UNIVERZA V MARIBORU

FAKULTETA ZA ELEKTROTEHNIKO,  
RAČUNALNIŠTVO IN INFORMATIKO

Marko Gluhak

Posebnosti razvoja spletnih rešitev s samopostrežnim zalednim sistemom v oblaku

Zaključno delo

Maribor, avgust 2020

Posebnosti razvoja spletnih rešitev s samopostrežnim zalednim sistemom v oblaku

Diplomsko delo

Študent: Marko Gluhak

Študijski program: Informatika in tehnologije komuniciranja

Smer: Informacijski sistemi

Mentor (ica): doc. dr. Luka Pavlič, univ. dipl. inž. rač. in inf.

Lektor (ica): TBA

Posebnosti razvoja spletnih rešitev s samopostrežnim zalednim sistemom v oblaku

**Ključne besede:**

**UDK:**

**Povzetek**

The specifics of the single page application development while using cloud-based backend as a service

**Keywords:**

**UDK:**

**Abstract**

Kazalo VSEBINE

[1 UVOD 1](#_Toc46845305)

[1.1 Opredelitev problema 1](#_Toc46845306)

[1.2 Cilji zaključnega dela 2](#_Toc46845307)

[1.3 Predpostavke in omejitve 2](#_Toc46845308)

[2 Evolucija storitev v oblaku 3](#_Toc46845309)

[2.1 Infrastruktura kot storitev 4](#_Toc46845310)

[2.2 Okolje kot storitev 4](#_Toc46845311)

[2.3 Samopostrežne oblačne rešitve 5](#_Toc46845312)

[2.4 Funkcija kot storitev 5](#_Toc46845313)

[3 Prednosti in omejitve samopostrežnih zalednih sistemov 6](#_Toc46845314)

[3.1 Najboljša samostojna razširljivost ob velikih bremenih 7](#_Toc46845315)

[3.2 Manjši čas odziva 8](#_Toc46845316)

[3.3 Nizki stroški delovanja – po porabi 9](#_Toc46845317)

[3.4 V splošnem se zmanjša kompleksnost aplikacije 11](#_Toc46845318)

[3.5 Potrebno je utrditi varnost 11](#_Toc46845319)

[3.6 Funkcije niso vedno idealne 12](#_Toc46845320)

[4 Enostranske spletne aplikacije 12](#_Toc46845321)

[4.1 Progresivne spletne aplikacije 13](#_Toc46845322)

[4.2 Utemeljitev izbora React-a 14](#_Toc46845323)

[5 kriterijI in izbor samopostrežnega zalednega sistema v oblaku 16](#_Toc46845324)

[5.1 Definicija kriterijev 16](#_Toc46845325)

[5.2 Rešitev z REST vmesnikom 18](#_Toc46845326)

[5.3 Amazon Web Services Amplify 19](#_Toc46845327)

[5.4 Firebase 21](#_Toc46845328)

[5.5 Primerjava 23](#_Toc46845329)

[5.6 Izbor 24](#_Toc46845330)

[6 Razvoj zalednega sistema Firebase in aplikacije 25](#_Toc46845331)

[6.1 Funkcionalnosti 25](#_Toc46845332)

[6.2 Arhitektura 26](#_Toc46845333)

[6.3 Priprava okolja 27](#_Toc46845334)

[6.4 Razvoj funkcionalnosti 30](#_Toc46845335)

[6.4.1 Avtentifikacija 30](#_Toc46845336)

[6.4.2 Naložitev fotografije 31](#_Toc46845337)

[6.4.3 Obdelava fotografije 33](#_Toc46845338)

[6.4.4 Prikaz obdelane fotografije 36](#_Toc46845339)

[7 Sklepi 39](#_Toc46845340)

[7.1 Možne izboljšave in nadaljnje raziskave 39](#_Toc46845341)

[8 Viri in Literatura 41](#_Toc46845342)

Kazalo slik

[Slika 2.2‑1: Razlike med monolitnimi, mikrostoritvenimi in FaaS modeli [1] 6](#_Toc46845408)

[Slika 2.2‑2: Oblačne storitve Microsoft Azure z različnimi modeli oblačnega računalništva [1] 6](#_Toc46845409)

[Slika 3‑1: Porazdeljena arhitektura samopostrežnih zalednih sistemov [6]. 8](#_Toc46845410)

[Slika 3‑2: Cenovni model AWS Lambda [8] 9](#_Toc46845411)

[Slika 3‑3: Primer zaračunanja stroškov delovanja konkretne funkcije [8]. 10](#_Toc46845412)

[Slika 4‑1: Rezultati raziskave o spletnih ogrodjih [14]. 15](#_Toc46845413)

[Slika 5‑1: Prikaz informativnega izračuna uporabe storitev Amplify [19] 20](#_Toc46845414)

[Slika 5‑2 Prikaz informativnega izračuna stroškov za uporabo Firebase storitev [22] 22](#_Toc46845415)

[Slika 6‑1 Diagram primerov uporabe za aplikacijo Moja Galerija 26](#_Toc46845416)

[Slika 6‑2 Arhitektura aplikacije, ki jo bomo razvili 27](#_Toc46845417)

[Slika 6‑3 Prikaz vodenih korakov vzpostavitve Firebase projekta 28](file:///D:\Faks\BachelorsThesis\Posebnosti%20razvoja%20spletnih%20rešitev%20s%20samopostrežnim%20zalednim%20sistemom%20v%20oblaku_2020_07_21_LP.docx#_Toc46845418)

[Slika 6‑4 Prikaz izbranih možnosti (v modri barvi) 30](#_Toc46845419)

[Slika 6‑5 Implementacija komponente za avtentifikacijo 31](#_Toc46845420)

[Slika 6‑6 Implementacija komponente za naložitev fotografije - UI del 32](#_Toc46845421)

[Slika 6‑7 Implementacija komponente za naložitev fotografije - del za delo s hrambo 33](#_Toc46845422)

[Slika 6‑8 Implementacija procesa obdelave slik na manjše ikone oz. thumbnail 35](#_Toc46845423)

[Slika 6‑9 Implementacija komponente galerije 37](#_Toc46845424)

[Slika 6‑10 Implementacija komponente za prenos in prikaz fotografije v brskalniku 38](#_Toc46845425)

Kazalo tabel

[Tabela 3-1: Primerjava orodij med seboj po primerjalni lestvici 24](#_Toc46595399)

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* |
|  |  |

# UVOD

V področju informatike je v zadnjih letih pričakovan hiter 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 je lahko problem odločiti se za pravilno. Kljub reševanju ogromno problemov kot so razširljivost, varnost in druge prepreke strojne opreme pa zahteva dodaten, 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 utečena za delo z REST vmesniki in predstavitvijo pridobljenih podatkov.

