UNIVERZA V MARIBORU

FAKULTETA ZA ELEKTROTEHNIKO,

RAČUNALNIŠTVO IN INFORMATIKO

Marko Gluhak

# Posebnosti razvoja enostranskih spletnih rešitev s samopostrežnim zalednim sistemom v oblaku

Zaključno delo

Maribor, avgust 2020

# Posebnosti razvoja enostranskih spletnih rešitev s samopostrežnim zalednim sistemom v oblaku

Diplomsko delo


| | |
|---|---|
| Študent: | Marko Gluhak |
| Študijski program: | Univerzitetni študijski program |
| | Informatika in tehnologije komuniciranja |
| Smer: | / |
| Mentor: | doc. dr. Luka Pavlič, univ. dipl. inž. rač. in inf. |
| Jezikovno pregledala: | Nina Katarina Bračko, dipl. inž. med. kom. (UN) |

# ZAHVALA

Iskrena zahvala mentorju, doc. dr. Luki Pavlič, za usmerjanje pri pisanju in nasvete, ko se je pisanje ustavilo. Prav tako se zahvaljujem družini, prijateljem in vsem ki ste me podprli na moji poti do sem.

# POSEBNOSTI RAZVOJA ENOSTRANSKIH SPLETNIH REŠITEV S SAMOPOSTREŽNIM ZALEDNIM SISTEMOM V OBLAKU

**Ključne besede:** samopostrežni zaledni sistemi v oblaku, enostranske spletne aplikacije, React, Firebase, AWS Amplify, Google, Amazon Web Services, REST

**UDK:** 004.383.3:004.738.5(043.2)

**Povzetek:**

*Paradigma samopostrežnih zalednih sistemov v oblaku je čedalje bolj popularna in zrela za uporabo. Enostranske spletne aplikacije so eden od potencialnih odjemalcev samopostrežnih zalednih sistemov v oblaku. V diplomskem delu smo raziskali, kako lahko ta učinkovito sodelujejo med seboj. Glavni namen dela je bil spoznati prednosti in omejitve tega sodelovanja ter definirati kriterije za izbor orodja za razvoj samopostrežnega zalednega sistema v oblaku. Nato smo zastavljene kriterije nanesli na tri orodja za razvoj samopostrežnega zalednega sistema v oblaku, primerjali smo lastne REST rešitve, Firebase in AWS Amplify. S Firebase-om, ki je ugotovljen kot najbolj primeren za nalogo, v paru z React-om, smo razvili enostavno spletno aplikacijo »Moja Galerija« in prikazali, kako se ta tehnološki par obnese v realnem življenju.*

# THE SPECIFICS OF SINGLE PAGE APPLICATION DEVELOPMENT WHILE USING CLOUD-BASED BACKEND AS A SERVICE

**Keywords:** Serverless, single page applications, React, Firebase, AWS Amplify, Google, Amazon Web Services, REST

**UDK:** 004.383.3:004.738.5(043.2)

**Abstract**

*The concept of serverless computing is gaining more traction over the past couple of years and therefore becoming more appealing for use in real world applications. Single page applications are one of the possible beneficiaries of the serverless paradigm. Throughout the diploma we will look at the collaboration of these technologies and what there is to gain. The main purpose of the diploma is getting to know the possibilities and limitations of this duality and to define the criteria for choosing the best tool to develop a serverless backend. After this we applied the established criteria to three tools for developing serverless solutions, the tools being: own REST solutions, AWS Amplify and Firebase. The scoring system determined Firebase as the winner of the comparison, we used it and React to develop a basic single web application "My Gallery" and demonstrated how this duality would function in a real-world scenario.*

# KAZALO VSEBINE

# KAZALO SLIK

# KAZALO TABEL

# SEZNAM UPORABLJENIH KRATIC

| | |
|---|---|
| REST | *Representational State Transfer* |
| IaaS | *Infrastructure as a Service* |
| PaaS | *Platform as a Service* |
| OS | *Operacijski sistem* |
| AWS | *Amazon Web Services* |
| BaaS | *Backend as a Service* |
| FaaS | *Function as a Service* |
| SPA | *Single Page Application* |
| DDoS | *Distributed Denial of Service* |
| HTML | *Hyper Text Markup Language* |
| CSS | *Cascading Style Sheets* |
| JSON | *JavaScript Object Notation* |
| XML | *Extensible Markup Language* |
| SSR | *Server-Side Rendering* |
| ES | *ECMAScript* |
| PWA | *Progressive Web Apps* |
| DOM | *Document Object Model* |
| JSX | *JavaScript XML* |
| IDE | *Integrated Development Environment* |
| npm | *Node Package Manager* |
| BLOB | *Binary Long Object* |

# 1 UVOD

Na področju informatike se v zadnjih letih pričakuje hitro odzivanje na razvoj programskih rešitev z možnostjo nagle rasti. Ko to uparimo z željo po zagotavljanju dobre uporabniške izkušnje, hitro ugotovimo, da tehnološki trendi nagibanja k samopostrežnim storitvam in enostranskim aplikacijam niso zgolj naključje. V poplavi ponudb zalednih sistemov kot storitev, se je lahko problem odločiti za pravilno. Kljub reševanju ogromno problemov, kot so razširljivost, varnost in druge prepreke strojne opreme, pa zahteva druge aspekte razvoja. Med te štejemo dobro zasnovan nivo abstrakcije strežniškega dela in dodatno kompleksnost pri pisanju kode. Enostranske aplikacije so odlična izbira za delovanje s takšnimi sistemi, saj so dobro utečene za delo z REST vmesniki in predstavitvijo pridobljenih podatkov.

V drugem poglavju smo pogledali te trende, preverili, kako so se razvile oblačne storitve in zakaj. Pogledali smo, kje smo danes in katere so glavne lastnosti razvoja oblačnih zalednih sistemov. V tretjem poglavju smo se omejili na samopostrežne zaledne rešitve (angl. »serverless«) in si podrobneje ogledali, kaj so prednosti in slabosti, ko se odločimo za delo z njimi. V četrtem poglavju smo preleteli enostranske spletne aplikacije in kaj imamo v mislih, ko s temi delamo s samopostrežnimi zalednimi sistemi v oblaku. V petem poglavju smo definirali kriterij za izbor najboljšega kandidata za razvoj samopostrežnih zalednih storitev v oblaku. Izbirali smo med AWS Amplify, Firebase in lastnimi REST rešitvami. Sledi ocena teh orodij in nato izbor najbolj primernega kandidata. V šestem poglavju smo z izbranim orodjem za razvoj samopostrežnih zalednih sistemov v oblaku razvili aplikacijo »Moja Galerija«. To aplikacijo smo upodobili s pomočjo React-a in zagotovili nekaj osnovnih funkcionalnosti.

## 1.1 Opredelitev problema

V zaključnem delu smo raziskali, kateri izmed glavnih ponudnikov samopostrežnih zalednih sistemov je najbolj primeren za delo z enostranskimi spletnimi

aplikacijami. Podrobneje smo pogledali komunikacijo teh storitev z enostranskimi aplikacijami in poskusili ugotoviti, katera rešitev je za te najbolj primerna. Za določitev najbolj primernega ponudnika je potreben zajem vseh kriterijev, ki so relevantni. Tukaj gre za arhitekturni stil, čas učenja, skupnost razvijalcev, podporo delovanja s temi tipi aplikacij, preglednost in sodobnost dokumentacije ter še druge sproti ugotovljene kriterije. Ker pa je izbor ponudnika samo en del celotnega postopka, smo za ugotovljeno najboljše orodje razvili preprosto rešitev in jo namestili v realni svet. Tako smo na osnovnem primeru tudi ugotovili, ali je naš zaključek izbora ponudnika primerljiv z realnim obnašanjem.

## 1.2 Cilji zaključnega dela

Cilj zaključnega dela je bil poiskati ponudnike samopostrežnih zalednih sistemov v oblaku in ugotoviti ustrezne kriterije za izbor najboljšega ponudnika za potrebe razvoja enostranskih spletnih aplikacij. Poleg tega je bil naš cilj ugotoviti, kakšna orodja nam ogrodja za razvoj enostranskih spletnih aplikacij sploh ponujajo in kako ta uporabiti na primeru. Na koncu naloge je bil cilj praktično demonstrirati preprostejšo aplikacijo z ugotovljenim najboljšim kandidatom za zaledje in predstaviti njegove prednosti na primeru.

## 1.3 Predpostavke in omejitve

Predpostavili smo, da se za enostransko aplikacijo uporabi knjižnico React, in da je v drugih enostranskih ogrodjih stvar podobna. T. i. Serverless zaledni sistem smo konkretizirali z rešitvijo, ki se je izkazala za najbolj obetavno. Primerjali smo naslednje predstavnike zalednih sistemov – lastne REST rešitve, Amazon Web Services in Google Firebase. Namestitev v realni svet je bila izvedena za Docker zabojnike.

# 2 EVOLUCIJA STORITEV V OBLAKU

Za razumevanje rešitev, ki jih samopostrežni zaledni sistemi v oblaku ponudijo, in zakaj so te tako atraktivne, moramo razumeti probleme, ki so to programersko paradigmo povzdignili na nov nivo. Podjetja se v času, ko je pozornost uporabnika med najbolj visoko cenjenimi surovinami, na vse načine trudijo to pozornost pridobiti in jo zadržati. Poleg kakovostne vsebine je treba zagotoviti tudi hitro serviranje te. Eden od izzivov je premagovanje geografskih omejitev in tako povečati svojo prisotnost na svetovnem trgu. Zagotoviti lastno infrastrukturo bi v tem primeru lahko predstavljalo tako velik izziv, kot sam razvoj vsebine na spletni strani. Oblačne rešitve nam te vidike občutno olajšajo, saj nam nudijo zanesljive, razširljive, cenovno učinkovite, sofisticirane storitve in rešitve, ki podjetjem omogočajo hitro posodobitev, prilagoditev in modernizacijo poslovnih procesov. Poleg teh prednosti pa zelo elegantno rešijo tudi večne probleme strojne opreme (od vzpostavitve do vzdrževanja), s katerimi se morajo podjetja spopadati z vsakim projektom posebej. Zaradi teh se je na področju oblačnega računalništva razvilo mnogo arhitekturnih paradigem, ki naslavljajo različne potrebe. Samopostrežni zaledni sistemi so zadnji izmed takšnih oblačnih modelov, ki se v osnovi osredotočajo na abstrahiranje strežnikov in upravljanje nizko nivojske infrastrukture pred razvijalci programske opreme. Torej ti sistemi igrajo veliko vlogo pri ohranjanju osredotočenosti razvijalcev na svojo primarno dejavnost – implementacijo poslovne logike in razširjanju posameznih funkcionalnosti neodvisno od strojne opreme. [1] [2] [3] [4]

Samopostrežni zaledni sistemi so v zadnjem desetletju dobili veliko zagovornikov, še posebej po tem, ko je Amazon leta 2014 predstavil svojo platformo pod imenom *AWS Lambda*. Ko je tržišče pokazalo interes, so se poslovnemu modelu pridružili še ostali tehnološki orjaki, Microsoft s svojimi t. i. Azure Functions in Google z *Google* Cloud Functions – oba leta 2016. Sledila sta še Oracle-ova Fn in IBM-ov OpenWhisk. Obstajajo tudi odprtokodna ogrodja za samopostrežne sisteme, kot

sta Serverless in Kubernetes, ki sta neodvisna od oblačnih ponudnikov in neodvisno delujeta v Docker in Kubernetes zabojnikih.

Da bi bolje razumeli samopostrežno računalništvo, je treba pogledati evolucijo strojne opreme in omrežne topologije od leta 1990. Takrat so podjetja še sama nakupovala in postavljala svojo strojno opremo ter postavljala svoje omrežne topologije za gostovanje lastnih aplikacij. To je bilo zelo učinkovito, saj so imeli direkten nadzor nad trenutnimi potrebami aplikacije in kako jih zagotoviti. Seveda to vpelje svoje probleme, kot je vzdrževanje in dolgoročne razširljivosti. Tu so prišle v igro različne oblačne rešitve, ki so nudile rešitve za večino problemov lastne postavitve in vzdrževanja. To so storile tako, da so ponudile različne nivoje abstrakcije za potrebovano strojno opremo [1].

## 2.1 Infrastruktura kot storitev

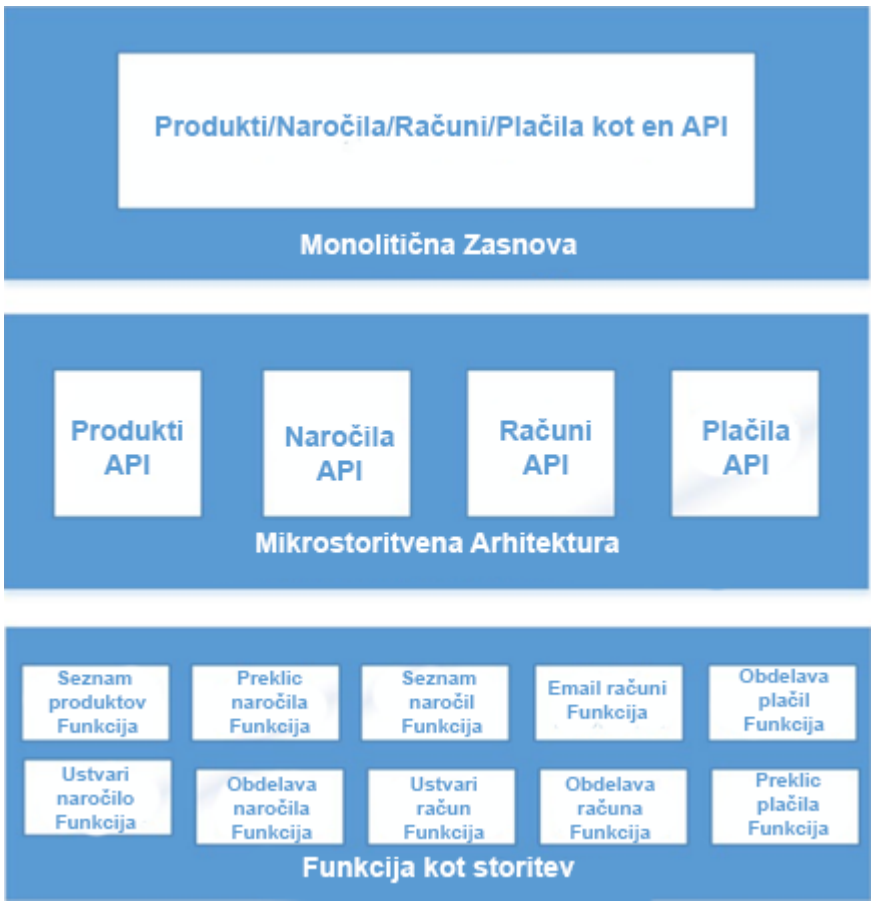
*Infrastruktura kot storitev* oz. *Infrastructure as a Service (IaaS)* je prva računalniška oblačna storitev, ki zagotavlja strojno opremo za organizacije z veliko različnimi potrebami. IaaS zagotovi virtualne naprave z različnimi OS, pomnilnikom in hranilnih možnosti za serviranje od majhnih do velikih obremenitev. V tem modelu delovanja mora organizacija sama skrbeti za OS, izvajalnike kode in drugo vmesno opremo. Celo namestitev aplikacije v IaaS ni povsem intuitivna za razvijalce, vendar pa nudi veliko svobode pri prilagoditvi gostitelja aplikacije [1].

