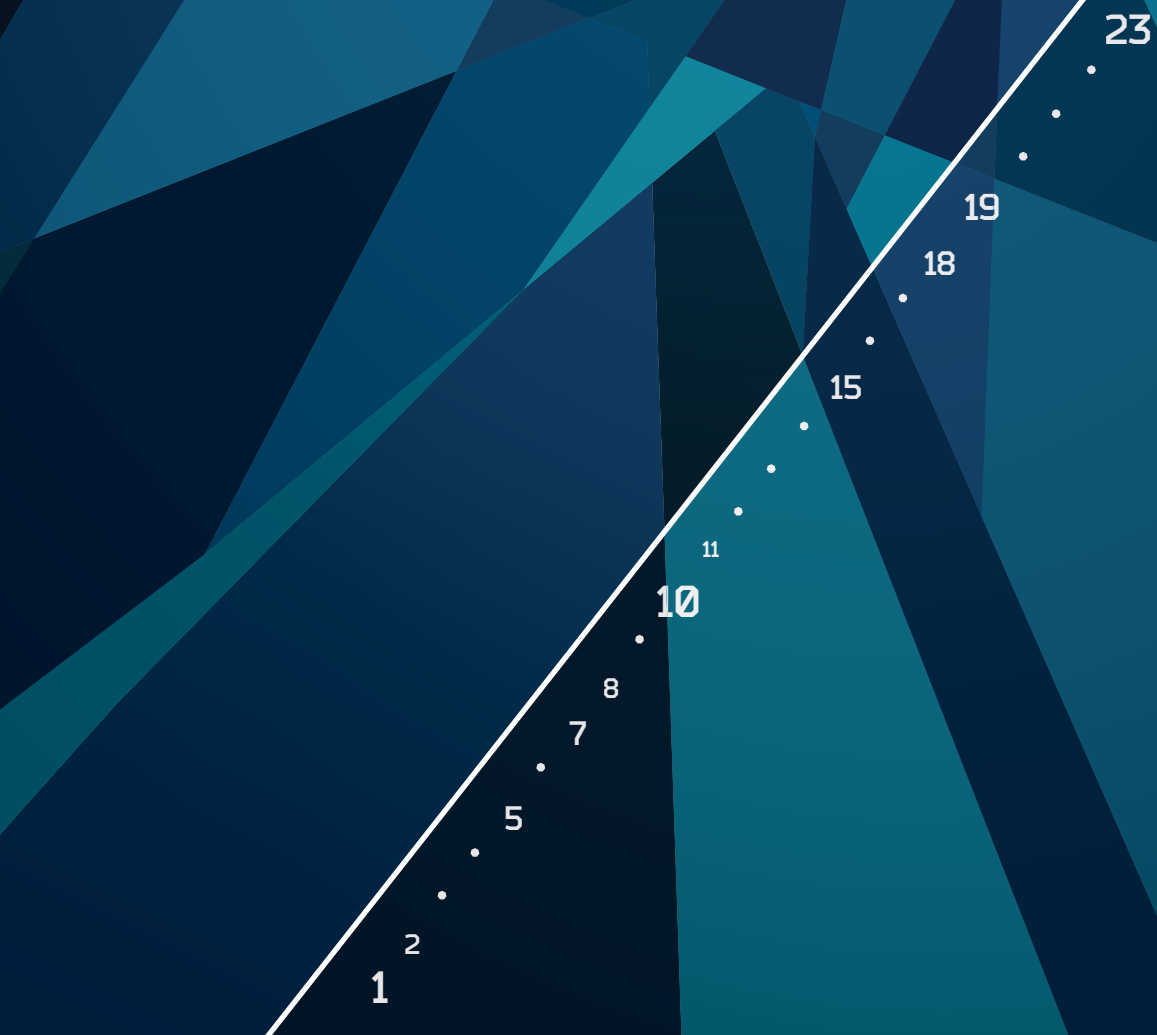
Univerzitetna založba
Univerze v Mariboru

24 konferenca sodobne informacijske tehnologije in storitve

ITS 2019



Urednika:
Marjan Heričko,
Katja Kous

Zbornik prispevkov

Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko

OTS 2019

Sodobne informacijske tehnologije in storitve

Zbornik štiriindvajsete konference, Maribor, 18. in 19. junij 2019

Urednika
Marjan Heričko
Katja Kous

Junij 2019

Naslov OTS 2019 Sodobne informacijske tehnologije in storitve

Podnaslov Zbornik štiriindvajsete konference, Maribor, 18. in 19. junij 2019

Title OTS 2019 Advanced Information Technology and Services

Subtitle Conference proceedings of the Twenty-fourth Conference, Maribor, June 18th & 19th, 2019

Urednika *Editors*
red. prof. dr. Marjan Heričko
(Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko)

asist. dr. Katja Kous
(Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko)

Tehnična urednika *Technical editors*
asist. Saša Kuhar
(Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko)

Jan Perša, mag. inž. prom.
(Univerzitetna založba Univerze v Mariboru)

Oblikovanje ovitka *Cover designer*
asist. Saša Kuhar
(Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko)

Grafika na ovitku *Cover graphics* Urednika

Grafične priloge *Graphics material* Avtorji prispevkov

Konferenca *Conference*
OTS 2019 Sodobne informacijske tehnologije in storitve

Kraj in datum *Location and date*
Maribor, 18. in 19. junij 2019

Programski odbor *Program committee*
prof. dr. Marjan Heričko (predsednik konference, vodja), prof. dr. Tatjana Welzer Družovec, dr. Tomaž Domajnko, dr. Boštjan Grašič, dr. Boštjan Kežmah, dr. Dean Korošec, dr. Luka Pavlič, dr. Boštjan Šumak, dr. Muhamed Turkanović, mag. Bojan Štok, mag. Ivan Lah in Milan Gabor.

Organizacijski odbor *Organizing committee*
dr. Katja Kous (vodja), Saša Kuhar, Boris Lahovnik, dr. Tina Beranič, dr. Maja Pušnik, Miha Strehar, Lucija Brezočnik, Mitja Gradišnik, Jernej Huber, dr. Gregor Jošt, dr. Sašo Karakatič, Blaž Podgorelec, Alen Rajšp, Patrik Rek in Viktor Taneski.

Založnik / *Published by*
Univerzitetna založba Univerze v Mariboru
Slomškov trg 15, 2000 Maribor, Slovenija
http://press.um.si, zalozba@um.si

Izdajatelj / *Co-published by*
Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko
Koroška cesta 46, 2000 Maribor, Slovenija
https://feri.um.si, feri@um.si

Izdaja *Edition* Prva izdaja

Vrsta publikacije *Publication type* E-knjiga

Dostopno na *Availabe at* http://press.um.si/index.php/ump/catalog/book/420

Izid *Published* Maribor, junij 2019

CIP - Kataložni zapis o publikaciji
Univerzitetna knjižnica Maribor

004.946.5:004.7(082)(0.034.2)

KONFERENCA Sodobne informacijske tehnologije in storitve (24 ; 2019 ; Maribor)
Sodobne informacijske tehnologije in storitve [Elektronski vir] : OTS 2019 : zbornik štiriindvajsete konference, Maribor, 18. in 19. junij 2019 / urednika Marjan Heričko, Katja Kous. - 1. izd. - Maribor : Univerzitetna založba Univerze, 2019

Način dostopa (URL): http://press.um.si/index.php/ump/catalog/book/420
ISBN 978-961-286-282-4 (pdf)
doi: 10.18690/978-961-286-282-4
1. Gl. stv. nasl. 2. Heričko, Marjan
COBISS.SI-ID 96825857

ISBN 978-961-286-282-4 (PDF)
978-961-286-283-1 (Broš.)

DOI https://doi.org/10.18690/978-961-286-282-4

Cena *Price* Brezplačni izvod

Odgovorna oseba založnika *For publisher* red. prof. dr. Zdravko Kačič, rektor Univerze v Mariboru

http://www.ots.si

Spoštovane in spoštovani,

informatika in informacijske tehnologije nedvomno omogočajo prebojne inovacije na vseh področjih poslovanja organizacij in našega vsakdanjega življenja. Vseprisotne rešitve in storitve prispevajo k digitalizaciji, poslovni odličnosti in višji kakovosti sodelovanja ter bivanja, a le v primeru, ko je umetna inteligenca nadgrajena in obogatena z razumom človeka. Zato je nujno, da se zavedamo pasti in izzivov razvoja tovrstnih rešitev ter jih hkrati znamo ustrezno ovrednotiti glede na namen uporabe in domeno apliciranja.

Konkretne izkušnje dokazujejo, da se ob primernem načrtovanju in razvoju tehnologija veriženja blokov lahko učinkovito in smiselno uporabi tudi v zahtevnih poslovnih procesih.

Zasnova in razvoj novih produktov že desetletja temelji na agilnih vrednotah, porazdeljenih interdisciplinarnih skupinah ter avtomatiziranih orodjih in virtualnih okoljih. Vse bolj porazdeljene in decentralizirane pa so tudi infrastrukture in aplikacije, ki so sposobne razpoznati enotno identiteto uporabnikov. Za vzpostavitev zaupanja morajo tehnološkim rešitvam slediti tudi ustrezni organizacijski pristopi in novi regulativni okviri ter hkratna nenehna skrb za ozaveščanje uporabnikov.

V zborniku prispevkov so tudi letos predstavljene inovativne informacijske rešitve in storitve ter skozi konkretne projekte pridobljene izkušnje s/z:

- vpeljavo tehnologij strojnega učenja in obogatene inteligence,
- uporabo tehnologij in platform veriženja blokov,
- razvojem šibko sklopljenih mikrostoritev,
- popolno virtualizacijo in izkoriščanjem porazdeljenih infrastruktur,
- zagotavljanjem kibernetske varnosti, zaupnosti in zasebnosti,
- skaliranjem agilnih metod v porazdeljenih projektnih skupinah,
- vpeljavo agilnih pristopov v sklopu avtomatiziranih in neprekinjenih procesov razvoja, testiranja, integracije in dostave,
- posodobitvijo in nadgradnjo obstoječih informacijskih sistemov,
- razvojem uporabniško prijaznih spletnih in mobilnih rešitev in
- uvajanjem sodobnih programskih jezikov in razvojnih okolij.

Prispevki v zborniku 24. konference OTS 2019 Sodobne informacijske tehnologije in storitve nedvomno omogočajo odličen vpogled v najaktualnejše tehnološke in organizacijske trende prav na vseh ključnih področjih računalništva in informatike. In kar je najpomembneje - predstavljene praktične izkušnje omogočajo kritično ovrednotenje uporabnosti posameznih pristopov, tehnologij in orodij.

dr. Katja Kous
vodja organizacijskega odbora OTS 2019

prof. dr. Marjan Heričko
predsednik 24. konference OTS 2019

KAZALO

OTS 2019
SODOBNE INFORMACIJSKE TEHNOLOGIJE IN STORITVE
M. Heričko in K. Kous (ur.)

Univerzitetna založba
Univerze v Mariboru

DOBRE PRAKSE AI NA PODROČJU ZDRAVSTVA OZIROMA ZAKAJ GOOGLANJE ZDRAVSTVENIH TEŽAV NI NAJBOLJŠA IDEJA

VID VISOČNIK, LUKA GRATEJ

Povzetek: *Problematiko dostopa do kakovostnih zdravstvenih informacij in storitev lahko zahvaljujoč tehnološkemu napredku v zadnjem desetletju rešujemo tudi s pomočjo pametnih asistentov (ang. chatbots), ki jih poganja obogatena inteligenca (ang. augmented intelligence). Primer digitalnega osebnega zdravstvenega asistenta, razvitega v podjetju Your.MD, prikazuje koncept prenosa razmišljanja in znanja zdravnikov v računalniku razumljivo obliko s poudarkom na treh ključnih nalogah: razumevanju uporabnikov s pomočjo strojnega procesiranja naravnega jezika (ang. natural language processing), algoritmih za nudenje zdravstvenih informacij in podajanju nadaljnjih korakov oz. rešitev v okviru mreže zunanjih ponudnikov zdravstvenih storitev pod imenom OneStop Health™.*

Ključne besede: *Your.MD, pametni asistent, obogatena inteligenca, procesiranje naravnega jezika, digitalizacija zdravstvenih storitev*

NASLOVA AVTORJEV: Vid Visočnik, Računalniške storitve, Vid Visočnik s.p., Maribor, Slovenija. e-pošta: vid.visocnik@gmail.com. Luka Gratej, Luka Gratej Consulting, s.p., Radlje ob Dravi, Slovenija. e-pošta: luka.gratej@gmail.com.

https://doi.org/10.18690/978-961-286-282-4.1 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 UVOD

Kako edinstvena je celota naravnih procesov, ki sestavlja in poganja naše telo, se ponavadi zavemo šele takrat, ko zbolimo. Zagotavljanje zdravja in dobrega počutja je neposredno povezano s kakovostjo našega življenja, zato v primeru bolezni in zdravstvenih težav stremimo k čimprejšnji ozdravitvi s pomočjo zaupanja vrednih rešitev: informacij, podatkov in kredibilnih nasvetov o nadaljnjih korakih na poti do ozdravitve.

Težava nastane, ko edini vir zaupanja vrednih informacij predstavlja zdravstveno osebje, ki ga v današnjem času kronično primanjkuje ne samo pri nas, temveč tudi na svetovni ravni. Tukaj nastaja priložnost, da nastajajočo vrzel zapolni visoko napredna tehnologija na podlagi obogatene inteligence (ang. Augmented Intelligence), ki lahko razbremeni del zdravstvenih posvetovanj, čemur v podjetju Your.MD pravijo predzdravstvena nega oziroma "pre-primary care".

Predzdravstvena nega zajema posvetovanja za simptome, ki jih lahko zdravimo sami doma, s tem pa znatno razbremenimo že tako preobremenjeno zdravstveno osebje.

Digitaliziranje zdravstvenih posvetovanj s področja predzdravstvene nege zahteva kombinacijo večih pristopov in novih paradigem, kjer si v izdatni meri pomagamo z zmogljivo strojno opremo in naprednimi algoritmi, ki so plod tesnega sodelovanja podatkovnih znanstvenikov in ekipe zdravnikov. Okviri delovanja sistema so jasno določeni: nadgradnje in izboljšave potekajo v nadzorovanih in kontroliranih pogojih pod strokovnim nadzorom, zato uporabljamo termin obogatena oz. augmentirana inteligenca.

Za nadaljnje razumevanje problema poglejmo tipične vzorce in odzive ob pojavu bolezenskih stanj.

2 RAZUMEVANJE ČLOVEKOVEGA ODZIVA

Raziskave poročajo, da ljudje ob poslabšanju zdravstvenega stanja zapademo v različne vzorce samorazlage in ignorance [1]. Premnogokrat lahko preberemo, da se bolezenskemu stanju lahko izognemo že s pravočasnim obiskom zdravnika.

Ob podrobni analizi vzorcev obnašanja, lahko korake odločanja razvrstimo v sledeče zaporedje:

1. Pojav bolezenskega stanja, ki ga ignoriramo v upanju, da se bo stanje izboljšalo samo od sebe
2. Spoznanje, da se stanje ne izboljšuje oz. celo slabša, zato se obrnemo po nasvet k partnerju ali družinskim članom
3. Iskanje informacij po svetovnem spletu
4. Obisk družinskega zdravnika ali specialista.

Preden preidemo iz prve do zadnje točke, lahko po nepotrebnem preteče precej dragocenega časa. Po odločitvi, da bomo o našem bolezenskem stanju ukrepali in pričnemo z raziskovanjem informacij na svetovnem spletu, se naenkrat znajdemo v poplavi vseh možnih informacij, podatkov, stanj, slik in nasvetov ter občasno grozljivih izkušenj uporabnikov, opisanih v spletnih skupnostih. Pri tem naletimo na nove težave: kako najti informacije, ki so značilne in relevantne za naš primer, ter ali so podane informacije zaupanja vredne.

Ne glede na informacije, ki nam jih ponuja svetovni splet, lahko obiščemo družinskega zdravnika oz. specialista, če nam to dopuščajo okoliščine. Namreč, Svetovna zdravstvena organizacija (WHO) in Združeni narodi (UN) v poročilu [2] navajata, da si trenutno polovica svetovnega prebivalstva ne more privoščiti osnovnih zdravstvenih storitev, do leta 2030 pa naj bi po podatkih World Bank Group v svetu imeli primanjkljaj 20 milijonov usposobljenih zdravstvenih delavcev. [3]

Primanjkljaj usposobljenih zdravstvenih delavcev lahko občutimo že v zahodnih oz. razvitih državah, v posameznih državah v razvoju pa je problem pomanjkanja usposobljenih zdravstvenih delavcev in z zdravstvom povezanih storitev že presegel kritično raven.

Dostop do zdravstvenih posvetovanj je del temeljne človekove pravice do zdravja, zato so v podjetju Your.MD leta 2012 pričeli z razvojem osebnega zdravstvenega pogovornega asistenta (ang. Personal Health Assistant) s podporo obogatene inteligence in ga leta 2014 predstavili javnosti. Do danes je bilo zabeleženih več kot dva milijona prenosov aplikacij z mobilnih platform in pametnih naprav.

3 DELOVANJE HIBRIDNEGA MODELA OSEBNEGA ZDRAVSTVENEGA ASISTENTA

Delovanje hibridnega modela osebnega zdravstvenega asistenta podjetja Your.MD lahko v grobem razdelimo na tri ključne komponente:

1. Razumevanje uporabnika
2. Zmanjševanje nabora možnih izidov
3. Podajanje nadaljnjih korakov

3.1 Razumevanje uporabnika

Medtem ko je razumevanje uporabnika in njegovega besedilnega vnosa razmeroma enostavna naloga za človeške možgane, še vedno predstavlja velik izziv za računalnike in programsko opremo. Ključna je izbira ustreznega pristopa, torej kako način razmišljanja in znanje zdravnikov prenesti v računalniku razumljivo obliko ter ga obenem nadgraditi s pomočjo algoritmov in upoštevanjem vhodnih parametrov za iskanje optimalne rešitve.

Proces strojne obdelave naravnega jezika (NLP - Natural Language Processing) opravlja več medsebojno povezanih komponent, ki iz uporabnikovega besednega vnosa najprej izluščijo namen oz. ga umestijo v eno izmed sledečih kategorij:

- Zahteva po posvetovanju (consultation)
- Zahteva po dodatnih informacijah (article search)
- Zahteva po besednem vprašalniku, samoocenjevalnem testu ali preverjanju znanja v obliki kviza (self assessment, test, quiz)
- Zahteva po specifičnih rešitvah v okviru mreže zunanjih ponudnikov storitev (OneStop Health)

Tako se npr. uporabnikova poizvedba "glavobol" oz. "I have a headache" obravnava kot zahteva po posvetovanju in sproži nize podvprašanj glede dodatnih simptomov, tako kot bi potekalo posvetovanje pri osebnem zdravniku. Po drugi strani se besedni vnosi tipa "informacije o glavobolu" oz. "info about headache" in "tell me about headache" obravnavajo kot uporabnikova zahteva po dodatnih informacijah, kjer bo sistem vrnil enega ali več ustreznih zdravstvenih člankov. Procesi odločanja potekajo tako po predefiniranih pravilih (ang. rule-based system) kot s pomočjo strojnega učenja (ang. machine learning) in modelov za klasifikacijo besedila.