## Opredelitev problema

V zaključnem delu bom raziskoval kateri izmed glavnih ponudnikov samopostrežnih zalednih sistemov je najbolj primeren za delo z enostranskimi spletnimi aplikacijami. Podrobneje bom pogledal komunikacijo teh storitev z enostranskimi aplikacijami in poskusil 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, podpora 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 bom za ugotovljeno najboljše orodje razvil preprosto rešitev in jo namestil v realno svet. Tako bom na osnovnem primeru tudi ugotovil ali je moj zaključek izbora ponudnika tudi primerljiv z realnim obnašanjem.

## Cilji zaključnega dela

Cilji mojega dela so poiskati ponudnike samopostrežnih zalednih sistemov v oblaku in ugotoviti ustrezne kriterije za izbor najboljšega za potrebe razvoja enostranskih spletnih aplikacij. Poleg tega je potrebno ugotoviti kakšna orodja nam ogrodja za razvoj enostranskih spletnih aplikacij sploh ponujajo in kako ta uporabiti na primeru. Sledi praktična demonstracija preprostejše aplikacije z ugotovljenim najboljšim kandidatom za zaledje in predstavitev njegovih prednosti na primeru.

## Predpostavke in omejitve

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

# Evolucija storitev v oblaku

Za razumevanje rešitev, ki jih samopostrežni zaledni sistemi v oblaku ponudijo in zakaj so te tako mamljive, moramo razumeti probleme, ki so to programersko paradigmo povzdignili na nove višave. Podjetja se v času, ko je pozornost uporabnika med najbolj visoko cenjenimi surovinami na vse načine trudijo pridobiti in zadržati le-to. Poleg kvalitetne vsebine je potrebno 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 pa nam te vidike občutno olajšajo, saj nam te 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 zadnja 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 – implementaciji poslovne logike in razširjevanju posameznih funkcionalnosti neodvisno od strojne opreme. [1] [2] [3] [4]

Samopostrežni zaledni sistemi so v zadnjem desetletju dobili veliko pristašev, še posebej po tem, ko je Amazon predstavil svojo platformo pod imenom *AWS Lambda* leta 2014. Ko je tržišče pokazalo interes so se poslovnemu modelu pridružili še ostali tehnološki orjaki, Microsoft s svojimi ti. AzureFunctions in Google z *Google* CloudFunctions – oba leta 2016. Sledila sta še Oracle-ova Fn in IBM-ov OpenWhisk. Obstajajo tudi odprtokodna ogrodja za samopostrežne sisteme, kot Serverless in Kubernetes, ki sta neodvisna od oblačnih ponudnikov in neodvisno delujeta v Docker in Kubernetes zabojnikih.

Da bi bolje razumeli samopostrežno računalništvo, je potrebno pogledati evolucijo strojne opreme in omrežne topologije od leta 1990. Takrat so podjetja še sama nakupovala in postavljala svojo strojno opremo in 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, 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].

## 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 povem intuitivna za razvijalce, vendar pa nudi veliko svobode pri prilagoditvi gostitelja aplikacije [1].

## 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].

## Samopostrežne oblačne rešitve

Samopostrežne oblačne rešitve so zadnja strategija ponudnikov oblačnih storitev, kje 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 popolna. 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].

## Funkcija kot storitev

*Funkcija kot storitev oz. Function as a Service* (FaaS) vzpodbudi razvijalce da izlušči majhne sklope funkcionalnosti iz več nivojske aplikacije in jih gostovati kot funkcije, ki se lahko razširijo neodvisno. Ta pristop je cenovno zelo učinkovit, 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 izgledajo bolj poznane storitve na Microsoft Azure z prej omenjenimi strategijami oblačnega računalništva.



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

# Prednosti in omejitve samopostrežnih zalednih sistemov

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

Prvo si oglejmo prednosti:

* Najboljša samostojna razširljivost ob velikih bremenih
* Brez vzdrževanja arhitekture
* Manjši čas odziva
* Podpora agilnih in hitrih razvojnih ciklov
* Nizki stroški obratovanja – plačaj kot porabiš
* Enostaven model namestitve
* V splošnem se zapletenost zmanjša

Omejitve:

* Orodja monitoringa, beleženja in razhroščevanja so še vedno v zgodnjih fazah razvoja
* Nekompatibilnost med ponudniki
* Zmogljivost je lahko v določenih primerih ozko grlo, kjer so potrebni klici med funkcijami
* Potrebno je utrditi varnostne sisteme
* Upravljanje stanj naj bo centralizirano
* Funkcije niso vedno idealne

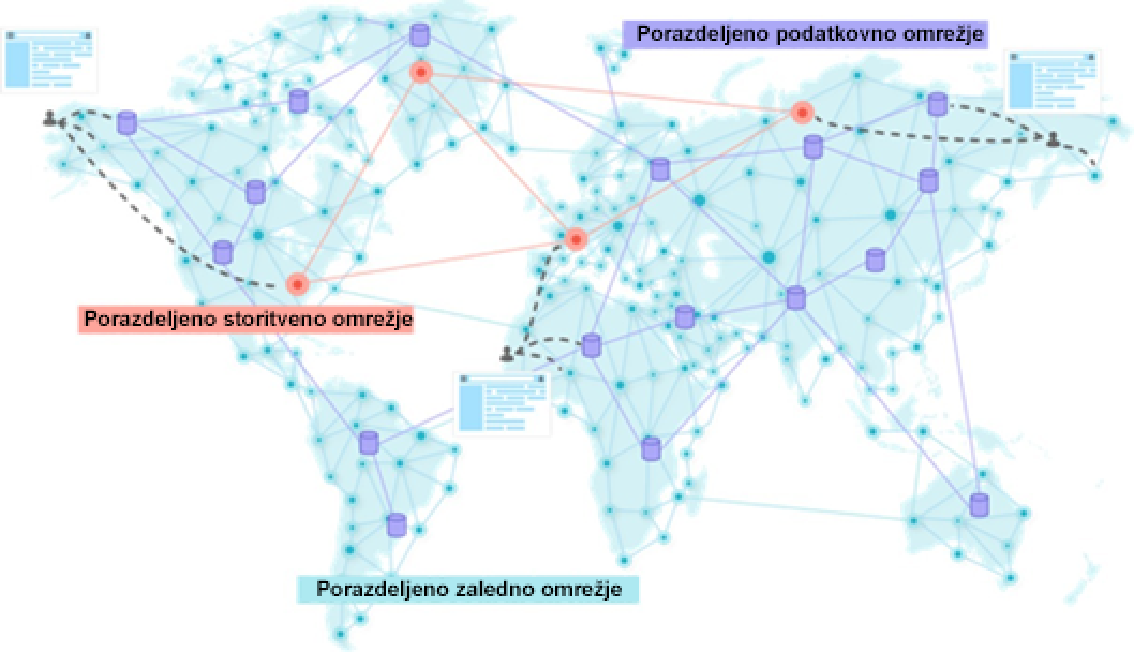
Hiter tempo po katerem samopostrežno računalništvo in temu sorodne rešitve zorijo kaže, da bo zahteven tekmovalec za svoje predhodnike. Trenutno je smatran kot najbolj zanesljiva nastajajoča paradigma. Podrobneje si poglejmo določene prednosti in omejitve, ki nam povedo več o sodelovanju tega paradigma z enostranskimi spletnimi aplikacijami. [1] [5]

