več argumentov, na prvem mestu je skoraj vedno objekt, ki hrani podatke o napaki. Tako lahko po dobljenem odzivu s stavki **if** obvladamo izvajanje in obnašanje kode glede na morebitne napake.

Node.js je tudi eden izmed uradno podprtih jezikov za izdelavo sistemov na AWS. S svojo asinhronostjo poskrbi za najboljši izkoristek procesorskega časa, zaradi česar je odličen za izvajanje funkcij na AWS, saj je tam izvajanje kode plačljivo glede na izvajalni čas in porabo pomnilnika. S pametno napisano kodo se lahko privarčuje nekaj denarja, saj bo Node.js poskrbel, da se koda izvaja tudi med čakanjem na ovirajočo operacijo.

Alternative

Ker je Javascript dinamičen jezik, se včasih lahko zgodi, da pričakujemo rezultat nekega tipa, dobimo pa rezultat drugačnega tipa, pri čemer ga bo Javascript poskusil pretvoriti v pričakovani tip in sistem pahnil v nekonsistenčno stanje. Zato je treba biti pri pisanju kode precej pazljiv, obstaja pa rešitev Typescript, ki Javascript pretvori v močnotipen jezik. Poleg Typescripta je seveda mogoče izbrati tudi kakšen drug programski jezik, saj AWS podpira še uporabo programskih jezikov Java, C#, GO in Python. Ti jeziki so močnotipni, zato je z njimi mogoče rešiti vprašanje neujemanja tipov.

Razlog za izbiro Node.js

Node.js smo izbrali, ker imamo z njim največ izkušenj in nam je programiranje v njem domače. Na AWS se koda v njem izvaja hitro, ima pa tudi številne uporabne knjižnice za kontrolo asinhronega toka operacij in knjižnico Mongoose za povezavo z bazo MongoDB. Ima res veliko dokumentacije in primerov na spletu, zato so bila vsa iskanja za reševanje težav v kodi hitra. Pythona nismo izbrali, ker je od vseh zgoraj naštetih jezikov najpočasnejši. V C# in GO še nismo delali, poleg tega ni nihče od sodelavcev najbolj tekoč v teh dveh jezikih, zato si nismo upali delati tako resnega projekta s čisto novim programskim jezikom. Java mi osebno ni najbolj všeč, poleg tega vsi člani naše ekipe uporabljajo Node.js, ki ima tudi več dokumentacije, kar je bil odločilni dejavnik za izbiro.

Morali smo se tudi odločiti za uporabo Typescripta, s čimer je mogoče rešiti nekaj težav. Vendar naš sistem ne pošilja HTTP-zahtevkov na zunanje sisteme, v bistvu le komunicira z bazo, za katero smo lahko prek knjižnice Mongoose določili sheme in ključem določili podatkovne tipe. Poskusili smo s Typescriptom, vendar z Mongoose funkcionira tako, da se določi shema za pričakovan objekt. To pomeni, da smo shemo objekta morali napisati enkrat za Mongoose – bazo, enkrat pa za Typescript, kar je seveda precej redundantno. Tako smo se dokončno odločili za uporabo navadnega Node.js.

3.1.2 Amazon Web Services

Amazon Web Services, oziroma krajše AWS[2], je hčerinsko podjetje Amazon.com in ponuja računalništvo v oblaku za posameznike in podjetja. Uporabnikom omogoča upravljanje virtualnega računalnika, ki ima tako kot fizičen računalnik svoje značilnosti. Tako lahko uporabniki, ki bolj potrebujejo procesiranje prek grafično-procesne enote (na primer za globoko učenje), izberejo virtualno napravo z močnejšo grafično enoto. Implementirano je na Amazonovih strežniških farmah po vsem svetu, zato deluje brez prekinitev. Amazon tudi skrbi za varnost, posodobitve in upravljanje navideznih naprav, zato uporabnika razbremeni vzdrževalnih nalog na strežniku, saj strežnika dejansko ni. AWS ponuja več kot 90 različnih servisov[24] za možnost celovite izdelave sistema, pri čemer ni treba uporabljati zunanjih komponent za podatkovne baze, avtentikacijo, pošiljanje e-pošte, potisnih obvestil in SMS-ov itd.

Za izdelavo sistema smo uporabili naslednje Amazonove servise:

- **AWS Lambda in vmesnik API** – anonimne Lambda funkcije, večino od njih proži API za odgovore na HTTP-zahtevke, nekatere pa tudi SNS;

- **Amazon Cognito** – servis za registracijo uporabnikov, prijavo v sistem in podpisovanje glave zahtevkov z avtorizacijo;

- **Simple Notification Service (SNS)** – servis za publish/subscribe arhitekturne dele sistema, ki se morajo izvesti ločeno od glavne izvajalne niti, to so na primer evalvacija pravil in pošiljanje obvestil;

- **Simple Email Service (SES)** – servis za pošiljanje e-pošte uporabnikom, po pozitivni evalvaciji pravil in sredstev.

Preden opišemo uporabljene servise, bomo pojasnili, zakaj smo se sploh odločili za AWS.

Zakaj ne fizični strežnik?

Najočitnejša alternativa je seveda uporaba fizičnega strežnika, na katerem teče instanca Node.js, ki dobiva zahteve na svoj vmesnik API in se nanje ustrezno odziva. Na tem strežniku se lahko hkrati postavi podatkovna baza, na primer v vsebniku Docker, in smo ubili dve muhi na en mah. Fizični strežniki so lahko tudi finančno ugodnejši, saj se najemnina naprave plačuje mesečno, na AWS pa se plačuje glede na izvajalni čas kode. AWS prvo leto podari 1.000.000 brezplačnih klicev na mesec, vsak milijon po tem pa stane 0,20 USD. V praksi to nanese nekaj več, saj se velikokrat še vpelje na primer SES za pošiljanje e-pošte, kar znese dodatnih 0,10 USD na 1000 sporočil, Cognito za upravljanje uporabnikov in njegove dodatne varnostne možnosti tudi prispevajo svoj delež k ceni. Tako veliko število uporabnikov in zahtev pomeni tudi višjo ceno. AWS je precej primernejši za manjše sisteme, ker ima večina servisov tudi svoj 'Free Tier', ki ima precej radodarne značilnosti, na primer 100.000 brezplačnih zahtev na mesec, vse skupaj do 50.000 brezplačnih Cognito uporabnikov, brezplačno podatkovno bazo DynamoDB z omejenim številom branj in pisanj itd. Vse to uporabniku omogoča gostovanje z njegovo malo zamislijo s stroški nekaj evrov na mesec.
Upoštevati pa je treba, da sta v ceno všteta celotno vzdrževanje in skrb za dosegljivost sistema. V praksi se lahko zgodi, da je strežnik nedosegljiv, zato morajo imeti razvijalci sistema urejen dostop do njega, da ga lahko popravijo. Strežniki so velikokrat na različnih Linuxovih distribucijah, kar pomeni, da morajo razvijalci imeti tudi dobro poznati skriptni jezik bash, če želijo popraviti morebitne sistemske napake. Namesto tega se lahko v ekipo vključi sistemski administrator ali inženir devops, ki bo skrbel, da strežnik teče, kot je treba. Seveda boljša strojna oprema stane precej več in ker sistem velikokrat ni stalno enako obremenjen, se zgodi, da se poraba pomnilnika ob intenzivni obremenitvi zelo poveča. V tem primeru je treba zakupiti več pomnilnika ali pa optimizirati kodo, narediti čakalne vrste za izvajanje delov sistema itd. Če pa je sistem gostovan na AWS, ni treba skrbeti za vse to, pomnilnik se alocira dinamično, ponujeno je tudi veliko servisov za nadzor

sistema (beleženje in izrisovanje grafov porabe virov).
Glede na navedeno bi težko opredelili, katera rešitev je zares boljša, saj ima vsaka svoje prednosti in slabosti. Sam sem izbral gostovanje na AWS, ker sem se tega želel naučiti, izkoristil sem tudi priložnost, da smo v podjetju že imeli pripravljene račune za delo z AWS.

Zakaj ne druge oblačne storitve?

Amazon seveda ni edino podjetje, ki ponuja računalništvo v oblaku. Zagotavljajo ga tudi Google, Microsoft in IBM. Vsaka rešitev je za nekatere primere cenovno boljša, performančno so vse izredno učinkovite. AWS je najstarejši med njimi, saj je bil izdan že leta 2006, zaradi česar ponuja res veliko servisov, mnogo več kot kateri drugi. Azure sicer počasi dohiteva AWS, vendar ima AWS še vedno kar 47-odstotni tržni delež med vsemi ponudniki oblačnih storitev[8]. To je skoraj polovica, kar pomeni, da ima veliko dokumentacije, vodičev in programerjev, ki se podajajo v razvoj.