Hibridni model chatbota pri umeščanju v kategorije in nadaljnjih korakih pri postopku posvetovanja poleg zgoraj navedenega upošteva še dejavnike kot so:

- Neuporaba knjižnega jezika oz. uporaba pogovornega jezika, kratic in okrajšav
- Slovnične in pravopisne napake
- Uporaba tujega jezika oz. prevajanje uporabniških vnosov v angleščino (trenutno na voljo v verziji za razvijalce)

V postopku razumevanja oz. prenosa informacij uporabljajo UMLS - Unified Medical Language System, kjer iz uporabnikovega prosto vnešenega besedila izluščimo CUI - Concept Unique Identifier. Tako npr. vnos "*Pollakisuria*" neposredno sovpada s CUI C0042023, enako velja za sopomenko *"frequent urination"*. Ker je od končnih uporabnikov nerealno pričakovati, da bodo uporabljali medicinske izraze, se strojno procesiranje jezika v resnici začne pri vnosih kot so *"urinating frequently"*, *"need to urinate frequently"*, *"need to frequently urinate"* ipd. Zaplete se v primerih, ko uporabnik izraza *"urination"* in *"frequent"* nadomesti s sopomenkami ali slengom oz. v primeru tipkarskih napak:

- *urinating often / urinating all the time / need to pee frequently*
- *need to urinate a lot / have to pee frequently / going to the toilet too often*
- *I need to drain the lizard 10 times a day / I take a whiz all the time*

V primerih uporabe sopomenk za pogostost mora sistem upoštevati še dodatne dejavnike in izračunati, kdaj gre za simptom prepogostega uriniranja in kdaj ne. Uporabnikov vnos z vključenimi pravopisnimi napakami *"I nid to take a leek every hour"* bo v tem primeru pravilno sovpadal s CUI C0042023, medtem ko zmanjšana

pogostost npr. *"I need to take a leak every 2 hours"* ali več ne sproži simptoma za pogosto uriniranje. Kot da primer ne bi bil dovolj zapleten, veljajo drugačna pravila, kadar je pogosto uriniranje omenjeno v kombinaciji s spanjem oz. v času noči, saj gre takrat za drug dejavnik pogostosti in drug simptom (pogosto uriniranje v nočnem času oz. angl. *Nocturia*, CUI C0028734).

3.2 Zmanjševanje nabora možnih izidov

Kadar je uporabnikov namen opredeljen kot posvetovanje oz. zahteva po informacijah, je potrebno množico možnih rezultatov zmanjšati oz. reducirati na nekaj izbranih rezultatov. V primeru posvetovanja med drugim uporabljamo Bayesovo statistiko (ang. Bayesian logic oz. Bayesian probability), kjer na podlagi medicinske oz. zdravstvene literature in podatkov določimo verjetnost posameznemu simptomu kot indikator za določeno zdravstveno stanje.

V postopku odločanja sistem zastavlja podvprašanja na način, da vsako naslednje vprašanje s podanimi dodatnimi simptomi v vsakem koraku kar najbolj zmanjšajo nabor možnih rezultatov. V postopku zmanjševanja oz. redukcije algoritem upošteva sledeče dejavnike:

- Razumevanje frekvence določenega pojava glede na izbrane vhodne podatke
- Upoštevanje uporabnikovega okolja: razumevanje pogostosti pojavov glede na sezonskost oz. letni čas, uporabnikovo regijo in pogostost iskanj v določenem obdobju oz. lokaciji
- Upoštevanje uporabnikovih dejavnikov (ang. influencing factors), npr. kajenje, prekomerna telesna teža, stres, ipd.
- Nadaljna in izločevalna vprašanja (ang. follow up questions)
- Odzivanje na nujne primere oz. potencialno ogrožajoče simptome (ang. red flag symptoms)
- Odzivanje v primerih, ko podani simptomi ne nakazujejo na zdravstveno stanje, ki bi potrebovalo nadaljnjo obravnavo

Pri tem je potrebno poudariti, da zgoraj opisani postopek redukcije deluje na principu upoštevanja značilnosti povprečnega pripadnika določene starostne skupine in spola.

3.3 Podajanje nadaljnjih korakov

Na podlagi zbranih podatkov z upoštevanjem lastnosti, ki veljajo za povprečnega pripadnika starostne skupine oz. spola, aplikacija pripravi poročilo (ang. consultation report), kjer predlaga naslednje korake, npr. specifične informacije o nadaljnjih korakih za določena zdravstvena stanja ali predlog za obisk zdravnika. Možnih rezultatov je lahko več in so razvrščeni glede na stopnjo verjetnosti.

Medtem ko aplikacija na podlagi vhodnih podatkov lahko prepozna možnost zdravstvenih stanj, ki ne potrebujejo strokovne obravnave, naletimo na omejitve tako pri kroničnih in redkih bolezenskih stanjih, kot pri zdravstvenih stanjih, ki potrebujejo nujno medicinsko pomoč. V tovrstnih primerih uporabnika napotimo k obisku zdravnika. Povedano drugače, tudi če obstaja velika verjetnost, da gre na podlagi vnesenih podatkov pri uporabniku za srčni zastoj ali pojav raka, bo aplikacija v tem primeru namesto specifičnega rezultata svetovala čim hitrejši obisk zdravnika.

Zaupanje in transparentnost sta kot eni izmed temeljnih vrednot podjetja vodilo tudi pri podajanju zdravstvenih informacij končnim uporabnikom, kjer imamo za vsebine vključno z zdravstvenimi članki licenco s strani britanske državne zdravstvene službe oz. National Health Service (NHS). Pri tem so zdravstvene informacije v člankih podane s strani zdravnikov oz. medicinskega osebja za zagotovitev kakovosti in uporabnikom čim bolj razumljivo in prijazno obliko.

Podajanje nadaljnjih korakov zaključuje mreža zunanjih ponudnikov zdravstvenih storitev pod imenom OneStop Health™, kjer so uporabnikom na voljo izbrani lokalni in globalni ponudniki zdravstvenih storitev in izdelkov: od posvetovanja z zdravnikom v živo oz. preko video klica, do možnosti naročanja krvnih testov na dom.

4 ZAKLJUČEK

Zaradi splošne namembnosti spletnih iskalnikov se iskanje informacij prične s ugotavljanjem relevantnega področja glede na uporabnikov iskalni niz, pri čemer iskalnik ne upošteva uporabnikovih bistvenih značilnosti. Tako lahko uporaba iskalnega niza s pogovorno frazo »I need to take a leak every hour« privede do iskalnih rezultatov s področja vodovodnih popravil, medtem ko uporaba domensko specifičnih storitev, kot je osebni zdravstveni asistent Your.MD, uporabnika že avtomatsko umesti v področje zdravstva – analogni ekvivalent tega predstavlja obisk zdravnika ali zdravstvene ustanove, kjer je ista uporabnikova fraza obravnavana v pravilnem kontekstu in rezultira v nadaljnji obravnavi s podvprašanji.

Področje zdravja in ugotavljanja možnih zdravstvenih stanj je preveč zapleteno, da bi znali učinkovito prepoznati problem na podlagi zgolj enega iskalnega niza oz. vprašanja, zlasti ob predpostavki, da gre pri posameznikovem telesu za skupek edinstvenih značilnosti oz. zdravstvenih težav. Z razvojem visokotehnološkega osebnega zdravstvenega asistenta se podjetje Your.MD sooča z vrsto izzivov, kako uporabnikom po vsem svetu omogočiti dostop do kakovostnih in zanesljivih zdravstvenih informacij s poudarkom na osebnem pristopu in preventivi v okviru predzdravstvene nege.

5 VIRI IN LITERATURA

[1] N. I. o. H. T. J. L. B. P. A. National Cancer Institute, „J Gen Intern Med,“ *Why do people avoid medical care? A qualitative study using national data.,* p. 290–297, 30 Marec 2015.
[2] Y. G. A. M. T. B. R. S. Jenny X. Liu, „Human Resources for Health,“ *Global Health Workforce Labor Market Projections for 2030.,* 3 Februar 2017.
[3] K. K. www.reuters.com, “Half of world's people can't get basic health services: WHO,” London, 2017, obiskano 16. 5. 2019

OTS 2019
SODOBNE INFORMACIJSKE TEHNOLOGIJE IN STORITVE
M. Heričko in K. Kous (ur.)

Univerzitetna založba
Univerze v Mariboru

PASTI PRI IMPLEMENTACIJI REŠITEV UMETNE INTELIGENCE

VILI PODGORELEC, SAŠO KARAKATIČ

Povzetek: *Razvoj informacijskih tehnologij, rešitev in storitev je pripeljal do obdobja, v katerem ljudje ustvarjamo, delimo in si izmenjujemo čedalje večje količine podatkov. Tovrsten razvoj prinaša uporabnikom vedno nove funkcionalnosti, za informatike – snovalce in razvijalce novih sistemov in rešitev – pa vse kompleksnejše zahteve in izzive. Sočasno z razvojem tehnologij se pogosto spreminjajo tudi koncepti uporabe le-teh, pri čemer vse bolj raste zavedanje o pomenu ustrezne analize obilice ustvarjenih podatkov in želja po uporabi znanja, skritega v njih. Vse bolj se zavedajoč potenciala umetne inteligence in strojnega učenja, kot dostopnega orodja za spopadanje z naštetimi izzivi, je v zadnjih letih strojno učenje postalo eden od temeljnih elementov informacijske tehnologije. Uporabniki so ozavestili inteligentno obnašanje naprav ter storitev in to od ponudnikov v vedno večji meri tudi zahtevajo. Podjetja so tako primorana nadgrajevati ter razvijati produkte in rešitve v smeri integracije z metodami in tehnologijami strojnega učenja oziroma s pomočjo umetne inteligence. Kljub enormnemu napredku na področju strojnega učenja pa je slednje še vedno zahtevna naloga, ki zahteva obilico domenskega znanja in izkušenj. Izhajajoč iz dveh desetletij praktičnih izkušenj, bomo v prispevku predstavili pogoste pasti, v katere se vse prehitro ujamejo premalo izkušeni razvijalci, ki pod vplivom pogosto prevelikih pričakovanj in nerealnih zahtev vodstva vidijo metode umetne inteligence kot novo »silver bullet« tehnologijo.*

Ključne besede: *umetna inteligenca, strojno učenje, podatkovna znanost, razvoj, pogoste pasti*

NASLOVA AVTORJEV: dr. Vili Podgorelec, redni profesor, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Maribor, Slovenija, e-pošta: vili.podgorelec@um.si. dr. Sašo Karakatič, docent, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Maribor, Slovenija, e-pošta: saso.karakatic@um.si.

https://doi.org/10.18690/978-961-286-282-4.2 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 UVOD

Razvoj informacijskih tehnologij je pripeljal do obdobja, v katerem ustvarjamo, delimo in izmenjujemo čedalje večje količine podatkov, tako v službenem kot zasebnem življenju. Napredek podatkovne analitike, inteligentne analize podatkov in vpeljave strojnega učenja v ta področja je ustvaril prepričljive ekonomske rezultate na različnih poslovnih področjih, kot so finance, trženje in logistika [1]. Medtem ko so vodilna podjetja že prejela znatne donose od naložb v IT na teh področjih, pa se mnogi šele spoznavajo s prednostmi naprednih rešitev umetne inteligence in podatkovne znanosti v svojih podjetjih in domenah. Zavedajoč se pomena avtomatiziranih, naprednih rešitev, zasnovanih na odkrivanju vzorcev oz. znanja, skritega v obilici ustvarjenih podatkov, so se odločevalci znašli pred zahtevnim izzivom – kako zasnovati, razviti in vpeljati rešitve umetne inteligence in strojnega učenja za uspešno analizo množice prepletenih podatkov in iz nje izluščiti relevantne informacije za gradnjo napovednih modelov. Ker količina razpoložljivih podatkov presega zmožnosti računalniških sistemov za njihovo učinkovito analizo z uporabo običajnih pristopov in metod, je postalo področje napredne podatkovne analitike ključno za spopadanje s tem izzivom.

V zadnjih letih smo tako priča razcvetu tehnologij, pristopov, metod in orodij na področju strojnega učenja oziroma umetne inteligence, ki igrajo ključno vlogo v širokem spektru aplikacij, kot so podatkovno rudarjenje, obdelava naravnega jezika, prepoznava slik in podobno. Tovrsten razvoj prinaša uporabnikom vedno nove funkcionalnosti, za informatike – snovalce in razvijalce novih sistemov in rešitev – pa vse kompleksnejše zahteve in izzive. Sočasno z razvojem tehnologij se pogosto spreminjajo tudi koncepti uporabe le-teh, pri čemer vse bolj raste zavedanje o pomenu ustrezne analize obilice ustvarjenih podatkov in želja po uporabi znanja, skritega v njih. Vse bolj se zavedajoč potenciala umetne inteligence in strojnega učenja, kot dostopnega orodja za spopadanje z naštetimi izzivi, je v zadnjih letih strojno učenje postalo eden od temeljnih elementov informacijske tehnologije in ključna sestavina za tehnološki napredek [2]. Uporabniki so ozavestili inteligentno obnašanje naprav ter storitev in to od ponudnikov v vedno večji meri tudi zahtevajo. Podjetja so tako primorana nadgrajevati ter razvijati produkte in rešitve v smeri integracije z metodami in tehnologijami strojnega učenja oziroma s pomočjo umetne inteligence.

V želji nadoknaditi zaostanek za konkurenti, ki so se vpeljave inteligentnih rešitev že lotili, ali morebiti že žanjejo prve ugodne rezultate tovrstnega truda, v vodstvu podjetij pritiskajo na svoje skupine razvijalcev z vse večjimi zahtevami po hitri, ekonomični in kakovosti vpeljavi rešitev umetne inteligence. A kljub enormnemu napredku na področju strojnega učenja je slednje še vedno zelo zahtevna naloga, ki zahteva obilico specializiranih strokovnih znanj in izkušenj. Kot nas je zgodovina naučila že na mnogih drugih področjih, se le z ustreznim upoštevanjem dobrih inženirskih praks, podprtih s strokovnim znanjem in pridobljenimi izkušnjami, lahko izognemo pastem, v katere se vse prehitro ujamejo premalo izkušeni razvijalci, ki pod vplivom pogosto prevelikih pričakovanj in nerealnih zahtev vodstva vidijo metode umetne inteligence kot novo »*silver bullet*« tehnologijo [3].

Izhajajoč iz dveh desetletij praktičnih izkušenj, bomo v članku tako predstavili nekatere najpogostejše pasti, v katere se zlahka ujamejo razvijalci pri implementaciji rešitev umetne inteligence. V grobem lahko pasti in izzive razdelimo v tri temeljne skupine:

1. Med **strateške pasti** štejemo vse morebitne neustrezne odločitve o razvoju in vpeljavi rešitev umetne inteligence v proces digitalne preobrazbe podjetja na najvišjem nivoju in so običajno posledica pomanjkljive poslovne vizije, napačnih poslovnih odločitev in/ali neustreznih pričakovanj. Za strateške napake praviloma niso odgovorni sami razvijalci, mora pa vodja razvoja nanje opozarjati vodstvo podjetja.
2. Kot **taktične pasti** razumemo predvsem morebitne neustrezne izbire in odločitve pri načrtovanju izvedbe zastavljenega strateškega cilja razvoja in vpeljave rešitev umetne inteligence. Za prepoznavanje in izogibanje taktičnim pastem skrbi predvsem vodja razvoja, nanje pa morajo opozarjati tudi sami razvijalci.
3. **Izvedbene pasti** so vse tiste, v katere se zlahka ujamemo pri sami implementaciji inteligentnih rešitev. Čeprav imajo težave, ki so posledica ujetja v izvedbene pasti, praviloma najbolj omejene posledice in si jih da odpraviti z manj vloženega truda kot taktične ali celo strateške napake, pa lahko vendarle tudi

te v skrajni obliki pripeljejo do nedelujoče rešitve. Izvedbenim pastem se morajo predvsem biti sposobni izogniti sami razvijalci.

2 UMETNA INTELIGENCA IN STROJNO UČENJE

Preden se poglobimo v tematiko, je pomembno, da dobro razumemo, kaj sploh je umetna inteligenca. V vsakdanji praksi so namreč v uporabi mnogi različni izrazi, ki se pogosto uporabljajo kot sinonimi. Umetno inteligenco lahko definiramo kot:

»Umetna inteligenca je področje računalništva in informatike, namenjeno reševanju problemov, ki sicer zahtevajo človeško inteligenco – na primer prepoznavanje vzorcev, učenje in posploševanje.«

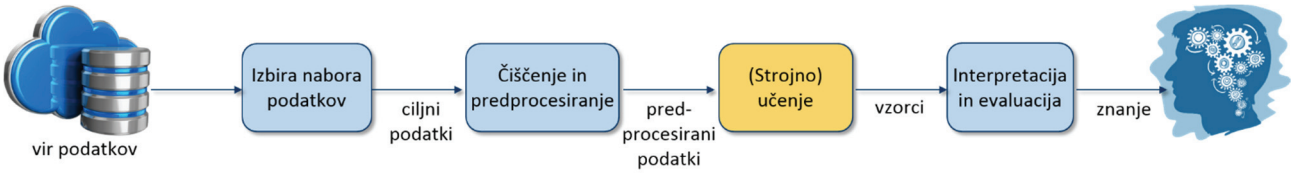
Izraz je v zadnjih letih prekomerno uporabljen za označevanje umetne splošne inteligence (*artificial general intelligence,* AGI), ki se nanaša na samo-zavedajoče se računalniške sisteme, ki so sposobni pravih kognitivnih funkcij. Kljub temu bo večina sistemov umetne inteligence v bližnji prihodnosti dejansko tisto, kar znanstveniki imenujejo kot "ozko umetno inteligenco" – kar pomeni, da bodo oblikovani tako, da bodo zelo dobro opravljali posamezno, precej ozko omejeno nalogo spoznavanja, namesto da bi resnično "razmišljali" sami. Za tovrstne namene pa se najpogosteje uporabljajo pristopi in tehnike strojnega učenja, ki ga lahko definiramo kot:

»Strojno učenje je podmnožica umetne inteligence, ki uporablja statistične in hevristične tehnike, da računalnikom omogoči, da se učijo iz podatkov, ne da bi bili za to izrecno programirani.«

Izraza umetna inteligenca in strojno učenje sta se v zadnjih letih izmenjaje uporabljala za opis praktičnih rešitev, zasnovanih na uporabi umetne inteligence, v mnogih podjetjih, predvsem zaradi velikega uspeha metod strojnega učenja. V strogem pomenu pa se moramo zavedati, da strojno učenje pomeni sposobnost samodejnega učenja iz podatkov, medtem ko umetna inteligenca vključuje učenje skupaj še z nekaterimi drugimi funkcijami.