## Najboljša samostojna razširljivost ob velikih bremenih

Ker sami ne upravljamo virov, ki so potrebni za tekoče delovanje infrastrukture se naša rešitev avtomatsko razširi po potrebi. Ker je naša aplikacija enostranska, pomeni tudi, da se ta prenese na stran uporabnika in se tam izvajajo ostale enostavnejše operacije. Posledica je, da vse kar bi lahko naši aplikacijo preobremenilo, je 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 funkcije ki imajo pogosto obremenitev so vedno pripravljene, da uporabnike postrežejo z odgovori in tako zagotovijo nenavadni hitre odzive. [5]

## Manjši čas odziva

Ponudniki samopostrežnih zalednih storitev imajo zelo dodelano arhitekturo in infrastrukturo. Za nas to pomeni, da bodo vedno lahko 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 naš 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.

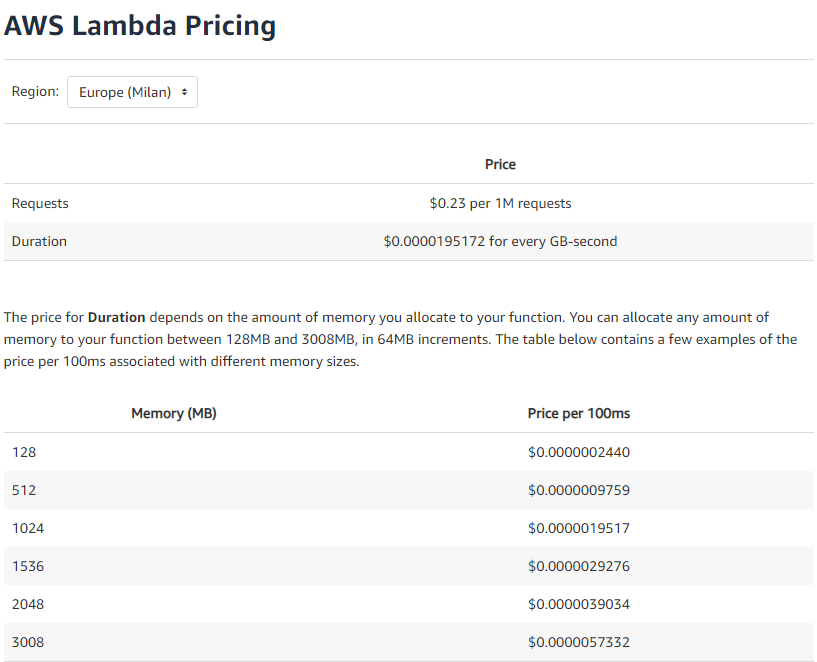


Slika 3‑1: Porazdeljena arhitektura samopostrežnih zalednih sistemov [6].

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. [7]

## Nizki stroški delovanja – po porabi

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 kaki cenovno ugodno so lahko funkcije ima AWS Lambda storitev zelo nazorno nakazano v svojen modelu cenitve Slika 2-4.



Slika 3‑2: Cenovni model AWS Lambda [8]

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



Slika 3‑3: Primer zaračunanja stroškov delovanja konkretne funkcije [8].

Torej če naši funkciji dodelimo 512MB spomina, jo izvedemo 3 milijon-krat v enem mesecu in se vsakič izvaja eno sekundo, bi bila skupna cena za ta mesec, za to funkcijo $18.74.

Poleg že tako ugodnega modela plačevanja, pa če funkcij nikoli ne uporabimo nas ne stanejo ničesar. [9]

## V splošnem se zmanjša kompleksnost aplikacije

Nekatere omejitve samopostrežnih funkcij delujejo tudi razvijalcem v prid. Med te omejitve štejemo to, da morajo te funkcije delovati ne glede na strojno opremo kjerkoli, brez potrebe po dodatnih zunanjih virov 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 koda in poslovna logika ne mešata s kodo, ki se izvaja v brskalniku uporabnika.

## Potrebno je utrditi varnost

Noben pregled samopostrežnih zalednih sistemov ne bi bil popoln brez da omenimo varnost in potencialne probleme s to, ko se odločamo za tak način razvoja. Zaradi skrbi o teh varnostnih problemih je ZDNet ustvaril seznam 10 takšnih potencialnih varnostnih tveganj [10], 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, katere 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 uspešno preobremenjene lahko te spodletijo v nudenju storitev več strankam naenkrat.

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

## Funkcije niso vedno idealne

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 funkcije naj ne tečejo več časa, saj lahko tako postanejo zelo cenovno potratne in se lahko veliko bolj izplača postaviti lastno strežniško arhitekturo. Torej v tem primeru ne gre za snovalno napako, temveč za vprašanje cenovne ugodnosti naše odločitve in potencialnih alternativah [7].

# 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 zahtevan osvežitev celotne spletne strani. Uporabniške interakcije, kot so pritiski na gumbe, izpolnjevanje obrazcev sprožijo GET ali pa POST zahtevno na spletni strežnik in ta mora ponovno pripraviti celotno HTML stran. To pa povzroči kratko utripanje strani v belo, ki 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 uporabnike, 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 jih 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, naložena na računalnik.

Stanje SPA ostane v brskalniku in JavaScript ogrodja kot so ReactJS, VueJS, AppRun,... upravljajo s stanji v brskalniku in posodabljajo prikazano vsebino deloma in dinamično. Da aplikacije ne postanejo neobvladljivo velike, ta ogrodja podpirajo uporabo komponent, kot gradniki za gradnjo SPA. Ti gradniki so organizirani in upravljani s pomočjo ECMAScript (ES) moduli. Komponente med seboj komunicirajo preko dogodkov [11] [12].

## Progresivne spletne aplikacije

Na hitro še beseda o progresivnih spletnih aplikacija (PWA), tu pa ne gre zgolj za eno orodje ampak za način razmišljanja, kako uporabnikom zagotoviti čim boljšo izkušnjo z našo aplikacijo. Ko uparimo samopostrežne zaledne rešitve in SPA, lahko zelo elegantno pridemo do PWA rešitve z malo dodatnega truda.