## 2.2 Okolje kot storitev

*Okolje kot storitev* oz. *Platform as a Service (PaaS)* je naslednja strategija za oblačno računalništvo. Zagotovi boljši nivo abstrakcije na strojno opremo in OS ter namestitev aplikacije v primerjavi z IaaS. PaaS rešitve so po zasnovi visoko razpoložljive in razširljive. V primerjavi z rešitvami, ki gostujejo na IaaS, so na PaaS rešitvah vse aktivnosti povezane s strojno opremo, vključno s posodobitvami OS in varnostnimi krpami obravnavane s strani ponudnika storitve [1].

## 2.3 Funkcija kot storitev

*Funkcija kot storitev* oz. *Function as a Service (FaaS)* spodbuja razvijalce, da izluščijo majhne sklope funkcionalnosti iz več nivojske aplikacije in jih implementirajo v obliki funkcije. Te funkcije lahko gostujemo kot samostojne enote, ki se razširijo neodvisno v oblaku. Ta pristop je cenovno zelo učinkovit, saj lahko individualno funkcijo razširimo na osnovi njene individualne obremenitve in ne celotne aplikacije. FaaS se razlikuje od tradicionalnih monolitnih oblik, kjer je celotna aplikacija stisnjena v eno enoto. Gre celo nivo nižje od mikrostoritve in razčleni aplikacijo v manjše funkcije. Slika 2.1 prikazuje razlike med monolitnimi, mikrostoritvenimi in FaaS arhitekturami na enostavnem modelu za upravljanje naročil [1].




Slika 2.2-1: Razlike med monolitnimi, mikrostoritvenimi in FaaS modeli [1]

Na Sliki 2.2 je ponazorjeno, kako so videti bolj poznane storitve na Microsoft Azure s prej omenjenimi strategijami oblačnega računalništva.


Slika 2.2-2: Oblačne storitve Microsoft Azure z različnimi modeli oblačnega računalništva [1]

## 2.4 Samopostrežne oblačne rešitve (angl. »Serverless«)

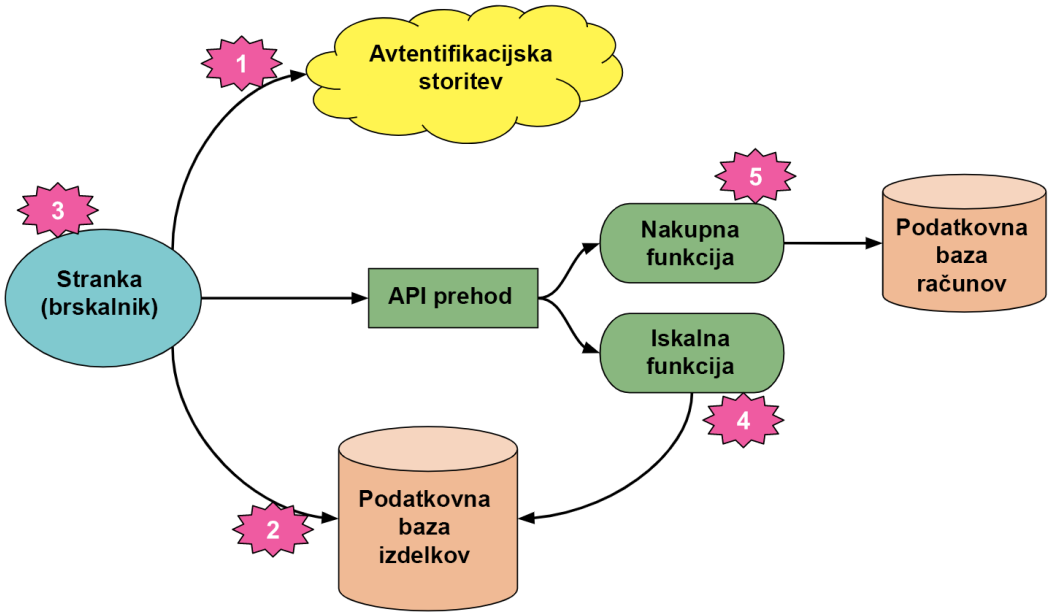
Samopostrežne oblačne rešitve so aplikacijske zasnove, ki uporabijo zunanje ponudnike *Backend as a Service* (BaaS), ki vključujejo po meri izvajano in upravljano kodo znotraj zabojnikov na *Function as a Service* (FaaS) platformi [5].

Samopostrežne oblačne rešitve so zadnja strategija ponudnikov oblačnih storitev, kjer so razvijalci aplikacij popolnoma izolirani od upravljanja strojne opreme. Angleški izraz za to je »*serverless*«, ki dobesedno preveden pomeni »*brez strežnika*«, kar pa ni res. Za tem izrazom se skriva nakazovanje na to, da je nivo abstrakcije upravljanja strežnika popoln. Samopostrežne storitve so se v osnovi začele kot *Backend as a Service* (BaaS) in se počasi razvile v *Function as a Service (FaaS).* BaaS rešitve so popolnoma spletno gostovane, kot na primer Google-ov Firebase in Microsoft-ov Azure Mobile App storitev, itd. Te ponujajo sklop funkcionalnosti, kot so shranjevanje podatkov, overitev, obvestila itd. Na drugi strani pa FaaS izvaja funkcije, zasnovane s strani razvijalcev, z uporabo programskih jezikov, kot so C#, Python itd. Te funkcije so izvedene na osnovi dogodkovno-vodenega modela s pomočjo prožilcev [1].

Med BaaS in samopostrežnimi storitvami pa je kljub velikemu prekrivanju kar nekaj razlik. Med dve najbolj preudarni uvrščamo način razširjevanja, in to, da samopostrežne storitve uporabljajo dogodkovno voden model. Samopostrežne storitve se bodo vedno avtomatsko razširjale glede na potrebe, BaaS pa se ne, če tega izrecno ne nastavimo – nekateri ponudniki tega sploh ne omogočajo. Dogodkovni model pa v primeru samopostrežnih storitev deli našo aplikacijo na funkcije, ki se prožijo ob določenih dogodkih. To za BaaS vedno ne drži, kar zahteva več virov. V manj pomembne razlike uvrščamo še, kako je naša aplikacija

vzpostavljena in pa kje se naša koda izvaja – s samopostrežnimi storitvami lahko podpremo t. i. »edge computing« oz. robno računalništvo [6] [7].

Končna arhitektura samopostrežne rešitve v oblaku je vidna na Sliki 2-3, ko so vse storitve med seboj na različnih lokacijah. Funkcije pa so vse dosegljive na enakem API prehodu, ki usmerja zahtevke na pravo mesto.




Slika 2-3: Primer arhitekture samopostrežne rešitve v oblaku [5].

# 3 PREDNOSTI IN OMEJITVE SAMOPOSTREŽNIH ZALEDNIH SISTEMOV

Sedaj ko vemo, kaj je samopostrežni zaledni sistem in kako smo do njega sčasoma prišli, definirajmo še, kaj dobrega nam omogoča in kje nas omejuje.

Najprej si oglejmo prednosti [1] [8]:

- najboljša samostojna razširljivost ob velikih bremenih,
- brez vzdrževanja arhitekture,
- manjši čas odziva,
- nizki stroški obratovanja – po porabi,

- enostaven model namestitve,
- v splošnem se zapletenost zmanjša.

Omejitve [1] [8]:

- orodja monitoringa, beleženja in razhroščevanja so še vedno v zgodnjih fazah razvoja,
- nekompatibilnost med ponudnik,
- treba je utrditi varnostne sisteme,
- funkcije niso vedno idealne.

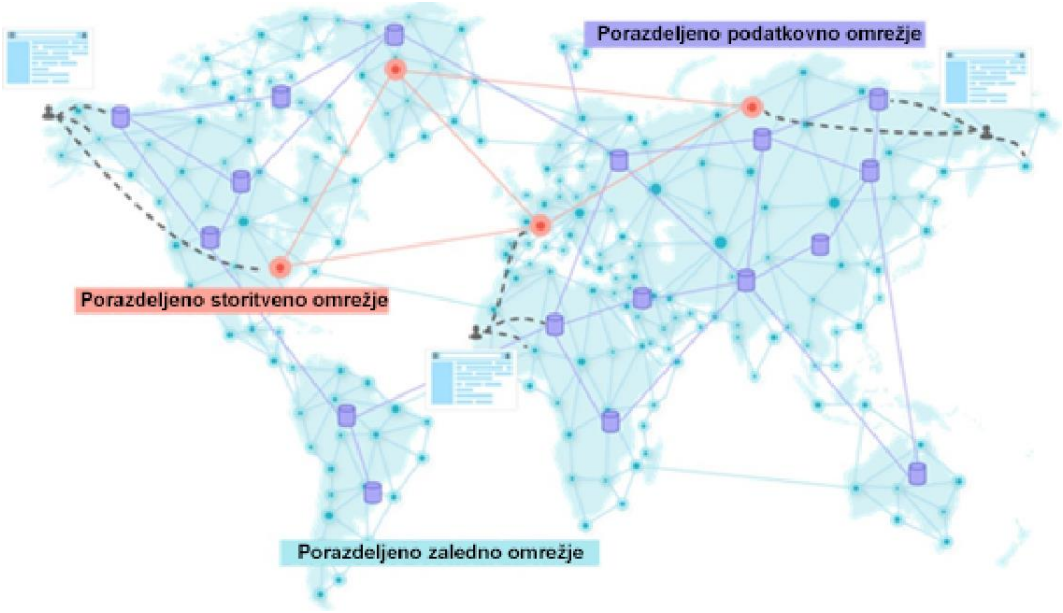
Ker se samopostrežno računalništvo in temu sorodne rešitve hitro izboljšujejo, bodo zahteven sotekmovalec za svoje predhodnike. Trenutno je samopostrežno računalništvo smatrano kot najbolj zanesljiva nastajajoča paradigma. Podrobneje si poglejmo prednosti in omejitve, ki nam povedo več o sodelovanju te paradigme z enostranskimi spletnimi aplikacijami [1] [8].

## 3.1 Prednosti

Ker sami ne upravljamo virov, ki so potrebni za tekoče delovanje infrastrukture, se naša rešitev avtomatsko razširi po potrebi. Ker je naša aplikacija enostranska, to pomeni tudi, da se ta prenese na stran uporabnika in se tam izvajajo ostale enostavnejše operacije. Posledica tega je, da je vse, kar bi lahko našo aplikacijo preobremenilo, v rokah ponudnikov samopostrežnih storitev in ti urejajo vse probleme. Če se mora funkcija izvesti v več instancah, se bodo strežniki ponudnika zagnali in ob koncu delovanja tudi ugasnili. To nas reši klasične fizične omejitve kapacitete strežnika, saj jih imajo ti ponudniki ogromno na zalogi – pogosto se to implementira z zabojniki. Posledica je tudi, da so funkcije, ki imajo pogosto obremenitev, vedno pripravljene, da uporabnike postrežejo z odgovori in tako zagotovijo nenavadno hitre odzive [8].

Ponudniki samopostrežnih zalednih storitev imajo zelo dodelano arhitekturo in infrastrukturo. Za nas to pomeni, da bodo lahko vedno zagotovili dobre odzivne čase, ne glede na lokacijo uporabnika. Na Sliki 2-3 vidimo uporabnika na zahodu

Afrike, Aljaske in vzhoda Rusije, ki se želijo povezati na našo aplikacijo. Vidimo mrežo porazdeljenih storitev (rdeče pike), zaledij (temneje modre pike) in podatkovnih baz (vijolični valji), ki nam jo zagotavlja izbrani oblačni ponudnik (ali več teh). Ko uporabnik zahteva izvesti določeno funkcionalnost, mu naš ponudnik/i zagotovi/jo izvedbo na tisti točki, ki bo to opravila najhitreje.




Slika 3-1: Porazdeljena arhitektura samopostrežnih zalednih sistemov [9].

Ko to uparimo z načinom, na katerega delujejo enostranske spletne aplikacije (SPA), imamo recept za zelo odzivno in z malo truda progresivno aplikacijo, ki je uporabnikom privlačna. O delovanju SPA in specifikah teh, si bomo pogledali v poglavju 2.3.