Elastic Compute Cloud 2 – EC2

Vse Lambda funkcije se izvajajo v Amazonovem navideznem okolju EC2, ki je tudi srce AWS[10]. Uporabnik lahko EC2 uporablja konstantno ali pa na zahtevo, enako kot funkcije Lambda. Za uporabo EC2 je treba kreirati Amazon Machine Image – AMI, ki se mu določijo specifikacije strojne opreme, operacijski sistem in vse želene aplikacije. Nato teče na Amazonovem oblaku, kar pomeni, da se ni treba ukvarjati s strojno opremo, posodobitvami in podobnimi sistemsko-administrativnimi deli. Uporabnik se na virtualno napravo poveže z dodeljenimi zasebnimi in javnimi ključi. Specifikacije strojne opreme lahko po želji nadgrajujemo ali degradiramo.

AWS Lambda

AWS Lambda[5] je dogodkovno vodena oblačna platforma, ki jo ponuja Amazon. V izbranem programskem jeziku uporabnik napiše anonimno funkcijo

Lambda, za katero nato določi prožilce. Tako lahko spišemo funkcijo, ki se izvede na HTTP-zahtevek ali na želene dogodke, kot je na primer objava na želeno temo SNS, naložitev datoteke na S3, registracija novega uporabnika v nabor uporabnikov (angl. *user pool*) AWS Cognito in še mnogo več. Funkcije Lambda se, v nasprotju z navadnimi strežniškimi sistemi, ne izvajajo stalno. Funkcija je v čakanju, dokler je ne pokliče želeni dogodek. Amazon funkcijo izvaja sam v svojem oblačnem servisu EC2.

Tako se v kombinaciji z vmesnikom API lahko uporablja za strežnik REST, ki ne porablja virov, ko ni HTTP-zahtevkov. API gateway namreč ob posameznem HTTP-zahtevku pokliče pripadajočo funkcijo Lambda, ki naredi, kar si je uporabnik zamislil, nato pa vrne HTTP-odgovor. Tako nam omogoča plačilo po porabi, funkcija Lambda se namreč obračuna na vsakih 100 ms (zaokroženo navzgor) porabljenega procesorskega časa. Prav tako ima zelo dobro integracijo avtorizacije AWS Cognito, ta namreč v glavo zahtevka doda vse ustrezne podatke za avtorizacijo, ki jih lahko funkcija Lambda trivialno prebere. Funkcija AWS Lambda prejme tri poimenovane parametre:

- **event** – objekt, ki vsebuje podatke, kot so glava in telo HTTP-zahtevka, vir dogodka (ali je bil na primer HTTP- ali SNS-prožilec);
- **context** – objekt, ki vsebuje podatke o izvajanju funkcije, kot so trenuten čas izvajanja, njena interna identifikacijska številka, podatki o avtorizaciji Cognito;
- **callback** – funkcija, ki se pokliče na koncu izvajanja z objektom za HTTP-odgovor (koda odgovora, telo in glava).

Kot je bilo že navedeno, smo za izdelavo zalednega sistema uporabili funkcije Lambda, ki jih je prožil vmesnik API glede na ustrezne dogodke. Napisali smo tudi dve funkciji Lambda, ki jih proži SNS. Prva je **funkcija za evalvacijo pravil**, ki se proži, ko se:

1. sredstvu, ki je dodeljeno enemu ali več pravilom, dodeli vrednost atributa;

2. sredstvu, ki je dodeljeno enemu ali več pravilom, spremeni vrednost atributa;

3. pravilu dodelijo pogoji;

4. pravilu dodelijo sredstva.

Za primera št. 1 in 2 se evalvirajo vsa pravila, ki imajo to sredstvo dodeljeno, vendar se ne evalvirajo vsa dodeljena sredstva, marveč le tisto, ki se mu je spremenila/dodelila vrednost. Za primer št. 3 se za to pravilo evalvirajo vsa sredstva. Za primer št. 4 se za želeno pravilo evalvirajo na novo dodeljena sredstva.

AWS Lambda ima v resnici zelo zapleten cenik. Uporabnik dobi določeno število brezplačnih sekund, ki se manjša glede na alocirano velikost pomnilnika. Ko se porabijo brezplačne sekunde, se na isti način določi cena na 100 ms. Tako so funkcije z več porabljenega pomnilnika na 100 ms dražje.

Cognito

Cognito[4] je Amazonov servis, ki ponuja avtentikacijo, avtorizacijo in omogoča upravljanje uporabnikov. Ker je del AWS, ima zelo močno integracijo z drugimi servisi, zato je to večinoma najpreprostejša rešitev za registracijo uporabnikov. Ločen je na dva dela, na nabor uporabnikov in nabor identitet (angl. *identity pool*). Nabor uporabnikov omogoča dejansko registracijo uporabnika v sistem. Pri ustvarjanju nabora uporabnikov lahko določimo, katere atribute mora posamezen uporabnik imeti, kako se uporabniki potrdijo (prek e-pošte ali SMS-sporočila) in ali želimo uporabljati večfaktorsko avtentikacijo. Določamo tudi dolžino in zapletenost gesla. Nabor identitet uporabniku da žeton za dostop, lahko je integriran v nabor uporabnikov ali pa se uporablja samostojno, na primer za omejen dostop neregistriranih uporabnikov. Sami smo uporabili oba naenkrat, saj neregistriranim uporabnikom nismo želeli pustiti dostopa do sistema, ker bi to pomenilo nesmotrno porabo sredstev podjetja.

Uporabnike v naboru uporabnikov nato registriramo prek lastnega obrazca, podatke pa pošiljamo tako, da se povežemo na Cognito API z identifikacijsko številko svojega ustvarjenega nabora uporabnikov. Ker je API odprt, je do njega mogoče dostopati prek zalednih sistemov in pročelnih delov sistema, treba je le imeti ustvarjen nabor uporabnikov na AWS.

Cognito tudi sinhronizira podatke o uporabnikih. Po registraciji, avtentikaciji, potrditvi e-poštnega naslova in drugih dogodkih lahko prožimo poljubne funkcije Lambda. To je lahko uporabno, če želimo še hraniti uporabnike v bazi, velikokrat je poizvedovanje po bazi preprostejše kot po Cognitu. Poleg vsega omogoča prijavo prek Googlovega, Facebookovega in Amazonovega računa.

Število mesečno aktivnih uporabnikov	Cena na uporabnika
Prvih 50.000	Brezplačno
Naslednjih 50.000	0,00550 USD
Naslednjih 900.000	0,00460 USD
Naslednjih 9.000.000	0,00325 USD
Več kot 10.000.000	0,00250 USD

Iz cenika je razvidno, da servis ni zelo drag, še posebej je ugoden v razponu do 50.000 mesečno aktivnih uporabnikov, kar v našem sistemu ne bo doseženo še nekaj časa.

Simple Notification Service – SNS

SNS[7] je servis za sporočanje publish/subscribe ter pošiljanje potisnih obvestil in SMS-sporočil na mobilne naprave. Zaradi arhitekture publish/subscribe se lahko z njim izvajajo funkcije zunaj glavnega toka programa, če se na primer nekaj shrani v bazo, pa je to treba še iz nekega razloga interno sprocesirati, lahko uporabniku vrnemo HTTP-odgovor, procesiranje pa se opravi v funkciji, ki je naročena na temo, na katero pošiljamo ustrezne podatke. Sami smo SNS uporabili, tako da smo ustvarili dve temi. Na eno se pošiljajo identifikacijske številke pravil in sredstev v skladu z zgoraj ome-

njenimi primeri evalvacije pravil. Funkcija, ki je naročena na temo, podatke dobi, evalvira in v primeru pozitivne evalvacije pošlje identifikacijsko številko pravila, sredstva in uporabnika na drugo temo. Na to temo je naročena funkcija, ki iz zgoraj omenjenih identifikacijskih števil dobi pravilo, sredstvo in uporabnikove podatke. Nato z uporabo SES uporabniku pošlje obvestilo prek e-pošte.

Med možnostmi za nadgradnjo je seveda uporaba SNS še za pošiljanje potisnih obvestil in SMS-sporočil, vendar je v tem primeru treba od uporabnika pri registraciji v Cognito zahtevati tudi telefonsko številko. SNS za uporabo publish/subscribe ne stane nič, plača se le čas izvajanja funkcij Lambda, ki so naročene na teme.

Simple Email Service – SES

SES[6] je servis, ki omogoča pošiljanje e-pošte. Je zelo preprost, treba je zgolj overiti e-naslov, ki je lahko naveden kot pošiljatelj, in že deluje. Pri tem nam ni treba skrbeti za nič, saj SES sam poskrbi, da bo e-poštno sporočilo prispelo do prejemnika. Omogoča tudi dodajanje priponk in uporabo HTML-predlog za oblikovanje videza e-poštnega sporočila.