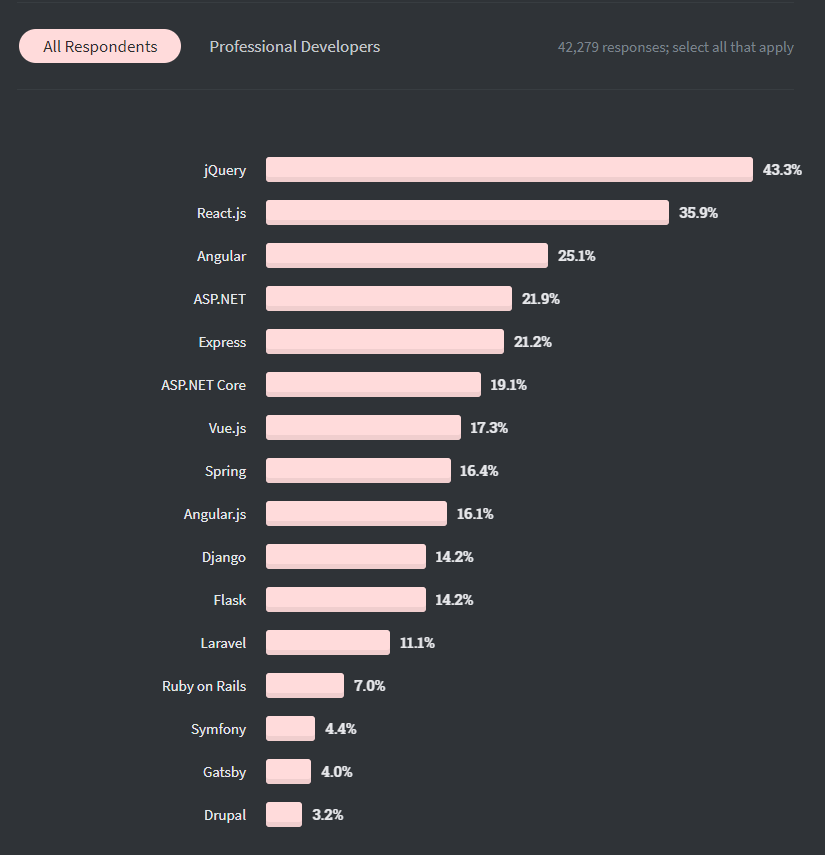
Pod ime PWA sodijo spletne aplikacije ki so odzivne, zanesljive in pritegnejo uporabnike tako, da jim postopoma popestrijo uporabniško izkušnjo ne glede na brskalnik, platformo ali napravo. Tu je ključnega pomena uporaba modernih API-jev, ki nam jih nudijo brskalniki.

Ne glede na izbiro orodja, ogrodja, platforme in programskega jezika, PWA morajo imeti sledeče lastnosti [13]:

* Takojšnja naložitev, aplikacija naj se naloži hitro in naj bo interaktivna čimprej.
* Neodvisna od povezave, z nič ali počasno nestabilno povezavo, mora aplikacija delovati nemoteno.
* Odzivni, mobilno-prvi, brez interneta-prvo snovanje, osredotočimo se na podobo na mobilni napravi prvo, ki ima manjše strojne zmogljivosti in mora biti popolnoma uporabna na mobilnih napravah.
* Opomniki, obveščanje uporabnika na posodobitve na spletni strani
* Občutek domorodnosti, aplikacija naj daje vtis domorodne rešitve. To se lahko zagotovi s pomočjo strojnih API-jev kot je Web Bluetooth.
* Varnost, ta je največje prioritete, PWA mora delovati preko HTTPS.
* Naložljiva, pomeni da bo dodana na domačo stran naprave in poganjana kot domorodna aplikacija
* Progresivna, ne glede na napravo mora naša aplikacija razvijati in pokriti nove funkcionalnosti in dati vsaki najboljšo možno uporabniško izkušnjo.

## Utemeljitev izbora React-a

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 – razvidno s slike 2-6, in hkrati tudi najbolj priljubljena med SPA ogrodji. Ta podatek je veljal za 42.279 vseh udeležencev ankete in 36.291 poklicnih udeležencev ankete [14]. Več o njihovi metodologiji je dostopno na njihovi spletni strani. Na voljo so tudi podatki o samih udeležencih, njihovem delu, lokaciji ipd. tako, da smemo sklepati, da je React dobro razširjen po vsem svetu in v profesionalni in ljubiteljski uporabi.



Slika 4‑1: Rezultati raziskave o spletnih ogrodjih [14].

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. Ti pripomore k temu, da razvijalci urejajo obnašanje in izgled komponent zelo učinkovito. Kljub vsej tej moči, pa React ostaja zelo lahko ogrodje v primerjavi z ostalimi bolj uporabljenimi [15].

# kriterijI in izbor samopostrežnega zalednega sistema v oblaku

## Definicija kriterijev

Ko izbiramo zaledno rešitev z 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 [7] [13]. Za naše potrebe sem definiral sledeče kriterije, na katere moramo biti pozorni:

* **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 možno in zaslužiti večje zneske.
* **Prilagodljivost:** ko orodje izberemo, kako težko je tega prilagoditi v nekaj kar je izven nastavitev tega? To vprašanje je lahko zelo pomembno na 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 tudi samo upravljanje enostavno s prijaznim grafičnim vmesnikom ipd.
* **Programski jezik:** katere jezike podpira orodje? Zelo pomembno je, da ponudnik storitve podpira jezika 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 se zavedati, če nismo eksperti da 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 potrebno premisliti kakšna je trenutna potreba po varnosti, kako občutljive podatke shranjujemo o uporabnikih in zagotovitev zakonodajne skladnosti, še posebno v okoliščinah ko sta zasebnost in varnost poudarjeni.
* **Strojno učenje:** današnje aplikacije imajo nek občutek zastarelosti, če vsaj ne razmišljajo u 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 potrebno 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 orodja, 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 jih tako uspemo zatreti, preden te povzročijo astronomske stroške [16]. 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.

## Rešitev z REST vmesnikom

Ko govorimo o lastni REST rešitvi, je lahko že sama odločitev za katero 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 teh. Ko razvijamo lasten REST vmesnik, je spet zelo odvisno od katerega izberemo, po navadi 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 nek 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 tudi vso odgovornost v rokah mi in je potrebno temu primerno oceniti čas trajanja in stroške razvoja naše rešitve [17].

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 teh stroškov, če planiramo izvajati neke operacije na zalednem sistemu, ki so neprimerne oz. cenovno neugodne za poganjanje na samopostrežni strežniški arhitekturi. Lahko pa kar nekaj stroškov pri izračunu dolgoročnega vzdrževanje 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 če se poslužimo dobro poznane in preverjene knjižnice lahko zagotovimo dovolj dobro varnost, ampak na nas vseeno pade posodabljanje in spremljanje novic o tej knjižnici ter menjavi le-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 nemogoč 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 [11].

## 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 tu 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, integracija z ostalimi storitvami je vrhunska, sploh s pomočjo amplify CLI, kjer s samo nekaj ukazi določimo storitve, ki jih želimo koristiti in se te v naš projekt dodajo avtomatsko s samodejno dodelitvijo virov v oblaku [18].