2.1 Proces strojnega učenja

Strojno učenje je namenjeno odkrivanju uporabnega znanja v surovih podatkih. Celoten proces strojnega učenja je netrivialen proces identifikacije veljavnih, novih, potencialno uporabnih in popolnoma razumljivih vzorcev v podatkih (slika 1). V tem kontekstu je vzorec ekvivalenten modelu, ki podaja relacije (strukturo) v podatkih. Vzorci so veljavni, če z določeno stopnjo gotovosti držijo za dane podatke. Razumljivost sme biti zagotovljena tudi z naknadnim procesiranjem, sami koraki procesa pa se običajno ponavljajo v več iteracijah, skozi katere na osnovi rezultatov predhodnih iteracij postopoma izboljšujemo oz. optimiziramo proces.



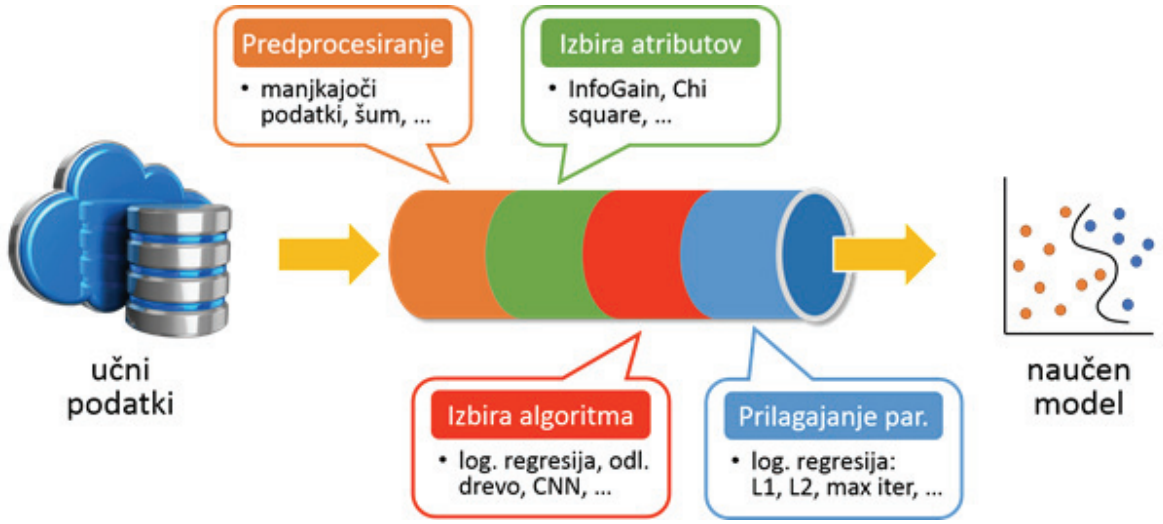
Slika 1: Proces strojnega učenja oz. odkrivanja znanja v podatkih.

Koraki procesa strojnega učenja so naslednji:

- Proces strojnega učenja zmeraj začnemo s fazo analize oz. spoznavanjem problemske/aplikacijske domene, ki obvezno vključuje določanje ciljev, ki bi jih radi z izvedbo procesa dosegli.
- Sledi oblikovanje ciljnega nabora podatkov. Obseg in kakovost podatkov sta ključna za vsak proces strojnega učenja, zato je le-temu potrebno nameniti dovolj pozornosti. Kakovostni podatki sami po sebi seveda še ne zagotavljajo uspešne inteligentne rešitve, dejstvo pa je, da smo brez kakovostnih podatkov gotovo obsojeni na neuspeh. Rezultat tega koraka je tako izbira podmnožice zapisov in spremenljivk, iz katerih bo izpeljano znanje.

- Naslednji korak je čiščenje in pred-procesiranje podatkov, k čemer med drugim sodi odstranjevanje šuma, obravnava manjkajočih polj, nepoznanih vrednosti in sekvenčnih informacij. Med pred-procesiranje štejemo še redukcijo in projekcijo podatkov, ki zajema iskanje alternativnih predstavitev podatkov ter izvajanje transformacij za zmanjševanje dimenzionalnosti in števila efektivnih spremenljivk.
- Sledi določitev funkcije strojnega učenja oz. inteligentne analize podatkov, pri čemer določimo namen izpeljanega modela (na primer: klasifikacija, regresija, povezovalna analiza, ...).
- Glede na izkazane potrebe, izbrane podatke in namen rešitve nato izberemo sam algoritem strojnega učenja glede na postavljene cilje in odločevalčeve potrebe. Za uspešen rezultat moramo nastaviti še zagonske parametre izbranega algoritma, saj lahko uglaševanje parametrov algoritma bistveno vpliva na rezultat. Sledi še dejanska izvedba izbranega algoritma z namenom odkritja relevantnih vzorcev v podatkih.
- Za poln izkoristek zgrajenih modelov znanja po izvedbi algoritma strojnega učenja sledi faza interpretacije, v kateri najprej odstranimo redundantne in nepomembne vzorce ter pretvorimo uporabne izpeljane vzorce v razumljivo predstavitev, obogateno z ustrezno vizualizacijo.
- Zadnja faza je ovrednotenje zgrajenih napovednih modelov. Pravilno ovrednotenje poiskanega znanja je ključno za uspešno uporabo rešitev umetne inteligence, saj nam rezultati vrednotenja povedo, kakšno stopnjo uspešnosti lahko pričakujemo od zgrajene rešitve pri dejanski uporabi v produkcijskem okolju.

Medtem, ko lahko v mnogih aktivnostih procesa strojnega učenja sodelujejo strokovnjaki, ki nimajo poglobljenih strokovnih znanj prav s področja umetne inteligence (poslovni analitiki, strokovnjaki za upravljanje podatkov, …), pa so predvsem za jedro procesa – optimizacijo cevovoda strojnega učenja (slika 2) – ključna specifična znanja, pomanjkanje katerih večinoma botruje neuspešnim rešitvam.



Slika 2: Standardni cevovod strojnega učenja. Od izbrane platforme, orodja oz. knjižnice je odvisno, katere korake je mogoče nadzorovati in do kakšne mere prilagajati svojim potrebam [4].

2.2 Zakaj je strojno učenje tako pomembno?

Obstaja več razlogov, zakaj je strojno učenje in na njem temelječe rešitve dandanes tako pomembno. Med ključne sodijo:

- razvoj informacijskih tehnologij za zajem, obdelavo, shranjevanje in upravljanje podatkov,
- razvoj zmogljivosti računalniških sistemov,
- izpopolnjeni algoritmi, in predvsem
- (poslovne) prednosti uporabe strojnega učenja.

Razvoj podatkovnih tehnologij je omogočil, da lahko danes v podjetjih brez večjih težav ohranijo prav vse generirane in zajete podatke. Izkaže se namreč, da lahko podatki, ki trenutno nimajo neposredne uporabne vrednosti, pripomorejo k sprejemanju ključnih odločitev v prihodnosti. S sistemi kot je npr. Hadoop je postalo shranjevanje in obdelava ogromnih količin podatkov zelo preprosto. Na osnovi teh ogromnih količin podatkov lahko algoritmi strojnega učenja nato zelo točno napovedujejo prihodnje rezultate. Podobno kot podatkovne tehnologije, eksponentno narašča tudi računska zmogljivost sodobnih računalniških sistemov. S prehodom na uporabo storitev v oblaku pa lahko tudi manjša podjetja uporabljajo zmogljivo infrastrukturo [5].

Sočasno z razvojem tehnologije pa se razvijajo in napredujejo tudi algoritmi, metode in orodja za inteligentno obdelavo ogromnih količin podatkov. Glavna naloga teh algoritmov je iskanje različnih vrst vzorcev v podatkih, njihovo analiziranje in dajanje pomembnih napotkov deležnikom za sprejemanje ustreznih odločitev v krajšem času. Pomagajo tudi pri zmanjševanju stroškov, ki nastanejo pri sprejemanju teh odločitev. Z zlivanjem fragmentiranih podatkov v enoten vir lahko bistveno povečamo uspešnost prihodnjih rezultatov.

3 PASTI, KI PREŽIJO NA RAZVIJALCE

Glavni razlog za vpeljavo rešitev umetne inteligence v podjetjih je potreba po napredovanju celotne digitalne transformacije. Z vpeljavo inteligentne programske opreme in storitev lahko namreč:

- podjetja izboljšajo zmožnosti izdelkov in kakovost storitev,
- bolje komunicirajo s strankami,
- racionalizirajo poslovanje, ter
- ustvarjajo napovedne in natančne poslovne strategije.

Glede na potencialne prednosti, ki jih omogočajo rešitve umetne inteligence za podjetja, je jasno, da si le-ta želijo razviti in vpeljati tovrstne rešitve. Vsak poslovni proces lahko z združevanjem podatkov pridobi, saj imajo različni oddelki večinoma opravka le s svojimi podatki. Ko se ti podatki združijo na ustrezen način in v ustreznem času, se lahko sprejmejo primerne odločitve, ki pomagajo podjetju rasti in napredovati. S strojnim učenjem lahko iz ogromnih količin podatkov izluščimo informacije z zanemarljivim posredovanjem ljudi. V idealnem primeru naj bi izvajanje tega koncepta prispevalo k eksponentni rasti, v resnici pa se lahko ujamemo v posamezne pasti, ki lahko rast izničijo. Strojno učenje daje organizacijam možnost, da sprejmejo natančnejše odločitve, ki jih usmerjajo podatki, in rešujejo probleme, ki jih tradicionalni pristopi ne zmorejo. Vendar strojno učenje ni magija. Predstavlja veliko podobnih izzivov kot druge tehnologije.

3.1. Strateške pasti strojnega učenja

Med strateške pasti štejemo vse morebitne neustrezne odločitve o razvoju in vpeljavi rešitev umetne inteligence v proces digitalne preobrazbe podjetja na najvišjem nivoju in so običajno posledica pomanjkljive poslovne vizije, napačnih poslovnih odločitev in/ali neustreznih pričakovanj. Za strateške napake praviloma niso odgovorni sami razvijalci, mora pa vodja razvoja nanje opozarjati vodstvo podjetja.

(Ne)podpora vodstva

Razvoj in vpeljava rešitev umetne inteligence v poslovni proces kateregakoli podjetja za sabo neločljivo potegne tudi spremembo načina vedenja in delovanja. Ena glavnih vlog vodstva, poleg tega, da že s sprejemom strategije pokaže svojo zavezanost tovrstni vpeljavi, je podpora in spodbujanje razvoja. Ob tem se morajo posamezni vodje, oboroženi s spoznanji in vpogledi iz zgrajenih modelov znanja, naučiti sprejemati več odločitev sami, pri čemer najvišje vodstvo določa le glavne usmeritve in se po potrebi vmeša le, ko se pojavijo izjeme. Za demokratizacijo uporabe analitike, ki bo zagotovila napredovanje s potrebnimi veščinami in spodbujanjem k souporabi podatkov, je seveda potreben določen čas.

3.1.1 *Pomanjkanje (dobrih) kadrov*

Pomanjkanje strokovnjakov strojnega učenja je še vedno očiten izziv in potreba po zaposlenih, ki lahko vodijo in ključno prispevajo k razvoju in vpeljavi inteligentnih rešitev, je vse večja. Zaposlovanje in obdržanje teh iskanih strokovnjakov je postalo pomemben poudarek za številne uspešne organizacije. Podatkovni znanstveniki in inženirji strojnega učenja potrebujejo edinstveno mešanico talenta in znanj s področij

računalništva in informatike, matematike in statistike ter ustreznega domenskega znanja. Prepričanje, da je razvoj rešitev umetne inteligence pač še eno izmed številnih področij in pričakovanje, da ga bodo obstoječi programski inženirji uspešno rešili tudi brez specialnih znanj, je ena najhujših napak, ki jih lahko vodstvo stori. Če se hočemo temu problemu izogniti, je nujno v razvojno ekipo uvrstiti ustrezno usposobljene strokovnjake, ki morajo biti poleg obvladovanja tehnične plati sposobni razumeti še poslovne razloge za razvoj.

3.1.2 *Pomanjkanje (dobrih) podatkov*

Čeprav se na strojno učenje praviloma gleda kot na postopek izboljševanja algoritmov, pa je grda resnica, da se večina časa porabi za pripravo podatkov in zagotavljanje njihove kakovosti. Kakovost podatkov je namreč bistvena za pridobivanje točnih rezultatov iz zgrajenih napovednih modelov. Z vidika kakovosti podatkov so predvsem problematični naslednji vidiki:

- Šum v podatkih – podatki vsebujejo veliko nasprotujočih si ali zavajajočih informacij.
- Umazani podatki – vsebujejo manjkajoče vrednosti, kategorične atribute z mnogimi ravnmi ter neskladne in napačne vrednosti.
- Redki podatki – vsebujejo zelo malo dejanskih vrednosti in so sestavljeni večinoma iz ničel ali manjkajočih vrednosti.
- Neustrezni podatki – nepopolni ali pristranski podatki.

Na žalost se lahko veliko napak prikrade v samih postopkih zbiranja in shranjevanja podatkov, vendar je potrebno za uspešno aplikacijo strojnega učenja te težave odpraviti ali jih vsaj čim bolj omejiti. Podjetja, ki imajo zgledno urejen sistem upravljanja s svojimi podatki, bodo tako imela ob vpeljavi rešitev strojnega učenja in podatkovne analitike bistveno manj težav.

3.1.3 *Nezadostna računalniška infrastruktura*

Za mnogo organizacij lahko upravljanje različnih vidikov infrastrukture, ki obdaja dejavnosti strojnega učenja, postane samo po sebi izziv. Že samo preverjeni in zanesljivi sistemi za upravljanje relacijskih podatkovnih baz lahko popolnoma odpovejo pod obremenitvijo in raznolikostjo podatkov, ki jih organizacije želijo zbrati in analizirati. Potem pa so tu pogosto še zahteve po izjemni računski moči, ki jih zahtevajo sodobni algoritmi strojnega učenja, ko se spopadajo z ogromnimi količinami podatkov.

Ker si mnoga, sploh manjša podjetja, ne morejo privoščiti niti nakupa, še manj pa vzdrževanja tovrstne infrastrukture, so v skladu s premikom paradigme v načinu gradnje tehnoloških rešitev v oblaku in z rastjo oblačnih storitev začeli mnogi veliki ponudniki ponujati možnost uporabe strojnega učenja kot storitve (MLaaS, *machine learning as-a-service*). Tovrstne platforme v računalniškem oblaku zagotavljajo osnovni zahtevi za izvajanje sistema umetne inteligence:

- učinkovite, stroškovno sprejemljive vire (v glavnem hramba podatkov), in
- računsko moč (zmožnost obdelave velikih količin podatkov).

3.1.4 *Prezgodnji razvoj oz. razvoj brez plana*

Vsako podjetje leta razvija svoje poslovne procese. Odločitev, kdaj naj se nove, praviloma naprednejše in celovitejše tehnologije vključijo v celovito strategijo podjetja, je težka naloga. Prehod na tehnike strojnega učenja morda niti ne bo potreben, dokler se IT in poslovne potrebe ustrezno ne razvijejo. Vsekakor je potrebno v organizaciji pred kakršnimkoli začetkom razvoja rešitev umetne inteligence jasno določiti strategijo in cilje, ki naj bi jih tovrstne rešitve prinesle. Nato je potrebno preveriti, ali je podjetje sploh sposobno razviti in vpeljati takšne rešitve ter ali so izpolnjeni vsi pogoji za tovrsten razvoj (kadri, infrastruktura, podatki, …)

3.2 Taktične pasti strojnega učenja

Kot taktične pasti razumemo predvsem morebitne neustrezne izbire in odločitve pri načrtovanju izvedbe zastavljenega strateškega cilja razvoja in vpeljave rešitev umetne inteligence. Za prepoznavanje in izogibanje taktičnim pastem skrbi predvsem vodja razvoja, nanje pa morajo opozarjati tudi sami razvijalci.

3.2.1 *Izbira načina razvoja in orodij za razvoj*

Včasih so algoritme in metode strojnega učenja večinoma uporabljali znanstveniki, tehnološki strokovnjaki ali domenski strokovnjaki. Danes je na voljo vse več storitev strojnega učenja MLaaS, ki so vse bolj dostopne širšemu spektru razvijalcev in raziskovalcev. Podobno, kakor tradicionalne spletne storitve pomagajo razvijalcem ustvariti aplikacije, tako tudi MLaaS omogočajo enostavno uporabo strojnega učenja povprečnemu razvijalcu. Prikrijejo kompleksnost pri ustvarjanju in uporabi modelov strojnega učenja, tako da se lahko razvijalci osredotočijo na pripravo podatkov, uporabniško izkušnjo, oblikovanje, eksperimentiranje in uporabo znanja ter odkrivanje vzorcev v podatkih. Za zmanjšanje kompleksnosti platforme MLaaS zmanjšujejo stopnjo nadzora, ki jo ima razvijalec nad posameznimi koraki v procesu strojnega učenja, posamezne platforme pa se med seboj precej razlikujejo. Izbira, ali bomo za razvoj uporabili posamezno specializirano platformo (in katero) ali bomo raje razvijali rešitve zgolj z uporabo osnovnih knjižnic in ogrodij, ni enostavna, lahko pa odločujoče vpliva na uspešnost razvite rešitve [4]. Pri tem je pomembno, da ne precenimo sposobnosti svojih razvijalcev.

3.2.2 *Delovanje algoritmov po principu črne škatle*

Nekateri algoritmi strojnega učenja zgradijo napovedne modele, ki so človeku povsem nerazumljivi, kar pomeni, da iz modelov ni razviden postopek odločanja. Ti algoritmi ali sistemi ne omogočajo vpogleda v notranje delovanje ali logiko za njimi. Tovrstne rešitve lahko hitro ustvarijo tehnične in kulturne težave. Če tak pristop ne doseže željenih rezultatov, ko se podatki precej spreminjajo, je lahko sam sistem zaradi nerazumevanja hitro ogrožen. Zelo težko je razložiti, zakaj model ni uspešen, kar lahko bistveno upočasni rast organizacije.