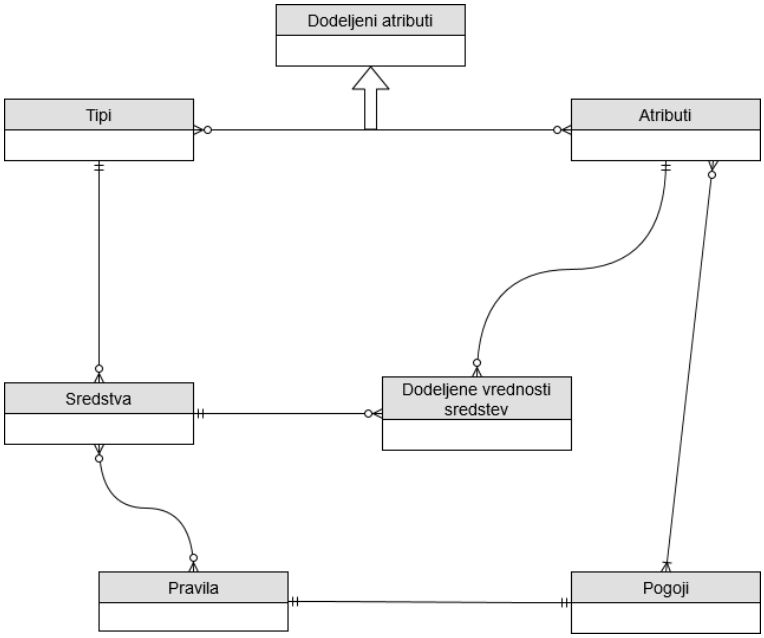
Pri testiranju smo morali e-naslove uporabnikov tudi overiti kot naslove pošiljateljev, saj je bil račun v načinu peskovnika, kjer je vključenih 1000 brezplačnih e-poštnih sporočil na mesec. Ta način je implementiran za testiranje ter zaščito pred prevarami in zlorabo.

Kaj?	Cena e-pošte na mesec	Dodatni stroški na mesec
Pošiljanje e-pošte iz aplikacije na EC2	Prvih 62.000 sporočil brezplačno, nato 0,10 USD na 1000 sporočil	0,12 USD na GB priponk
Pošiljanje e-pošte iz drugih aplikacij	0,10 USD na 1000 poslanih sporočil	0,12 USD na GB priponk
Prejemanje e-pošte v aplikaciji na EC2	Prvih 1000 prejetih sporočil je brezplačnih, nato 0,10 USD na 1000 prejetih sporočil	0,09 na delček (angl. *chunk*) sporočila USD
Prejemanje e-pošte iz drugih aplikacij	0,10 USD na 1000 prejetih sporočil	0,09 na delček sporočila

Že sam cenik uporabnike spodbuja k uporabi funkcij Lambda ali gostovanje zalednega sistema na EC2, saj nam 'podari' 1000 poslanih in prejetih e-sporočil na mesec.

3.2 Podatkovna baza

Pred izbiro podatkovne baze je smiselno izrisati entitetno-relacijski (ER) diagram baze.



Slika 3.1: Diagram ER

Na prvem mestu je vedno izbira med relacijsko in nerelacijsko podatkovno bazo. Kot je razvidno iz diagrama, ima naš sistem več relacij, zato so relacijske podatkovne baze precej primerne za naš primer. Ker je sistem gostovan v oblaku, smo pregledali več oblačnih variacij baz. Pri relacijskih bazah se trenutno najbolj uporablja PostgresSQL, ima tudi svojo oblačno različico na Googlovem oblaku. AWS tudi ponuja uporabo PostgresSQL-a z njihovim servisom Amazon Relational Database Service.

Uporaba nerelacijske podatkovne baze je velikokrat primerna za shranjevanje dokumentov in nasploh med seboj neodvisnih podatkov. Med podatki že zaradi njihove narave velikokrat obstajajo relacije, v praksi še nismo srečali kompleksnejšega primera, ki ne bi vseboval relacij. Zato so tudi različne nerelacijske podatkovne baze pričele izdelovati svojo obliko tujih ključev. Sami smo izbirali med Amazonovo podatkovno bazo v izvedbah DynamoDB in MongoDB v oblaku Mongo Atlas.

DynamoDB[9] je nerelacijska podatkovna baza, ki jo uporablja tudi sam

Amazon. Namenjena je hitremu izvajanju poizvedb v lastnem poizvedbenem jeziku. Kot večina Amazonovih servisov ima tudi ta brezplačno različico do 25 GB porabljenih podatkov ter 200 milijonov operacij pisanja in branj na mesec.

MongoDB[22] je ena izmed prvih odprtokodnih nerelacijskih podatkovnih baz, nedolgo nazaj je ista ekipa tudi začela ponujati MongoDB v oblaku, poimenovano Mongo Atlas[13]. Ta ima tudi brezplačno obliko, ki dodeljuje 512 MB prostora in neomejeno število operacij na mesec. MongoDB ima svoj lastni poizvedbeni jezik, ki je precej močan, saj omogoča tudi uporabo tujih ključev. Prek operacije $lookup izvede JOIN. Ponuja pa še veliko več, zelo močan je na primer agregacijski cevovod, v katerega nanizamo poljubne operacije, pri čemer rezultat prejšnje operacije postane vhod naslednje operacije. Z novejšimi različicami MongoDB ponuja tudi naročanje na dogodke iz baze, tako lahko vemo, kdaj se neki dokument pobriše iz baze, vstavi vanjo ali spremeni. V podjetju smo uporabljali MongoDB na projektu iz mobilnosti, ker ima zelo močno podporo za geoprostorske poizvedbe. Ponuja operacijo geoNear, ki za vsak element poračuna razdaljo do neke točke in jih uredi po razdalji. Ima tudi veliko operacij za določanje, ali je neka točka v geolokacijskem poligonu, krogu okrog neke točke. Lahko se tudi poljubno definira neki prostor, v katerem določimo poligone, kjer ne velja, da se točka nahaja v prostoru (kot da bi delali nekakšen kolobar). Pri vsem tem tudi upošteva ukrivljenost Zemlje.

3.2.1 Izbira podatkovne baze

Na koncu smo izbrali MongoDB Atlas. DynamoDB ima sicer več prostora v brezplačni različici, vendar menimo, da raba DynamoDB ni intuitivna. Zgradba baze ni shema SQL, ali pa objekt JSON kot v MongoDB, ampak je popolnoma sama svoja. Poizvedbeni jezik tudi ni najbolj eleganten, bližji nam je SQL in Mongo, zato je bil DynamoDB izločen prvi.

Kot je razvidno iz ER-diagrama, ima sistem veliko relacij, zato bi bila zelo primerna uporaba PostgresSQL. Vendar ne Amazon ne Google ne ponujata

brezplačne različice. Poleg tega SQL, v nasprotju z MongoDB, ne ponuja velike količine geoprostorskih funkcionalnosti.

Glavni razlog za izbiro so bile dosedanje izkušnje z MongoDB, saj sta nam njegov poizvedbeni jezik in upravljanje baze Mongo dobro znana. Pomemben razlog je bil tudi, da njegov poizvedbeni jezik ponuja veliko operacij nad geolokacijskimi podatki, ki so tudi glavna motivacija za izdelavo sistema. Z najnovejšo različico MongoDB bi lahko veliko pravil izvajali z naročanjem na spremembe v bazi. Tako bi lahko za pravilo, v skladu s katerim se uporabnik obvesti en teden pred želenim datumom, ustvarili objekt v bazi, ki se izbriše točno en teden pred želenim datumom. Ko bi ta objekt šel iz baze, bi uporabniku elegantno poslal obvestilo. Ta možnost sicer še ni podprta v brezplačni različici, je pa že podprta v najcenejši plačljivi različici. Pogoje pravil smo na začetku hoteli ustvariti tako, da bi se prevedla v poizvedbeni jezik Mongo, ki bi se nato izvajal neposredno nad sredstvi in njihovimi dodeljenimi vrednostmi. Zaradi dobrega poznavanja jezika in odlične podpore za geoprostorske poizvedbe pri pisanju nismo imeli nobenih težav.