Cenitev AWS Amplify je lahko na trenutke zahrbtna in varljiva, saj že da pridemo da samih cenitev storitev je potrebno kar nekaj brskanja in kopanja znotraj različnih menijev. Poleg tega je cenitev še odvisna od virov ki jih funkciji namenimo in tako dalje. AWS sicer nudi nekaj primerov cenitve teh storitev, vendar ni ravno vizualno privlačno, prikazano na sliki 3.1.



Slika 5‑1: Prikaz informativnega izračuna uporabe storitev Amplify [19]

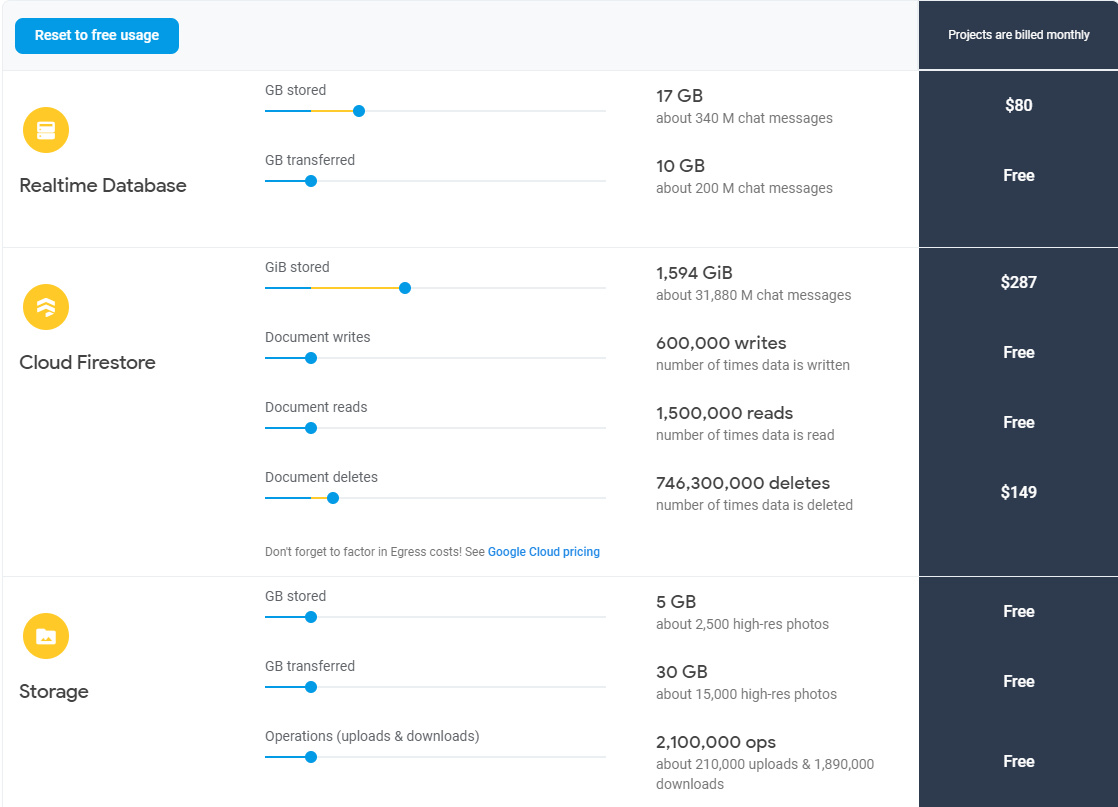
Postopek iskanja cenitve in potem samostojno preračunati vsak strošek posebej je potrebno ponoviti za vsako storitev, ki jo želimo dodati projektu. Ta postopek je zelo neprijazen in neodporen na napake v izračunih brez kakršnekoli odgovornosti s strani AWS. AWS Amplify sicer nudi brezplačne storitve do neke mere, vendar zahteva podatke za plačilo morebitnih stroškov, saj ko porabimo brezplačne storitve naprej računa po redni ceni. Ponovno lahko nastanejo ne željeni stroški, če si predhodno sami ne nastavimo celovitega sistema obveščanja ob porastu stroškov [20].

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 ne intuitivna. Uporaba strojnega učenja je dobra, funkcionalnosti ki jih zagotavlja je kar nekaj od prepoznavanja slavnih oseb, besedila, generiranje besedila ipd. [19]

## 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 nadoknadi 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. Velikost skupnosti ni tako velika, je pa ta zelo zagnana in pripravljena priskoči 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 [20]. Uradni programski jeziki so Node.js (JavaScript), Java, Python, Go, C# [21].

Cenitev pri Firebase je veliko bolj prijetna in intuitivna kot pri Amplify. Vsak lahko za svoje primere pogleda in nastavi informativni izračun za lastne potrebe in 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 [22]

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 [18] [20] [23].

Za varnost je poskrbljeno na vse možne načine, preko več različnih poti overjanja uporabnika, telefonskega klica, sms-a, večstopenjsko avtentifikacijo in podobne zadeve. 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 recimo hrana itd. [22]

## Primerjava

Za lažjo vizualizacijo problema, bom vse tri možne kandidate zvrstil v preglednico in kategoriziral njihove uspehe v določeni kategoriji z vrednostmi od 1 do 3. To pomeni, da bo primerjava narejena v vrstnem redu, katero orodje je v posamezni kategoriji najboljše. Vrednost 1 predstavlja da je orodje najslabše vrednost 3 pa 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 |
| Velikost skupnosti | 1 | 3 | 2 |
| Programski jeziki | 3 | 1 | 2 |
| Cenitev storitev | 3 | 2 | 1 |
| Varnost | 1 | 2 | 3 |
| Orodja za razvijalce | 3 | 1 | 2 |
| Beleženje in monitoring | 1 | 2 | 3 |
| Strojno učenje | 1 | 2 | 3 |
| Skupaj: | 18 | 19 | 23 |

Tabela 3-1: Primerjava orodij med seboj po primerjalni lestvici

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

## Izbor

Kot pokazano v prejšnjem poglavju, za namene enostranskih spletnih aplikacij je najboljše orodje Firebase. Do tega sklepa smo prišli zato, saj 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 je dobra dokumentacija, zavzeta skupnost in transparentno cenjenje storitev med najbolj privlačnimi.