V nekaterih panogah, kot so npr. medicina, bančništvo ali zavarovalništvo, modeli znanja enostavno morajo biti razložljivi. V nekaterih reguliranih panogah razlaga, dokumentiranje in utemeljevanje kompleksnih modelov strojnega učenja tako predstavljajo še dodatno breme.

3.2.3 *Ustvarjanje tehničnega dolga*

Tehnični dolg se v okviru razvoja programske opreme nanaša na situacije, v katerih se razvijalci odločajo za kodo, ki sicer deluje in se jo enostavno implementira v kratkem času, ni pa dolgoročno ustrezna z vidika vzdrževanja. Pri uporabi strojnega učenja je možnosti za ustvarjanje tehničnega dolga zelo veliko – prepletajoči se cevovodi, neupoštevana odvisnost med podatki, skrite povratne zanke, … Problemu tehničnega dolga se lahko izognemo z upoštevanjem inženirskega pristopa pri razvoju. Pomembno se je zavedati, da so podatkovni znanstveniki nagnjeni predvsem k izboljševanju rezultatov in manj h kakovosti same programske kode.

3.3 Izvedbene (operativne) pasti strojnega učenja

Izvedbene pasti so vse tiste, v katere se zlahka ujamemo pri sami implementaciji inteligentnih rešitev. Čeprav imajo težave, ki so posledica ujetja v izvedbene pasti, praviloma najbolj omejene posledice in si jih da odpraviti z manj vloženega truda kot taktične ali celo strateške napake, pa lahko vendarle tudi te v skrajni obliki pripeljejo do nedelujoče rešitve. Izvedbenim pastem se morajo predvsem biti sposobni izogniti sami razvijalci.

3.3.1 *Preskakovanje priprave podatkov*

Pred uporabo pristopov grajenja modelov znanja je podatke potrebno ustrezno pripraviti – predprocesirati. Tukaj ne mislimo le na primerno poimenovanje spremenljivk, vstavljanje manjkajočih podatkov in poenotenje zapisov – to je del čiščenja podatkov in ne predpriprave. Pri predpripravi podatkov pa imamo podatke že očiščene in združene v pravi obliki, vseeno pa jih je potrebno še do določene mere spremeniti. Poglejmo si primer kategorične (nominalne) vrednosti, ki jo lahko zakodiramo v podatke na več možnih načinov. Če imamo kategorično spremenljivko, ki lahko zavzema vrednosti štirih kategorij, je prva misel za kodiranje kategorij kar preprosto z zaporednim mestom kategorije (0-prva kategorija, 1-druga kategorija…), kar pa je napačen pristop. Če kategorije kodiramo kot zaporedje, bo algoritem predpostavljal, da obstaja vzorec v vrstnem redu določanja kodiranja. Če je temu res tako, nimamo opravka s kategorično spremenljivko, temveč z vrstnim redom. Če pa vrstni red ni pomemben, pa moramo tudi algoritmu to povedati, da ne bo sklepal na pomembnost vrstnega reda kodiranja. To najlažje naredimo tako, da iz ene kategorične spremenljivke naredimo N novih

slamnatih spremenljivk, kjer je N enak številu kategorij. Vsaka izmed teh slamnatih spremenljivk bo zavzela vrednost bodisi 0 (če primer ni tiste kategorije) ali vrednost 1 (če primer je tiste kategorije). S tem pristopom bo algoritem vsako slamnato spremenljivko obravnaval kot svojo neodvisno spremenljivko in ne bo zmotnega sklepanja o vrstnem redu kategorij.

Pogosto se spregleda tudi transformacija številskih spremenljivk – te je namreč potrebno spraviti na enako enoto. Zakaj? Predstavljajmo si primer, ko imamo v podatkih zapis o ceni pijače, ki ima razpon med 5 in 10€ in količini pijače, ki ima razpon med 330 in 1500ml. Nekateri algoritmi gradnje modelov znanja lahko sklepajo, da večje razlike v eni vrednosti, pomenijo tudi večje razlike v splošnem. Primer:

- Pijača 1: 330ml, 6€
- Pijača 2: 350ml, 10€

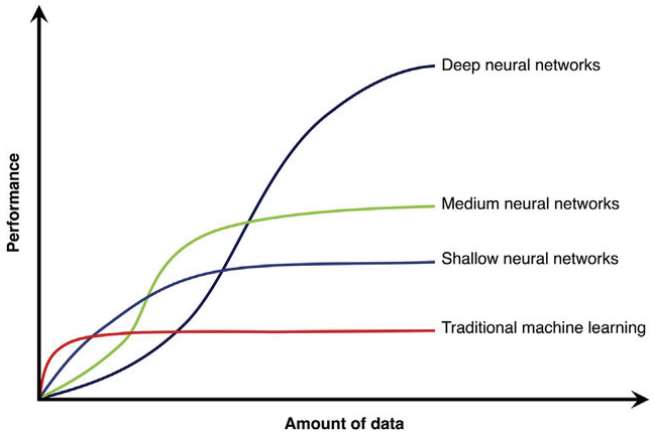
Razlika v količini pijače je 20ml, razlika v količini cene pa 4€. Ker algoritmi ne poznajo enot, je razlika 20 za algoritem bolj pomembna kot razlika 4, pa tudi če je nam ljudem jasno, da temu ni tako. Zaradi tega je potrebno vse vrednosti spraviti na enako enoto: *z-vrednost*. S standardizacijo vsake izmed spremenljivk dosežemo, da bo povprečje spremenljivke vseh primerov 0, standardni odklon pa 1. Po normalizaciji je razlika 20ml le še 0,03 , razlika 4€ pa je kar 0,6.

3.3.2 *Napačna izbira značilk*

Z vidika kakovosti morajo biti podatki po eni strani čim bolj točni, zanesljivi in brez šuma, po drugi pa vsebinsko takšni, da je iz njih sploh mogoče napovedati željen rezultat. Za vsebinsko ustreznost podatkov je ključnega pomena pravilna izbira značilk (atributov), ki jih bomo uporabili v fazi učenja napovednih modelov. Izbira ustreznih značilk ni preprosta naloga, ki je v veliki meri odvisna od poznavanja same domene in konkretnega problema, ki ga skušamo rešiti. V splošnem velja, da je izbira ustreznih atributov ena najpomembnejših nalog pri strojnem učenju, celo pomembnejša od izbire primernega algoritma [6].

3.3.3 *Premalo (ustreznih) podatkov*

Preprosto dejstvo je, da je uspešnost strojnega učenja odvisna od podatkov, na katerih se sistem uči. Pri tem sta pomembna tako kakovost samih podatkov kot njihova količina. Z vidika količine podatkov seveda velja, da lahko boljše rezultate dosežemo s čim večjo količino podatkov. Je pa pri tem zanimivo, da nad manjšo količino podatkov bolje delujejo nekateri klasični učni algoritmi, medtem ko najnaprednejše tehnike globokega učenja dosegajo boljše rezultate šele nad zelo veliko količino podatkov (slika 3).



Slika 3: Odvisnost napovedne uspešnosti od količine podatkov [7].

3.3.4 *Človeška pristranskost*

Izbiro algoritmov in ovrednotenje rezultatov opravijo ljudje, ki so lahko (nevede) pristranski. Znano je, da ljudje najraje izbirajo rešitve, ki jih dobro poznajo. Tako bo posameznik, ki je imel v preteklosti dobre izkušnje z nekim algoritmom, le-tega z večjo verjetnostjo izbral tudi za naslednjo nalogo, pa čeprav le-ta zanjo ne bo

optimalen. Podobno so ljudje, ki morajo ocenjevati kakovost rešitev, h katerim so sami prispevali (z zasnovo procesa strojnega učenja), nagnjeni k pretirano optimističnim ocenam. Prav tako lahko v želji po čim boljšem rezultatu (spet nevede) v postopek gradnje napovednih modelov vnesejo prepovedane informacije (uhajanje podatkov), kar prispeva k bistveno slabšim rezultatom v produkcijskem okolju od predvidenih z opravljenimi testi. V izogib tovrstnim pastem je potrebno dosledno sistematično preverjanje pravilnosti opravljenih korakov.

3.3.5 *Izbira primernega algoritma*

V splošnem ne obstaja noben algoritem strojnega učenja, ki bi bil v splošnem najboljši. Zato je izbira ustreznega algoritma glede na izkazane potrebe, zbrane podatke in namen rešitve izredno pomembna. Vsak algoritem ima svoje prednosti in slabosti, zato izbira napačnega kvečjemu poslabša poslovni rezultat. Teoretično bi seveda lahko za vsako nalogo preskusili kopico različnih algoritmov in izbrali najboljšega, vendar v praksi tak pristop ne deluje, saj izredno poveča kompleksnost samega procesa. Kot bomo videli kasneje, je za dobro evalvacijo modelov namreč še zmeraj potrebno človeško posredovanje. Zato je pri izbiri algoritma potrebno dobro poznati in ustrezno pretehtati lastnosti posameznih algoritmov.

3.3.6 *Uporaba najnaprednejših pristopov za vsako ceno*

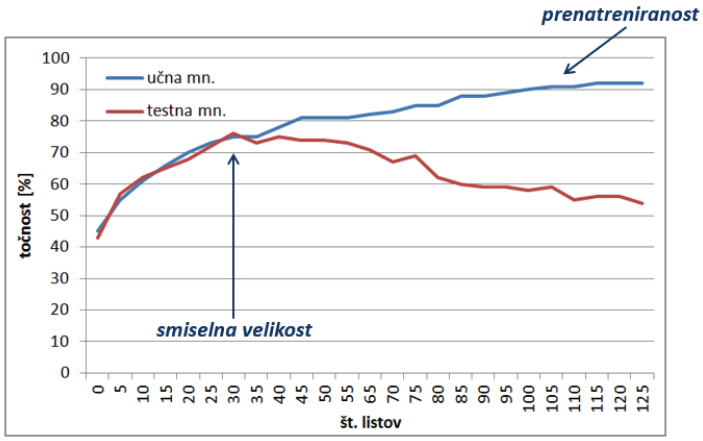
V času konstantnih napredkov na področju strojnega učenja se iz dneva v dan pojavljajo nove in boljše rešitve ter pristopi za uporabo strojnega učenja na raznovrstnih problemih. Najnovejši pristopi zelo pogosto prekašajo starejše na specifični domeni, kar nam daje občutek, da so le najnovejši pristopi primerni za uporabo. Temu največkrat ni tako. Vsak pristop, algoritem ali metoda ima svoje specifike, svoje pozitivne lastnosti, svoje negativne lastnosti in domene, kjer deluje odlično ter domene kjer deluje slabo. Dobro poznavanje teh lastnosti je ključ do tega, da se pri implementaciji ne uporabijo napačni modeli. Hkrati je potrebno tudi dobro poznati problem, ki ga skušamo rešiti. Nesmiselno je uporabiti zahtevne nevronske mreže, če nimamo za to dovolj podatkov ali pa je vzorec, ki ga iščemo, nižje kompleksnosti. Najnovejši (in mnogokrat oglaševani kot najboljši) pristopi so največkrat specializirani za en specifičen, mnogokrat zelo zahteven, problem. Slepa uporaba takega modela na enostavnejših oz. drugačnih podatkih bo prinesla slabše rezultate, kot pa če pri tem uporabimo enostavnejše pristope. Kompleksnejši pristopi so namreč bolj nagnjeni k prenatreniranosti, saj sklepajo, da vsaka variabilnost v podatkih že kaže na določen vzorec. Ta problem se še potencira, če nimamo dovolj podatkov za uporabo kompleksnih modelov – vsaka vrednost bo interpretirana kot signal in šum bo spregledan. Ključ, da se izognemo tem problemom je, da se najprej poglobljeno spoznamo s podatki. Šele na to pa preizkusimo več pristopov (algoritmov) za reševanje našega problema – po korakih napredujemo od bolj preprostih, do bolj zahtevnih.

3.3.7 *Uglaševanje zagonskih parametrov algoritmov*

Pri strojnem učenju predstavlja optimizacija oz. uglaševanje zagonskih parametrov problem izbire niza optimalnih parametrov za izbran učni algoritem. Kakovost zgrajenega modela znanja je zelo odvisna od nastavitve parametrov, katerih vrednosti se uporabljajo za nadzor samega učnega procesa. Izbran algoritem strojnega učenja lahko zahteva različne izbire, omejitve, uteži ali stopnje učenja za posploševanje različnih vzorcev podatkov. Da lahko algoritem optimalno reši zastavljen problem strojnega učenja, je potrebno parametre ustrezno prilagoditi. Ker optimalnih vrednosti v splošnem ni možno analitično izračunati, se v ta namen pogosto uporabljajo različni pristopi optimizacije, pri čemer pa je potrebno zelo paziti, da ne pride do prenatreniranosti modela ali do uhajanja podatkov [8].

3.3.8 *Prenatreniranost modelov*

V želji po čim boljših rezultatih se manj izkušeni razvijalci hitro ujamejo v past pretiranega treniranja (angl. *overfitting*) napovednih modelov. Po principu Ockhamove britve so bolj splošni modeli znanja bolj primerni od zelo specializiranih, saj praviloma z njimi dosegamo boljše rezultate na novih, nepoznanih primerih, ki jih bomo dejansko morali reševati v realnem, produkcijskem okolju. Ker imamo v laboratorijskem okolju na voljo le znane podatke, s pretiranim prilagajanjem dobimo modele, ki so izredno dobri zgolj na teh podatkih in praviloma bistveno slabši kasneje na realnih (slika 4).



Slika 4: Problem prenatreniranosti napovednih modelov.

3.3.9 *Uhajanje podatkov*

Zelo nevarna past je uhajanje podatkov (angl. *data leakage*), kjer razvijalci testirajo zgrajene modele znanja na učnih podatkih. Vrednotenje modelov mora potekati izključno na podatkih, ki niso bili uporabljeni v procesu gradnje ali optimizacije modelov. Če modele vrednotimo na prej videnih podatkih, obstaja nevarnost, da v procesu učenja modela ne izluščimo vzorcev iz podatkov, temveč silimo model, da si podatke zapomni »na pamet«. Posledica je drugačna kot pri prenatreniranosti – točnost pri testiranju je namreč zelo visoka, ampak ta točnost pade, ko model znanja uporabimo v produkciji – torej, ko naleti na še nepoznane primere. Najpogostejši primer, ko pride do uhajanja podatkov, je pri sami optimizaciji algoritmov kreacije modelov znanja. Ko spreminjamo nastavitve algoritmov, je potrebno te tudi ovrednotiti. V tem koraku za vrednotenje še ne smemo uporabiti testnih podatkov, ampak za to uporabimo del učnih podatkov, ki pa ne sodelujejo pri procesu učenja (kreacije modela). Tem podatkom pravimo validacijski podatki. Posledično torej ne delimo naše množice podatkov na dva, ampak kar na tri dele:

- **učne podatke** (ki služijo le gradnji modelov),
- **validacijske podatke** (ki služijo vrednotenju nastavitev modelov) in
- **testne podatke** (ki služijo za vrednotenje modelov, ko smo zaključili s procesom gradnje in optimizacije).

3.3.10 *Merjenje kakovosti modelov*

Če se ukvarjamo s klasifikacijo podatkov v vnaprej definirane razrede, najlažje merimo kvaliteto modela znanja klasifikacije z metriko, ki jo imenujemo **točnost**. Ta metrika nam pove delež pravilno klasificiranih podatkov. Zdaj pa si poglejmo primer, ko klasificiramo podatke v dva zelo neuravnotežena razreda: učnih podatkov, ki so razreda A je 9990, učnih podatkov, ki so razreda B pa je 10. Pri gradnji modelov znanja prav vsak algoritem sprejme določeno mero napake, saj želi razbrati le vzorce in s tem generalizirati znanje, brez da bi le pomnil podatke na pamet. To pomeni, da bodo nekateri učni podatki spregledani pri gradnji modela znanja – v našem primeru bo to najverjetneje tistih 10, ki so razreda B. Najverjetneje bi preprost algoritem v našem primeru našel le eno pravilo: vsi podatki so razreda A. Če ovrednotimo tak model zgolj z vidika točnostjo, se model izkaže kot zelo kvaliteten, saj je njegova točnost kar 99,9%. Seveda pa je tak model popolnoma brez znanja in posledično neuporaben, saj ne zna pravilno klasificirati prav nobenega izmed primerkov razreda B!

Taki primeri neuravnoteženih podatkov niso redkost, zato že obstajajo pristopi za spopadanje z njimi. Najenostavnejši pristop je uporaba alternativnih metrik za merjenje kakovosti takih modelov znanja. Ena takih je **preciznost** ali natančnost, ki nam pove delež pravilno identificiranih primerkov posameznega razreda napram vsem identificiranim primerkom tega razreda. Podoben je tudi **priklic** ali senzitivnost, ki nam pove delež identificiranih primerkov posameznega razreda napram vsem primerkom tega razreda. Ti metriki lahko tudi združimo, s čimer dobimo bolj uravnoteženo oceno: lahko uporabimo aritmetično sredino in tako dobimo **F-mero**, ali pa uporabimo geometrično sredino in tako imamo **G-mero**. Vse štiri naštete metrike kvalitete se izračunajo le za posamezne razrede, te pa lahko agregiramo v eno vrednost, da dobimo kvaliteto modela znanja

v splošnem, ne le za en razred. Agregiramo lahko s preprostim povprečenjem, lahko povprečimo uravnoteženo glede na razmerja razredov ali pa povprečimo vrednosti pred samim izračunom – uporaba je odvisna od naših potreb in ciljev.

4 ZAKLJUČEK

Uporaba strojnega učenja se z uvedbo grafičnih uporabniških vmesnikov v orodjih in številnih spletnih učilnic zdi precej enostavna. Seveda pa zgolj dejstvo, da so algoritmi in metode strojnega učenja na voljo praktično vsakomur, še ne pomeni, da bo pri njihovi uporabi vsak poskus pravilen in uspešen. Če znamo zapisati delujočo kodo za uporabo modula strojnega učenja, še ne pomeni, da ga znamo uporabljati pravilno. V celotnem procesu strojnega učenja se skriva ogromno pasti, v katere se ob neustreznem delovanju hitro ujamemo. Knjige, vodiči in drugi viri nam mnogokrat zamolčijo pogoste pasti pri uporabi pristopov strojnega učenja – sposobnost njihovemu izogibanju se pridobijo predvsem z izkušnjami. Namen prispevka je, da bralce seznani s številnimi pastmi, ki smo jih z izkušnjami tekom našega raziskovanja, uporabe, implementacije in kreacije pristopov strojnega učenja pridobili. Tako so v prispevku predstavljene številne zmote in nepravilni pristopi, na katere moramo biti pozorni pri uporabi algoritmov strojnega učenja, sestavi ekipe umetne inteligence, obdelavi podatkov in interpretaciji modelov. Čeprav so v prispevku opisane le najpogostejše in najnevarnejše pasti slepe uporabe strojnega učenja, je poznavanje že teh velik korak k bolj uspešni uporabi strojnega učenja in interpretaciji rezultatov.