Vse to dosegamo tudi z razlogom, ker smo ponudniku predali v roke celotno skrb za arhitekturo. To pomeni, da privarčujemo ogromno pri usposabljanju osebja za potrebe vzdrževanje poleg vseh ostalih stroškov nabave itd. Če poenostavimo koncept – ko to arhitekturo predamo tako velikemu ponudniku, kot je recimo Google, pridobimo ogromno pokritost sveta in poleg tega še varčujemo na sami skrbi vseh teh naprav.

Za doseganje tako velike učinkovitosti z zagotovitvijo naprav na pravih lokacijah je sam namestitveni postopek zelo enostaven. V primeru, da sami vzpostavimo

podatkovne centre na vseh teh lokacijah, bi morali razviti to tehnologijo in jo sami posodabljati. Tako pa ponovno vse to prepustimo v rokah našega ponudnika [10].

Kar se tiče računalniške moči in človeških virov, se samopostrežne storitve obrestujejo. Nesmiselno je plačevati za ponovno implementacijo avtorizacije, zaznavanja prisotnosti, obdelave slik, prepoznavanja obrazov in drugih operacij. Enako je s stroški za vzdrževanje strojne opreme in človeških virov, ki so za to potrebni. Za boljšo idejo o tem, kako cenovno ugodne so lahko funkcije, ima AWS Lambda storitev zelo nazorno nakazano v svojem modelu cenitve (Slika 2-4).

## AWS Lambda Pricing

Region: Europe (Milan)

| | Price |
|---|---|
| Requests | $0.23 per 1M requests |
| Duration | $0.0000195172 for every GB-second |

The price for **Duration** depends on the amount of memory you allocate to your function. You can allocate any amount of memory to your function between 128MB and 3008MB, in 64MB increments. The table below contains a few examples of the price per 100ms associated with different memory sizes.

| Memory (MB) | Price per 100ms |
|---|---|
| 128 | $0.0000002440 |
| 512 | $0.0000009759 |
| 1024 | $0.0000019517 |
| 1536 | $0.0000029276 |
| 2048 | $0.0000039034 |
| 3008 | $0.0000057332 |

Slika 3-2: Cenovni model AWS Lambda [11]

Za boljšo predstavo tega, kako cenovno ugodno je to, so pri Amazon-u pripravili še nekaj konkretnih primerov, kako bi deloval njihov cenik v produkciji (Slika 2-5).


If you allocated 512MB of memory to your function, executed it 3 million times in one month, and it ran for 1 second each time, your charges would be calculated as follows:

**Monthly compute charges**

The monthly compute price is $0.00001667 per GB-s and the free tier provides 400,000 GB-s.

Total compute (seconds) = 3M * (1s) = 3,000,000 seconds

Total compute (GB-s) = 3,000,000 * 512MB/1024 = 1,500,000 GB-s

Total compute – Free tier compute = Monthly billable compute GB- s

1,500,000 GB-s – 400,000 free tier GB-s = 1,100,000 GB-s

**Monthly compute charges = 1,100,000 * $0.00001667 = $18.34**

**Monthly request charges**

The monthly request price is $0.20 per 1 million requests and the free tier provides 1M requests per month.

Total requests – Free tier requests = Monthly billable requests

3M requests – 1M free tier requests = 2M Monthly billable requests

**Monthly request charges = 2M * $0.2/M = $0.40**

**Total monthly charges**

**Total charges = Compute charges + Request charges = $18.34 + $0.40 = $18.74 per month**

Slika 3-3: Primer zaračunanja stroškov delovanja konkretne funkcije [11].

Če naši funkciji dodelimo 512MB spomina, jo izvedemo 3 milijonkrat v enem mesecu in se vsakič izvaja eno sekundo, bi bila skupna cena za ta mesec za to funkcijo $18.74. Če funkcij nikoli ne uporabimo, nas ne stanejo ničesar [12].

Nekatere omejitve samopostrežnih funkcij delujejo tudi razvijalcem v korist. Med te omejitve štejemo to, da morajo te funkcije delovati kjerkoli, ne glede na strojno opremo, brez potrebe po dodatnih zunanjih virih za dodatno kodo. To pomeni, da so te funkcije samostojne enote in da morajo biti enostavno napisane. To naredi

pregrado vstopa razvijalcem zelo nizko. Za enostranske aplikacije je to še posebno dobro, saj pomeni, da se ponavljajoča se koda in poslovna logika ne mešata s kodo, ki se izvaja v brskalniku uporabnika.

## 3.2 Omejitve

Noben pregled samopostrežnih zalednih sistemov ne bi bil popoln, ne da omenimo varnosti in potencialnih problemov s tem, ko se odločamo za tak način razvoja. Zaradi skrbi o teh varnostnih problemih je ZDNet ustvaril seznam 10 takšnih potencialnih varnostnih tveganj [13], med katere sodijo:

- Vrivanje podatkov dogodka, kar je napad vrivanja SQL stavka na strežnik, ki poganja samopostrežne funkcije;
- Nevarno samopostrežno namestitev, ki lahko povzroči vrsto napak na administratorskem delu in pusti samopostrežni računalnik odprt na napade prestrezanja;
- Nezadostno nadzorovanje in beleženje funkcij, ki bi lahko administratorjem namignile o tem, da napadalci izvajajo preiskovalne akcije;
- Nevarne knjižnice tretjih oseb - samopostrežne funkcije lahko kličejo knjižnice tretjih oseb, ki potencialno vsebujejo zlonamerno kodo in izpostavijo podatke nevarnosti;
- Napadi zanikanja storitve (DDoS) - če so napadi na samopostrežne platforme uspešni, so te preobremenjene in lahko spodletijo v nudenju storitev več strankam naenkrat [8].

Pri enostranskih aplikacijah je varnost še posebej ogrožena, saj teče komunikacijski kanal med napravo uporabnika in ponudniki oblačnih storitev.

Samopostrežne storitve delujejo najbolj optimalno, ko so klicane pogosto v kratkih intervalih. To zagotovi, da so vedno pripravljene na delovanje in nimajo zamika, da se storitev zažene. Vendar pa naj funkcije ne tečejo več časa, saj lahko tako postanejo zelo cenovno potratne in se lahko veliko bolj splača postaviti lastno

strežniško arhitekturo. Torej v tem primeru ne gre za snovalno napako, temveč za vprašanje cenovne ugodnosti naše odločitve in potencialnih alternativ [10].

Trenutno je tehnologija monitoringa in razhroščevanja še vedno nekoliko v infantilnih fazah in je lahko to eden od zadržkov uporabe samopostrežnih zalednih sistemov. Problem je, da aplikacija deluje na zunanjem ponudniku in sami nimamo dostopa do vseh informacij o tem, kaj se na sistemu dogaja. Informacije, ki nam jih ponudnik dovoli opazovati, včasih niso dovolj za temeljito razhroščevanje in nadzorovanje naše aplikacije. Če bi se aplikacija izvajala na strežniku v podjetju, bi imeli boljši vpogled v celovito obnašanje aplikacije, ponudniki se na tem področju precej razlikujejo.

Kot je ugotovljeno iz prednosti uporabe samopostrežnih zalednih sistemov v oblaku, nam ponudniki oblačnih rešitev olajšajo življenje v precejšnji meri. To je zelo dobro, dokler nam ponudnik ustreza. Če bi želeli ponudnika zamenjati, je to lahko velik glavobol, saj se redkokatere funkcionalnosti med ponudniki prekrivajo tako, da bi delovale po enakem principu. Skoraj vedno so potrebne nekakšne manjše spremembe. Če pa bi se odločili uporabljati samo krovne funkcionalnosti izbranega ponudnika v imenu lažje menjave, če bi ta bila potrebna, pa se odrečemo veliko orodjem, ki bi nam pomagala pri razvoju.

# 4 ENOSTRANSKE SPLETNE APLIKACIJE

Splet se je začel s statičnimi spletnimi stranmi, ki so postregle HTML dokumente. Ti si vsebovali hiperpovezave do drugih dokumentov, porazdeljenih po drugih spletnih strežnikih po celem svetu. Kasneje so spletne strani postale spletne aplikacije, ko so spletni strežniki bili sposobni generiranja dinamičnih vsebin na podlagi uporabnikovih vnosov in navigacij. Ti so uporabljali tehnologijo na strani strežnika, kot so ASP, JSP ali PHP za pridobivanje in posodabljanje podatkov s podatkovnih baz in generiranja HTML strani dinamično. Temu pravimo *server-side rendering* (SSR).

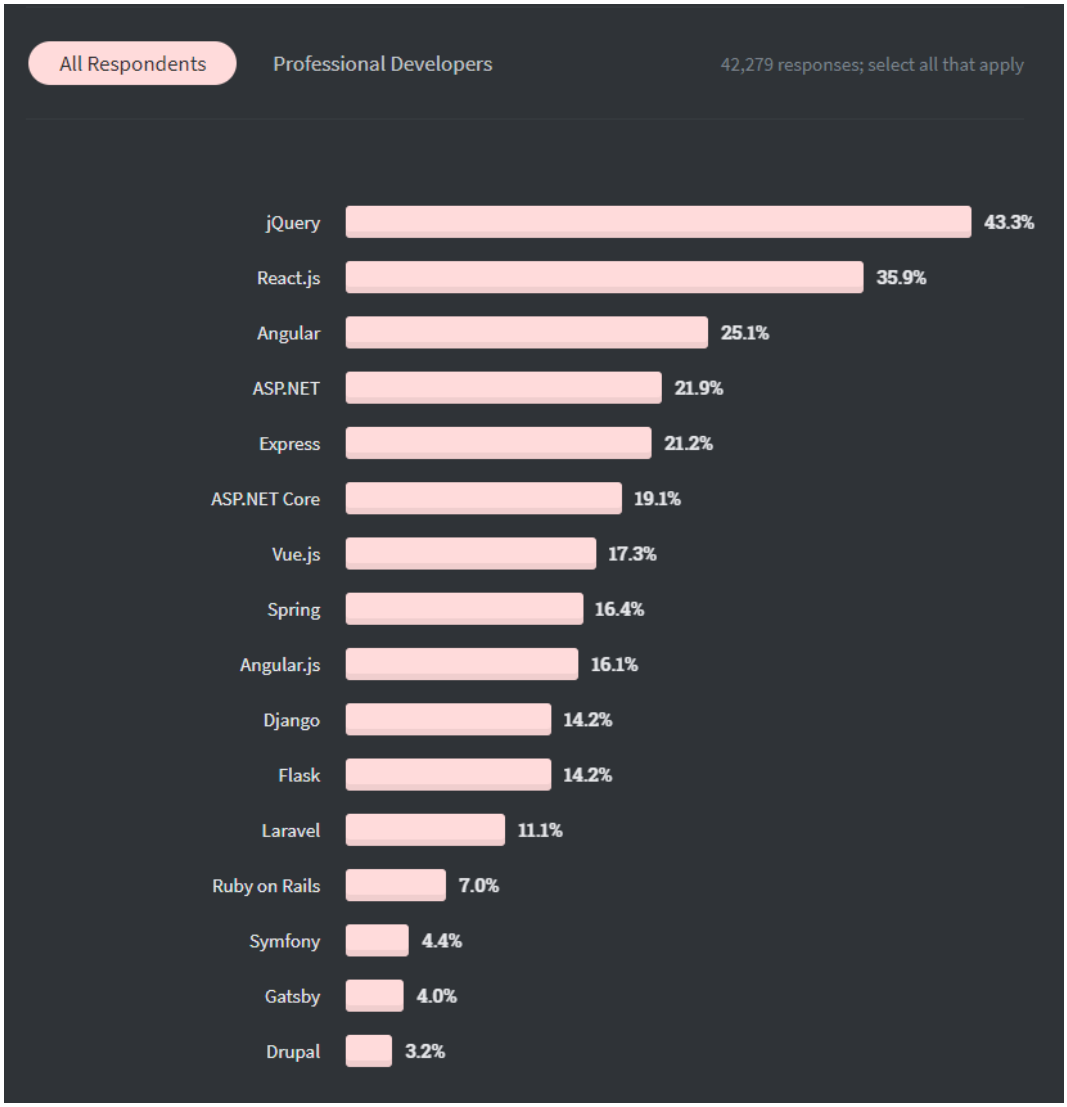
Običajno ima SSR problem, da za vsako interakcijo uporabnika zahteva osvežitev celotne spletne strani. Uporabniške interakcije, kot so pritiski na gumbe, izpolnjevanje obrazcev, sprožijo GET ali pa POST zahtevo na spletni strežnik in ta mora ponovno pripraviti celotno HTML stran. To pa povzroči kratko utripanje strani v belo, kar poslabša uporabniško izkušnjo in je moteče. Vsekakor tudi dodatno in nepotrebno obremeni strežnik. Strežnik mora poznati celotno stanje aplikacije v brskalniku, kot recimo ID vpisanega uporabnika, številko strani, vsebino obrazcev za ponovno upodobitev spletne strani. Usklajevanje stanja med brskalnikom in strežnikom je težavno.

Programski vmesnik spletne aplikacije (API), ter Asinhroni JavaScript in XML (Ajax) sta bila ustvarjena, da rešita probleme SSR, kar pa je postopoma vodilo v enostranske spletne aplikacije (SPA). SPA so tehnologije, ki urejajo stanje aplikacije in logiko v glavnem v brskalniku. Ko aplikacija potrebuje dinamične podatke, pošlje svoje zahteve na spletni API. Ta nato pridobi podatke iz podatkovne baze in pošlje nazaj pridobljene podatke v JSON obliki. Spletna aplikacija upodobi stran v brskalniku in tako smo se rešili problema ponovnega nalaganja celotne strani. Pridobili smo sposobnost, da posodobimo le del spletne strani, kar zagotovi tekočo uporabniško izkušnjo, podobno kot bi uporabljali namizno aplikacijo, naloženo na računalnik.