# Razvoj zalednega sistema Firebase in aplikacije

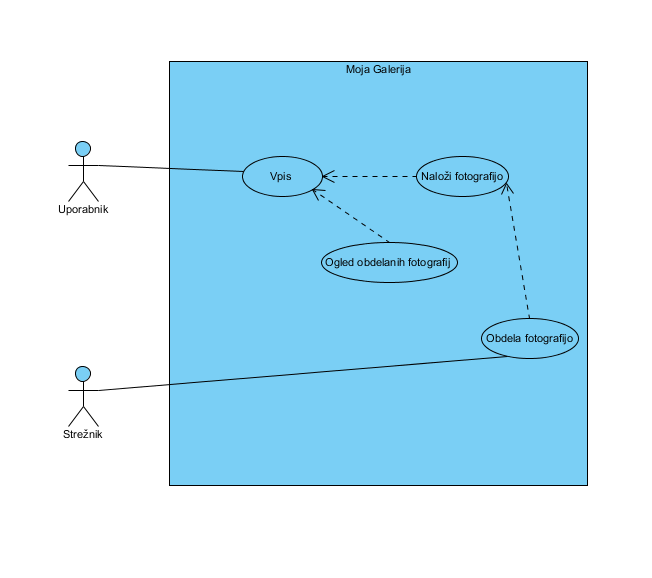
## Funkcionalnosti

Za potrditev ugotovitve prejšnjega poglavja – da je Firebase najboljši za uporabo v kombinaciji z enostranskimi spletnimi aplikacijami, bomo razvili enostavno spletno aplikacijo. Razvili bomo 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 bo gostovana na Firebase in javno dostopna preko HTTPS. Tako bomo 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

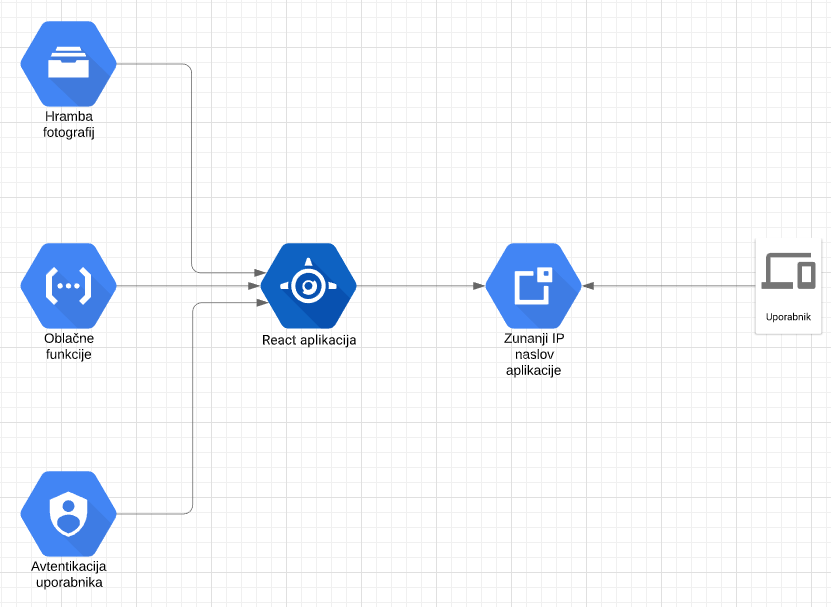
Tako bomo dobili jedrnat vpogled v sodelovanje Firebase-a z React-om in lahko ocenili ali je res tako razvijalcu prijazno, kot je zatrjeno v prejšnjih poglavjih. Primere uporabe si lahko za lažjo predstavo narišemo s pomočjo diagrama primerov uporabe Slika 6-1. S tega diagrama je tudi razvidno, da je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



Slika 6‑1 Diagram primerov uporabe za aplikacijo Moja Galerija

## Arhitektura

Za boljšo predstavo, kako našo aplikacijo uporaba oblačnih storitev modulira, si oglejmo diagram arhitekture, Slika 6-2. 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 za sebe deluje in uporaba ostalih na njo ne vplivajo, kar zagotovi najhitrejšo in tekočo uporabniško izkušnjo.

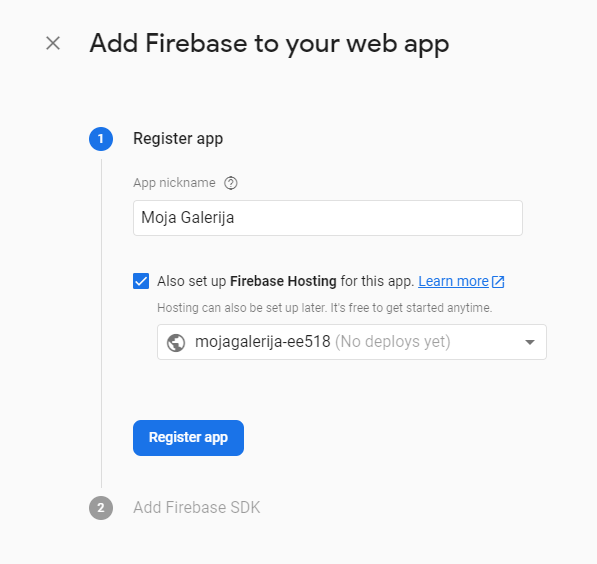


Slika 6‑2 Arhitektura aplikacije, ki jo bomo razvili

## Priprava okolja

Za razvoj samopostrežnega zalednega sistema s Firebase je preprosto. Sistemi, ki že razvijajo s pomočjo tehnologije React, že imajo večino namestitve urejene. Potreben je nek 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)