5 LITERATURA

[1] BRYNJOLFSSON Erik, McAFEE Andrew "The business of artificial intelligence", Harvard Business Review, 2017.

[2] SMOLA Alex, Introduction to Machine Learning, Cambridge University Press, 2010.

[3] MUNCASTER Phil, »IT Leaders Believe AI is a 'Silver Bullet' for Threats«, Info Security Magazine, avgust 2018.

[4] KARAKATIČ Sašo, VRBANČIČ Grega, FLISAR Jernej, PODGORELEC Vili "Umetna inteligenca za telebane – platforme strojnega učenja", Sodobne tehnologije in storitve OTS 2018: Zbornik triindvajsete konference, Maribor, Fakulteta za elektrotehniko, računalništvo in informatiko, Inštitut za informatiko, 175-188.

[5] VRBANČIČ Grega, PODGORELEC Vili " Analiza inteligentnih oblačnih storitev na primeru prepoznave obrazov", Sodobne tehnologije in storitve OTS 2018: Zbornik triindvajsete konference, Maribor, Fakulteta za elektrotehniko, računalništvo in informatiko, Inštitut za informatiko, 189-201.

[6] DIETTERICH Thomas G., *et al.* »Ensemble learning«, The handbook of brain theory and neural networks, 2002, 110-125.

[7] TANG An, *et al.* »Canadian Association of Radiologists White Paper on Artificial Intelligence in Radiology«, Canadian Association of Radiologists Journal, letnik 69, april 2018, str. 120-135.

[8] BERGSTRA James; BENGIO Yoshua »Random search for hyper-parameter optimization«, Journal of Machine Learning Research, februar 2012, str. 281-305.

OTS 2019
SODOBNE INFORMACIJSKE TEHNOLOGIJE IN STORITVE
M. Heričko in K. Kous (ur.)

Univerzitetna založba
Univerze v Mariboru

KORAK K SAMODEJNI INTEGRACIJI IN TESTIRANJU NAPOVEDNIH MODELOV STROJNEGA UČENJA

GREGA VRBANČIČ, VILI PODGORELEC

Povzetek: *Z razcvetom strojnega učenja in prodorom v vse veje gospodarstva, se nenehno povečuje tudi potreba po ustaljenih razvojnih procesih pri razvoju napovednih modelov strojnega učenja, ki bi olajšala njihovo vpeljavo in integracijo v obstoječe informacijske sisteme. Podobno kot je to že ustaljena praksa pri programskem inženirstvu, je potreba in želja gospodarstva, da se tudi pri razvoju napovednih modelov strojnega učenja razvijejo smernice in prakse, ki bi omogočale enostavno vpeljavo, hiter in prilagodljiv razvoj ter nadzor nad kvaliteto dostavljenih napovednih modelov. V prispevku bomo predstavili smernice, izzive in potencialne rešitve pri vpeljavi napovednih modelov v proces samodejne neprekinjene integracije ter prikazali praktičen primer vpeljave napovednega modela.*

Ključne besede: *samodejna neprekinjena integracija, napovedni modeli, strojno učenje, testiranje, razvoj programske opreme*

NASLOVA AVTORJEV: Grega Vrbančič, mladi raziskovalec, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Maribor, Slovenija, e-pošta: grega.vrbancic@um.si.
dr. Vili Podgorelec, redni profesor, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Maribor, Slovenija, e-pošta: vili.podgorelec@um.si.

https://doi.org/10.18690/978-961-286-282-4.3 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 UVOD

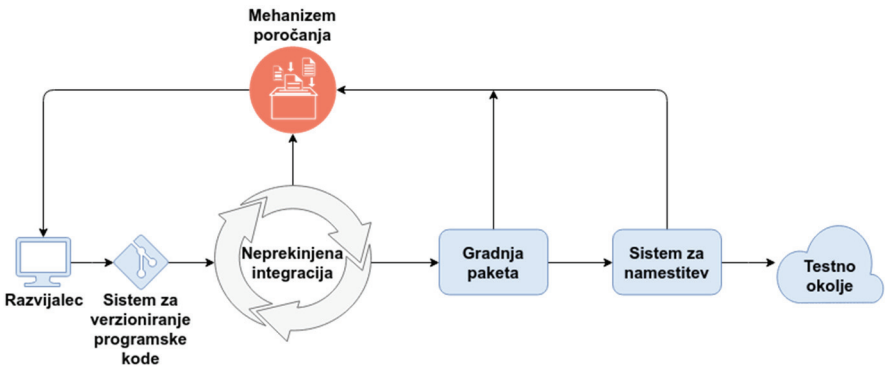
S prodorom umetne inteligence in tehnik strojnega učenja v praktično vse veje gospodarstva, še posebej pa na področje informacijskih tehnologij, se vedno pogosteje zastavlja tudi vprašanje o integraciji in zagotavljanju kakovosti takšnih rešitev. V nasprotju z na primer programskim inženirstvom, kjer so procesi integracije in testiranja programske kode ter programske metrike že dodobra opredeljene in uveljavljene, pa pri razvoju napovednih modelov trenutno še ni dobro definiranih okvirjev oz. strategij, kako jih pravilno in učinkovito integrirati, testirati in zagotavljati kakovost. Sprejeti in uveljavljeni procesi testiranja in zagotavljanja kakovosti napovednih modelov so nujni za njihovo množično integracijo v obstoječe ali celo kritične dele informacijskih sistemov. Da bi lahko dobro definirali procese testiranja in zagotavljanja kakovosti napovednih modelov, pa moramo najprej definirati, kaj na področju strojnega učenja sploh pomeni testiranje. Primarno se izraz testiranje, v povezavi z umetno inteligenco, navezuje na testiranje uspešnosti napovednih modelov. Na uspešnost napovednih modelov vpliva velik nabor odvisnih in neodvisnih spremenljivk, zato je na mestu vprašanje, kako sploh testirati napovedne modele? Kako zagotoviti, da bodo delovali v skladu z našimi zahtevami in pričakovanji?

V prispevku bomo predstavili problematiko oziroma izzive samodejne neprekinjene integracije in testiranja ter zagotavljanja kakovosti napovednih modelov ter poizkušali odgovoriti na zastavljena vprašanja. Predstavili bomo že uveljavljen koncept samodejne neprekinjene integracije ter ga razširili z vpeljavo korakov in komponent, specifičnih za razvoj napovednih modelov strojnega učenja. Prav tako bomo opisali obstoječe smernice in pristope zagotavljanja kakovosti napovednih modelov ter naslovili problematiko integracije napovednih modelov v obstoječe informacijske sisteme in na praktičnem primeru prikazali končen doprinos uvedbe testiranja na delovanje napovednega modela.

2 SAMODEJNA NEPREKINJENA INTEGRACIJA

Integracija programske opreme je pristop povezovanja podsistemov in komponent programske opreme z namenom izgradnje enotnega sistema. Integracija je del vsakega razvojnega življenjskega cikla programske opreme, saj ekipa inženirjev programsko opremo razvija skozi različne faze. [1] Leta je integracija programske opreme predstavljala tiičen in nepredvidljivosti poln korak v življenjskem ciklu razvoja programske opreme. Že več kot dve desetletji pa problematika tega koraka počasi pojenja, zahvaljujoč pojavitvi samodejne neprekinjene integracija (angl. Continuous Integration, CI). Začetki CI izhajajo iz Kent Blockovih dvanajstih praks razvoja programske opreme, poznanega kot eXtreme Programming (XP) [2]. Avtor je CI okarakteriziral kot proces, v katerem je nova programska koda integrirana v trenuten sistem v največ nekaj urah, pri čemer je sistem vedno zgrajen v celoti od začetka, spremembe programske kode pa se ohranijo v primeru, da so bili uspešno prestani vsi testi. [3].

Tipičen scenarij procesa neprekinjene integracije programske kode (Slika 1) se začne z razvijalcem, ki prispeva (angl. commit) programsko kodo v skupen repozitorij. Pri delu na tipičnem projektu lahko deležniki, pogosto v različnih vlogah, prispevajo spremembe, ki sprožijo cikel CI. Na primer, razvijalci sprožijo cikel s spremembami programske kode, administratorji podatkovnih baz s spremembami definicij entitet, ekipe, ki skrbijo za izgradnjo in namestitev programske opreme, s spremembami konfiguracijskih datotek, itd.



Slika 1: Konceptualni prikaz procesa neprekinjene integracije programske kode.

Koraki v omenjenem tipičnem scenariju procesa integracije programske kode si v splošnem sledijo [4]:

1. Najprej deležnik prispeva spremembo v skupen repozitorij. Medtem CI strežnik neprekinjeno na določen interval opreza za morebitnimi spremembami na repozitoriju ali posluša na izpostavljenem spletnem vmesniku (angl. webhook) za morebiten klic, ki sporoča o dogodku – spremembi na skupnem repozitoriju.
2. Kmalu po oddanem prispevku v repozitorij, CI strežnik zazna spremembo in pridobi najnovejšo različico programske kode ter prične z izvajanjem CI cikla.
3. Po koncu cikla CI strežnik generira poročilo, ki ga posreduje vsem deležnikom oz. poročilo ponudi na voljo v enostavno dostopni spletni obliki.
4. Opcijsko zapakira uspešno integrirano, prevedeno programsko kodo v izvršljiv paket in ga namesti na testno izvajalno okolje.

2.1 Komponente neprekinjene integracije

Kljub temu, da je samodejna neprekinjena integracija postala standard v programskem inženirstvu, v stroki še vedno ni konsenza, kako bi takšen sistem moral biti implementiran. Na voljo so različna orodja (Jenkins, Travis, GitLab CI...), ki omogočajo razvoj programske opreme z uporabo pristopa samodejne neprekinjene integracije, sicer pa pristop sam ne zahteva uporabe nobenega specifičnega orodja. Glede na omenjeno, obstaja množica napotkov in dobrih praks [4], [5], povezanih s CI in vpeljavo le-te, ki jih lahko obravnavamo kot nekakšne smernice. Martin Fowler v njegovem množično citiranem prispevku [5] predstavi deset ključnih praks, potrebnih za vzpostavitev učinkovitega CI sistema:

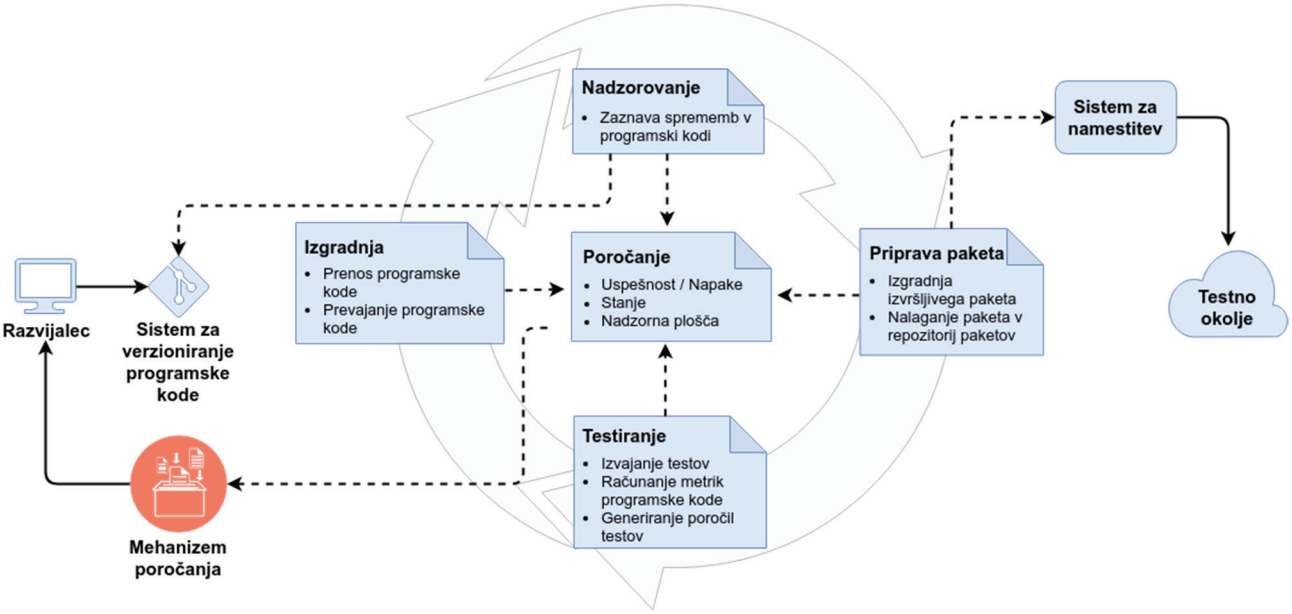
- vzdrževanje enotnega vira – repozitorija programske kode,
- avtomatizacija izgradnje paketa,
- izgradnja naj bo samo testirajoča (angl. self-testing),
- vsi razvijalci dnevno prispevajo kodo v repozitorij,
- vsak prispevek mora biti zgrajen na strežniku, ki poganja integracijskih sistem,
- nedelujoče izgradnje morajo biti takoj popravljene,
- izgradnja naj bo hitra,
- testiranje se naj izvaja na klonu produkcijskega okolja,
- dostopnost do zadnjega zgrajenega izvajalnega paketa naj bo enostavna in omogočena vsem deležnikom,
- vsak ima možnost spremljati, kaj se dogaja,
- avtomatizacija namestitve.

Podobno kot Fowler, tudi Duvall [4] predstavi sedem temeljev neprekinjene integracije programske opreme, ki v splošnem povzamejo Fowlerjeve ključne prakse, dodatno pa eksplicitno izpostavi, da naj ima popravilo neuspešnih integracij najvišjo prioriteto. Kot enega izmed ključnih temeljev izpostavi tudi pregled generiranih poročil, ki vsebujejo analizo kvalitete programske kode.

2.2 Integracijski cikel

Kot predstavljeno je proces samodejne neprekinjene integracije programske opreme sestavljen iz več gradnikov. Ključna komponenta integracijskega procesa, sestavljena iz več gradnikov, je integracijski cikel. Vsak integracijski cikel (Slika 2), se izvaja na CI strežniku, ki v splošnem izvede naslednje korake [6]:

1. Zazna spremembe na skupnem repozitoriju (npr.: Git, Subversion) programske kode.
2. Pridobi programsko kodo iz repozitorija ter jo prenese na CI strežnik.
3. Prevede programsko kodo, v kolikor je to potrebno.
4. Izvede preverjanje kakovosti – izvede sorodne naloge kot so testiranje enot (angl. unit testing) in verifikacija avtomatizirane arhitekture, analiza kvalitete programske kode, itd. Te naloge so v procesu samodejne neprekinjene integracije opcijske, vendar priporočljive, saj je čimprejšnje vpeljevanje nadzora kvalitete v proces integracije ena izmed ključnih lastnosti pristopa samodejne neprekinjene integracije.
5. Prevedeno programsko kodo vrne nazaj v repozitorij.
6. Poroča o stanju izgradnje vsem deležnikom.
7. Opcijsko zapakira uspešno integrirano prevedeno programsko kodo in jo namesti v testno okolje.



Slika 2: Podrobna predstavitev cikla neprekinjene integracije.

2.3 Testiranje

Faza testiranja, poznana tudi kot neprekinjeno testiranje (angl. Continuous testing), v ciklu CI uporablja avtomatizirane pristope ter tako imenovan pomik v levo (angl. shift-left) pristop z namenom pohitritve testiranja, ki posledično v proces CI in faze razvoja programske opreme vključuje tudi problematiko zagotavljanja kakovosti. Uporaba takšnega pristopa lahko vključuje nabor samodejnih testnih delovnih tokov, kateri lahko zajemajo tudi analitiko in metrike programskega inženirstva (LOC[1], CBO[2], CC[3]...), z namenom zagotavljanja čiste, pregledne, na dejstvih temelječe slike kvalitete dostavljene programske opreme. [6]

Z uporabo pristopa neprekinjenega testiranja deležniki dobijo povratne informacije o kakovosti programske opreme, ki jo gradijo. Prav tako jim omogoča, da testirajo prej ter z večjo pokritostjo z odstranitvijo ozkega grla kot je dostop do deljenega testnega okolja in čakanjem na razvoj oz. stabilizacijo uporabniškega vmesnika.

Torej, če želimo razviti programsko opremo, ki je zanesljiva, moramo zagotoviti zanesljivost na nivoju objektov, katero pa lahko dosežemo le s skozi uspešno testiranje enot. Seveda pa nam zgolj napisani testi enot nujno ne zagotavljajo zanesljivosti. Testi morajo efektivno preverjati uporabo objektov in morajo biti pogosto zagnani. Ker objekti komunicirajo med seboj, morajo biti testi pognani, kadarkoli v programski opremi pride do kakršnekoli spremembe. Za celovito in učinkovito testiranje programske opreme je nujno potrebno avtomatizirati skupek raznovrstnih testov [4]:

- **Testi enot**: preverjajo obnašanje manjših elementov programske opreme, ki so navadno predstavljeni kot samostojen razred. Nekateri testi enot zahtevajo minimalne zunanje odvisnosti, ki so implementirane samo kot drugi samostojni razredi. Takšni razredi so sami po sebi enostavni in nimajo globokega grafa objektov. Občasno testi enot uporabljajo t.i. prototipe (angl. mock), ki so prav tako enostavni razredi, katerih vloga je nadomestitev bolj kompleksnih enot za namene testiranja.
- **Testi komponent:** testi komponent ali testi podsistemov preverjajo specifične dele oz. komponente programske opreme in navadno zahtevajo kakšne zunanje odvisnosti kot so podatkovne baze, datotečni sistemi itd. Takšni testi preverjajo, ali komponente delujejo na način, da ustvarjajo pričakovano agregirano obnašanje. Ker tovrstni testi zajemajo večjo količino programske kode v vsakem testnem primeru, se temu primerno tudi dalj časa izvajajo kot na primer testi enot. Pogosto posamezni testi komponent testirajo obnašanje določenih komponent preko izpostavljenega aplikacijskega programskega vmesnika. Takšni testi so poznani tudi kot integracijski testi.