Stanje SPA ostane v brskalniku in JavaScript ogrodja, kot so ReactJS, VueJS, AppRun idr. upravljajo s stanji v brskalniku in posodabljajo prikazano vsebino deloma in dinamično. Da aplikacije ne postanejo neobvladljivo velike, ta ogrodja podpirajo uporabo komponent kot gradnikov za gradnjo SPA. Ti gradniki so organizirani in upravljani s pomočjo ECMAScript (ES) moduli. Komponente med seboj komunicirajo preko dogodkov [14] [15].

Za namene tega diplomskega dela sem kot reprezentativnega predstavnika delovanja SPA izbral knjižnico React. React je JavaScript knjižnica za gradnjo uporabniških vmesnikov. Je deklarativna, osnovana na komponentah in deluje na več platformah. React sem izbral, saj je StackOverflow Developer Study 2020

pokazala, da je ta tehnologija med spletnimi ogrodji najbolj popularna in hkrati tudi najbolj priljubljena med SPA ogrodji – razvidno s Slike 2-6. Ta podatek je veljal za 42.279 vseh udeležencev ankete in 36.291 poklicnih udeležencev ankete [16]. Na voljo so tudi podatki o samih udeležencih, njihovem delu, lokaciji ipd. Tako da smemo sklepati, da je React dobro razširjen po vsem svetu tako v profesionalni kot v ljubiteljski uporabi.




Slika 4-1: Rezultati raziskave o spletnih ogrodjih [16].

Kot že omenjeno, je React orientiran na komponente. Skoraj vsa vsebina in element v aplikaciji sta predstavljena kot ponovno uporabljiva komponenta. Podpira tudi navidezni *Document Object Model* (DOM), kar naredi proces upodabljanja in spreminjanja komponent opazno hitreje in nadzorljivo. Poleg tega

omogoča pisanje HTML in JavaScript datotek z uporabo *JavaScript XML* (JSX) sintakse. Nato prevajalniki, kot so Babel, pretvorijo JSX v JavaScript kodo. To pripomore k temu, da razvijalci urejajo obnašanje in videz komponent zelo učinkovito. Kljub vsej tej moči, pa React ostaja zelo lahko ogrodje v primerjavi z ostalimi bolj uporabljenimi [17].

## 4.1 Samopostrežni zaledni sistemi in enostranske spletne aplikacije

Zakaj so samopostrežni zaledni sistemi tako dobri pri sodelovanju z enostranskimi spletnimi aplikacijami in orodji, kot je React? Na podlagi izbora Firebase-a v knjigi »Progressive Web Apps with Angular« ugotovimo, da je avtorja ta prepričal s svojo enostavnostjo uporabe in možnostjo dopolnitve pomanjkljivosti s funkcijami kot storitev [18]. Če bolje definiramo lastnosti samopostrežnih zalednih sistemov v oblaku za enostranske spletne aplikacije, so te:

- enostavnost vzpostavitve,
- nabor uporabnih funkcionalnosti,
- dopolnitev manjkajočih funkcionalnosti,
- abstrakcija zaledja,
- podprtost znotraj izbranega orodja.

Abstrakcija zaledja je zelo močan pripomoček pri razvoju enostranskih spletnih aplikacij, saj nam omogoči popolno osredotočenost na razvoj tistega dela aplikacije, ki bo imel stik z uporabniki. Enostranske spletne aplikacije se poslužujejo manipulacije svojega stanja. Tako imajo možnost naročanja na različne dogodke ob spremembi tega stanja, tako da lahko dobro izkoristimo, kdaj so podatki z našega zaledja na voljo, v čakanju ali nedosegljivi.

Poleg abstrakcije zaledja nam samopostrežni zaledni sistemi v oblaku ponudijo že kar nekaj funkcionalnosti znotraj tega, kot je delo s podatkovno bazo, hrambo datotek in vpeljavo različnih metod avtentifikacije. Seveda se ponudba vnaprej pripravljenih funkcionalnosti močno spreminja od ponudnika do ponudnika. Pomembno je tudi, da v primeru nepopolne ponudbe funkcionalnosti, naš ponudnik omogoča definiranje lastnih funkcionalnosti.

Vse to pomeni, da ob izboru pravega ponudnika, ki nudi dovolj funkcionalnosti, ki jih želimo uporabiti, omogoči enostavno vzpostavitev projekta. Naš projekt je nato pripravljen za nadaljnje delo in širjenje funkcionalnosti, glede na izkazane potrebe pridobljenih uporabnikov. Prednosti enostavne vzpostavitve so tudi hiter čas na trgu, hitro preverjanje, ali je naša aplikacija finančno upravičena.

Za lažji razvoj naše aplikacije je vedno dobrodošla dobra podpora orodij, s katerimi delamo. Pri sodelovanju med samopostrežnimi zalednimi sistemi v oblaku in enostranskimi spletnimi aplikacijami se ta kaže v obliki knjižnic, saj te naredijo veliko za razvijalce v obliki abstrahiranja ponavljajoče kode in poenostavitvi težjih konceptov.

# 5 PRIMERI REŠITEV SAMOPOSTREŽNIH ZALEDNIH SISTEMOV V OBLAKU

## 5.1 Definicija kriterijev

Ko izbiramo zaledno rešitev za našo aplikacijo, je ključnega pomena, da se zavedamo naših potreb in omejitev. Nepravilna odločitev lahko v primeru oblačnih ponudnikov povzroči ogromne stroške in dolgoročne probleme, saj nas ponudniki s svojim sistemom dela omejijo le na njihove storitve. Zato je eden od večjih dejavnikov odločitve ponudnika transparentnost zaračunavanja storitev. Za naše potrebe smo definirali naslednje kriterije, na katere moramo biti pozorni [10] [18]:

- **Cenitev storitev:** Ključnega pomena je, kako se naše storitve zaračunajo in kakšen nadzor imamo nad tem. Je možno nastaviti dogodke, ko se približujemo določenim vsotam? Je možno omejiti stroške in storitev zapreti, ko dosežemo določene stroške? Lahko storitve poženemo brez podatkov kreditne kartice? Vse to mora močno vplivati na našo odločitev, saj je v končnih fazah aplikacije vedno cilj plačati najmanj in zaslužiti večje zneske.
- **Prilagodljivost:** Kako težko je izbrano orodje prilagoditi v nekaj, kar je izven nastavitev tega? To vprašanje je lahko zelo pomembno za našo

dolgoročno vizijo, ali želimo na ponudnika ostati vezani trajno ali samo v infantilnih fazah projekta.

- **Integracija z ostalimi storitvami:** Katere ostale storitve nam lahko orodje poleg primarnega namena še zagotovi? Potrebe današnjih podjetij se zelo naglo spreminjajo in ponudba domorodne podatkovne baze, strojnega učenja, sistemov e-pošte lahko pritegne mnogo strank. Takšne potrebe po hitrih obvestilih preko e-pošte ali česa podobnega se lahko pojavijo čez noč zaradi kakšnih novih zakonov ali drugih sprememb v okolju projekta.
- **Enostavnost uporabe:** Kako enostavno je orodje za uporabo? Ko je ena izmed glavnih prednosti ponudnikov ta, da zagotavljajo enostavno izkušnjo razvoja, mora biti tudi samo upravljanje enostavno s prijaznim grafičnim vmesnikom ipd.
- **Programski jezik:** Katere jezike podpira orodje? Zelo pomembno je, da ponudnik storitve podpira jezik, s katerimi smo si domači, saj lahko nasproten primer močno oteži postopek razvoja. Če uporabljamo jezik, ki ga ponudnik ne podpira, ali so določene funkcionalnosti še vedno na voljo?
- **Velikost skupnosti:** Kadarkoli se lotevamo novega projekta, je zelo pomembno, da se zavedamo, da če nismo eksperti, bomo potrebovali pomoč v vsaj neki meri. Skupnost lahko v tem primeru močno vpliva na odločitev, ki jo bomo sprejeli in olajša naše življenje. Skupnost je lahko tudi eden izmed faktorjev, ki vpliva na kredibilnost naše aplikacije.
- **Varnost:** Kako je poskrbljeno za varnost? Kakšna je podpora avtentifikacije, avtorizacije ipd. zadev? Dobro je treba premisliti, kakšna je trenutna potreba po varnosti, kako shranjujemo občutljive podatke o uporabnikih in zagotovitev zakonodajne skladnosti, še posebno v okoliščinah, ko sta zasebnost in varnost poudarjeni.
- **Strojno učenje:** Današnje aplikacije dajejo občutek zastarelosti, če vsaj ne razmišljamo o uporabni nekakšnega strojnega učenja. Kako nam lahko

ponudnik to olajša? Razvoj lastnih algoritmov strojnega učenja je lahko mukotrpno delo in navadno lastne rešitve, če se v njih res ne specializiramo, niso konkurenčne tistim, ki so jih večji ponudniki že dovršili.

- **Enostavnost vzpostavitve:** Sama vzpostavitev projekta je lahko eden izmed najbolj zahtevnih korakov, saj je treba predvideti trenutne in nadaljnje potrebe projekta. Kakšen je postopek tega z našim izbranim orodjem in ali uporabi že prej nastavljeno tehnologijo, ki jo bomo uporabljali za predstavitveni del aplikacije?
- **Orodja za razvijalce:** Kako naša izbira olajša delo razvijalcem? Kakšne so možnosti razhroščevanja, je zelo pomemben del naše odločitve za orodje, saj je razhroščevanje najboljši pripomoček pri odkrivanju napak in mora biti podprto v zreli obliki.
- **Beleženje in monitoring:** Tudi ta kriterij je zelo pomemben pri odločitvi za določeno orodje, saj je to naš vpogled v delovanje aplikacije v realnem svetu. Dobro dodelano beleženje v kombinaciji z monitoringom naše aplikacije nas lahko opozori na morebitne pomanjkljivosti v aplikaciji in nam jih tako uspe zatreti, preden te povzročijo astronomske stroške [19]. Lahko pa naredi tudi kar nekaj v smeri varnosti, saj lahko z dobro nastavljenim monitoringom opazimo, da je aplikacija napadana in jo s tem znanjem upravljamo.

## 5.2 Rešitev z REST vmesnikom

Ko govorimo o lastni REST rešitvi, je lahko že sama izbira, za katero rešitev se odločiti, projekt zase. Ob poplavi različnih ogrodij za spletne vsebine si lahko izberemo katerikoli jezik, ki ga poznamo. Prilagodljivosti teh sistemov je spet odvisna od samega izbora, pa vendar je tu vse v naših rokah, da se odločimo in izberemo. Torej noben oblačni ponudnik ne more preseči prilagodljivosti lastne REST rešitve. Integracija z ostalimi rešitvami tudi ni problematična, saj vse, kar podpira komunikacijo brez stanja, lahko komunicira z našim orodjem. Enostavnost uporabe je vsekakor slabša kot pri ponudnikih oblačnih storitev, saj je to ena od

glavnih prednosti tega načina delovanja. Ko razvijamo lasten REST vmesnik, je spet zelo odvisno, katerega izberemo. Pogosto je tudi tukaj lahko zelo različno, saj ogrodja, kot so Laravel, močno lajšajo naše življenje, ampak zahtevajo kar nekaj truda za prilagoditev nekemu bolj robnemu primeru. Velikost skupnosti je lahko zelo različna, saj so nekatera orodja bolj zrela kot druga in je težko konkretno določiti neki skupni imenovalec za vsa ta orodja. To je nekako tematika tega, da si sami razvijemo REST vmesnik, saj imamo vso kontrolo v rokah mi, imamo pa v rokah tudi vso odgovornost in je treba temu primerno oceniti čas trajanja in stroške razvoja naše rešitve [20].

Cenitev storitve je lahko dokaj zapletena, ko govorimo o lastnem REST vmesniku. Začetna investicija postavitve lastnega sistema bo nedvomno dražja, vendar pa na dolgi rok lahko pričakujemo povrnitev stroškov, če načrtujemo izvajanje nekaterih operacij na zalednem sistemu, ki so neprimerne oz. cenovno neugodne za poganjanje na samopostrežni strežniški arhitekturi. Lahko pa kar nekaj stroškov pri izračunu dolgoročnega vzdrževanja zgrešimo in si tako naredimo medvedjo uslugo. Stroški namenskih delavcev za vzdrževanje, zagotovitev ugodnih prostorov po svetu za delovanje naših strežnikov in podobni problemi lahko hitro porastejo v cenovne poraze [3].

Varnost je mogoče najbolj vprašljiva zadeva, ko se je lotevamo sami. Tu je dobra praksa vedno uporabiti nekaj, kar je že bilo vnaprej pripravljeno in temeljito testirano s strani ljudi, ki se s tem dalj časa ukvarjajo. Ko sami pišemo prijave in registracije, se pogosto zgodi, da na kakšne malenkosti pozabimo in je naša aplikacija varnostno vprašljiva. Vsekakor lahko, če uporabimo dobro poznano in preverjeno knjižnico, zagotovimo dovolj dobro varnost. Ampak na nas vseeno pade posodabljanje in spremljanje novic o tej knjižnici ter menjavi te, če se izkaže, da ta ni več zadovoljiva. Orodja za razvijalce tukaj pridejo v obliki IDE okolij in so, seveda odvisno od uporabljenega IDE, lahko vrhunec tehnologije ali pa osnovno razhroščevanje. Kar teče na naših lokalnih napravah, lahko vedno bolj temeljito spremljamo in iščemo probleme s pomočjo raznih orodij za spremljanje delovanja aplikacije in nadzorovanjem porabe virov sistema. Oblačni ponudniki vedno šepajo

v ozadju, ko jih primerjamo s temi orodji, pa vendar je zrelost nekaterih, kot je Firebase, že skoraj ujela ta nemogoči tempo. Vključitev strojnega učenja v aplikacijo je ponovno povsem v naših rokah, ampak se v večini primerov preda zunanjim ponudnikom, saj so ti v tej panogi bolj izkušeni [14].

