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](#_Toc44000553)

[1.1 Opredelitev problema 1](#_Toc44000554)

[1.2 Cilji zaključnega dela 2](#_Toc44000555)

[1.3 Predpostavke in omejitve 2](#_Toc44000556)

[2. SAMOPOSTREŽNI ZALEDNI SISTEMI V OBLAKU ZA ENOSTRANSKE SPLETNE APLIKACIJE 3](#_Toc44000557)

[2.1 Evolucija storitev v oblaku 4](#_Toc44000558)

[3. VIRI IN LITERATURA 6](#_Toc44000559)

[References 6](#_Toc44000560)

Kazalo slik

[Slika 2.1: Razlike med monolitnimi, mikrostoritvenimi in FaaS modeli 6](#_Toc44026261)

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* |
|  |  |

# UVOD

V področju informatike je v zadnjih letih na udaru 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.

# SAMOPOSTREŽNI ZALEDNI SISTEMI V OBLAKU ZA ENOSTRANSKE SPLETNE APLIKACIJE

Za razumevanje rešitev, ki nam 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.

## Evolucija storitev v oblaku

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.

### Okolje kot storitev

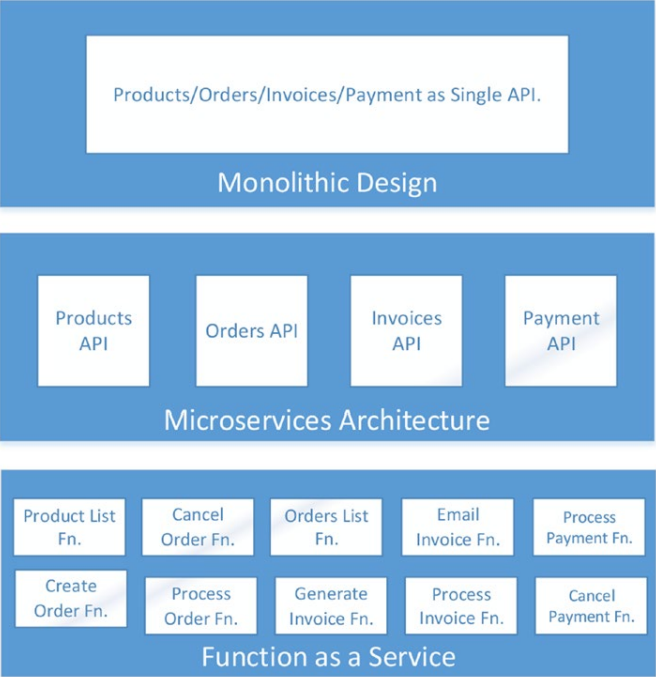
Okolje kot storitev oz. Platform as a Service (PaaS) je naslednja strategija za oblačno računalništvo. Zagotovi boljši nivo abstrakcije na strojno opremo in OS, ter namestitev aplikacije v primerjavi z IaaS. PaaS rešitve so po zasnovi visoko razpoložljive in razširljive. V primerjavi z rešitvami ki gostujejo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

### 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.

### 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.



Slika 2‑1: Razlike med monolitnimi, mikrostoritvenimi in FaaS modeli

Na sliki 2.2 je ponazorjeno kako izgledajo bolj poznane storitve na Microsoft Azure z prej omenjenimi strategijami oblačnega računalništva.



Slika 2‑2: Oblačne storitve Microsoft Azure z različnimi modeli oblačnega računalništva

## 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

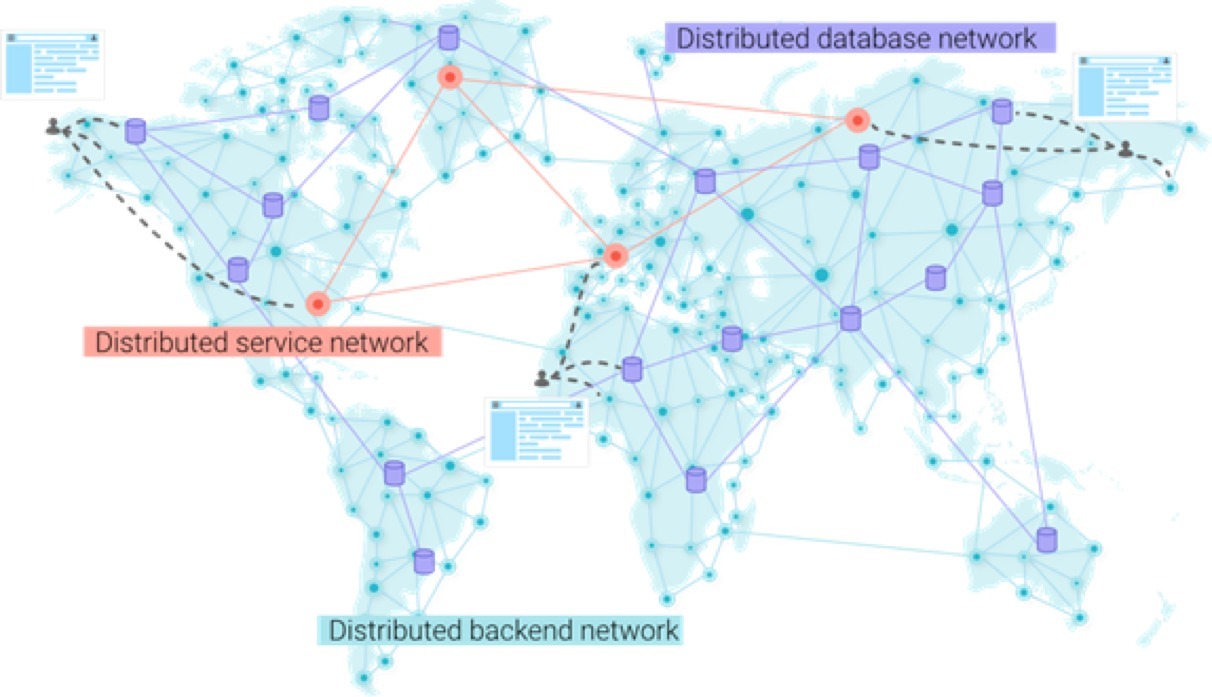
Hiter tempo po katerem samopostrežno računalništvo in temu sorodne rešitve zorijo kaže, da bo zahteven tekmovalec za svoje predhodnike. Trenutno je smatran kot najbolj zanesljiva nastajajoča paradigma. Podrobneje si poglejmo določene prednosti in omejitve, ki nam povedo več o sodelovanju tega paradigma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