[1] Število vrstic kode (angl. Lines Of Code – LOC)
[2] Število razredov, ki so vezani na specifičen razred (angl. Coupling Between Objects – CBO)
[3] Ciklična kompleksnost (angl.Cyclomatic Complexity – CC)

- **Sistemski testi:** preverjajo delovanje celotne programske opreme in tako zahtevajo popolno nameščen sistem, kot na primer vsebnik servletov in povezano podatkovno bazo. Takšni testi preverjajo, ali zunanji vmesniki kot so spletne strani, spletne storitve ali grafični uporabniški vmesnik delujejo v povezavi s preostalim razvitim sistemom po pričakovanjih.
- **Funkcionalni testi:** kot nakazuje ime, testi funkcionalnosti preverjajo delovanje posameznih funkcionalnosti programske opreme s stališča uporabnika, kar pomeni da testi sami posnemajo interakcijo uporabnikov s programsko opremo.

Glavne koristi, ki jih deležniki z vpeljavo neprekinjenega testiranja pridobijo, so tako [6]:

- Pomik aktivnosti testiranja "v levo" v življenjskem ciklu razvoja programske opreme in integracijo le-teh v razvojne aktivnosti.
- Združevanje testnih, razvojnih in administrativnih ekip v vsakem koraku življenjskega cikla razvoja programske opreme.
- Avtomatizacija testiranja v največji meri z namenom neprekinjenega testiranja ključnih lastnosti oz. zmogljivosti dostavljene programske opreme.
- Omogoča, da poslovnim partnerjem dostavljamo zgodnjo in neprekinjeno povratno informacijo o lastnostih in zmogljivostih razvijajoče se programske opreme.
- Odstranjuje ozka grla dostopnosti do testnega okolja.
- Aktivno in neprekinjeno nadzoruje kvaliteto skozi celoten razvojni cikel programske opreme.

2.4 Izzivi vpeljave neprekinjene integracije

Skozi leta razvoja in vpeljav CI procesa v proces razvoja programske opreme je faza testiranja postala ena izmed najpomembnejših delov uspešnega izvajanja CI. Po drugi strani pa glede na poročanje številnih študij, faza testiranja predstavlja izvor največjih ovir in izzivov, povezanih z vpeljavo CI v proces razvoja programske opreme.

Med pogostejšimi izzivi, povezanimi s testiranjem programske kode, se tako pojavlja problem avtomatizacije različnih tipov testov kot so testiranje enot, integracijsko testiranje, testiranje uporabniških vmesnikov [7], [8]. Vzrok za težave pri avtomatizaciji testov lahko bodisi izvira iz pomanjkljive infrastrukture, odvisnosti od specifične strojne opreme, bodisi iz slabega izvajanja testno vodenega razvoja (angl. test driven development, TDD) [9]. Drug pogost izziv, povezan s testiranjem programske opreme, pa je nizka kvaliteta napisanih testov, kar vključuje nezanesljive teste, nizko pokritost s testi, dolgo izvajajoče se teste, ipd. [7], [10]

Poleg izzivov, povezanih s testiranjem programske opreme, so avtorji članka [9] kot pomemben izziv vpeljave CI identificirali tudi odpravljanje konfliktov v programski kodi. Med pogostejšimi razlogi za težave pri odpravljanju konfliktov sta izpostavljena odvisnost od modulov tretjih oseb ter močno sklopljena zasnova oz. arhitektura programske opreme. [7], [10]

2.5 Koristi neprekinjene integracije

Raziskave, opravljene na področju programskega inženirstva, kot pomembnejše koristi neprekinjene integracije izpostavljajo [5], [9]:

- Pridobivanje hitrejšega povratnega odziva o uspešnosti izgradnje programske opreme kot tudi odziva uporabnikov.
- Hitrejša identifikacija, lažje obvladovanje ter hitrejša odprava hroščev v programski opremi.
- Pogoste in zanesljive izdaje programske opreme, ki vodijo k zvišanju stopnje zadovoljstva končnih uporabnikov programske opreme.
- Skozi vpeljavo CI v proces razvoja programske opreme podjetja pogosto vpeljejo tudi proces neprekinjene dostave, kar ima za posledico izboljšanje povezovanja in komunikacije med razvijalci ter skrbniki infrastrukture ter posledično povečano stopnjo avtomatizacije procesov.

3 VKLJUČITEV NAPOVEDNIH MODELOV V PROCES SAMODEJNE NEPREKINJENE INTEGRACIJE IN TESTIRANJA

Razvoj napovednih modelov strojnega učenja v sami osnovi ni precej drugačen od razvoja tradicionalne programske opreme. Tako kot pri razvoju običajne programske opreme je tudi pri razvoju napovednih modelov to proces s celotnim življenjskim ciklom, ki vključuje načrtovanje, implementacijo, optimizacijo, testiranje in dostavo oziroma namestitev. S tem ko napovedni modeli postajajo iz leta v leto čedalje bolj vključeni v raznovrstne informacijske sisteme, se tudi njihova vloga čedalje bolj povečuje. Z vedno bolj pomembno vlogo napovednih modelov pa se zvišuje tudi potreba, da je življenjski cikel razvoja ter integracije napovednega modela upravljan na sistematičen in bolj rigiden način kot je to do sedaj že uveljavljena praksa pri razvoju konvencionalne programske opreme. [11] O tem priča tudi porast znanstvenih in strokovnih člankov [2], [11], [12], ki naslavljajo temo razvojnega življenjskega cikla napovednih modelov strojnega učenja in integracijo ter namestitev takšnih modelov v že obstoječe programske rešitve.

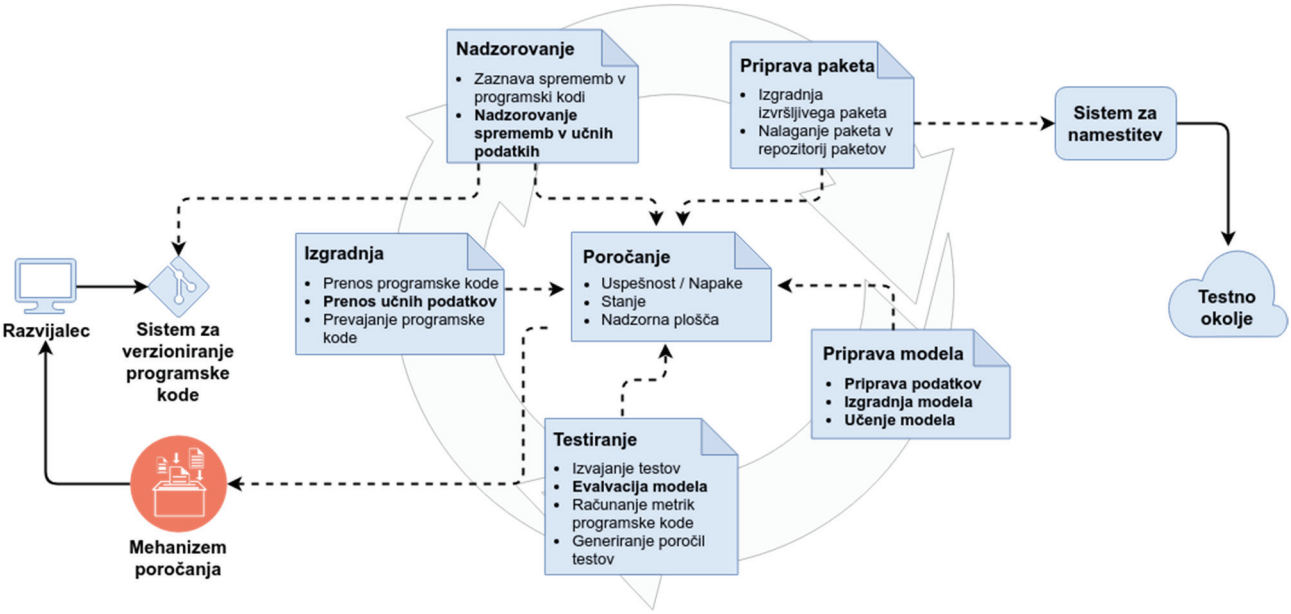
Kot rečeno, se sam pristop razvoja napovednega modela v samem bistvu kaj dosti ne razlikuje od razvoja tradicionalne programske opreme, ampak ga v splošnem samo razširja. Tipičen proces razvoja napovednega modela bi lahko opredelili s sledečimi koraki:

1. Identifikacija problema
2. Načrtovanje rešitve
3. Pridobivanje in pred-obdelava podatkov
4. Izgradnja napovednega modela
   4.1. Izbira algoritma
   4.2. Učenje modela
5. Evalvacija modela
6. Namestitev

3.1 Samodejna neprekinjena integracija napovednih modelov

Izmed prej naštetih korakov procesa razvoja napovednih modelov sta za vpeljavo napovednih modelov strojnega v proces CI ključna izgradnja napovednega modela in evalvacija modela. Izgradnja uspešnega napovednega modela v splošnem zahteva veliko učno množico (podatki), bolj ali manj kompleksen algoritem, s katerim bomo zgradili napovedni model ter strategijo učenja in ovrednotenja napovednega modela. Ob tem je pomembno dodati, da je izgradnja napovednega modela tipično dolgo trajajoč postopek, katerega rezultat ni deterministične. Ne-determinističnost napovednih modelov pa prinaša tudi številne izzive v področje nadzorovanja in zagotavljanja kakovosti delovanja napovednega modela, zato je ključnega pomena, da v obstoječ CI proces pri vpeljavi napovednih modelov strojnega učenja poleg mehanizmov priprave modela vključimo tudi potrebne mehanizme za evalvacijo le-teh.

Konceptualna zasnova (Slika 3) prikazuje razširjen proces CI z vključitvijo napovednega modela strojnega učenja. Kot je razvidno iz slike je poleg razširitev obstoječih komponent procesa dodana tudi nova komponenta – priprava modela. Priprava modela vsebuje korake priprave podatkov, izgradnjo samega modela ter učenje modela. V komponenti nadzorovanje je dodan korak nadzorovanja sprememb nad učnimi podatki, ti so navadno hranjeni bodisi v obstoječem sistemu za verzioniranje programske kode, bodisi v za to namenjenem sistemu za hrambo podatkov kot na primer v podatkovnih bazah, datotečnih sistemih ipd. Razširjena je tudi komponenta izgradnja, ki poleg prenosa ter opcijsko prevajanja programske kode vsebuje tudi korak, v katerem se opravi prenos učnih podatkov iz specifičnega vira v sam proces CI. Prav tako ključna je razširitev komponente testiranje, ki v razširjenem procesu CI vsebuje tudi korak evalvacije modela.



Slika 3: Konceptualna zasnova razširjenega procesa neprekinjene integracije z vključitvijo napovednih modelov strojnega učenja.

3.2 Izzivi

Z vključitvijo dodatne komponente ter razširitvijo omenjenih komponent procesa samodejne integracije in testiranja napovednih modelov, se moramo soočiti tudi z izzivi, ki jih le-te prinesejo. Kot ključne izzive so avtorji v prispevkih [11]–[13] izpostavili predvsem problematiko kompleksnosti testiranja in zagotavljanja kakovosti napovednih modelov ter izzive, povezane z učenjem napovednih modelov.

Med izzive, povezane z učenjem napovednih modelov, prav gotovo spadajo izzivi, ki se navezujejo na učne in testne podatke. Kvaliteta in kvantiteta podatkov, tako učnih kot tudi testnih, je kritična za razvoj uspešnega napovednega modela. V primeru razvoja napovednih modelov, temelječih na algoritmih globokega učenja, kot so na primer konvolucijske nevronske mreže ali nevronske mreže s povratno zanko, pa je predvsem količina podatkov še toliko bolj pomembna, saj takšni algoritmi za uspešno učenje potrebujejo večjo količino prečiščenih podatkov kot konvencionalni algoritmi. Na tej točki velja izpostaviti pomembnost čiščenja podatkov, ki je pri razvoju in integraciji napovednih modelov pogosto spregledana. V splošnem obstajata dve skupini tehnik oziroma pristopov, s katerimi naslavljamo negativni efekt neprečiščenih podatkov oziroma podatkov s šumom: odpornost na šum v podatkih ter odstranitev šuma. Pristopi, odporni na šum, vključujejo izgradnjo robustnih algoritmov, ki so tolerantni do podatkov, ki vključujejo šum in tako ne potrebujejo predprocesiranja in prečiščevanja podatkov. Tehnike za odstranjevanje šuma pa delujejo po principu odstranjevanja učnih primerkov, v kolikor le-ti vsebujejo šum. [2]

Prav tako pogosti izzivi, povezani z učenjem napovednih modelov, so hranjenje podatkov oziroma izzivi z uporabo sistema za verzioniranje. Velikost napovednih modelov lahko presega nekaj gigabajtov, povrhu vsega pa so ti navadno hranjeni v binarni obliki, kar zna predstavljati problem ob uporabi sistema za verzioniranje kot je Git, saj ta ni bil zasnovan za takšne namene. Sicer lahko tovrsten problem rešimo z uporabo razširitve Git LFS (Git Large File Storage), vendar pa ni nujno da je uporaba le-te podprta s strani izbranega sistema za neprekinjeno integracijo. Z vpeljavo napovednih modelov v proces CI pa je potrebno poleg napovednih modelov slediti in hraniti tudi spremembe v učnih in testnih podatkih. Sicer je mogoče to opravljati v sklopu repozitorija programske kode, lahko pa tak pristop hitro privede do težje sledljivosti in zmede pri nadzorovanju in upravljanju sprememb. Obetajoč kandidat, ki naslavlja omenjeno problematiko, je orodje Data Version Control (DVC), splavljeno leta 2018, ki je razširitev Git orodja s poudarkom na upravljanju z velikimi količinami podatkov in možnostjo reprodukcije napovednih modelov. [2]

Ena izmed pomembnejših lastnosti procesa CI je hitra in pogosta integracija, ki vzpodbuja, da razvijalci večkrat dnevno prispevajo programsko kodo v repozitorij. Kot poročajo avtorji, se že pri razvoju konvencionalne programske opreme pogosto pojavljajo izzivi, povezani z dolgo trajajočimi integracijskimi cikli. Ti pa se z

vpeljavo napovednih modelov v proces CI še dodatno drastično povečajo, saj je učenje napovednih modelov računsko zahteven in dalj časa trajajoč proces. Tipično se pri razvoju konvencionalne programske opreme težava dolgo trajajočega CI cikla rešuje z razbijanjem izgradnje programske opreme na več manjših komponent. Enakega pristopa pa žal ne moremo uporabiti pri razvoju napovednih modelov, saj je proces izgradnje in učenja močno sklopljen. Omenjene težave lahko naslovimo z delegiranjem učenja napovednih modelov na specifično strojno opremo, optimizirano za učenje napovednih modelov, ali pa z uporabo ogrodij in sistemov za porazdeljeno učenje. [2]

Kot poznamo že s področja programskega inženirstva je testiranje pomembna strategija za povečanje zanesljivosti, zmanjševanje tehničnega dolga in dolgoročno zniževanje stroškov vzdrževanja. Ključna razlika testiranja napovednih modelov strojnega učenja v primerjavi s testiranjem konvencionalne programske opreme je v kompleksnosti. Kompleksnost testiranja napovednih modelov izvira iz njihove narave, saj je obnašanje le-teh močno odvisno od podatkov in dejstva, da jih ni mogoče strogo specificirati. Na omenjen izziv lahko gledamo z vidika prispodobe učenja napovednega modela s prevajanjem programske kode, kjer je v primeru razvoja napovednega modela programska koda prispodoba tako za dejansko programsko kodo kot tudi učne podatke. Tako je potrebno, z vidika te prispodobe, testirati tako programsko kodo kot tudi podatke, naučen napovedni model pa moramo obravnavati na podoben način kot obravnavamo izvršljive pakete programske opreme. [13] Na tej točki si je torej smiselno zastaviti vprašanje: Kaj in koliko sploh testirati?

3.3 Testiranje in evalvacija napovednih modelov

Testiranje programske opreme je v programskem inženirstvu v splošnem dobro raziskano, podobno je tudi s področjem strojnega učenja. Težava pa se pojavlja na preseku omenjenih področij, torej testiranju napovednih modelov strojnega učenja. Še najboljši približek smernicam oziroma dobrim praksam, kot jih poznamo na področju programskega inženirstva, je predstavljen v prispevku [13], pripravljenem s strani Googlovih zaposlenih. V nadaljevanju bomo predstavili nekaj pomembnejših smernic za celovito testiranje napovednih modelov ter zmanjševanje potencialnega tehničnega dolga.