## 5.3 Amazon Web Services Amplify

AWS Amplify je zelo dovršena in zrela rešitev, ki nudi ogromno finih nastavitev svojih storitev in bogato ponudbo komplementarnih storitev, ki olajšajo delo razvijalca. Problem nastopi pri iskanju vseh teh nastavitev, saj grafični vmesnik ni tako kvalitetno dodelan in zahteva kar nekaj učenja, preden znamo te stvari dobro nastaviti. Posledično enostavnost uporabe tukaj trpi. Skupnost je ogromna, saj je AWS vodilni igralec med ponudniki tovrstnih storitev in ima trenutno največji tržni delež. Velik faktor tega izhaja iz dejstva, da je bil tudi prvi na sceni s ponudbo teh storitev in vsi ostali ponudniki efektivno trgajo od tega deleža s svojimi ponudbami. AWS Amplify podpira iOS, Android, spletne in React domorodne aplikacije. Za spletne aplikacije podpira tudi globoko integracijo z React, Ionic, Angular in pa Vue.js. Kot že prej omenjeno, je integracija z ostalimi storitvami vrhunska, sploh s pomočjo Amplify CLI, kjer s samo nekaj ukazi določimo storitve, ki jih želimo koristiti in te se v naš projekt dodajo avtomatsko s samodejno dodelitvijo virov v oblaku [21].

Cenitev AWS Amplify je lahko na trenutke zahrbtna in varljiva. Že za to, da pridemo do samih cenitev storitev, je treba kar nekaj brskanja in kopanja znotraj različnih menijev. Poleg tega je cenitev odvisna še od virov, ki jih funkciji namenimo. AWS sicer nudi nekaj primerov cenitve teh storitev, vendar ni ravno vizualno privlačno - prikazano na Sliki 3.1.

Example 1

A startup team with 5 developers have an app that has 300 daily active users. The team commits code 2 times per day.

**Monthly build & deploy charges**

- Assumptions: Average build time = 3 mins; Number of work days/month = 20
- Total build time per month = num of devs * num of commits/day * num of days * avg. build time = 5*2*20*3 = 600 build mins per month
- **Monthly build & deploy charges = 600*.01 = $6**

**Monthly hosting charges**

- Assumptions: Web app size = 25 MB, average size of page requested = 1.5 MB
- Monthly GB served = Daily active users * average page size * days = 300 * (1.5/1024) * 30 = 13.18 GB
- Monthly GB stored = web app size * number of monthly builds = (25/1024)*(5*2*20) = 4.88 GB
- **Monthly hosting charges = 13.18*$0.15 + 4.88*$0.023= $1.97 + $0.11 = $2.08**

**Total monthly charges**

**Total charges = Build & deploy charges + Hosting charges = $6+$2.08 = $8.08 per month**

Slika 5-1: Prikaz informativnega izračuna uporabe storitev Amplify [22]

Postopek iskanja cenitve in potem samostojno preračunati vsak strošek posebej, je treba ponoviti za vsako storitev, ki jo želimo dodati projektu. Ta postopek je zelo neprijazen in neodporen na napake v izračunih brez kakršnekoli odgovornosti s strani AWS. AWS Amplify sicer nudi brezplačne storitve do neke mere, vendar zahteva podatke za plačilo morebitnih stroškov, saj ko porabimo brezplačne storitve, naprej računa po redni ceni. Ponovno lahko nastanejo neželeni stroški, če si predhodno sami ne nastavimo celovitega sistema obveščanja ob porastu stroškov [23].

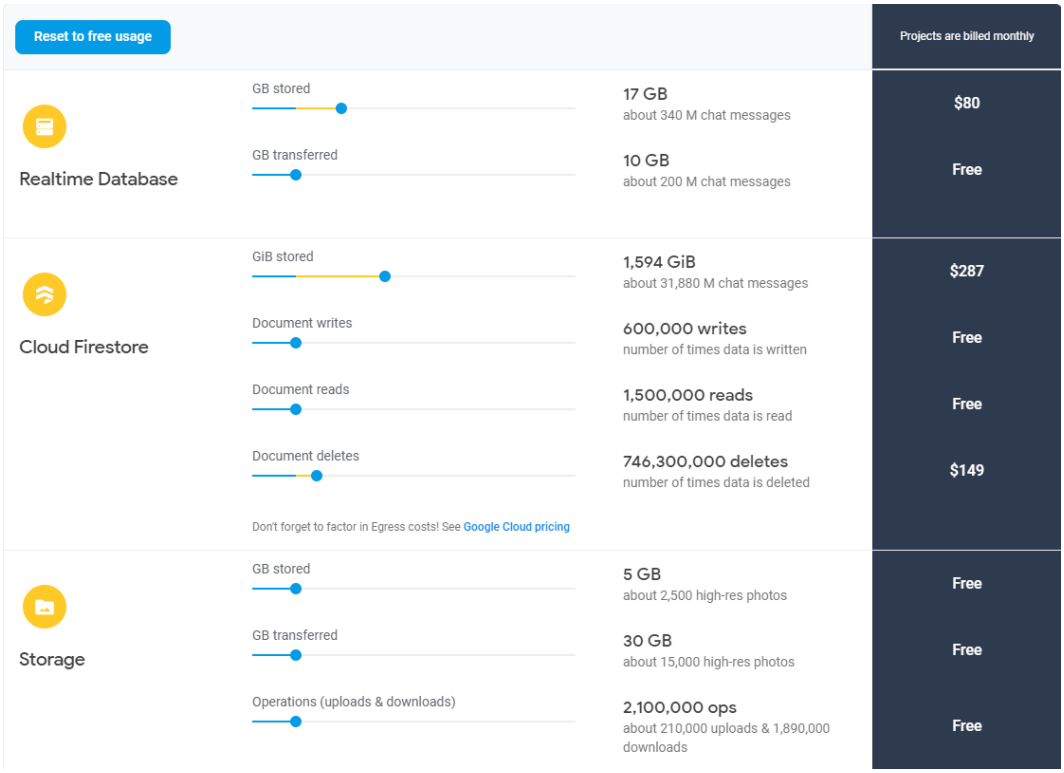
Varnost je zelo dobra na strani AWS Amplify, saj nudi različne oblike overjanja uporabnikove identitete (sms, e-mail, klic, biometrija, prepoznavanje obraza …), in pa seveda večstopenjsko avtentifikacijo. Orodja za razvijalce so zelo dobra, kot se pričakuje od ponudnika oblačnih storitev, ki je najdlje na tržišču. S pomočjo konzole in analitičnih izpisov je rešitev še kako kompetentna za vsakdanje potrebe razvijalcev, četudi ob določenih trenutkih nekoliko neintuitivna. Uporaba strojnega učenja je dobra, funkcionalnosti, ki jih zagotavlja, je kar nekaj - od prepoznavanja slavnih oseb, besedila, generiranje besedila ipd. [22]

## 5.4 Firebase

Firebase je Googlov prispevek na sceno ponudnikov oblačnih storitev, vendar je podjetje v začetku še bilo samostojno. Gre za zelo uporabniku prijazno in

enostavno izkušnjo pri postavitvi vsega potrebnega za nadomestitev strežnika. Ni toliko prilagodljiva rešitev kot AWS, pa vendar nadomesti svojo pomanjkljivost z enostavno postavitvijo in razširitvijo z drugimi storitvami. Firebase za razliko od Amplify nudi tudi gostovanje spletnih vsebin z dinamično vsebino in ne le statično. Skupnost ni tako velika, je pa ta zelo zagnana in pripravljena priskočiti na pomoč razvijalcem. Ogromno število pozitivno naravnanih člankov priča o vzponu potencialnega novega vodilnega ponudnika, vendar bo za to spremembo potrebnega še kar nekaj časa [23]. Uradni programski jeziki so Node.js (JavaScript), Java, Python, Go, C# [24].

Cenitev pri Firebase je veliko prijetnejša in intuitivna kot pri Amplify. Vsak lahko za svoje primere pogleda in nastavi informativni izračun za lastne potrebe ter na enem mestu so zbrani vsi izdelki, kot je prikazano na Sliki 3.2.




Slika 5-2: Prikaz informativnega izračuna stroškov za uporabo Firebase storitev [25]

Veliko člankov na spletu se strinja v tem, da je Firebase cenovno bolj ugoden za manjša podjetja, saj se Amplify poceni, ko zakupimo dovolj veliko število virov [21] [23] [26].

Za varnost je poskrbljeno na vse možne načine, preko več različnih poti overjanja uporabnika, telefonskega klica, sms-a, večstopenjske avtentifikacije in podobnih zadev. Orodja za razvijalce so v tem primeru zelo dovršena in smatrana kot najboljša trenutno na voljo med ponudniki oblačnih storitev. Firebase nudi tudi analitiko spletnega mesta kot samostojno storitev in je tudi ta zelo izčrpna ter enostavna za vzpostavitev. Poleg temeljitega monitoringa in beleženja nam Firebase nudi tudi možnost testiranja aplikacije na raznih mobilnih napravah, kar pa nam sicer za enostransko spletno aplikacijo kaj dosti ne koristi. Poleg tega je Googlov Vision API med najboljšimi za strojno učenje. Lahko ga tudi sami naučimo klasifikacijskega prepoznavanja predmetov na sliki, kot je recimo hrana itd. [25]

## 5.5 Primerjava

Za lažjo vizualizacijo problema smo vse tri možne kandidate zvrstili v preglednico in kategorizirali njihove uspehe v določeni kategoriji z vrednostmi od 1 do 3. To pomeni, da je primerjava narejena v vrstnem redu, katero orodje je v posamezni kategoriji najboljše. Vrednost 1 predstavlja, da je orodje najslabše, vrednost 3 pa pomeni, da je najboljše. Rezultati so prikazani v Tabeli 3.1.

| Kategorija\Orodje | Lasten REST | AWS Amplify | Google Firebase |
|---|---|---|---|
| **Prilagodljivost** | 3 | 2 | 1 |
| **Integracija z ostalimi storitvami** | 1 | 2 | 3 |
| **Enostavnost uporabe** | 1 | 2 | 3 |
| **Enostavnost vzpostavitve** | 1 | 2 | 3 |
| **Velikost skupnosti** | 1 | 3 | 2 |
| **Programski jeziki** | 3 | 1 | 2 |
| **Cenitev storitev** | 1 | 2 | 3 |

| | | | |
|---|---|---|---|
| **Varnost** | 1 | 2 | 3 |
| **Orodja za razvijalce** | 3 | 1 | 2 |
| **Beleženje in monitoring** | 1 | 2 | 3 |
| **Strojno učenje** | 1 | 2 | 3 |
| **Skupaj:** | **17** | **21** | **28** |

Tabela 5-1: Primerjava orodij med seboj po primerjalni lestvici

Vidimo lahko, da je zmagovalec Firebase. Imeli smo dvome glede tega, katero orodje bo na vrhu, ampak se je izkazalo, da ko kriterije nastavimo za namene razvoja enostranskih spletnih aplikacij, je Firebase vodilna odločitev.

Vse ocene so osnovane na lastnih ugotovitvah v raziskovanju teh orodij in preteklih izkušnjah. Utemeljitev ocen:

- **Prilagodljivost**: Nad lastnimi REST rešitvami imamo največji nadzor nad tem, kaj vse nam ponujajo, zato torej dobijo največ točk – 3. AWS Amplify sledi s prilagodljivostjo in dobi 2 točki, saj je pri naboru ponujenih funkcionalnosti cel kup nastavitev. Firebase je najslabši, saj nam ne nudi ogromno finih nastavitev.
- **Integracija z ostalimi storitvami:** Lastne REST storitve niso privzeto narejene za integracijo z zunanjimi storitvami, torej dobijo 1 točko. AWS Amplify ponuja veliko lastnih storitev in nekaj zunanjih, tako da dobi 2 točki. Firebase ponuja ogromno lastnik storitev, saj spada v Google-ov poslovni model in tako dostopa do vseh teh storitev zelo elegantno – zato dobi 3 točke.
- **Enostavnost uporabe:** Tu je lastna REST rešitev zopet zadnja, saj zagotavlja prilagodljivost, ne pa enostavnosti – 1 točka. AWS Amplify zaradi svojih množičnih finih nastavitev in neprijaznimi grafičnimi vmesniki ne zagotavlja enostavnosti uporabe – 2 točki. Firebase pa zaradi

svoje neprilagodljivosti in uporabniškega vmesnika zagotavlja najenostavnejšo uporabo in prejme 3 točke.

- **Enostavnost vzpostavitve:** Zopet je REST rešitev najslabša z 1 točko, saj smo sami odgovorni za postavitev celotnega sistema. AWS Amplify je v sredini, saj je še vedno lažje kot pri REST rešitvi in prejme 2 točki, Firebase pa vodi s svojo enostavnostjo vzpostavitve zaradi malega števila nastavljivih parametrov – 3 točke.
- **Velikost skupnosti:** Ko razvijamo lasten REST vmesnik, smo vedno omejeni na orodje, s katerim želimo delati in na dejstvo, da je trend takšnih rešitev v zatonu, zato imamo najmanjšo skupnost, ki nam lahko pomaga – 1. AWS Amplify s svojim vstopom na trg, kot prvi ponudnik tovrstnih storitev, ima največjo bazo uporabnikov, ki so zelo vešči na tej platformi – 3 točke. Firebase je tukaj v sredini, saj je novejši med ponudniki in z manjšo bazo uporabnikov, so pa ti med bolj zagretimi za Google-ove tehnologije – 2 točki.
- **Programski jeziki:** Lastna REST rešitev nam omogoča popoln nadzor nad tem, s katerim jezikom bomo razvijali in si prisluži 3 točke. Ker nam je bolj pomembno, koliko jezikov že podpira ogrodje brez dodatnega dela, si AWS Amplify prisluži 1 točko, čeprav bi lahko z dodatnim delom sami razvili vmesnike za delo z drugimi jeziki. Firebase podpira več jezikov brez dodatnega dela in si prisluži 2 točki.
- **Cenitev storitev:** Lastna REST storitev nas lahko stane ogromno, saj jo moramo poleg razvoja še namestiti nekam in to bo vedno dražje kot uporaba oblačnega ponudnika – 1 točka. AWS Amplify je naravnan tako, da kratkoročno kar nekaj stane in začetne cene so dokaj visoke, ko pa porabimo več in več virov, pa te postanejo cenejše. Ker je v ta kriterij zajet tudi nivo transparentnosti in enostavnost izračuna stroškov uporabe storitev, si prisluži le 2 točki. Firebase prejme 3 točke, saj je pri manjši porabi bolj ugoden in izračun stroškov je uporabniku prijazen.