Na koncu se je izkazalo, da bi bil sistem s takšnim načinom izdelave pogojev preveč odvisen od specifične baze. Če bi kdaj želeli migrirati na SQL ali katero drugo nerelacijsko bazo, bi bilo treba prepisati celoten sistem. Pogoje smo nato preverjali programsko v funkcijah Lambda. Glede na to in na dejstvo, da še ni bilo mogoče uporabljati najnovejše različice MongoDB, bi lahko mirno izbrali PostgresSQL, a ker je sistem že deloma funkcioniral, smo se odločili, da bi veliko dela s prepisom prineslo premalo koristi. Uporabljali smo namreč knjižnico Mongoose, ki omogoča izdelavo shem za MongoDB ter doda validacijo, ročno dodajanje indeksov in tudi tuje ključe

3.2.2 Mongoose

Mongoose[14] je knjižnica za Node.js, ki skrbi za neprekinjeno povezavo z bazo MongoDB, validacijo in ustvarjanje shem za posamezno zbirko podatkov. Je zelo uporaben, saj je pisanje vseh teh funkcij na roke lahko kar mučno,

posledica česar je grša koda. MongoDB sam ne podpira shem, a perečo težavo reši Mongoose, kjer le definiramo shemo s posameznimi atributi, ki jim določimo tipe, obveznost, privzete vrednosti in validatorje po meri. Tako lahko definiramo najrazličnejše sheme in z validacijo preprečimo neujemanje podatkovnih tipov. Omogoča tudi velik nabor vmesne programske opreme za posamezno shemo, ki se proži ob dogodkih v naslednjem vrstnem redu:

1. pred validacijo,
2. po validaciji,
3. pred shranjevanjem,
4. po shranjevanju,
5. pred branjem,
6. po branju.

V posameznih funkcijah lahko tudi izvemo, katera polja v objektu so bila spremenjena in ali je objekt na novo ustvarjen ali že obstoječ. Tako je implementacija polja, ki hrani datum nastanka, in polja, ki hrani datum spremembe, zelo preprosta. Prav tako lahko z njimi po potrebi spreminjamo vrnjene objekte. Mongoose je zaradi vseh funkcionalnosti ena izmed najbolj razširjenih knjižnic Node.js.

3.3 Evalvacija pravil

Evalvacija pravil se je izvajala programsko v funkcijah Lambda v programskem jeziku Javascript. Pred izdelavo lastnega algoritma za evalvacijo smo preverili, ali obstaja že kakšna implementacija. Po krajšem iskanju smo našli odprtokodno knjižnico *json-business-rules-engine*[11]. Ta ima že implementirano poljubno globoko in široko definiranje pravil. Hkrati ponuja pogon za evalvacijo, pri katerem lahko ob pozitivni evalvaciji prožimo dogodke. Z

izdelavo posebnih dogodkov se nismo ukvarjali, saj smo želeli le vedeti, ali je bilo sredstvo evalvirano kot ustrezno ali ne.

Knjižnica privzeto že ponuja nekaj osnovnih operatorjev, kot so enakost, neenakost, večje, manjše ali enako, vsebovanost v tabeli itd. Hkrati ponuja preprosto možnost dodajanja operatorjev po želji, tako je bilo dodajanje operatorjev za geolokacijske in datumske vrednosti trivialno. Veriženje operatorjev se izvaja podobno kot v naši gramatiki, ključ 'all' pomeni, da morajo biti vsi pogoji, navedeni v tabeli, ovrednoteni za pravilne, če hočemo, da je tudi ta. Tako funkcionira kot pogoj in. Kot pogoj ali pa je ključ 'any', pri katerem je zadostno, da se kot pravilno evalvira le eno izmed pravil, navedenih v tabeli. Te pogoje lahko gradimo poljubno široko in globoko, saj ima lahko posamezen ključ 'all' ali 'any' v tabeli še en sestavljen pogoj, ki ga spet začnemo z relacijo 'all' ali 'any'. Za lažje razumevanje delovanja knjižnice si poglejmo kratek primer v kodi Javascript.

```
1   const pogoji = {
2     any: [{
3       fact: "stanjeBaterije",
4       operator: "equal",
5       value: 100
6     }, {
7       all: [{
8         fact: "statusBaterije",
9         operator: "equal",
10        value: "Nova"
11      }, {
12        fact: "lokacijaBaterije",
13        operator: "vGeoRadiju",
14        value: {
15          sirina: 45,
16          dolzina: 16,
17          radij: 1000
18        }
19      }]
20    }]
21  };
```

```
22  const pogon = new Engine();
23  // dodamo nov operator vGeoRadiju
24  pogon.addOperator("vGeoRadiju", (dejanskaVrednost,
        vrednostPogoja) => {
25    // koda, ki preveri, ali se dejanska vrednost nahaja v
          radiju
26    return dejanskaVrednost.sirina,
          dejanskaVrednost.dolzina v krogu
          vrednostPogoja.sirina, vrednostPogoja.dolzina z
          radijem vrednostPogoja.radij;
27  });
28  pogon.add({
29    conditions: pogoji,
30    event: {
31      type: "Uspesna evalvacija"
32      params: {
33        message: "Baterija ustreza pogojem!"
34      }
35    }
36  });
37
38  const baterija = {
39    stanjeBaterije: 60,
40    statusBaterije: "Nova",
41    lokacijaBaterije: {
42      sirina: 45.1,
43      dolzina: 16.05
44    }
45  };
46
47  pogon
48    .run(baterija)
49    .then((dogodki) => {
50      if (dogodki.length > 0) {
51        // baterija ustreza pogojem
52      } else {
53       // baterija ne ustreza pogojem
54      }
```

```
55      });
56    }
```

Zgornja koda evalvira pravilo kot ustrezno, če ima baterija vrednost atributa 'stanjeBaterije' 100, ali če ima vrednost atributa 'statusBaterije' enako 'Nova' in se nahaja v 1000-metrskem radiju okrog geolokacije s širino 45 in dolžino 16. Za to tudi ustvari nov operator 'vGeoRadiju', ki preveri, ali se podana vrednost nahaja v radiju okrog geolokacije pogoja. Nato na koncu, ko se kliče pogon.run, dobimo nazaj tabelo proženih dogodkov. Ko smo ta tako imenovani 'event' dodali v pogon, mu dodamo atribut 'type' in 'message', ki nam lahko pomaga razločevati med različnimi uspešnimi dogodki. Za naš primer je zadostno le, da se proži ta edini dogodek, če se proži, pomeni, da je bila evalvacija uspešna. Če se ni prožil noben dogodek (drugih sploh ni), baterija ne ustreza pogojem.
Knjižnica ima podobno sintakso, kot smo si jo zamislili mi, deluje precej preprosto in ponuja programsko reševanje evalvacije pravil, zato smo jo uporabili kot jedro dela za evalvacijo pravil.

3.4 Nadzorna plošča

Odločili smo se za izdelavo nadzorne plošče, ki se uporablja kot vmesnik za interakcijo s sistemom. Uporaba HTML s knjižnico jQuery dandanes ne zadošča več, uporabniški vmesniki morajo imeti dobro uporabniško izkušnjo in nekaj dodatnih, a intuitivnih funkcionalnosti. Začele so se uporabljati enostranske spletne strani, kjer s kodo spreminjamo videz in vsebino strani, namesto da se za vsako novo stran zahteva nova HTML-datoteka s strežnika. Tako prihranimo veliko klicev na strežnik in pospešimo izrisovanje spletne strani, saj ni treba čakati na strežnikov odziv, da izrišemo želeno spletno stran. Velikokrat je priročno tudi hranjenje podatkov iz strežnika v kodi v spremenljivki oziroma shrambi (angl. *store*). Tako imajo vsi deli kode možnost branja in/ali pisanja v shrambo, nam pa ni vedno treba klicati strežnika za pridobitev podatkov.

Prva večja knjižnica, ki je poskušala ponuditi rešitev za to z arhitekturo MVC, je bila AngularJS, in sicer leta 2010. Angular je bila v bistvu ogrodje za izdelavo spletne strani, saj vsebuje že veliko lastnih gradnikov, za katere ni treba iskati in vključevati novih knjižnic. Čez leta se je nato razvilo več knjižnic, kot so ReactJS, Angular (popolna nadgradnja Typescript za AngularJS), Vue.js, Ember.js in mnogo več. Najaktualnejše so ReactJS[18], Angular[1] in Vue.js[21], te so bile tudi glavne kandidatke za knjižnico, ki jo bomo uporabili za izdelavo nadzorne plošče.

Razlikujejo se predvsem po tem, da sta ReactJS in Vue.js knjižnici, Angular pa ogrodje za izdelavo spletnih strani. Tako je z Angularjem lažje pričeti, a je manj fleksibilen, pri drugih dveh lahko namreč brez težav nadomestimo posamezen gradnik z drugo knjižnico (na primer spustni meni). ReactJS in Vue.js imata nižjo vstopno točko, oba funkcionriata kot knjižnici Javascript z nekaj drugačnostmi. Izdelava spletne strani Hello world! je preprosta, kodo je mogoče pisati v navadnem Javascriptu, zato lahko vsak programer kar hitro prične razvijanje. Angular pa kot ogrodje vsebuje veliko lastne namenske kode nad Javascriptom, ki je sicer zelo močna, ampak se z njo občutno zviša točka vstopa.