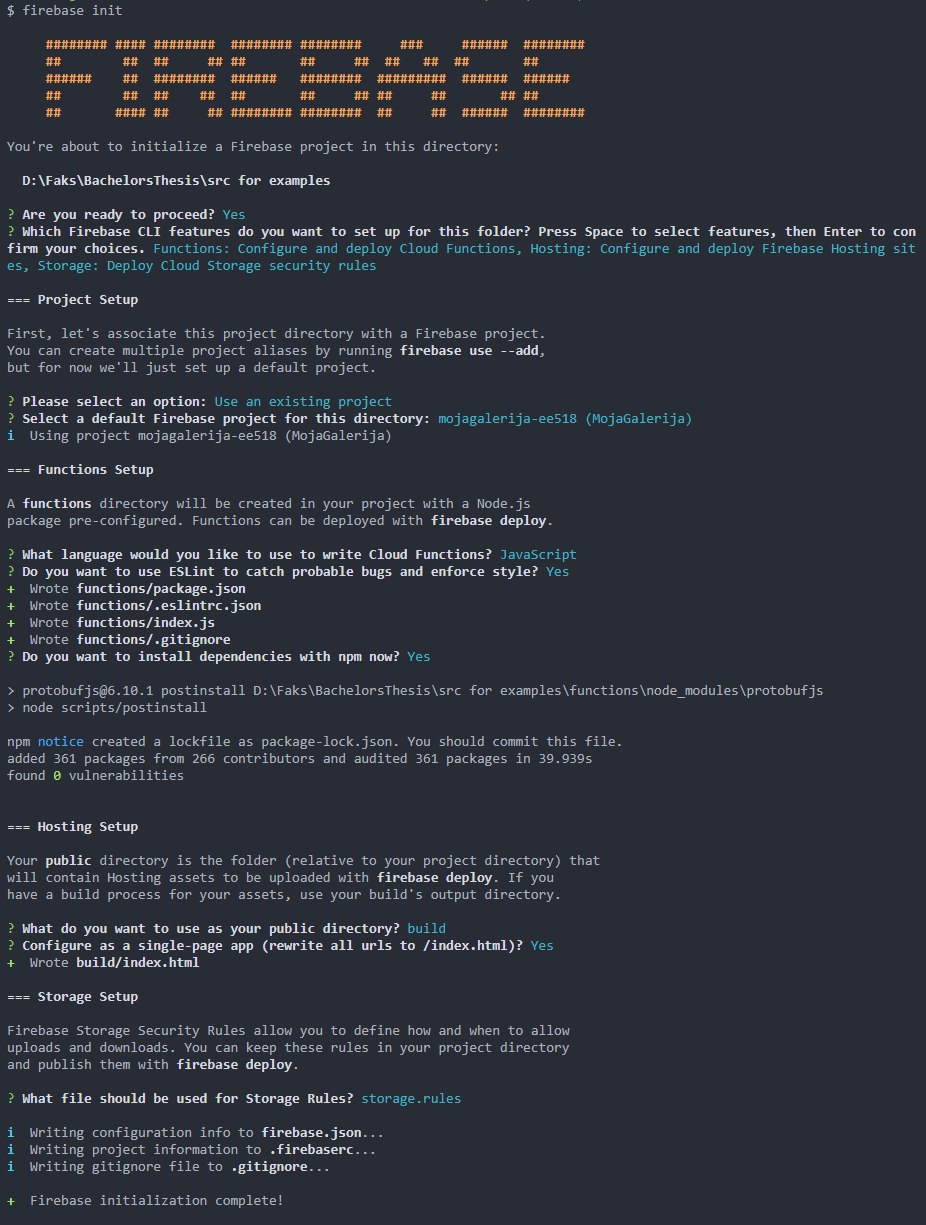
Predpostavil bom, da so NodeJS, npm, npx in Visual Studio Code že naloženi, torej ostane samo še namestitev Firebase Tools, skozi katerega nas bodo vodila nadaljnja navodila. S pomočjo ukaza »npx create-react-app mojagalerija«, ustvarimo nov React projekt, ki ga bomo registrirali kot Firebase projekt. Nato obiščemo spletno stran [https://console.firebase.google.com](https://console.firebase.google.com/u/0/), 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 potrebno 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 6-3. 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 [24], tega bomo izvedli kasneje znotraj naše aplikacije.

Slika 6‑3 Prikaz vodenih korakov vzpostavitve Firebase projekta

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

Preden izvedemo korak 4., se postavimo v datotek 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 sledeče opcije, razvidne s slike 6-4. Izbrane možnosti so označene s svetlo modro barvo v konzoli. Preden React aplikacijo namestimo na gostovanje, je potrebno ustvariti produkcijsko zgradnjo aplikacije, z ukazom »npm run build«. In nato »Firebase deploy«, da spravimo našo aplikacijo na javni naslov z SSL certifikatom.

Ker smo naredili kar veliko, naj povzamem. 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 galerija.

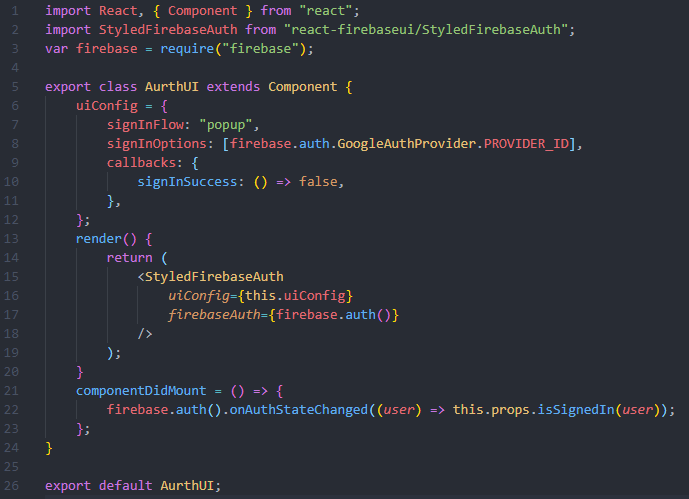


Slika 6‑4 Prikaz izbranih možnosti (v modri barvi)

## Razvoj funkcionalnosti

### Avtentifikacija

Prvo 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 6-5. 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.



Slika 6‑5 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.

### Naložitev fotografije

Oglejmo si kako izgleda komponenta na naložitev fotografije v našo galerijo. Za lažje obravnavanje komponente, si jo bomo ogledali v dveh slikah 6-6 in 6-7. 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.



Slika 6‑6 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 procentov naše datoteke se je že naložilo ali pa poslušamo za napake – v tem primeru poslušamo samo za napake in jih beležimo v konzolo.



Slika 6‑7 Implementacija komponente za naložitev fotografije - del za delo s hrambo

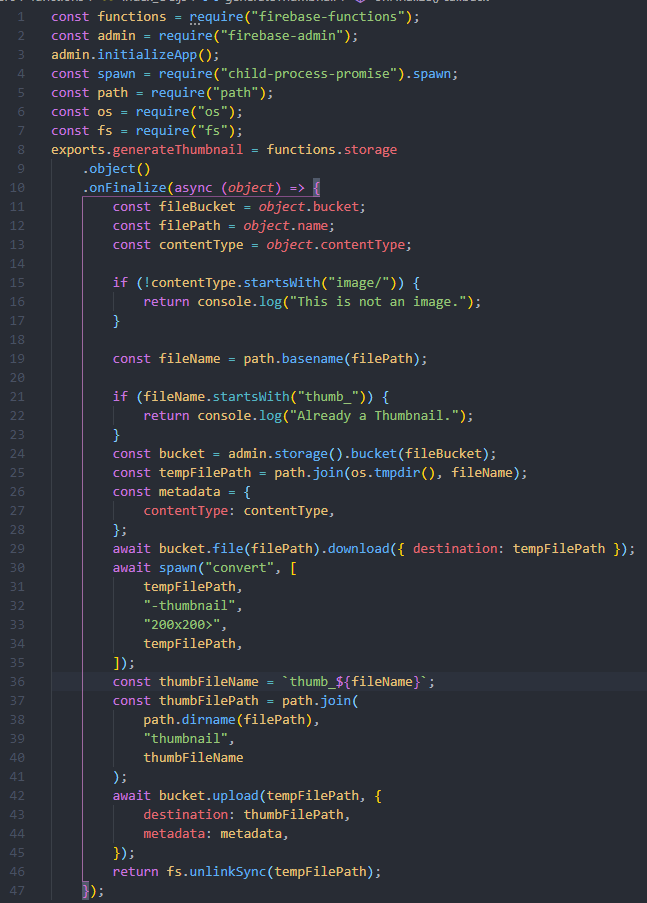
Ponovno je samo delo s Firebase zelo enostavno in je problem rešen v nekaj vrsticah. Poleg enostavne rešitve pa nudi robusten vpogled v samo dogajanje, če si to želimo.