- **Varnost:** Lasten REST je lahko zelo varna rešitev, vendar le, ko tej posvetimo velik del razvoja, zato prejme le 1 točko. Ostali rešitvi namreč ponujata večjo stopnjo varnosti z zelo malo truda. AWS Amplify in Firebase nudita podobno stopnjo rešitev varnosti, vendar Firebase to ponuja na razvijalcem in uporabnikom prijazen način, zato prejme 3 točke, AWS Amplify pa 2.
- **Orodja za razvijalce:** Lasten REST nam ponuja največ načinov vpogleda v delovanje aplikacije z naprednimi IDE in drugimi orodji, torej prejme 3 točke. AWS Amplify omogoča manj kot Firebase in prejme 1 točko, Firebase pa 2.
- **Beleženje in monitoring:** Pri lastni REST rešitvi je ponovno potrebno ta orodja razviti in vzdrževati samostojno, torej prejme 1 točko. AWS Amplify nudi kar nekaj funkcionalnosti, vendar do manjše mere kot Firebase, zato prejme 2 točki. Firebase nudi najlepše oblikovane in napredne storitve v tem pogledu in prejme 3 točke.
- **Strojno učenje:** Lasten REST nam v tem ne pomaga nič, razen možnosti uporabe zunanjih knjižnic, ki jih moramo spet uporabiti sami in prejme 1 točno. AWS Amplify nudi dovršene rešitve za strojno učenje, vendar ga Firebase s pomočjo ogromno podatkov pridobljenih od Google-a prekosi. AWS Amplify prejme 2 točki, Firebase pa 3.

## 5.6 Izbor

Kot je pokazano v prejšnjem poglavju, je za namene enostranskih spletnih aplikacij najboljše orodje Firebase. Do tega sklepa smo prišli, ker podpira največ funkcionalnosti, a ob tem še vedno ostane zelo enostaven in cenovno ugoden. Problemi se znajo pojaviti pri aplikacijah večjega obsega, vendar za namene našega primera ne bi smelo biti nikakršnih problemov in je odločitev s tem znanjem skoraj samoumevna. Poleg tega je za nekoga, ki se samopostrežnih sistemov šele loteva prvič, dobra dokumentacija, zavzeta skupnost in transparentno cenjenje storitev, med najbolj privlačnimi.

# 6 RAZVOJ ZALEDNEGA SISTEMA FIREBASE IN APLIKACIJE

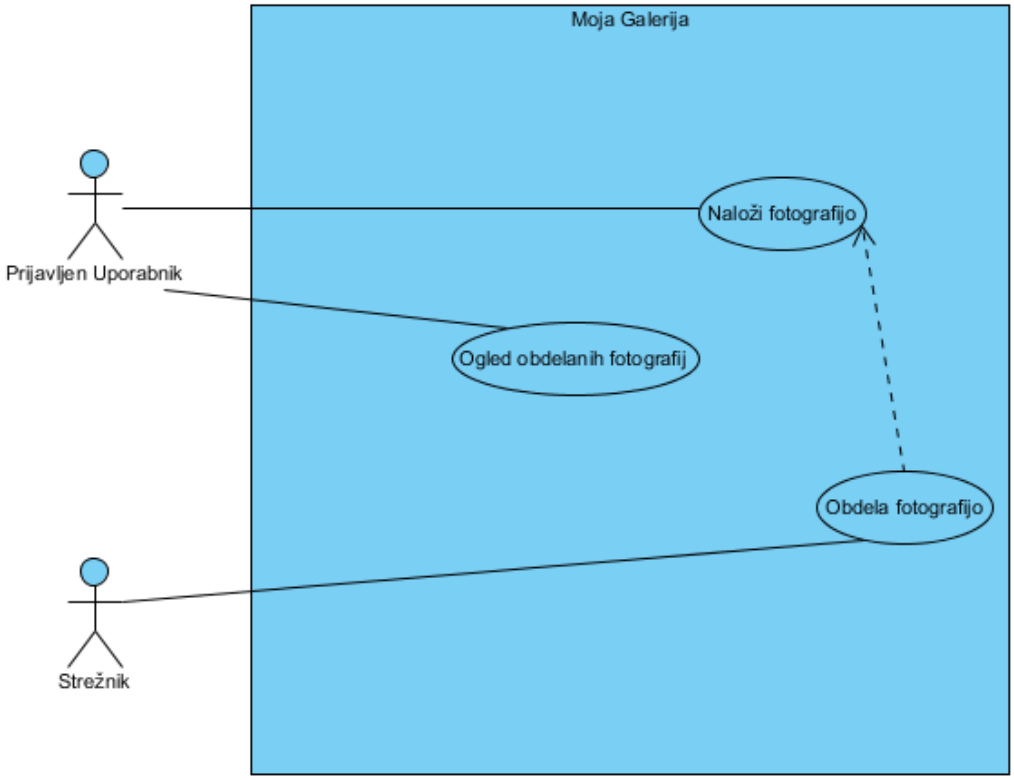
## 6.1 Funkcionalnosti

Za potrditev ugotovitve prejšnjega poglavja – da je Firebase najboljši za uporabo v kombinaciji z enostranskimi spletnimi aplikacijami, smo razvili enostavno spletno aplikacijo. Razvili smo enostavno spletno galerijo, kamor bodo lahko uporabniki nalagali svoje slike. Te se bodo stisnile s pomočjo oblačnih funkcij in se stisnjene prikazale uporabnikom, ki se bodo prijavili z Google računom. Celotna aplikacija je gostovana na Firebase in javno dostopna preko HTTPS. Tako smo zajeli uporabo štirih storitev Firebase (gostovanje, shranjevanje, avtentikacijo in oblačne funkcije) in preverili ali je res delo s Firebase enostavno in primerno za delo z React-om.

Funkcionalnosti lahko ločimo na:

- vpis,
- naložitev fotografije,
- obdelava fotografije,
- prikaz obdelane fotografije.

Tako smo dobili jedrnat vpogled v sodelovanje Firebase-a z React-om in lahko ocenili, ali je res tako razvijalcu prijazno, kot je zatrjeno v prejšnjih poglavjih. Primere uporabe si lahko za lažjo predstavo narišemo s pomočjo diagrama primerov uporabe (Slika 6-1). S tega diagrama je tudi razvidno, da je za dostop do funkcionalnosti potreben prijavljen uporabnik.



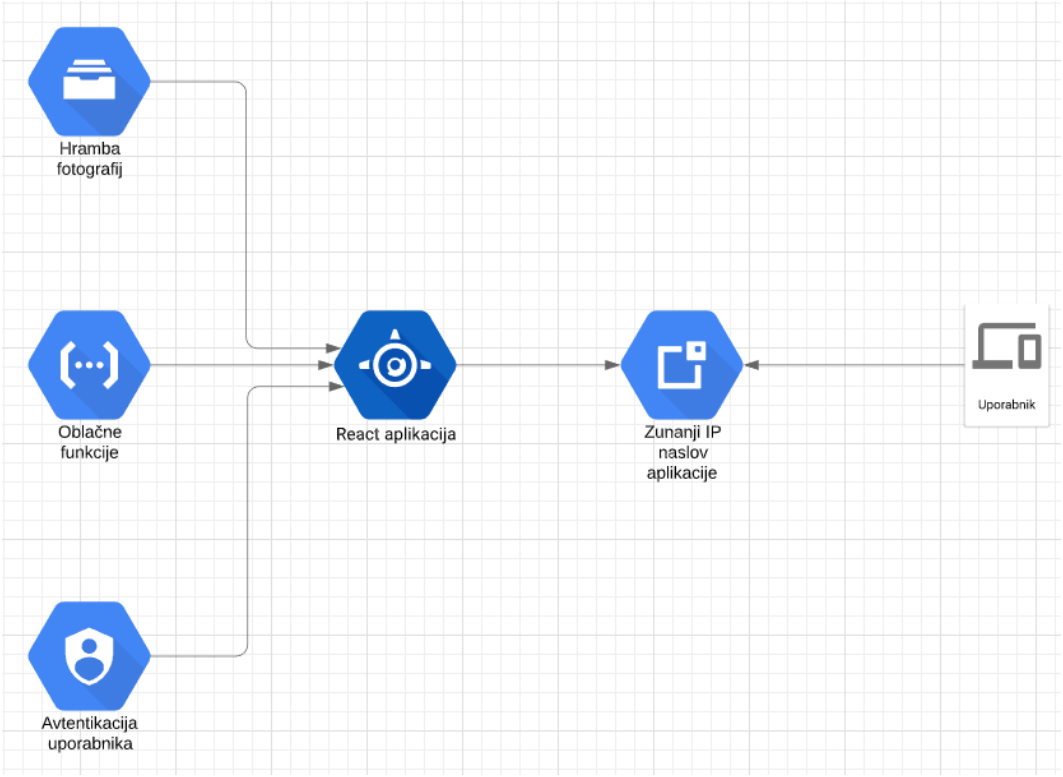

Slika 6-1: Diagram primerov uporabe za aplikacijo Moja Galerija

Končen cilj aplikacije lahko vidimo na Sliki 6-2. Vidimo osnovno obliko spletne aplikacije s prikazanimi obdelanimi fotografijami na standardno velikost za našo aplikacijo.


Slika 6-2: Končna aplikacija "Moja Galerija"

## 6.2 Arhitektura

Za boljšo predstavo, kako našo aplikacijo uporaba oblačnih storitev modulira, si oglejmo diagram arhitekture (Slika 6-3). Vidimo, da se uporabnik povezuje na zunanji IP naslov, ki nam ga zagotovi uporaba Firebase gostovanja. Ta poskrbi, da ni pomembno, od kod se uporabnik povezuje na našo aplikacijo, vedno je odzivni čas hiter. Uporabnik dobi servirano našo React aplikacijo, ki se naloži na njegovo napravo in se tam naprej izvaja. Ko aplikacija za svoje nadaljnje delovanje potrebuje dodatne zunanje storitve, kliče te. To pomeni, da vsaka storitev deluje za sebe in uporaba ostalih na njo ne vpliva, kar zagotovi najhitrejšo in tekočo uporabniško izkušnjo.




Slika 6-3: Arhitektura aplikacije, ki smo jo razvili

## 6.3 Priprava okolja

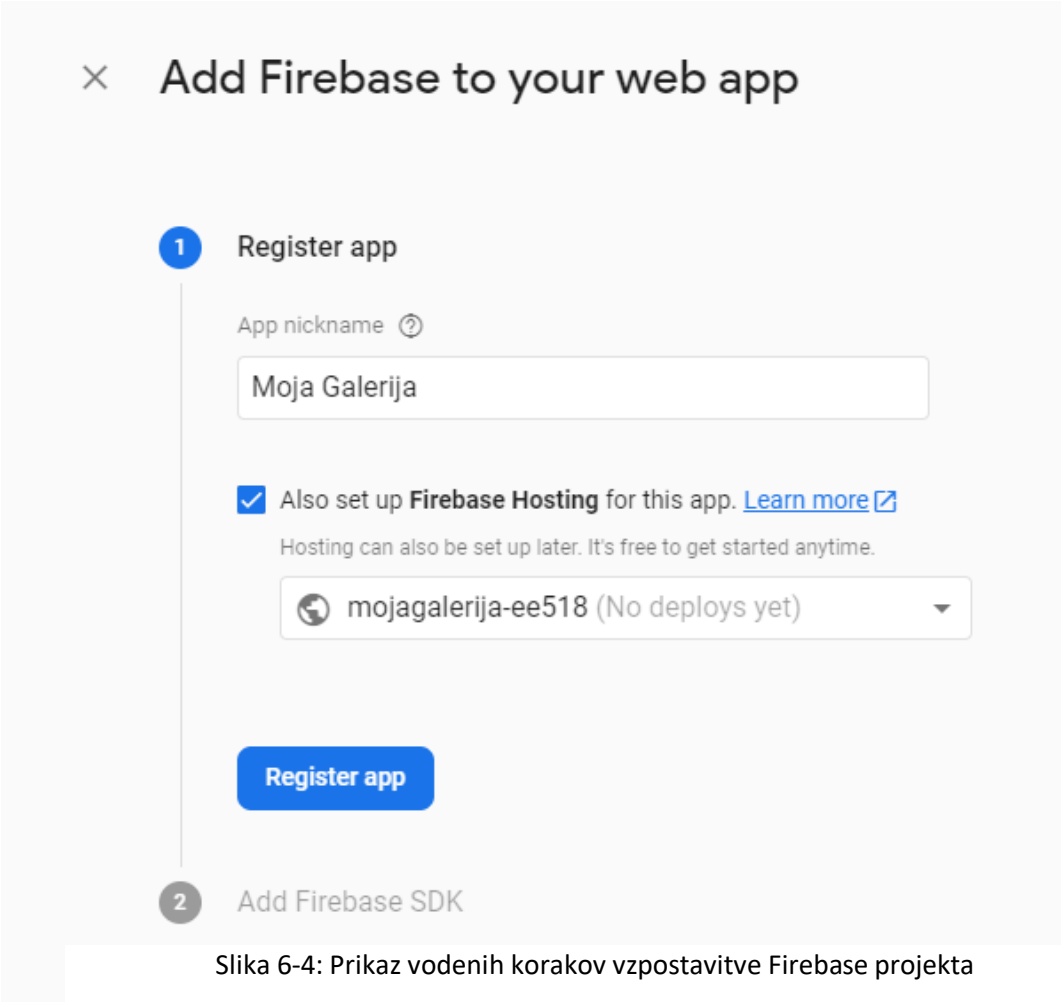
Razvoj samopostrežnega zalednega sistema s Firebase je preprost. Sistemi, ki že razvijajo s pomočjo tehnologije React, že imajo večino namestitve urejene. Potreben je neki osnovni urejevalnik kode, konzola za poganjanje (Node Package

Manager) npm ukazov in pa seveda NodeJS. Tehnološki sklad uporabljen za razvoj v tem primeru in verzije:

- Visual Studio Code,
- NodeJS (10.16.0),
- Npm (6.9.0),
- Npx (10.2.2),
- Firebase Tools (8.6.0).