Zaradi pomanjkljivih izkušenj z razvijanjem pročelnih sistemov, omejenih na nekaj dela z ReactJS in AngularJS, smo želeno kodo spisali veliko hitreje v Reactu, saj smo lahko pisali tako rekoč navaden Javascript. Na podlagi tega smo Angular takoj izključili, poleg tega je Angular treba pisati v Typescriptu, ki bi se ga sicer želel naučiti, vendar izdelava diplomske naloge ni bila primerna priložnost za to.

Knjižnici Vue.js in ReactJS sta si precej podobni, obe se zanašata na virtualen DOM (Document Object Model – vrsta razčlenitve HTML-strani na posamezen element), obe sta tudi knjižnici za izdelavo vmesnikov, preostala opravila, kot je na primer usmerjanje (angl. *routing*), prepustita drugim knjižnicam, za kar seveda večinoma zagotavljata močno podporo. V času

odločitve je bil ReactJS veliko bolj razširjen, ponujal je veliko dokumentacije in vodičev za izdelavo spletnih strani z uporabo Reacta, Vue.js pa je bil v porastu. Poleg tega so bile dotedanje skromne izkušnje z Reactom precej navdušujoče, zato smo se odločili za ReactJS. Med nastajanjem naloge je Vue.js že precej zrasel, a je ReactJS še vedno večji in bolj razširjen. ReactJS ima tudi različico React Native, namenjeno izdelavi mobilnih aplikacij, ki je zelo podobna ReactJS. Razlika je v tem, da je v ReactJS mogoče pisati z navadno HTML-kodo, v React Native je mogoče pisati le v JSX, Reactovi 'združitvi' Javascripta in HTML-ja, ki navadne HTML- elemente pretvori v elemente React, da lažje komunicirajo s shrambo in drugimi knjižnicami.

Kot je bilo že nakazano zgoraj, je bilo treba uporabiti nekaj zunanjih knjižnic, a o tem več po opisu ReactJS in JSX.

3.4.1 ReactJS

Kot je bilo navedeno zgoraj, je to knjižnica za razvoj uporabniških vmesnikov z uporabo jezika Javascript. Ustvaril jo je Facebookov inženir Jordan Walke. Reactov zato razvija, vzdržuje in nadzira Facebook kot podjetje samo. Deluje po načelu stanja in komponent, vsak gradnik je lahko svoja komponenta, ki hrani in bere podatke iz stanja (angl. *state*). Komponenta ima svojo metodo render, ki služi za izris elementov, to so večinoma HTML-elementi oziroma HTML-elementi, nadgrajeni z uporabo zunanjih knjižnic. Logika komponent je spisana v Javascriptu, ne v HTML, zato je v stanju mogoče hraniti bogate podatke. Zagotavlja enosmerni tok podatkov, posamezna komponenta vedno dobi lastnosti (angl. *props*) samo za branje, če namreč dobi spremenjene lastnosti, se znova pokliče metoda za izris.

Ponuja tudi virtualen DOM, ki ga React upravlja sam. Programer tako kodo napiše, kot da se vedno na novo izriše celotna stran, React pa poskrbi, da se na novo izrišejo le komponente, na katere so vplivale spremembe lastnosti.

Posamezna komponenta ima tudi življenjski cikel:

- shouldComponentUpdate je funkcija, ki jo pokličemo z booleanom, in povemo komponenti, ali mora ponovno pognati svoj izris;

- componentWillMount je funkcija, ki se izvede, tik preden se komponenta doda v DOM;
- componentDidMount se izvede takoj po tem, ko je komponenta dodana v DOM;
- render je funkcija za izris, ki jo mora imeti vsaka komponenta, vsebuje tudi kodo JSX.

JSX

JSX je sintaktični podaljšek navadnega Javascripta. Programerjem omogoča kombiniranje kod HTML in Javascript brez večjih težav, na primer:

```
1    const element = <h1>Zdravo, svet!</h1>;
```

Ta del kode v spremenljivko element shrani naslov tipa h1 z vsebino 'Hello world!'. Nato bi v metodi render le poklicali ime spremenljivke, na primer:

```
1  render() {
2    return (
3      element
4    );
5  }
```

Tako z JSX lahko pišemo HTML-značke, vključno s CSS-jem, in jih shranjujemo v spremenljivke Javascript. To je izredno močno orodje, ki nam zelo olajša izrisovanje elementov glede na neke pogoje. Omogoča nam tudi ločevanje izrisa po funkcijah, tako lahko izrišemo elemente glede na parametre funkcije.

Še ena posebnost ReactJS in JSX je uporaba lastnosti in stanja. Ko React zazna uporabniško definirano komponento, ji vhod pošlje zapakiran v lastnostih samo za branje.

```
1  function Welcome(props) {
2    return <h1>Zdravo, {props.name}</h1>;
3  }
4  const ime = "Kerry";
```

```
5   const element = <Welcome name={{ ime }} />;
6
7   render() {
8     return(
9       element
10    );
11  }
```

Tako lahko s spremenljivkami pokličemo Reactove komponente. To je izredno močno orodje, ki nam zelo preprosto omogoča pogojni izris in nasploh neposredno branje iz kompleksnejših objektov.
Izdelovanje HTML-obrazcev je tudi zelo preprosto, saj imajo komponente za vnos React privzeto metodo onChange, ki skrbi za spreminjanje vrednosti vnosa. Nato ima posamezna komponenta atribut 'value', v katerem hrani vrednost. Za lažje razumevanje ponazarjamo uporabo s preprostim primerom v kodi.

```
1   class Obrazec extends React.Component {
2     handleChange(event) {
3       console.log("Nova vrednost je: ", event.target.value
            );
4     }
5
6     handleSubmit(values) {
7       console.log("Posiljam vrednost obrazca: ", values);
8       // izpise JS objekt { test: <vrednost> }
9     }
10
11    render() {
12      return (
13        <form onSubmit={this.handleSubmit}>
14          <input type="text" name="test" onChange={
                this.handleChange } />
15          <button type="submit" text="Poslji" />
16        </form>
17      );
18    }
```

```
19    }
```

Tako lahko vse vnesene vrednosti spreminjamo po želji, omogočeno je tudi spreminjanje vrednosti takoj po vnosu.
JSX je torej izredno močno orodje, ki pa je hkrati zelo intuitivno za uporabo. Vsakemu HTML-elementu se zelo preprosto priredi veliko poljubnih funkcij za različne dogodke.

React redux

Izdelovanje enostranskih dinamičnih spletnih strani ni preprosta naloga, kaj hitro namreč lahko privede do slabe kode, polne stavkov if-else. Velikokrat je treba tudi dobiti podatke z nekega strežnika – ali pa jih poslati in ko jih strežnik prejme, tudi želimo uporabnika o tem obvestiti. V ReactJS je vse to mogoče, rešitev pa ni vedno nujno lepa. Spremembe se velikokrat verižno povezujejo, kar lahko privede do sprememb nepričakovanih komponent. Redux[19] vpelje spreminjanje spremenljivke stanje prek asinhronih akcij. Akcija je funkcija, ki prejme parametre in nato izvede spremembo stanja. Vsaka sprememba stanja mora biti opisana z akcijo. Namesto da bi neposredno spreminjali stanje, tako pokličemo akcijo, ki ga spremeni namesto nas. S tem se poenostavi razhroščevanje, ker se vse spremembe dogajajo na enem mestu. Akcija pošlje objekt, ki vsebuje tip in poljubno število preostalih atributov, vendar je dobra praksa vse zapakirati v en ključ. Sam sem ga poimenoval 'payload'.

```
1    const action = {
2      type: "Test",
3      payload: podatki
4    };
5    dispatch(action);
```

Na akcije čakajo deli sistema, ki se jim reče reducer.Reducer je cilj vseh akcij, sami pa s stavki if-else ali switch določimo, katere akcije sploh upošteva.Nato reducer spremeni svoje stanje, tako da naredi kopijo in jo nato spremeni.

Tako lahko lažje spremljamo zgodovino sprememb in tudi zavrnemo nove spremembe stanja. Primer kode:

```
1   const zacetniState = { vrednost: null }; // pomozna
       spremenljivka, ki definira kak je state na zacetku
2   function testReducer(state = zacetniState, action) {
3     switch (action.type) {
4       case "Test":
5         return Object.assign({}, state, {
6           vrednost: action.payload
7         });
8         // Object.assign v nov objekt {} shrani state,
             nato pa posodobi vrednost na action.payload
9     }
10  }
```

Sprememba stanja v reducerju spremeni glavno shrambo podatkov. Ko se pomnilnik spremeni, se pokliče izris vseh trenutno aktivnih komponent. Shramba funkcionira kot enotno skladišče podatkov, njen posamezni atribut je po navadi en reducer. Tako lahko za izdelavo shrambe kombiniramo več reducerjev. Reducerje lahko definiramo kar glede na posamezno komponento ali pa glede na tematsko povezan sklop komponent. Komponenta se nato odzove na spremembo v shrambi za ta posamezni reducer. Spreminjanje podatkov v reducerju je slaba praksa, to je bolje storiti v akciji.