Ključna razlika med konvencionalno programsko opremo ter napovednimi modeli strojnega učenja je, da obnašanje napovednih modelov ni določeno neposredno s programsko kodo, ampak na njegovo obnašanje vplivajo podatki. Zato je posledično smiselno v procesu neprekinjene integracije z vpeljanimi napovednimi modeli testirati tudi učne podatke. Pri testiranju podatkov je smiselno preverjati ali so le-ti v pravilni oz. v pričakovani obliki ter ali so vrednosti posameznih značilk znotraj predvidenega definiranega območja. Dodatno je smiselno s testi enot preverjati tudi programsko kodo za izluščevanje in/ali izgradnjo značilk, čeprav se navadno zdi, da je precej enostavna in ni potrebe po testiranju. Napake v vrednosti značilk, ko so že v procesu izluščevanja in nadalje v procesu učenja napovednega modela, je namreč skorajda nemogoče zaznati. [13]

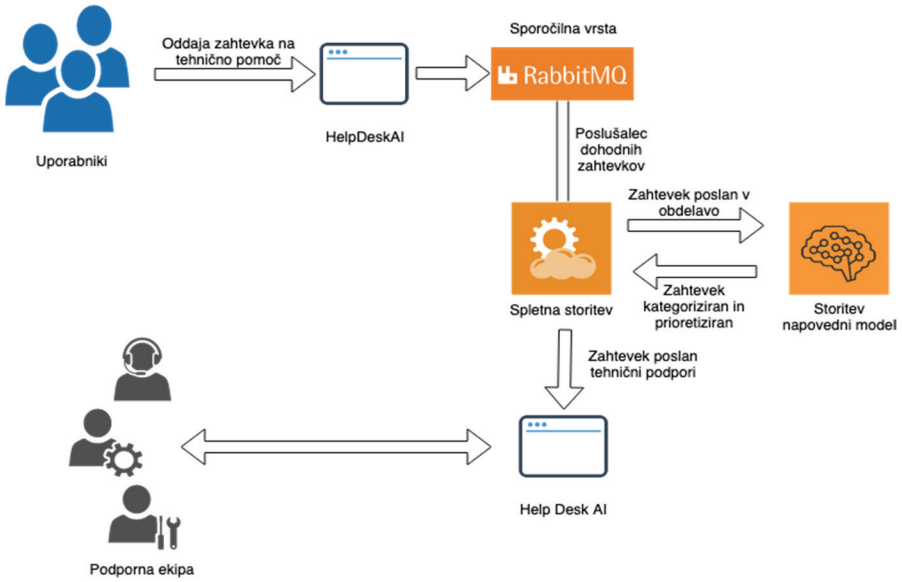
Za razliko od področja strojnega inženirstva, kjer se je že razvil širok nabor smernic in dobrih praks, so smernice in dobre prakse razvoja zanesljivih napovednih modelov strojnega učenja še v razvoju. Algoritmi strojnega učenja navadno dopuščajo oz. celo zahtevajo nastavitve velikega števila t.i. hiper-parametrov, s katerimi definiramo delovanje algoritmov, učnih strategij, ipd. Izbira primernih hiper-parametrov ima velik vpliv na sposobnost učenja modela kot tudi na končno uspešnost napovednih modelov. Hiper-parametre je potrebno skozi razvoj prav tako prilagajati glede na spremembe v zasnovi učnega algoritma kot tudi glede na spremembe v podatkih. V ta namen je priporočljiva uporaba bolj ali manj enostavnih strategij iskanja hiper-parametrov, ki lahko v splošnem poleg zvišanja kvalitete napovedi razkrije tudi težave, povezane z zanesljivostjo napovednih modelov. Za namene zagotavljanja kakovosti napovednih modelov je ključno vpeljati teste, kjer preverjamo ali je uspešnost napovednega modela na vnaprej določeni učni podmnožici v skladu z našimi pričakovanji. Na primer, »uspešnost oz. točnost napovednega modela za podmnožico x mora biti nad 95 %« je lahko eden izmed takšnih testov. Priporočljiva je tudi vpeljava več testov takšnega tipa, za različne podmnožice, saj je na tak način lažje zaznati večje padce v uspešnosti napovednega modela iz iteracije v iteracijo. Na tej točki je smiselno izpostaviti tudi, da je validacijo kakovosti napovednega modela nujno potrebno opraviti pred namestitvijo modela v izvajalno okolje. [13] Pri procesu zagotavljanja kakovosti s preverjanjem uspešnosti modela pa ima velik pomen tudi izbira metrike, s katero merimo uspešnost napovednega modela. Med pogosteje uporabljanimi metrikami je zagotovo metrika točnost (angl. accuracy), ki predstavlja delež pravilno napovedanih primerkov. Zaradi enostavnosti same metrike pa lahko le-ta prikrije

pomembne informacije o uspešnosti napovednega modela. Zato je priporočljiva uporaba kombinacij metrik oz. uporaba metrik, ki bolj celovito kot metrika točnost predstavljajo uspešnost napovednega modela (npr.: F1 metrika, AUC metrika). [2] Prav tako kot pri tipični programski opremi je tudi pri napovednih modelih nujno zagotoviti možnost enostavne povrnitve modela na prejšnjo verzijo v primeru, da je zaznana nepravilnost v delovanju oz. v primeru nepričakovanega padca uspešnosti napovednega modela strojnega učenja. [13]

4 PRAKTIČNI PRIMER

Za praktični prikaz primera vpeljave razvoja napovednih modelov z uporabo razširjenega procesa CI smo si zastavili razvoj spletne aplikacije – sistem za tehnično podporo s samodejno kategorizacijo in prioretizacijo uporabniških zahtevkov. Razvit sistem naslavlja scenarij (Slika 4), kjer uporabniki preko pročelne spletne aplikacije vnesejo zahtevek tehnični podpori, ta pa je preko sporočilne vrste dostavljen zalednemu delu spletne aplikacije. Zaledni del spletne aplikacije preko REST komunicira z napovednim modelov strojnega učenja, izpostavljenim v obliki spletne storitve, ki prejet zahtevek kategorizira in mu določi prioriteto. Obdelan uporabniški zahtevek je nato posredovan na pročelni del spletne aplikacije, namenjen tehnični podpori.

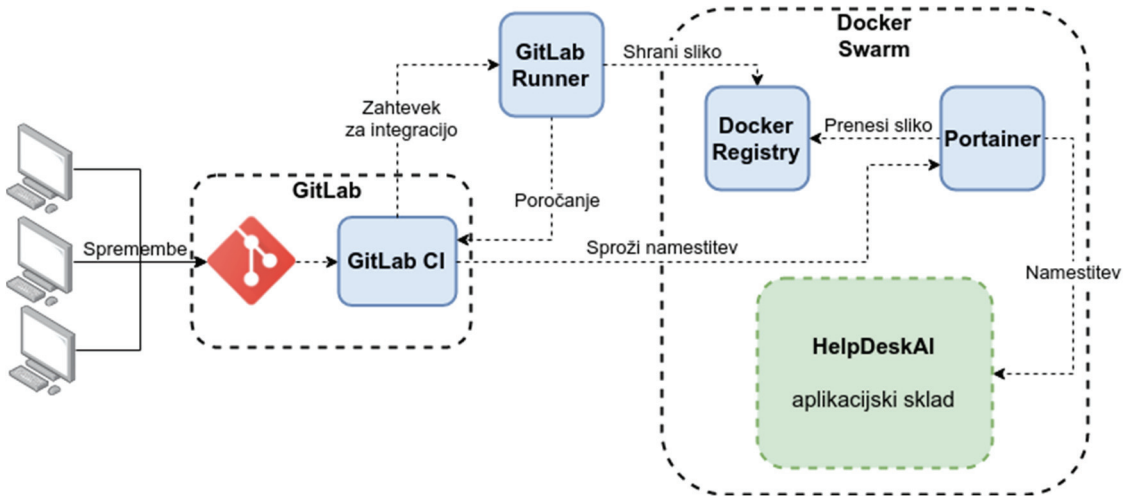
Tehnološki sklad, izbran za implementacijo praktičnega primera, sestoji iz sporočilne vrste RabbitMQ, zaledni del spletne storitve je implementiran z uporabo Java ogrodja Spring Boot, pročelna aplikacija temelji na knjižnici React, napovedna modela (en zadolžen za kategorizacijo, drugi pa za prioretizacijo) sta razvita s pomočjo Pythonove knjižnice sklearn ter servirana v obliki REST s pomočjo ogrodja Flask. Vsak izmed omenjenih delov spletne aplikacije je za namen namestitve na gruči Docker Swarm zapakiran v Docker sliko, za orkestracijo zabojnikov skrbi Portainer, za verzioniranje kode ter kot sistem za neprekinjeno integracijo in dostavo pa služi GitLab.



Slika 4: Scenarij praktičnega primera – sistem za tehnično podporo s samodejno kategorizacijo in prioretizacijo uporabniških zahtevkov.

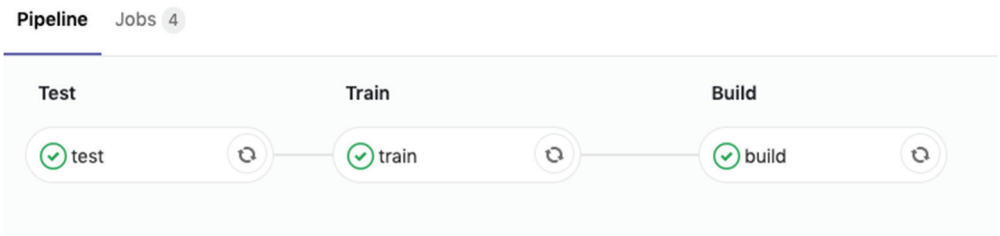
Pri tem praktičnem primeru se bomo osredotočili na razvoj in integracijo storitve, upoštevajoč smernice, ki smo jih nakazali oziroma predstavili v prejšnjem poglavju. Cilj je vpeljati razvoj napovednega modela strojnega učenja v predstavljen razširjen proces CI. Slika 5 prikazuje prilagojen koncept vpeljave razvoja napovednega modela strojnega učenja v proces samodejne neprekinjene integracije. V našem primeru je programska koda kot tudi učni podatki shranjena v sistemu za verzioniranje programske kode Git. Ob vsaki spremembi se torej sproži proces neprekinjene integracije, ki ga v našem primeru izvaja t.i. GitLab Runner. GitLab Runner je uraden GitLabov agent, namenjen za izvajanje tovrstnih nalog pri uporabi sistema GitLab, ki ima vgrajeno samodejno poročanje, kar nam omogoča sprotno spremljanje poteka procesa samodejne neprekinjene integracije. V zadnji fazi omenjen agent zgradi Docker sliko, ki napovedna modela zapakira v obliko, neodvisno od platforme in pripravljeno za takojšnje izvajanje. Sliko nato shrani v privaten register slik, integracijski sistem pa na orkestracijskem sistemu sproži zahtevek za namestitev. Kot omenjeno smo za

orkestracijski sistem izbrali Portainer, ta je zadolžen za izvajanje, orkestracijo ter posodabljanje storitev, definiranih znotraj našega aplikacijskega sklada (HelpDeskAI). V primeru, da kakšen izmed korakov integracije ni uspešen, se celoten proces zaustavi, s čimer ne pridemo do situacije, ko bi namestili nedelujočo storitev oz. storitev, ki ni v skladu z našimi zahtevami in pričakovanji.



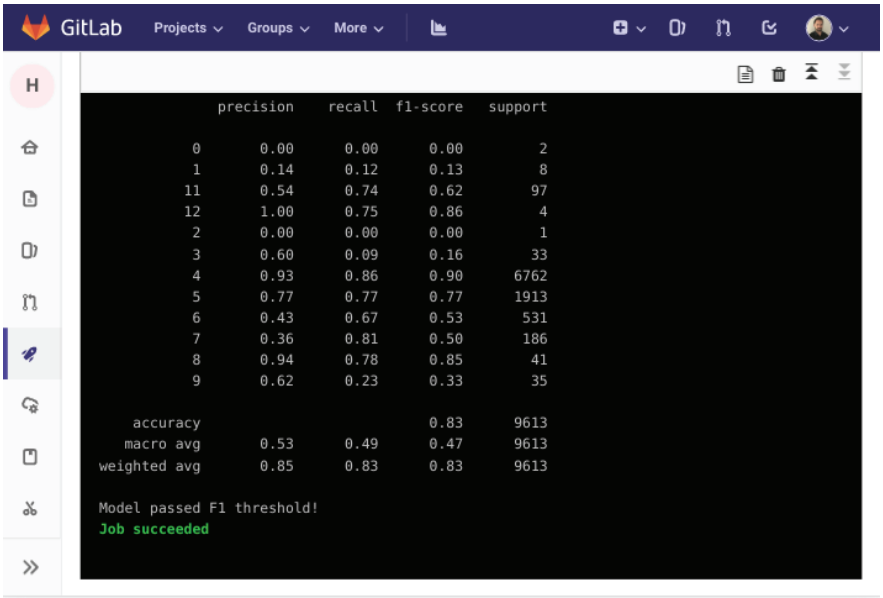
Slika 5: Proces samodejne neprekinjene integracije za razvoj napovednega modela strojnega učenja.

Sam proces integracije napovednega modela smo opravili v treh korakih (Slika 6): testiranje, učenje in gradnja. Ker so uporabljeni učni podatki bili že predhodno prečiščeni in preoblikovani v obliko, primerno za učenje napovednih modelov, smo korak pred-procesiranja izpustili, sicer bi v tipičnem primeru tak korak bil nujno potreben. V našem primeru se je tako integracija pričela s korakom testiranja, kjer smo poleg poznanih konceptov iz programskega inženirstva testirali tudi podatke. Preverjali smo predvsem nabor vrednosti posameznih značilk znotraj učne množice, mogoče pa bi bilo opraviti preverjanje distribucije testnih primerkov, preverjanje informacijskih vrednosti posameznih značilk, ipd.



Slika 6: Koraki integracije napovednega modela.

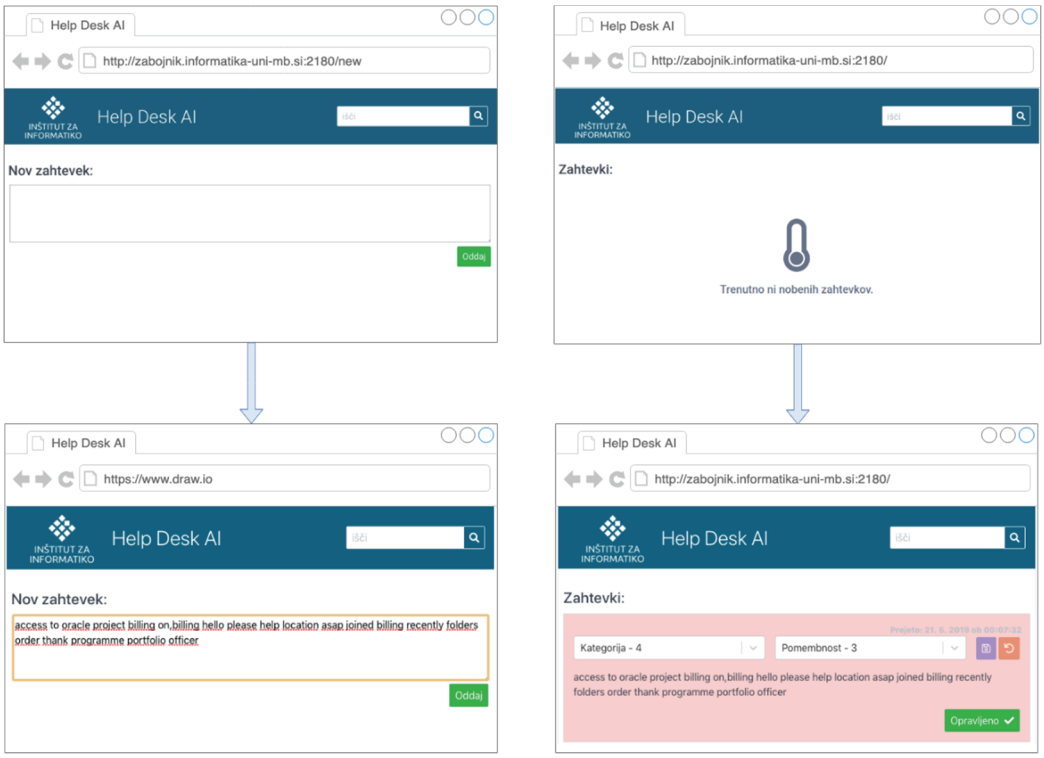
V drugem koraku smo izvedli učenje in evalvacijo obeh naučenih modelov. Kot rečeno smo učili dva specializirana modela. Naloga prvega je glede na podano vsebino zahtevka določiti kategorijo, v katero zahtevek spada, naloga drugega pa je določiti prioriteto posameznega zahtevka. Oba modela smo učili na enak način, z uporabo klasifikatorja naivni Bayes ter razdelitvijo učnih in testnih podatkov v razmerju 80 % proti 20 %. Uporabili pa smo tudi pristop grid search za iskanje optimalne nastavitve hiper-parametrov klasifikatorja. Po učenju posameznega modela smo izračunali ter izpisali tudi matriko zmede z osnovnimi vrednostmi klasifikacijskih metrik (Slika 7) kot so točnost, natančnost (angl. Precision), priklic (angl. Recall) ter F1 metriko. Uspešnost posameznega napovednega modela smo evalvirali na podlagi povprečne obtežene vrednosti F1 metrike ter določili mejo 80 %, nad katero je napoveden model dovolj uspešen za uporabo oz. namestitev. V primeru, da model ne doseže vnaprej določene meje, se faza zaključi kot neuspešna ter se tako prekine celoten proces samodejne neprekinjene integracije. Na tej točki bi veljalo izpostaviti, da bi za namene evalvacije napovednih modelov, podobno kot je za ocenjevanje kvalitete programske kode, prav prišla razna ogrodja, knjižnice in sistemi, ki bi omogočali lažjo integracijo in pregled nad rezultati evalvacije. V našem primeru je bilo namreč potrebno evalvacijo in mehanizme preverjanja in prestajanja uspešnosti implementirati ročno.



Slika 7: Matrika zmede in vrednosti izbranih metrik napovednega modela za klasifikacijo zahtevkov v kategorije.

Po uspešno prestani fazi učenja in evalvacije napovednih modelov se v fazi grajenja naučena modela s pomočjo ogrodja Flask izpostavita v obliki REST, celotna storitev pa se nato zapakira v Docker sliko, ki jo naložimo na privaten register Docker slik. Integracija je z uspešno izvedenim zadnjim korakom tako zaključena, integracijski sistem pa za potrebe namestitve izvrši zahtevek na orkestracijski sistem Portainer. Ta poskrbi za prenos najnovejše Docker slike iz registra slik ter zagon zabojnika na podlagi prenesene slike. V kolikor je zagon zabojnika uspešen, je predhodna različica storitve zabojnika zamenjana za novejšo.

Iz slike 8 so razvidni zaslonski posnetki pročelnega dela razvitega sistema za tehnično podporo. Zgornji dve sliki prikazujeta prazna pogleda za uporabnika (levo) in za člana tehnične podpore (desno). Spodaj levo je prikazan vnos zahtevka s strani uporabnika, na desni strani pa je prikazan ta isti zahtevek po opravljeni kategorizaciji in prioretizaciji s strani naših razvitih napovednih modelov.



Slika 8: Zaslonske slike spletne aplikacije za tehnično podporo.

Obstoječ sistem bi bilo smiselno razširiti s hrambo sprememb kategorij oz. stopnje pomembnosti s strani članov tehnične podpore. Hrambo teh sprememb bi lahko izrabili za pridobivanje novih oz. izboljšanih učnih podatkov, s čimer bi dodatno pripomogli k večji uspešnosti napovednega modela. V takšnem primeru bi dodatno veljalo uvesti tudi periodičen zagon cikla neprekinjene integracije napovednega modela, saj bi bilo ob pogostih spremembah učnih podatkov s strani članov tehnične podpore smiselno tudi pogosteje učiti in evalvirati napovedna modela.

5 ZAKLJUČEK

V prispevku smo na začetku predstavili osnoven koncept samodejne neprekinjene integracije pri razvoju konvencionalne programske opreme ter izpostavili ključne prednosti uporabe takšnega pristopa. V nadaljevanju smo obstoječ, uveljavljen koncept, razširili ter prilagodili za potrebe razvoja in integracije napovednih modelov strojnega učenja. Naslovili smo nekaj ključnih izzivov, ki jih prinaša vpeljava razvoja napovednih modelov, ter izpostavili morebitne smernice oziroma dobre prakse.

Predlagane smernice in dobre prakse smo predstavili tudi v obliki praktičnega primera, kjer smo z vpeljavo napovednih modelov strojnega učenja razvili sistem za tehnično podporo, ki samodejno, z uporabo napovednih modelov, kategorizira in prioretizira posamezne uporabniške zahtevke.