### 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 6-8, potek dela je zelo direkten – na objekt »exports« vežemo funkcije z željenim imenom, v našem primeru je ime »generateThumbnail«. Vežemo jo na prožilec objekta »storage«, ko se zaključi delo z nekom objektom. Ko se to izvrši, se začne izvajati naša funkcije, 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 željene parametre. Nato datoteko zapišemo na disk in jo naložimo nazaj na Firebase hrambo podatkov, ter brisanje začasne datoteke z diska.



Slika 6‑8 Implementacija procesa obdelave slik na manjše ikone oz. thumbnail

Ponovno je proces zelo enostaven, saj pišemo funkcijo enako kot bi se izvajala na lokalni napravi, vendar imamo dodatne možnosti s poslušanjem za dogodke v oblaku. Če funkcijo spremenimo se bo ukazu »firebase deploy« samodejno posodobi.

### Prikaz obdelane fotografije

Za prikaz obdelanih fotografij ponovno ustvarimo namensko komponento, ki ima v stanju polje za zapis poti do datotek, slika 6-9. 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 6-10. 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 rezuktat. In tako je prikaz fotografij modularno urejen in pripravljen tudi na večje projekte.



Slika 6‑9 Implementacija komponente galerije



Slika 6‑10 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.

# Sklepi

Tekom diplomskega dela smo si ogledali trenutno stanje razvoja programske opreme in zakaj so samopostrežne rešitve tako popularne. Preleteli smo tudi ostale rešitve, ki omogočajo različne načine razvoje 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 so te tehnologije zelo dobro opremljene za medsebojno sodelovanje in je končni izdelek lahko fleksibilen, robusten in očesu privlačen. Ker je število ponudnikov oblačnih storitev kar nekaj in ker samo 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. Izkazalo se je, da je med tremi najbolj primeren Firebase za delo z enostranskimi spletnimi aplikacijami, saj je najenostavnejši za uporabo in ponuja dovolj veliko skupnost razvijalcev in nudi najboljšo integracijo z ostalimi sistemi, ki jih ponuja Google. 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, povezava med projektom na naši napravi in vsemi uporabljenimi storitvami.

## Možne izboljšave in nadaljnje raziskave

Vsekakor bi bilo v kakšno podobno primerjavo zajeti Microsoft in njihove oblačne ponudbe, ki so v večjih podjetjih vsekakor zelo popularne. V sklopu te naloge sem se osredotočil na prednosti in slabosti za enostranske spletne aplikacije, kar pa je dokaj ozek spekter uporabe in ta orodja zagotovo lahko prispevajo ogromno na drugih področjih. Med mojo raziskavo sem imel v mislih manjše ekipe ali celo samostojne razvijalce, katerim te storitve res ogromno olajšajo. Zanimivo bi bilo preveriti koliko lahko ti sistemi samostojno razviti in ti zunanjih ponudnikov nudijo velikim podjetjem v poslovnih okoljih, kjer je zahtevana podatkovna tajnost ipd. katere vidike bi bilo potrebno poudariti v tem primeru?

# 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] | 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]. |
| [6] | 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]. |
| [7] | Cloudflare, „Cloudflare,“ [Elektronski]. Available: https://www.cloudflare.com/learning/serverless/why-use-serverless/. [Poskus dostopa 30 6 2020]. |
| [8] | Amazon, „AWS Lambda Pricing,“ [Elektronski]. Available: https://aws.amazon.com/lambda/pricing/. [Poskus dostopa 30 5 2020]. |
| [9] | J. Hanson, „Hackernoon,“ 26 6 2017. [Elektronski]. Available: https://hackernoon.com/five-advantages-of-serverless-technology-68160c1f884e. [Poskus dostopa 30 6 2020]. |
| [10] | 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]. |
| [11] | Y. Sun, Practical Application Development with AppRun, Berkeley: Apress, 2019. |
| [12] | P. Späth, „Building Single-Page Web Applications with REST and JSON,“ v *Beginning Jakarta EE*, APress, 2019, pp. 113-143. |
| [13] | M. Hajian, Progressive Web Apps with Angular, Berkeley: Apress, 2019. |
| [14] | StackOverflow, „StackOverflow,“ 2 2020. [Elektronski]. Available: https://insights.stackoverflow.com/survey/2020#technology-web-frameworks-all-respondents2. [Poskus dostopa 6 7 2020]. |
| [15] | T. Vo, Web Application Development with React and Google Firebase, Turku, Finland: Turku University of Applied Sciences, 2020. |
| [16] | 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]. |
| [17] | M. Podplatnik, Primerjava ogrodij za zaledne sisteme mobilnih aplikacij : diplomsko delo, Maribor: M. Podplatnik, 2019. |
| [18] | P. A. Kheta, „Preeti,“ 4 5 2020. [Elektronski]. Available: https://p3420a1.wixsite.com/preeti/post/aws-amplify-vs-firebase. [Poskus dostopa 9 7 2020]. |
| [19] | Amazon, „AWS Amplify,“ Amazon, [Elektronski]. Available: https://docs.amplify.aws. [Poskus dostopa 16 3 2020]. |
| [20] | 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]. |
| [21] | Google, „Google,“ Google, 6 7 2020. [Elektronski]. Available: https://firebase.google.com/docs/admin/setup. [Poskus dostopa 9 7 2020]. |
| [22] | Google, „Firebase,“ Google, [Elektronski]. Available: https://firebase.google.com/docs. [Poskus dostopa 16 3 2020]. |
| [23] | N. Broda, „StackShare,“ 17 6 2020. [Elektronski]. Available: https://stackshare.io/stackups/aws-amplify-vs-firebase. [Poskus dostopa 9 7 2020]. |
| [24] | Firebase, „Firebase,“ 2020. [Elektronski]. Available: https://firebase.google.com/docs/web/setup?authuser=0#node.js-apps. [Poskus dostopa 26 7 2020]. |
| [25] | Facebook, „React,“ Facebook, [Elektronski]. Available: https://reactjs.org/. [Poskus dostopa 16 3 2020]. |
| [26] | D. Lamas, F. Loizides, L. Nacke, H. Petrie, M. Winckler in P. Zaphiris, Human-Computer Interaction – INTERACT 2019, Cham: Springer, 2019. |
| [27] | L. Moroney, The Definitive Guide to Firebase, Berkeley: Apress, 2017. |
| [28] | 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. |
| [29] | A. Freeman, Pro Windows 8 Development with HTML5 and JavaScript, Berkeley: Apress, 2012. |
| [30] | L. Baresi in M. Garriga, Microservices: The Evolution and Extinction of Web Services?, Cham: Springer, 2019. |