Z uporabo redux je vzpostavljen enosmerni tok podatkov, ki jih lahko spremljamo vse od klica akcije do spremembe v reducerju in prihoda odziva do komponente. Konkreten primer uporabe je naslednji tok:

1. Uporabnik vnese svoje ime v komponenti VnosImena in klikne gumb ‘Pošlji’.

2. Pokliče se akcija, ki njegovo ime shrani na strežnik. Ko strežnik odgovori, akcija pokliče dodelitev (angl. *dispatch*) tipa ‘spremembaImena’ in vsebino (angl. *payload*), ki je uporabnikovo vneseno ime.

3. Reducer imena 'uporabnik' dobi akcijo tipa 'spremembaImena' in njeno vsebino zapiše v svoje stanje pod atribut 'ime'.

4. Shramba vidi spremembo stanja za reducer uporabnik in atribut ime, zato pokliče vnovičen izris komponente VnosImena.

5. Komponenta VnosImena dobi spremenjeno shrambo, iz *store.uporabnik.ime* dobi vneseno ime in izpiše alert 'Dobrodošli <vneseno ime>!'.

Druge pomembne knjižnice

React Bootstrap

React Bootstrap[15] je reimplementacija izredno priljubljene knjižnice Bootstrap z uporabo Reacta. Bootstrap je orodje, ki programerjem omogoča uporabo oblikovno nadgrajenih HTML-elementov. Ustvarjen je ovoj okrog navadnih HTML-elementov, nadgrajen je videz, dodanih je še nekaj funkcionalnosti. Tako imajo lahko gumbi ne le lepšo obliko, ampak tudi možnost za na primer delovanje kot spustni meni več gumbov. Navadnim inženirjem, kakršen sem sam, je s tem zelo olajšana izdelava vizualno privlačnih spletnih strani, ne da bi se bilo treba veliko ukvarjati s stiliziranjem CSS.

Redux form

Redux form[20] je knjižnica, ki služi kot ovoj okrog navadnih HTML-obrazcev. Deluje po načelu Reduxa, vrednost obrazca je hranjena v objektu stanje, za spremembe vrednosti se pokliče akcija. Vsak obrazec ima svoj reducer, ki kot pri navadnem reduxu čaka na akcije in spreminja stanje obrazca. Polja obrazca ponujajo dodatne zmožnosti, kakršno je klicanje funkcij ob dogodkih, ko na primer polje dobi žarišče (angl. *focus*) in ga izgubi. Ponuja tudi možnost formatiranja in normaliziranja vrednosti, takoj ko se ta spremeni. Implementiran je tudi preprost način za asinhrono validacijo obrazcev z že vgrajenim prikazom napak.

React router

Moja nadzorna plošča je po načelu Reacta enostranska spletna stran. To

pomeni, da poti v URL-ju ne delujejo enako kot na spletnih straneh, ki prek URL-ja kličejo strežnik. Treba je opredeliti, katere komponente naj se izrišejo za kateri URL. React router[16] je knjižnica, ki to omogoča zelo preprosto, hkrati pa v spletno stran prinese zgodovino predmeta (angl. *object history*). V zgodovini se hranita URL trenutne komponente in stanje. Vanjo lahko dodajamo in iz nje odstranjujemo nove dele spletne strani. Omogoča tudi klicanje novega prikaza z vnaprej določenim stanjem, tako ni treba klicati izrisa komponente iz trenutne komponente, ampak v zgodovino dodamo URL, na katerem se nahaja druga komponenta. Priključimo še stanje, v katerem lahko elegantno posredujemo najnujnejše podatke za izris druge komponente. Tako lahko preprosto rešimo primer, ko želimo v tabeli klikniti na objekt, ki ga želimo posodobiti. Namesto da bi to hranili v storu, pošljemo v stanju pri dodajanju izrisa komponente za posodobitev v zgodovino.

React table

Namesto izrisovanja HTML-tabele smo uporabili knjižnico React table[17]. Ta za obliko uporablja Bootstrap in se samodejno prilega velikosti okna. Preprosto sprejme tabelo podatkov kot vhod. Opredelimo le imena stolpcev in pot do želenega podatka pri posameznem objektu. Privzeto že podpira razvrščanje po številkah, datumih in nizih znakov. V celice tabele lahko dajemo tudi gumbe in vse JSX in HTML-elemente. Omogoča tudi razdeljevanje celic na več podcelic, ima že vgrajeno paginacijo. Ponuja že filtriranje po enakosti za številke in nize znakov, omogočena je tudi definicija poljubnih filtrov. Zanje je treba spisati videz vmesnika za vnos vrednosti in metode, po katerih naj tabela filtrira vsako vrstico.

AWS Amplify

Je Amazonova uradna knjižnica za mobilne aplikacije in tudi knjižnica za izdelavo pročelnih sistemov[3]. Obstaja tudi različica za ReactJS. Služi kot vmesnik med pročelnim sistemom in zalednim sistemom AWS. Pri inicializaciji knjižnice vnesemo konfiguracijske podatke, v našem primeru tako spletni naslov zalednega sistema in podatke o Cognitu. Sama komunicira s poda-

nim naborom uporabnikov Cognito, da pridobi ustrezne podatke za podpis HTTP-zahtevka na zaledni sistem AWS. Podatke podpiše v skladu z Amazonovim protokolom za podpis zahtevkov aws4. Ta zahteva:

- identifikacijsko številko ključa za dostop do sistema AWS,
- trenutni datum,
- algoritem za izračun podpisa,
- izračunan podpis ključa za dostop na podlagi datuma in podatkov o trenutni seji uporabnika.

Podatke o trenutni seji uporabnika dobi prek žetona seje, ki ga Cognito vrne po uporabnikovi prijavi. Žeton je veljaven eno uro, AWS Amplify sam skrbi za njegovo obnovo, ko poteče. Ponuja preproste rešitve za registracijo, prijavljanje in odjavljanje uporabnikov. Reproduciranje vsega, kar ponuja, zahteva veliko kode in dela, zato je uporaba te knjižnice pri večini izdelav pročelnega sistema za komunikacijo z zalednim sistemom AWS daleč najboljša možnost. Omogoča tudi konfiguracijo klica več različnih vmesnikov API, saj si konfiguracijo shrani in ob klicu na želeni sistem samodejno prebere ustrezne podatke. Poleg tega omogoča pošiljanje potisnih sporočil, vmesnik za GraphQL in mnogo več, mi smo knjižnico uporabljali le za registracijo in prijavo uporabnikov ter seveda podpisovanje HTTP-zahtevkov na naš sistem.

Poglavje 4

Pilotna izdaja sistema

Za testiranje delovanja sistema smo uporabili podatke o električnih avtomobilih iz testnega sistema Avant2Go. Napisali smo preprost vmesnik Node.js med sistemoma, podatke o avtomobilih smo sinhronizirali na pol ure iz sistema Avant2go v naš sistem. Za posamezen avtomobil smo beležili stopnjo napolnjenosti baterije in njegovo geolokacijo. Prav tako smo določili fiktiven datum, kdaj avtomobilu poteče registracija, da smo lahko testirali še časovna pravila. Ustvarili smo pet pravil:

1. Avtomobil je v BTC.
2. Avtomobila ni v BTC.
3. Stanje napolnjenosti baterije avtomobila je nad 70 %.
4. Stanje napolnjenosti baterije avtomobila je pod 70 %.
5. Datum registracije avtomobila je oddaljen manj kot en teden (časovno pravilo).

Ustvarili smo torej štiri pravila glede na spremembe vrednosti in eno časovno pravilo. Časovno pravilo se evalvira vsake pol ure za vsa njemu dodeljena sredstva. Pravilnost izvedbe časovnega pravila smo ocenjevali tako, da smo primerjali datum in uro prejetja e-pošte z dejanskim datumom in

uro registracije avtomobila. To pravilo je delovalo brezhibno, saj obvestil ni pošiljalo dvakrat, vsa so prispela ob prvi polni uri, ki je bila oddaljena manj kot en teden od datuma registracije. Preostala štiri pravila smo namerno oblikovali tako, da bo vsak avtomobil ali v BTC ali pa zunaj njega, ali pa bo imel stanje baterije več kot 70 % ali manj. Tako smo prejemali obvestila na svoj e-naslov in imeli zgodovino sprememb vrednosti. Zaradi ne najboljše zasnove beleženja, ali je bilo obvestilo že poslano, pa se je pojavila **težava sočasnega dostopa**.