Predpostavili smo, da so NodeJS, npm, npx in Visual Studio Code že naloženi, torej ostane samo še namestitev Firebase Tools, skozi katerega nas bodo vodila nadaljnja navodila. S pomočjo ukaza »npx create-react-app mojagalerija«, ustvarimo nov React projekt, ki ga bomo registrirali kot Firebase projekt. Nato obiščemo spletno stran https://console.firebase.google.com, kjer ustvarimo nov projekt z imenom »MojaGalerija«. Preden registriramo našo aplikacijo, se s pomočjo desnega stranskega menija pomaknemo na »Storage«, kjer pritisnemo na »Getting started« in sprejmemo privzete nastavitve. Nato je treba spremeniti plačilni račun na »Blaze«, saj oblačne funkcije zahtevajo podatke o plačniku, preden lahko namestimo aplikacijo. Potem se vrnemo domov in kliknemo ikono za registracijo spletne aplikacije. Ko izberemo ime, označimo tudi možnost, »Also set up Firebase Hosting for this app.«, (Slika 6-4). To bo zagotovilo, da bo naša aplikacija dostopna na javni domeni. Naslednji korak »Add Firebase SDK«

preskočimo, saj je postopek za postavitev React aplikacij drugačen [27]. Tega bomo izvedli kasneje znotraj naše aplikacije.

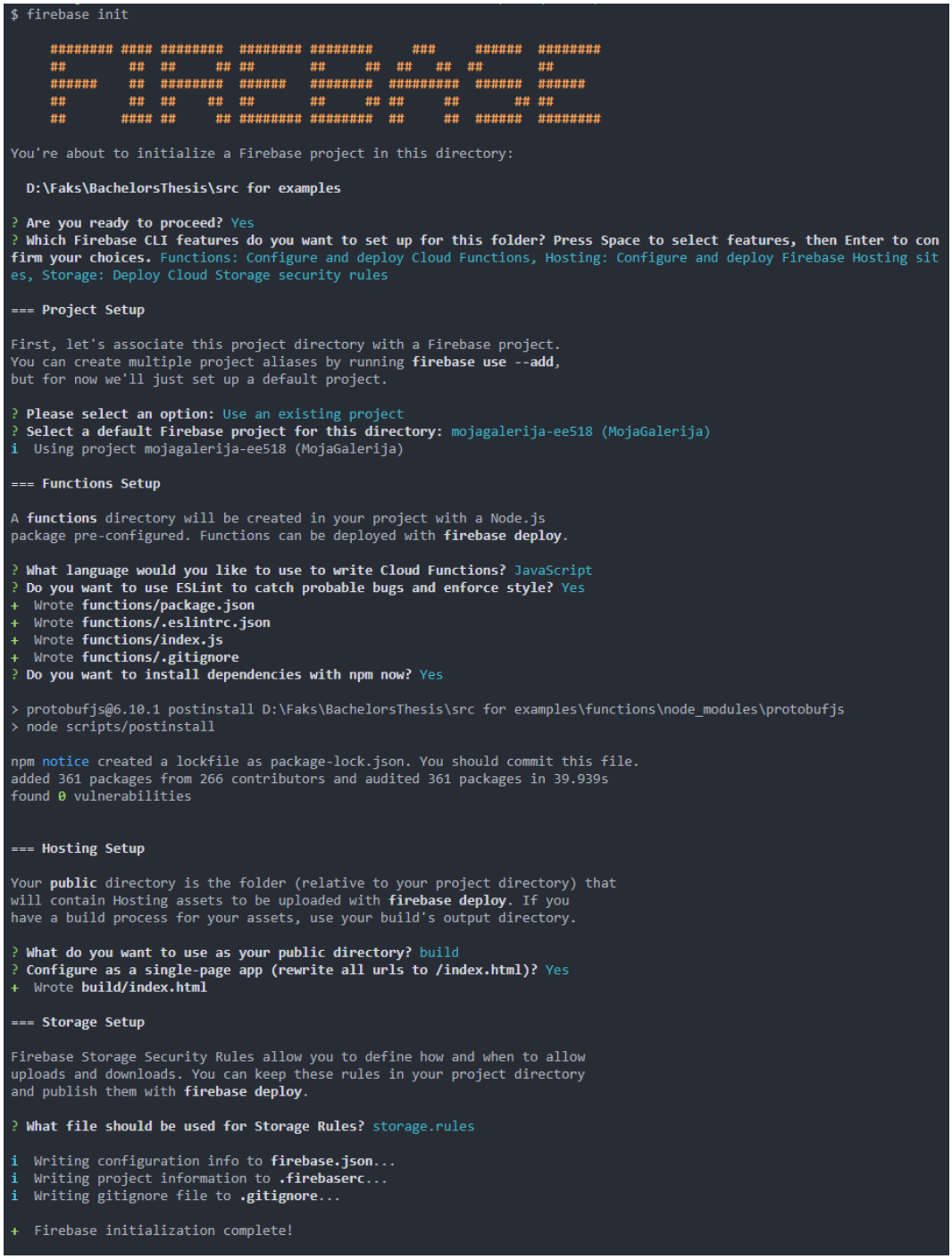



Slika 6-4: Prikaz vodenih korakov vzpostavitve Firebase projekta

Tretji korak je namestitev Firebase Tools. Ukaz »npm install -g firebase-tools« izvedemo v konzoli z dostopom do npm.

Preden izvedemo korak 4, se postavimo v datoteko z našim projektom in dodamo v njega knjižnice za delovanje Firebase z »npm install firebase –save«. Da naš React projekt povežemo s tem, ki smo ga ustvarili v konzoli, se moramo najprej vpisati z ukazom »firebase login«. In nato zaženemo »firebase init«, ter izberemo opcije, razvidne s Slike 6-4. Izbrane možnosti so označene s svetlo modro barvo v konzoli. Preden React aplikacijo namestimo na gostovanje, je treba ustvariti produkcijsko zgradnjo aplikacije, z ukazom »npm run build«. In nato »Firebase deploy«, da spravimo našo aplikacijo na javni naslov s SSL certifikatom.

Ker smo naredili veliko korakov, naj te na kratko povzamemo. Pripravili smo Firebase projekt s pomočjo njihove spletne konzole in zgradili privzeto React aplikacijo, ki se generira ob zagonu ukaza »npx create-react-app«. Ti aplikaciji smo nato povezali s pomočjo »firebase init« in omogočili storitve, ki jih bomo tekom razvoja naše galerije uporabili (gostovanje, hramba in oblačne funkcije). Kar smo prej zgradili, smo nato namestili na gostovanje Firebase s pomočjo ukaza »firebase deploy«. Sedaj imamo osnovno aplikacijo z omogočenim vsem potrebnim za razvoj naše galerije.




Slika 6-5: Prikaz izbranih možnosti (v modri barvi)

## 6.4 Razvoj funkcionalnosti

### 6.4.1 Avtentifikacija

Najprej si poglejmo razvoj avtentifikacije. Za namene tega bomo izdelali namensko React komponento, ki bo ob svoji pritrditvi preverila, kateri uporabnik je trenutno

vpisan in ga poslala preko lastnosti osrednji aplikaciji – vrstica 22 na Sliki 6-6. Z uvažanjem komponente StyledFirebaseAuth pridobimo tudi elegantno rešitev za grafični vmesnik. Ta nam na podlagi objekta uiConfig generira obrazce in gumbe za vpis in/ali registracijo uporabnika. V tem priemru bomo izbrali samo Google račun za način avtentifikacije.

```
1  import React, { Component } from "react";
2  import StyledFirebaseAuth from "react-firebaseui/StyledFirebaseAuth";
3  var firebase = require("firebase");
4
5  export class AurthUI extends Component {
6      uiConfig = {
7          signInFlow: "popup",
8          signInOptions: [firebase.auth.GoogleAuthProvider.PROVIDER_ID],
9          callbacks: {
10             signInSuccess: () => false,
11         },
12     };
13     render() {
14         return (
15             <StyledFirebaseAuth
16                 uiConfig={this.uiConfig}
17                 firebaseAuth={firebase.auth()}
18             />
19         );
20     }
21     componentDidMount = () => {
22         firebase.auth().onAuthStateChanged((user) => this.props.isSignedIn(user));
23     };
24 }
25
26 export default AurthUI;
```

Slika 6-6: Implementacija komponente za avtentifikacijo

Na tem primeru je jasno, da je implementacija zunanjega ponudnika avtentifikacije s Firebase zelo enostavna. Ne samo, da smo robustno poskrbeli za varnost s podjetjem, ki ima za te namene od nas boljšo infrastrukturo, še dolgoročno smo poskrbeli, da ne bo problemov z grafičnim vmesnikom, saj ga bodo ti posodobili, ko bo to potrebno.

### 6.4.2 Naložitev fotografije

Oglejmo si, kako je videti komponenta za naložitev fotografije v našo galerijo. Za lažje obravnavanje komponente, si jo bomo ogledali v dveh Slikah 6-7 in 6-8. Na Sliki 6-6 vidimo, da imamo v stanju pripravljeno spremenljivko za našo sliko in da ob pritrditvi komponente ustvarimo instanco objekta za delo s hrambo.

Komponenta za svoj predstavitveni del generira enostavno vnosno polje za datoteke in gumb za potrditev vnesenega. Ob spremembi vnosnega polja se kliče metoda »handleChange«, ki preveri, ali kakšna datoteka sploh obstaja. Če to drži, se ta zapiše v stanje komponente.

```
import React, { Component } from "react";
import firebase from "firebase";

export class UploadImage extends Component {
  storage = null;
  state = {
    image: null,
  };
  componentDidMount() {
    this.storage = firebase.storage();
  }

  handleChange = (e) => {
    if (e.target.files[0]) {
      this.setState({ image: e.target.files[0] });
    }
  };
  render() {
    return (
      <div>
        <h1>Naložite sliko</h1>
        <input
          type="file"
          name="image"
          id="image"
          onChange={this.handleChange}
        />
        <button onClick={this.handleUpload}>Naloži!</button>
      </div>
    );
  }
```

Slika 6-7: Implementacija komponente za naložitev fotografije - UI del

Oglejmo si drugi del naše komponente na Sliki 6-7. Ta obravnava, kako bodo datoteke naložene v našo hrambo. Najprej ustvarimo instanco opravila za nalaganje datotek in z metodo »ref« nastavimo, kam naj se v oblaku datoteka naloži in s katerim imenom. Z metodo »put« pa povemo, kaj želimo tja naložiti.

Ustvarjeno opravilo ima tudi možnost prijave na dogodke, kot so »state_changed« oz. sprememba stanja. Tako bi lahko zelo enostavno spremljali, koliko odstotkov naše datoteke se je že naložilo ali pa poslušamo za napake – v tem primeru poslušamo samo za napake in jih beležimo v konzolo.

```
33    handleUpload = () => {
34      const uploadTask = this.storage
35        .ref(`images/${this.state.image.name}`)
36        .put(this.state.image);
37      uploadTask.on(
38        "state_changed",
39        (snapshot) => {},
40        (error) => {
41          console.log(error);
42        }
43      );
44    };
45  }
46
47  export default UploadImage;
48
```

Slika 6-8: Implementacija komponente za naložitev fotografije - del za delo s hrambo

Ponovno je samo delo s Firebase zelo enostavno in je problem rešen v nekaj vrsticah. Poleg enostavne rešitve pa nudi robusten vpogled v samo dogajanje, če si to želimo.

### 6.4.3 Obdelava fotografije

Obdelava fotografije se zgodi po tem, ko je ta že naložena v oblak s pomočjo oblačnih funkcij. To je zelo priročno, saj tako uporabnik prenaša fotografijo v oblak samo enkrat in se nam znižajo stroški branja in pisanja fotografij.

Delo z oblačnimi funkcijami je prav tako preprosto, saj smo ob vzpostavitvi projekta obkljukali uporabo teh in so orodja Firebase-a za nas ustvarila imenik z imenom »functions« oz. funkcije. Tam se nahaja datoteka z imenom »index.js«, ki vsebuje vse naše funkcije. Poleg tega najdemo tudi ostale datoteke za delo z JavaScript in npm, ki bodo uporabljene na strani oblačnega ponudnika.