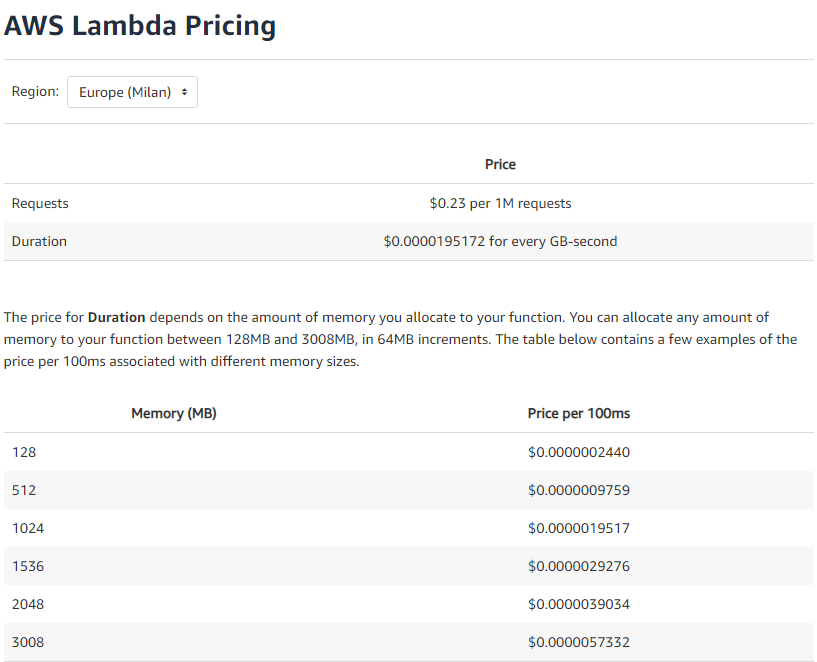


Slika 2‑3: Porazdeljena arhitektura samopostrežnih zalednih sistemov

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. [6]

### Nizki stroški delovanja – plačaj kot porabiš

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 2‑4: Cenovni model AWS Lambda [7]

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 2‑5: Primer zaračunanja stroškov delovanja konkretne funkcije [7]

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. [8]

### 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 [9], 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 [6].

## 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 [10] [11].

### 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 [12]:

* 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.

### React

# 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] | Cloudflare, „Cloudflare,“ [Elektronski]. Available: https://www.cloudflare.com/learning/serverless/why-use-serverless/. [Poskus dostopa 30 6 2020]. |
| [7] | Amazon, „AWS Lambda Pricing,“ [Elektronski]. Available: https://aws.amazon.com/lambda/pricing/. [Poskus dostopa 30 5 2020]. |
| [8] | J. Hanson, „Hackernoon,“ 26 6 2017. [Elektronski]. Available: https://hackernoon.com/five-advantages-of-serverless-technology-68160c1f884e. [Poskus dostopa 30 6 2020]. |
| [9] | 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]. |
| [10] | Y. Sun, Practical Application Development with AppRun, Berkeley: Apress, 2019. |
| [11] | P. Späth, „Building Single-Page Web Applications with REST and JSON,“ v *Beginning Jakarta EE*, APress, 2019, pp. 113-143. |
| [12] | Google, „Firebase,“ Google, [Elektronski]. Available: https://firebase.google.com/docs. [Poskus dostopa 16 3 2020]. |
| [13] | Amazon, „AWS Amplify,“ Amazon, [Elektronski]. Available: https://docs.aws.amazon.com/index.html?nc2=h\_ql\_doc\_do\_v. [Poskus dostopa 16 3 2020]. |
| [14] | Facebook, „React,“ Facebook, [Elektronski]. Available: https://reactjs.org/. [Poskus dostopa 16 3 2020]. |
| [15] | D. Lamas, F. Loizides, L. Nacke, H. Petrie, M. Winckler in P. Zaphiris, Human-Computer Interaction – INTERACT 2019, Cham: Springer, 2019. |
| [16] | M. Podplatnik, Primerjava ogrodij za zaledne sisteme mobilnih aplikacij : diplomsko delo, Maribor: M. Podplatnik, 2019. |
| [17] | L. Moroney, The Definitive Guide to Firebase, Berkeley: Apress, 2017. |
| [18] | 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. |
| [19] | A. Freeman, Pro Windows 8 Development with HTML5 and JavaScript, Berkeley: Apress, 2012. |
| [20] | M. Hajian, Deploying to Firebase as the Back End, Berkeley: Apress, 2019. |
| [21] | L. Baresi in M. Garriga, Microservices: The Evolution and Extinction of Web Services?, Cham: Springer, 2019. |