4.1 Težava sočasnega dostopa

Težava sočasnega dostopa se pojavi, ko je en dokument iz podatkovne baze hkrati odprt v dveh ali več instancah, ki spreminjata vrednosti atributov tega dokumenta. Če na primer prva shrani spremembe in jih nato želi še druga, ne da bi prej posodobila dokument, se po navadi zgodita dva scenarija:

1. V bazo se shrani dokument z druge instance, ki pa še nima spremenjenih vrednosti iz prve instance, zato se spremembe prve zavržejo.

2. Orodja, kot je Mongoose, hranijo tudi različice dokumenta, tako bi druga instanca dobila vrnjeno napako, ker ima staro različico dokumenta, zato bi se beležile le spremembe iz prve instance.

V našem primeru se je seveda zgodil drugi scenarij, saj smo uporabili Mongoose.

4.1.1 Zakaj se je pojavila težava?

Težava je nastala, ker smo za vsako sredstvo hranili zastavico, če je bilo obvestilo že poslano na pravilu. Tako je ob sinhronizaciji vrednosti dokument odprlo štirinajstkrat, za vsak avtomobil enkrat. Če je bilo treba spremeniti vrednost zastavice, je bilo upoštevanih približno pol sprememb, preostala polovica je prejela napako od Mongoosa zaradi neujemanja različic dokumenta.

Spremembe se niso shranile, posledic česar je bilo dvojno pošiljanje obvestil oziroma se zastavica ni nastavila na neresnično, zato ob naslednji pozitivni evalvaciji obvestilo sploh ni bilo poslano.

Težava je bila tudi posledica tega, da smo evalvacijo pravil izvajali ločeno od glavnega toka programa, uporabljali smo namreč SNS. Poleg tega AWS pri delovanju dosega velike hitrosti, tako je štirinajst posodobitev vrednosti avtomobilov izvedel izredno hitro in se je evalvacija pravil velikokrat prožila vzporedno za več sredstev. Funkcija za evalvacijo je imela dokument odprt večkrat in ga tudi večkrat naenkrat poskušala posodobiti. Časovne razlike shranjevanja so bile približno 15 pikosekund.

4.1.2 Kako rešiti težavo?

Uporabili bi lahko več pristopov k reševanju težave. Lahko bi implementirali čakalno vrsto in evalvacijo vedno izvajali sekvenčno. Rešili bi jo tudi tako, da bi ob prejeti napaki o neujemanju različic dokument še enkrat naložili iz baze in ga poskusili spret shraniti. To bi nato izvajali, dokler ne bi bil končno shranjen. Obe možnosti sta sicer v redu, prva je verjetno boljša in manj nevarna, a smo se odločili za tretjo. Težava je namreč nastala zaradi naše lastne slabe zasnove beleženja zastavic za vsa sredstva na enem pravilu. Rešili smo jo tako, da smo dodali nov gradnik sistema, namenjen beleženju zastavic. Posamezen dokument je kot atribute imel:

- identifikacijsko številko pravila,
- identifikacijsko številko sredstva,
- zastavico, ali je bilo obvestilo poslano.

S tem se je za posamezno pravilo beležila zastavica še posebej za vsako sredstvo. Težava sočasnega dostopa tako sploh ni mogla nastati, saj se ne odpira več en dokument, ampak se za posamezno sredstvo v evalvaciji odpre dokument, za to sredstvo in trenutno pravilo v evalvaciji. Tako je bila hitrost sistema še vedno ista, ni se pa odpiral isti dokument v več sočasnih instancah.

Na zbirko v bazi smo prek Mongoosa tudi dodali omejitev, da mora biti par identifikacijske številke pravila in identifikacijske številke sredstva unikaten. Dva ista para teoretično sicer nista mogoča, vendar se velikokrat izkaže, da je stvar bolje preveriti prevečkrat kot nikoli.

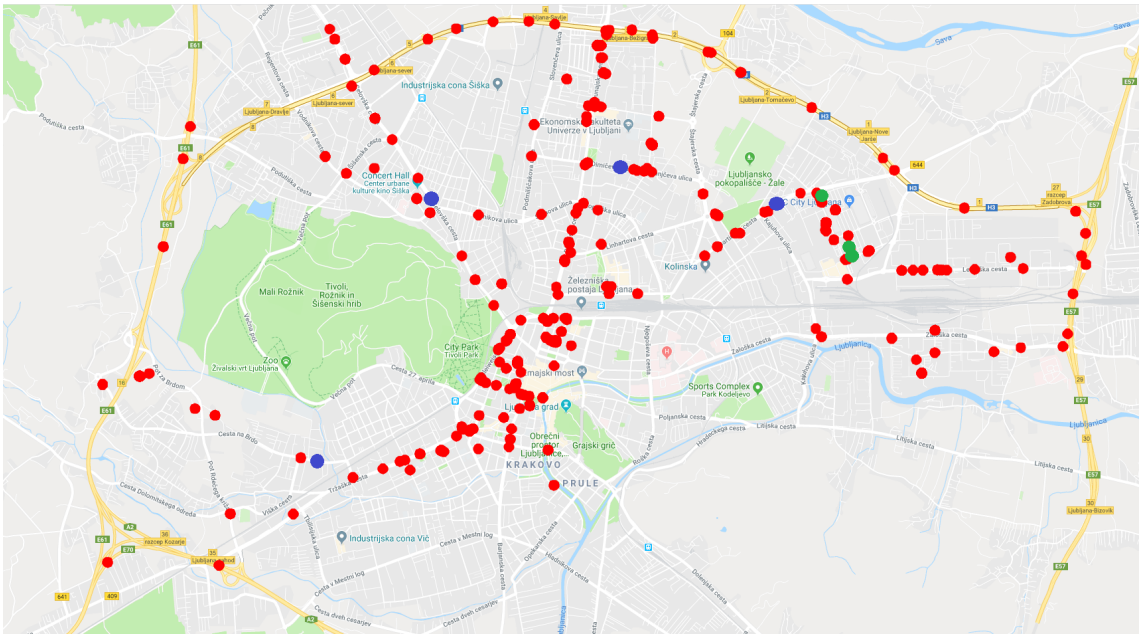
4.2 Rezultati

Po odpravljeni težavi sočasnega dostopa je en teden tekla pravilno delujoča pilotna različica. Po enem tednu smo analizirali e-sporočila, da bi videli, kolikokrat je kateri avtomobil izpolnil katero pravilo. Podatki niso najbolj reprezentativni za dejansko rabo, saj se nanašajo le na trinajst testnih avtomobilov, večina katerih je stalno stala v garaži. Prav tako se podatki niso sinhronizirali v realnem času, ampak le na pol ure, da smo porabili čim manj sredstev AWS. Vseeno se je nekaj avtomobilov uporabljalo redno, zato ni bilo malo uspešnih evalvacij.

Avtomobil	Stanje baterije nad 70	Stanje baterije pod 70	V BTC	Zunaj BTC
DU 373 HO	2	2	0	1
DU 372 HO	2	1	0	1
LJ 599 RB	1	1	5	5
DU 379 HO	1	0	0	1
DU 389 HO	1	0	0	1
LJ 033 SC	2	2	1	2
LJ 048 TF	3	4	1	1
LJ 13 ZJV	1	1	0	1
LJ 12 ZJV	1	0	0	1
LJ 056 DU	1	0	0	1
LJ 032 SC	2	2	0	1
LJ 020 SC	0	1	0	1
LJ 043 SC	3	2	0	1
Vsota	20	16	7	18

Veliko avtomobilov z uspešno evalvacijo enkrat za lokacijo zunaj BTC in stanjem baterije pod ali nad 70 je novih in stojijo v garaži. Iz rezultatov je razvidno, da se je večina avtomobilov od samega začetka nahajala zunaj BTC. Nekaj avtomobilov, ki so se uporabljali redno, je šlo v BTC le sedemkrat. Kot je bilo že navedeno, je namreč veliko avtomobilov v garaži, nekaj pa je testnih avtomobilov podjetja AvantCar, ki nima prostorov v bližini BTC, zato avtomobilov verjetno niso vozili tja. Pogostejše je bilo tudi stanje baterije nad 70, kar pomeni, da so jih pri nizki napolnjenosti baterije večinoma takoj priključili na polnilnico. Polnjenj in praznjenj ni bilo veliko, saj so namenjeni za mestno vožnjo, poleg tega imajo vsi baterijo z dosegom, večjim od sto kilometrov.