Oglejmo si Sliko 6-9 – potek dela je zelo direkten. Na objekt »exports« vežemo funkcije z želenim imenom, v našem primeru je ime »generateThumbnail«. Vežemo jo na prožilec objekta »storage«, ko se zaključi delo z nekim objektom. Ko se to izvrši, se začne izvajati naša funkcija, ki prvo v spremenljivke shrani podatke o objektu in preveri, ali sploh gre za sliko. Nato izluščimo ime datoteke in preverimo, ali že gre za obdelano fotografijo (kasneje bomo vstavili pred vse obdelane fotografije »thumb_«). Nato se pripravi vse potrebno, da lahko datoteko prenesemo in jo obdelamo na želene parametre. Nato datoteko zapišemo na disk in jo naložimo nazaj na Firebase hrambo podatkov, ter izbrišemo začasne datoteke.

```
const functions = require("firebase-functions");
const admin = require("firebase-admin");
admin.initializeApp();
const spawn = require("child-process-promise").spawn;
const path = require("path");
const os = require("os");
const fs = require("fs");
exports.generateThumbnail = functions.storage
    .object()
    .onFinalize(async (object) => {
        const fileBucket = object.bucket;
        const filePath = object.name;
        const contentType = object.contentType;

        if (!contentType.startsWith("image/")) {
            return console.log("This is not an image.");
        }

        const fileName = path.basename(filePath);

        if (fileName.startsWith("thumb_")) {
            return console.log("Already a Thumbnail.");
        }
        const bucket = admin.storage().bucket(fileBucket);
        const tempFilePath = path.join(os.tmpdir(), fileName);
        const metadata = {
            contentType: contentType,
        };
        await bucket.file(filePath).download({ destination: tempFilePath });
        await spawn("convert", [
            tempFilePath,
            "-thumbnail",
            "200x200>",
            tempFilePath,
        ]);
        const thumbFileName = `thumb_${fileName}`;
        const thumbFilePath = path.join(
            path.dirname(filePath),
            "thumbnail",
            thumbFileName
        );
        await bucket.upload(tempFilePath, {
            destination: thumbFilePath,
            metadata: metadata,
        });
        return fs.unlinkSync(tempFilePath);
    });
```

Slika 6-9: Implementacija procesa obdelave slik na manjše ikone oz. thumbnail

Ponovno je proces zelo enostaven, saj pišemo funkcijo enako, kot bi se ta izvajala na lokalni napravi, vendar imamo dodatne možnosti s poslušanjem za dogodke v

oblaku. Če funkcijo spremenimo, se ob ukazu »firebase deploy« samodejno posodobi.

### 6.4.4 Prikaz obdelane fotografije

Za prikaz obdelanih fotografij ponovno ustvarimo namensko komponento, ki ima v stanju polje za zapis poti do datotek (Slika 6-10). Nato v metodi ob pritrditvi komponente s pomočjo objekta »storage« pridobimo vse datoteke, ki se nahajajo na »images/thumbnail« lokaciji v oblaku in jih zapišemo v stanje. Pridobljene poti bomo posredovali komponenti, ki je posebej namenjena za prikaz fotografije (Slika 6-11). Tu bomo s pomočjo objekta »ref«, ki ga pridobimo z objekta »storage«, prenesli Binary Long Object oz. BLOB fotografije na stran brskalnika in jo prikazali v znački »<img>«. Pridobljeni BLOB lahko zapišemo neposredno v »src« atribut značke »<img>« in dosežemo želeni rezultat. In tako je prikaz fotografij modularno urejen in pripravljen tudi na večje projekte.

```jsx
import React, { Component } from "react";
import UploadImage from "./UploadImage";
import SingleImage from "./SingleImage";
import firebase from "firebase";

export class Gallery extends Component {
    state = {
        imagePaths: [],
    };

    componentDidMount() {
        const storageRef = firebase.storage().ref();
        var listRef = storageRef.child("images/thumbnail");
        const imagePaths = [];
        listRef
            .listAll()
            .then(function (res) {
                res.items.forEach(function (itemRef) {
                    imagePaths.push(itemRef.location.path);
                });
            })
            .catch(function (error) {
                console.log(error);
            })
            .then(() => {
                this.setState({
                    imagePaths,
                });
            });
    }

    render() {
        return (
            <div>
                <h1>Moja Galerija</h1>
                <ul>
                    {this.state.imagePaths.map((imagePath) => (
                        <SingleImage imagePath={imagePath} />
                    ))}
                </ul>
                <UploadImage />
            </div>
        );
    }
}

export default Gallery;
```

Slika 6-10: Implementacija komponente galerije

```
import React, { Component } from "react";
import firebase from "firebase";

export class SingleImage extends Component {
    state = {
        image: null,
    };

    componentDidMount() {
        const storageRef = firebase.storage().ref();
        let image = null;
        storageRef
            .child(this.props.imagePath)
            .getDownloadURL()
            .then(function (url) {
                var xhr = new XMLHttpRequest();
                xhr.responseType = "blob";
                xhr.onload = function (event) {};
                xhr.open("GET", url);
                xhr.send();
                image = url;
            })
            .catch(function (error) {
                console.log(error);
            })
            .then(() => this.setState({ image }));
    }

    render() {
        return (
            <li>
                <img src={this.state.image} alt="Fotografija" />
            </li>
        );
    }
}

export default SingleImage;
```

Slika 6-11: Implementacija komponente za prenos in prikaz fotografije v brskalniku

Spet ugotovimo, da je delo zelo enostavno pri rokovanju z oblačnimi storitvami Firebase. Imamo dobro pripravljene metode za ugotavljanje lokacij in nato prenos s teh lokacij v brskalnik.

# 7 SKLEPI

V diplomskem delu smo si ogledali trenutno stanje razvoja programske opreme in zakaj so samopostrežne rešitve tako popularne. Preleteli smo tudi ostale rešitve, ki omogočajo različne načine razvoja s storitvami v oblaku. Nato smo se posvetili samopostrežnim oblačnim rešitvam in preučili, kaj jih pravzaprav naredi tako atraktivne za uporabo kot zaledni sistem naše aplikacije. Predstavljene so tudi slabe plati razvoja na tak način in kdaj ta ni primeren za naš scenarij. Ko smo ugotovili, kaj naredi samopostrežne rešitve privlačne, smo se lotili preučevanja delovanja teh tehnologij v paru. Ugotovili smo, da:

- Enostranske spletne aplikacije in samopostrežni zaledni sistemi v oblaku dobro sodelujejo med seboj;
- Sodelovanje teh tehnologij ustvari fleksibilen, robusten, interaktiven in očesu privlačen izdelek.

Ker je število ponudnikov oblačnih storitev kar nekaj in ker sami lahko razvijemo primerljivo rešitev, ne glede na izbrano orodje, smo na primerjavo postavili lastne REST rešitve, Firebase in AWS Amplify. S pregledom vseh treh možnosti in predhodnim znanjem o samopostrežnih oblačnih rešitvah in enostranskih spletnih aplikacijah smo definirali kriterij za primerjavo teh orodij in primerjavo tudi izvedli. Zaradi naslednjih lastnosti se je izkazalo, da je za delo z enostranskimi spletnimi aplikacijami, najbolj primeren Firebase:

- enostaven za uporabo,
- integracija dodatnih storitev,
- enostavnost vzpostavitve projekta,
- možnost dodatnih storitev.

Opremljeni s tem znanjem smo razvili aplikacijo »Moja Galerija«, ki je z osnovnimi funkcionalnostmi avtentifikacije, nalaganja, obdelave in prikazovanja naloženih slik služila kot vpogled v obnašanje Firebase-a v realnem svetu. Ugotovili smo, da je delo res enostavno in razvijalcu prijazno – od same nastavitve v ponudnikovi

konzoli, povezave med projektom na naši napravi in vsemi uporabljenimi storitvami.

## 7.1 Možne izboljšave in nadaljnje raziskave

V sklopu te naloge smo se osredotočili na prednosti in slabosti za enostranske spletne aplikacije, kar pa je dokaj ozek spekter uporabe. Ta orodja zagotovo lahko prispevajo ogromno na drugih področjih. Med raziskavo smo imeli v mislih manjše ekipe ali celo samostojne razvijalce, ki jim te storitve res ogromno olajšajo delo. Omejili smo se na uporabo avtentifikacije, hrambe datotek, oblačnih funkcij in gostovanja. Nadaljnje raziskave bi lahko primerjale npr. storitve strojnega učenja, podatkovne baze v realnem času idr. Storitev. Zanimivo bi bilo preveriti, koliko lahko ti sistemi v oblaku (samostojno razviti in od zunanjih ponudnikov) nudijo velikim podjetjem v poslovnih okoljih, kjer je zahtevana podatkovna tajnost ipd. Katere vidike bi bilo treba poudariti v tem primeru?

# 8 VIRI IN LITERATURA


[1] R. Vemula, Integrating Serverless Architecture, Visakhapatnam: Apress, 2019.

[2] C. Wodehouse, „Upwork,“ 2 10 2018. [Elektronski]. Available: https://www.upwork.com/hiring/development/a-beginners-guide-to-back-end-development/. [Poskus dostopa 24 6 2020].

[3] C. G. Kim, A Study of Utilizing Backend as a Service, Springer, Cham, 2019.

[4] C. SPOIALA, „Assist Software,“ 23 April 2019. [Elektronski]. Available: https://assist-software.net/blog/pros-and-cons-serverless-computing-faas-comparison-aws-lambda-vs-azure-functions-vs-google. [Poskus dostopa 29 6 2020].

[5] M. Roberts, „Martin Fowler,“ Martin Fowler, 22 5 2018. [Elektronski]. Available: https://www.martinfowler.com/articles/serverless.html. [Poskus dostopa 12 8 2020].

[6] Cloudflare, „Cloudflare,“ [Elektronski]. Available: https://www.cloudflare.com/learning/serverless/glossary/backend-as-a-service-baas/. [Poskus dostopa 7 8 2020].

[7] M. Clark, „Back4App,“ [Elektronski]. Available: https://blog.back4app.com/baas-vs-serverless/. [Poskus dostopa 7 8 2020].

[8] B. Vigliarolo, „Tech Republic,“ 1 Maj 2019. [Elektronski]. Available: https://www.techrepublic.com/article/serverless-computing-pros-and-cons-5-benefits-and-3-drawbacks/. [Poskus dostopa 29 Junij 2020].

[9] B. D. Rooms, „InfoWorld,“ InfoWorld, 13 2 2020. [Elektronski]. Available: https://www.infoworld.com/article/3526480/whats-next-for-serverless-architecture.html. [Poskus dostopa 12 7 2020].

[10] Cloudflare, „Cloudflare,“ [Elektronski]. Available: https://www.cloudflare.com/learning/serverless/why-use-serverless/. [Poskus dostopa 30 6 2020].

[11] Amazon, „AWS Lambda Pricing,“ [Elektronski]. Available: https://aws.amazon.com/lambda/pricing/. [Poskus dostopa 30 5 2020].

[12] J. Hanson, „Hackernoon,“ 26 6 2017. [Elektronski]. Available: https://hackernoon.com/five-advantages-of-serverless-technology-68160c1f884e. [Poskus dostopa 30 6 2020].

[13] C. Osborn, „ZDNet,“ Zero Day, 17 1 2018. [Elektronski]. Available: https://www.zdnet.com/article/the-top-10-risks-for-apps-on-serverless-architectures/. [Poskus dostopa 1 7 2020].

[14] Y. Sun, Practical Application Development with AppRun, Berkeley: Apress, 2019.

[15] P. Späth, „Building Single-Page Web Applications with REST and JSON,“ v *Beginning Jakarta EE*, APress, 2019, pp. 113-143.

[16] StackOverflow, „StackOverflow,“ 2 2020. [Elektronski]. Available: https://insights.stackoverflow.com/survey/2020#technology-web-frameworks-all-respondents2. [Poskus dostopa 6 7 2020].

[17] T. Vo, Web Application Development with React and Google Firebase, Turku, Finland: Turku University of Applied Sciences, 2020.

[18] M. Hajian, Progressive Web Apps with Angular, Berkeley: Apress, 2019.

[19] N. C. V, „Hackernoon,“ Hackernoon, 23 7 2018. [Elektronski]. Available: https://hackernoon.com/how-we-spent-30k-usd-in-firebase-in-less-than-72-hours-307490bd24d. [Poskus dostopa 9 7 2020].

[20] M. Podplatnik, Primerjava ogrodij za zaledne sisteme mobilnih aplikacij : diplomsko delo, Maribor: M. Podplatnik, 2019.

[21] P. A. Kheta, „Preeti,“ 4 5 2020. [Elektronski]. Available: https://p3420a1.wixsite.com/preeti/post/aws-amplify-vs-firebase. [Poskus dostopa 9 7 2020].

[22] Amazon, „AWS Amplify,“ Amazon, [Elektronski]. Available: https://docs.amplify.aws. [Poskus dostopa 16 3 2020].

[23] W. H. A. Sitanggang, „Mitrais,“ Mitrais, 13 2 2020. [Elektronski]. Available: https://www.mitrais.com/news-updates/aws-amplify-vs-google-firebase-which-is-better/. [Poskus dostopa 9 7 2020].

[24] Google, „Google,“ Google, 6 7 2020. [Elektronski]. Available: https://firebase.google.com/docs/admin/setup. [Poskus dostopa 9 7 2020].

[25] Google, „Firebase,“ Google, [Elektronski]. Available: https://firebase.google.com/docs. [Poskus dostopa 16 3 2020].

[26] N. Broda, „StackShare,“ 17 6 2020. [Elektronski]. Available: https://stackshare.io/stackups/aws-amplify-vs-firebase. [Poskus dostopa 9 7 2020].

[27] Firebase, „Firebase,“ 2020. [Elektronski]. Available: https://firebase.google.com/docs/web/setup?authuser=0#node.js-apps. [Poskus dostopa 26 7 2020].

[28] Facebook, „React,“ Facebook, [Elektronski]. Available: https://reactjs.org/. [Poskus dostopa 16 3 2020].

[29] D. Lamas, F. Loizides, L. Nacke, H. Petrie, M. Winckler in P. Zaphiris, Human-Computer Interaction – INTERACT 2019, Cham: Springer, 2019.

[30] L. Moroney, The Definitive Guide to Firebase, Berkeley: Apress, 2017.

[31] B. Choudhary, C. Pophale, A. Gutte, A. Dani in S. S. Sonawani, Case Study: Use of AWS Lambda for Building a Serverless Chat Application, Singapore: Springer, 2020.

[32] A. Freeman, Pro Windows 8 Development with HTML5 and JavaScript, Berkeley: Apress, 2012.

[33] L. Baresi in M. Garriga, Microservices: The Evolution and Extinction of Web Services?, Cham: Springer, 2019.