Vozilo z registrsko številko LJ 599 RB je naš testni avtomobil, iz rezultatov je razvidno, da se je premikal največ, vsak dan je bil v BTC in tudi zunaj njega. Hkrati je razvidno, da avtomobila nismo velikokrat polnili, saj na našem parkirišču ni polnilnice zanj. Za vsak dogodek je v e-sporočilu razviden tudi datum, tako je bilo vidno, kdaj je avtomobil na kateri lokaciji in kakšno je stanje negove baterije. Možna nadgradnja bi bila, da za vsako sredstvo beležimo še zgodovino sprememb njegovih vrednosti atributov. V pilotni različici tega še nismo implementirali, saj funkcionalnost ni ključna za demonstracijo delovanja. V sistemu Avant2Go se za vsak avtomobil beleži geolokacija, ob spremembi se zapiše nova. Tako lahko geolokacijske podatke in datume zapisa v bazo na sistemu Avant2Go kombiniramo z datumi uspešne evalvacije za stanje baterije za posamezen avtomobil in izrišemo sliko, kje se je avto nahajal in kakšno je bilo stanje njegove baterije. Za hitro demonstracijo bomo uporabili vozilo LJ 048 TF, pri katerem se je stanje baterije največkrat spremenilo. Rdeče pike so vse njegove znane lokacije, zelene pike so lokacije, kjer se je baterija napolnila nad 70 odstotkov, modre pa lokacije, na katerih se je spraznila pod 70 odstotkov.



Slika 4.1: Lokacije avtomobila LJ 048 TF.

Sistem torej ponuja še več možnosti za nadgradnje, s katerimi se še poveča obseg njegovih funkcionalnosti. Z njim lahko popolnoma spremljamo življenjski cikel sredstva in vrednost vseh njegovih atributov v realnem času, ali pa za poljuben čas v njegovi zgodovini.

Poglavje 5

Sklepne ugotovitve

Računalništvo se hitro spreminja, vsako leto postajajo priljubljene nove tehnologije, ki bistveno vplivajo na izdelovanje programske opreme. V letu 2017 se je vse več govorilo o AWS in ReactJS, objavljeni so bili številni članki na to temo, ki so prinesli veliko pozitivnih pa tudi negativnih mnenj. Po našem prepričanju si mora v računalništvu mnenje vsak oblikovati sam. Lahko preberemo še toliko člankov, a je najbolje vzeti novo tehnologijo v svoje roke in z njo nekaj izdelati. Za uporabo zelo novih tehnologij smo se odločili predvsem zato, da jih lahko preizkusimo sami. Kot vedno je bil začetek nekoliko težji, še posebej glede AWS, saj nenadoma ne programiramo več strežnika, ampak več ločenih funkcij. Vsem uporabljenim tehnologijam smo se hitro privadili, pri čemer so bili v veliko pomoč sodelavci z več izkušnjami pri delu z njimi. Izkušnje so pozitivne, saj smo si lahko oblikovali mnenje o uporabljenih tehnologijah in se veliko naučili. Sam kot programer bolje razumem, kako vse deluje, lahko rečem tudi, da zdaj tehnologije znam uporabljati. Pri nekaterih vprašanjih bi bila zame osebno uporaba druge tehnologije preprostejša. V vseh teh primerih smo sicer poiskali rešitev s trenutnim naborom tehnologij, kar je zahtevalo nekaj več časa, a tudi prineslo več znanja. Izdelava sistema je veliko pripomogla k mojemu programerskemu razvoju, močno sem izboljšal svoje programerske sposobnosti ter sposobnosti analitičnega in algoritmičnega razmišljanja. Z uporabo sodobnega nabora tehnologij sem tudi

spremenil svoj pogled na industrijo, sedaj sem bistveno bolj na tekočem. Zdaj ne le berem o ReactJS, AWS, Mongo Atlas itd., ampak imam dejanske projektne izkušnje in mnenje, ki sem si ga ustvaril sam.

S pilotno izdajo različice sistema in analizo rezultatov smo dokazali, da ima sistem praktično uporabo. Z njim je mogoče samostojno rešiti vprašanje upravljanja poljubnega tipa sredstev in za to ni treba kupovati več namenskih sistemov. Našli smo tudi možnost za izboljšavo, pri čemer bi beležili spreminjanje vrednosti sredstev in vodili zgodovino sprememb za vsa sredstva. Tako bi lahko vedeli, kdaj in kje se je zgodila uspešna evalvacija pravila, kaj se je dogajalo s sredstvom prej in kaj potem. Za implementacijo sistema smo porabili približno 600 delovnih ur, kar seveda ni malo, vendar menimo, da nam je vendarle uspelo sestaviti pilotno različico v spodobnem času. Porabljeni čas bi se lahko občutno zmanjšal, če ne bi izdelovali še nadzorne plošče v Reactu. Menimo, da bi bilo porabljeni čas mogoče bolje izkoristiti, če bi ga posvetili zalednemu sistemu in evalvaciji, tako bi lahko projekt dokončali prej in dodali še več funkcionalnosti. Sam kljub temu odločitve ne obžalujem, saj sem večino dosedanjih delovnih ur preživel kot razvijalec zalednih sistemov, z nadzorno ploščo pa sem končno postavil še spodoben pročelni sistem in dobil res veliko izkušenj. Do polne izdaje sistemu verjetno manjka še kakšna funkcionalnost na zalednem sistemu in polepšava nadzorne plošče, vendar bomo s pridobljenim znanjem to izvedli brez težav.

Literatura

[1] Angular. Dosegljivo: https://angular.io/. [Dostopano: 16.6.2018].

[2] AWS. Dosegljivo: https://aws.amazon.com/what-is-aws/. [Dostopano: 16.6.2018].

[3] AWS Amplify. Dosegljivo: https://aws.github.io/aws-amplify/media/developer_guide. [Dostopano: 16.6.2018].

[4] AWS Cognito. Dosegljivo: https://aws.amazon.com/cognito/?nc2=h_m1. [Dostopano: 16.6.2018].

[5] AWS Lambda. Dosegljivo: https://aws.amazon.com/lambda/?nc2=h_m1. [Dostopano: 16.6.2018].

[6] AWS SES. Dosegljivo: https://aws.amazon.com/ses/?nc2=h_m1. [Dostopano: 16.6.2018].

[7] AWS SNS. Dosegljivo: https://aws.amazon.com/sns/?nc2=h_m1. [Dostopano: 16.6.2018].

[8] Cloud providers marketshare. Dosegljivo: https://www.skyhighnetworks.com/cloud-security-blog/microsoft-azure-closes-iaas-adoption-gap-with-amazon-aws/. [Dostopano: 17.6.2018].

[9] DynamoDB. Dosegljivo: https://aws.amazon.com/dynamodb/?nc2=h_m1. [Dostopano: 16.6.2018].

[10] EC2. Dosegljivo: https://aws.amazon.com/ec2/?nc2=h_m1/. [Dostopano: 16.6.2018].

[11] JSON rules engine. Dosegljivo: https://github.com/CacheControl/json-rules-engines. [Dostopano: 16.6.2018].

[12] Magento. Dosegljivo: https://magento.com/. [Dostopano: 17.6.2018].

[13] Mongo Atlas. Dosegljivo: https://www.mongodb.com/cloud/atlas. [Dostopano: 16.6.2018].

[14] Mongoose. Dosegljivo: http://mongoosejs.com/. [Dostopano: 16.6.2018].

[15] React Bootstrap. Dosegljivo: https://react-bootstrap.github.io/getting-started/introduction. [Dostopano: 16.6.2018].

[16] React Router. Dosegljivo: https://github.com/ReactTraining/react-router. [Dostopano: 16.6.2018].

[17] React Table. Dosegljivo: https://react-table.js.org/#/story/readme. [Dostopano: 16.6.2018].

[18] ReactJS. Dosegljivo: https://reactjs.org/. [Dostopano: 16.6.2018].

[19] Redux. Dosegljivo: https://redux.js.org/. [Dostopano: 16.6.2018].

[20] Redux Form. Dosegljivo: https://redux-form.com/7.4.0/docs/gettingstarted.md/. [Dostopano: 16.6.2018].

[21] Vue.js. Dosegljivo: https://vuejs.org/v2/guide/index.html. [Dostopano: 16.6.2018].

[22] Kristina Chodorow. *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. "O'Reilly Media, Inc.", 2013.

[23] David Flanagan. *JavaScript: the definitive guide*. "O'Reilly Media, Inc.", 2006.

[24] Bernard Golden. *Amazon Web Services for Dummies*. John Wiley & Sons, 2013.

[25] Graham Klyne and Chris Newman. Date and time on the internet: Timestamps. Technical report, 2002.

[26] Stefan Tilkov and Steve Vinoski. Node. js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, 2010.