Izpostaviti velja, da je poleg manka uveljavljenih dobrih praks in smernic, kot je to pri programskem inženirstvu, mogoče zaznati tudi manko namenskih orodij ali knjižnic, še posebej za potrebe evalvacije napovednih modelov ter generiranja in pregleda poročil. Menimo, da bo v naslednjih letih področje integracije in evalvacije napovednih modelov ter zagotavljanje kakovosti le-teh doživelo silovit razcvet, saj potreba po rešitvah, ki naslavljajo omenjeno problematiko v gospodarstvu iz leta v leto narašča.

6 LITERATURA

[1] S. Hamdan in S. Alramouni, „A Quality Framework for Software Continuous Integration“, *Procedia Manuf.*, let. 3, str. 2019–2025, 2015.
[2] A. Herczeg, „Continuous Integration in Machine Learning Towards fully automated continuous learning“, 2018.
[3] K. Beck, „Embrance Change with Extreme Programming“, *IEEE Comput. Mag.*, št. c, str. 70–77, 1999.
[4] P. M. Duvall, S. Matyas, in A. Glover, *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.
[5] M. Fowler in M. Foemmel, „Continuous integration“, *Thought-Works) http//www. thoughtworks. com/Continuous Integr. pdf*, let. 122, str. 14, 2006.
[6] M. Erder in P. Pureur, „Continuous Architecture and Continuous Delivery“, v *Continuous Architecture*, Elsevier, 2016, str. 103–129.
[7] M. Leppänen *idr.*, „The highways and country roads to continuous deployment“, *IEEE Softw.*, let. 32, št. 2, str. 64–72, mar. 2015.
[8] A. Debbiche, M. Dienér, in R. Berntsson Svensson, „Challenges When Adopting Continuous Integration: A Case Study“, Springer, Cham, 2014, str. 17–32.
[9] M. Shahin, M. Ali Babar, in L. Zhu, „Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices“, *IEEE Access*, let. 5, št. Ci, str. 3909–3943, 2017.
[10] G. G. Claps, R. Berntsson Svensson, in A. Aurum, „On the journey to continuous deployment: Technical and social challenges along the way“, v *Information and Software Technology*, 2015, let. 57, št. 1, str. 21–31.
[11] C. Renggli *idr.*, „Continuous Integration of Machine Learning Models with ease.ml/ci: Towards a Rigorous Yet Practical Treatment“, 2019.
[12] B. Derakhshan, „Continuous Deployment of Machine Learning Pipelines“, str. 397–408, 2019.
[13] E. Breck, S. Cai, E. Nielsen, M. Salib, in D. Sculley, „The ML test score: A rubric for ML production readiness and technical debt reduction“, v *Proceedings - 2017 IEEE International Conference on Big Data, Big Data 2017*, 2018, let. 2018-Janua, str. 1123–1132.

OTS 2019
ADVANCED INFORMATION TECHNOLOGY AND SERVICES
M. Heričko in K. Kous (eds.)

University of Maribor Press

D3LEDGER: THE DECENTRALIZED DIGITAL DEPOSITORY PLATFORM FOR ASSET MANAGEMENT BASED ON HYPERLEDGER IROHA

NIKOLAI IUSHKEVICH, ANDREI LEBEDEV, ROK ŠKETA, MAKOTO TAKEMIYA

Abstract: *Blockchains organize data in a linked list of tamper-resistant blocks. Transactions are appended to the blockchain via the decision of a decentralized network. Tokenized securities provide investors with new transparency and opportunities for liquidity by decreasing times for carrying out traditional capital market operations. This allows creating a global post-trade infrastructure for digital assets: tokenized securities and security tokens. This article describes how the D3ledger project allows the tokenization of traditional securities, as well as the safekeeping of security and currency tokens from the Ethereum and Bitcoin blockchain networks, using Hyperledger Iroha as an underlying blockchain network.*

Key words: *D3ledger, Hyperledger Iroha, Bitcoin, Ethereum, digital assets, tokenization, depository*

CORRESPONDENCE ADDRESS: Nikolai Iushkevich, Soramitsu Labs, Russia, e-mail: nikolai@soramitsu.co.jp. Andrei Lebedev, Soramitsu Labs, Russia, e-mail: andrei@soramitsu.co.jp. Rok Šketa, KDD, Central Securities Clearing Corporation, Slovenia, e-mail: rsketa@kdd.si. Makoto Takemiya, Soramitsu Holdings, Switzerland, e-mail: takemiya@soramitsu.co.jp.

https://doi.org/10.18690/978-961-286-282-4.4 ISBN 978-961-286-282-4
Available at: http://press.um.si

1 INTRODUCTION

1.1 Problem

In recent years, distributed systems have received significant attention from institutions, as new blockchain-based systems have emerged that promise to provide a transition from centralized business models to decentralized. Bitcoin was the first successful implementation of a blockchain-based decentralized system to create a so-called cryptocurrency. The idea, proposed by Satoshi Nakamoto in the Bitcoin paper [1], was to create a digital asset based on cryptographic proofs to enforce digital scarcity, rather than trust. Bitcoin thus intermediates third parties, because the system as a whole is the third party, enforcing consensus constraints as a form of third-party account. Nick Szabo expressed similar ideas as requirements for so-called smart contracts, where asymmetric cryptography and tamper-evident transaction logs can be used to create automatically executing contracts, thus removing counterparty risk and reducing the need for reliance upon mediators or other third parties [2].

Smart contracts were implemented in a restricted form in Bitcoin as a non-Turing complete scripting language for output address scripts, that checks inputs for a given logical statement expressed in the script when an address is used in the input of a transaction. Later on, Ethereum was released as a platform for decentralized applications. The Ethereum platform allows the deployment of Turing-complete bytecode to a distributed network of nodes. Once deployed, the bytecode cannot be modified, however every network participant can execute and validate state changes in a key-value store. The deployed bytecode is called a smart-contract. A deployed smart contract can be triggered by sending a transaction to invoke a method and attaching a transaction fee as an incentive for a miner to include the transaction in a block.

As of May 6th 2019, there are 185,189 ERC-20 token contracts [3] and 1,460 ERC-721 token contracts [4] deployed in Ethereum – these tokens are used for buying services, gambling, speculation, and other purposes. Bitcoins, tokens used as incentives for block creators (miners) in the Bitcoin network, increased their value from 0.003 USD in Mar 2010 to 3,930 USD in February 2019 per 1 BTC [5].

The reason why Bitcoin, tokenized securities and security tokens gained their value over the last years is linked to the following properties of such assets [6]:

- asymmetric cryptography provides a solution for double-spending problem of payments with tokenized assets
- increased global acceptance rate of such assets
- scarcity of some of the assets, or limited monetary supply for assets like Bitcoin
- decentralization, and resulting improbability of a third, central party intervention to value manipulation in case of demand fluctuations

Although these factors refer mainly to phenomenon of currency tokens, they also give a clue to the rising popularity of the idea of asset-backed tokens. Tokenization of assets enables creation of value added services for investors, issuers and professional markets players, that goes far beyond what traditional book-entry-and-messaging paradigm can offer in terms of efficiency and digitization level. The direct access of all market participants into global decentralized platforms that use tokens as a tool to transfer ownership rights, and smart contracts as a tool allows a significant increase of speed, and decrease of risks and costs of settlement and asset servicing on global markets.

While this form of assets has its advantages – it is hard to apply existing business processes in capital markets to security tokens in public networks. Public blockchain networks provide participants anonymity (network peers and transaction creators), and inability to control assets in the lack of private key presence. There is a lack of governance for assets exchange processes. Infrastructural centralization of exchanges for cryptoassets and the inability to revert fraudulent or error actions (similar to recent transfer from Canadian investment fund to a Bitcoin exchange cold wallet that passed away [7]) makes the **risks**, related to investments in security tokens, Bitcoin, Ethereum, ERC20-based tokens or other currency tokens, **prevail financial appeal** for institutional investors.

1.2 Solution

This problem has been already perceived by investment funds, central securities depositories in certain jurisdictions. Technical solutions, based on distributed-ledger technology (DLT or private blockchains) for the problems mentioned above already exist in a form of proof of concept projects: e.g. project Ubin in Singapore [8], or project Jasper in Canada [9]. International Securities Services Association, comprised of participants from central securities depositories (CSDs) and central counterparty clearing houses (CCPs), has already proposed recommendations for crypto-assets infrastructure [10]. D3ledger solution has been designed with these recommendations in mind, and provides a solution which is **horizontally scalable**. *D3ledger* (Decentralized Digital Depository) is a decentralized financial infrastructure that enables market participants to safely work with crypto and traditional financial assets. *D3ledger* uses distributed ledger technology to provide safekeeping, settlement, and asset servicing functions accessible without any intermediaries. Such architecture guarantees the high level of robustness and reliability. Unlike exiting DLT initiatives, *D3ledger* is going to operate across several jurisdictions, providing horizontally scalable solution to institutional investors and minimizing risks of political and infrastructural intervention in its architecture [11].

This article focuses on technical aspects of *D3ledger* project: how securities are tokenized on Hyperledger Iroha platform, that acts as a intermediary blockchain; and how *D3ledger* operates with security tokes from Bitcoin and Ethereum networks via a two-way peg.

2 HYPERLEDGER IROHA: DLT-PLATFORM FOR ASSET AND IDENTITY MANAGEMENT

Hyperledger Iroha [12] is unique implementation of a blockchain platform governed by Hyperledger and hosted by The Linux Foundation. Hyperledger Iroha is written in C++, incorporating chain-based Byzantine Fault Tolerant consensus algorithm, called Yet Another Consensus [13]. Peers in Iroha network are universal — which means that every peer acts as a validator and has the same vote weight as any other peer during network consensus.

2.1 Iroha API: Commands and Queries

The manipulation of accounts and digital assets is supported with a set of commands and queries. For example, `CreateAsset` command allows creation of a new asset, `AddAssetQuantity` command increases monetary supply of a given asset; `GetAccountAssets` query will return the state of assets linked to the account [14]. Commands are atomic state-changing actions, that can be put in a single transaction, whereas Queries allow Iroha clients to get a snapshot of peer state, called World State View – or in a simple words to get: balance of account, history of transactions, and account information.

2.2 Relational state

In order to perform actions, expected from Iroha API, peers construct relational state over the data, stored in blockchain. The state is stored in PostgreSQL and used during the state construction and subsequent validation of API calls [14]. In a couple of sections below, we give an in-depth example of how business objects are represented.

2.2.1 *Account*

Iroha entity which is able to perform a specified set of actions. Each account belongs to one of the existing domains. An account has some number of roles (can be none) associated. The roles are just a collection of permissions.

account

Attribute	Datatype
account_id	VARCHAR(288)
domain_id	VARCHAR(255)
quorum	INT
data JSONB	JSONB

2.2.2 *Asset*

Any commodity or value represented as an asset. Each asset relates to one of the existing domains in Iroha.

asset

Attribute	Datatype
asset_id	VARCHAR(288)
domain_id	VARCHAR(255)
precision	INT
data	JSON

account_has_asset

Attribute	Datatype
account_id	VARCHAR(288)
asset_id	VARCHAR(288)
amount	DECIMAL

2.3 Role-Based Access Control

Decentralized permission model, based on role-based access control, allows Iroha accounts to perform state-changing actions and retrieve information only if it is permitted to an assigned role. This allows separation of accounts related to tokenization, audit, and asset transfers. Initial roles and permissions [14] are set in the genesis (initial) block of the blockchain. Roles and their assignment to users might be dynamic if corresponding changes exist in the genesis block.

2.4 Universal Peer Role

Unlike some of the DLT frameworks, in order to establish and maintain the network of Iroha nodes, system administrators they only need to deploy Iroha peer. Every peer has the same rank and responsibilities for validation, which makes the system decentralized [15].

2.5 Multi-signature transactions

Transactions in Iroha require attached signatures since Iroha peers use asymmetric crptography for validation of incoming messages. Some of the transactions may have more than a single signature attached – they are called multisignature transactions. Multisigniture is a powerful feature that allows decentralization of transaction generation.

Required number of signatures is associated with 'quorum', a number associated to an Account that defines minimum number of signatures that are required for transaction to be successfully validated. By increasing the quorum, Iroha clients can improve security of their accounts. Peers only process transactions, containing M out of N signatures, where M is the quorum number and N is the total amount of public keys, associated with the account. By default, $M, N = 1$.

However, Iroha clients do not have to always form and send transactions, containing exactly M signatures as peers share transactions [14] using a Gossip protocol [16]. Therefore Iroha clients may send partially-signed transactions to any Iroha peer in the network, expecting that in the end their transactions will gather enough signatures – and will be passed to the next consensus round for validation.

2.6 Transaction Batches

Transactions batch is a feature that allows sending several transactions at once to Iroha peer, preserving their order. A batch can contain transactions created by different accounts. In this case they have to be signed by every account in order to pass validation step. Partially-signed transactions, that don't contain enough signatures for validation can be retrieved from MST endpoint, similarly to a multi-signature transaction.

Each transaction within a batch includes batch meta information. Batch meta contains batch type identifier (atomic or ordered) and a list of reduced hashes of all transactions within a batch. The order of hashes prescribes transactions sequence.

3 MANAGEMENT OF SECURITIES: IROHA FEATURES APPLIED TO D3LEDGER NODES

D3ledger nodes provide different services, namely transaction validation using custom business logic and notary that tracks events on external public networks: Bitcoin and Ethereum. They are responsible for registering facts of security creation, distribution, and fulfillment of corporate actions. This section describes actions, possible in *D3ledger* network, and how they are made on Iroha peer level.

3.1 Tokenization

Tokenization process is a process of asset issuance, where asset represent specific security, and possession of any asset by network participant represents ownership rights for a given security. With respect to traditional securities *D3ledger* is particularly good for debt and equity securities. For derivative securities such as futures, options and similar instruments *D3ledger* network has to have connection to data source about conditions that trigger derivative contract condition. Neither *D3ledger* node nor Iroha node are limited in term of architecture – in fact Iroha was used in a project for weather derivatives [17], where Iroha was used for interest rate payouts based on fluctuations of weather conditions.

Tokenization process essentially consists of a single transaction with `CreateAsset` command. This command has to specify:

- `asset name` – the identifier of a security
- `domain` – domain, or managing organization, associated with the asset and responsible for corporate actions over the security
- `precision` – degree of allowed asset fractionalization

Asset creator has to possess corresponding permissions in their role set. After this transaction is finalized (written in blockchain) the asset is created and now can be distributed in the network.

If a governing organization would like to change supply of a given asset or security they may initiate transaction with any of the following commands:

- `AddAssetQuantity` – issuance of additional tokens to the governing Iroha account (money creator)
- `SubtractAssetQuantity` – burning tokens (taking assets from circulation) from the governing Iroha account (money creator)

It is worth mentioning that Iroha permission model allows limiting asset supply upper and lower boundaries if a particular Iroha network does not have assets creation and quantity modification rights in genesis block. As a result there isn't any role assigned to modify asset supply. It might be useful to tokenize a fraction of possessed assets that cannot modify their supply amount in order to realize economic models similar to Bitcoin network.

3.2 Delivery-Versus-Payment Settlements

Delivery versus payment (DVP) is a securities industry settlement method that guarantees transfer of securities only after payment has been made. DVP stipulates that the buyer's payment for securities must be made prior to or at the same time as the delivery of the security [18].

In case of Iroha this settlement might be done with a transaction batch, where sender (party A) puts:

- `Delivery` – transaction with a transfer from party A to party B
- `Payment` – transaction with a transfer from party B to party A
- `signature(s)` of party A – the batch can be partially-signed or fully-signed

This transaction batch is sent to any Iroha peer, which processes the batch similarly to any other partially-signed transaction: party B can retrieve information about batches that require signing from any peer and similarly can send their signatures to any Iroha peer in any arbitrary order.

When both parties signed-off the batch it is processed during the next consensus round. If none of the validation rules are violated (i.e. all parties have delivery and payment as payment for securities is not made prior to the delivery), the atomic swap, or DVP settlement operation is done and blockchain is appended with a block containing this batch of transactions.

3.3 Asset transfer

In order to use *D3ledger* as a payment network to perform actions related to capital markets payments Iroha features an atomic action of asset transfer.

Such transaction may have single or more `TransferAsset` commands, containing:

- `Source account ID` – ID of the account to withdraw the asset from
- `Destination account ID` – ID of the account to send the asset to
- `Asset ID` of party A – ID of the asset to transfer
- `Description` of party A – message to attach to the transfer
- `Amount` of party A – amount of the asset to transfer

Straightforwardness of this API allows its extension and implementation of arbitrary validation rules over its description message or account information of sender or receiver.

4 D3LEDGER INTERCHAIN METHOD: TWO-WAY PEG WITH A FEDERATION OF NOTARIES

In order to provide a seamless interface between *D3ledger* and existing public networks with security tokens, *D3ledger* node features notary services, that monitor actions happening in a public network and perform a group decision on asset tokenization and transfers. One of the main requirements for the system design was atomicity of transfers between public blockchains such as Ethereum or Bitcoin and *D3ledger*. We define atomic payment as confirmed if and only if both sides have completed their payments, and the corresponding transactions are validated and stored in the ledgers. The technical problem to be addressed were probabilistic consensus algorithms, because several forks, or different versions of the chain could exist at one point of time.

The proposed solution for the atomicity requirement is to minimize the probability that the payment transaction will only appear in a fork, and to use an intermediate ledger with a deterministic consensus algorithm.

Two-way peg [19] is an exchange protocol between a main chain and a secondary chain and vice versa, which does not require to use a third party. In other words, the exchanges between public networks should not include third parties. A two-way peg is a voting system, where votes are defined by the consensus algorithm of a particular system, such as digital signatures or hashing power.

Multi-signature federation of notaries [20] is a group of identities in blockchains which control a single multi-signature account in each of the chains.

In the next subsections we describe the general pipelines for *D3ledger* use cases.

4.1 Cross-chain Deposit

The following steps describe the case when an asset owner wants to deposit tokens already owned in Bitcoin or Ethereum to the *D3ledger* system in order to use them to trade for other tokens.

- Create a new user account in *D3ledger* system by generating their keys (Registration service)
- There will be an empty account — so we need to deposit assets
- D3 client deposits tokens to a specified address in Ethereum or Bitcoin
- Wait until this transaction is finalized
- Each D3 notary certifies the fact of transfer by a multi-signature transaction
- When everyone certified — the tokens are created in Iroha and user can check the balance in web app

4.2 Cross-chain Withdrawal

The withdrawal process, when an asset owner wants to withdraw tokens to their native trade systems to use them for payments or safekeeping outside D3ledger system, is described in the next steps.

- Create a withdrawal request in D3 to get money back to Ethereum or Bitcoin from Iroha
- This request is noticed by Withdrawal service
- This service goes over each notary and asks them to present a proof (signed approval of withdrawal)
- When all proofs are collected — Withdrawal service asks native chain contract to check proofs and send money back
- Contract sends tokens to a specified address in request

4.3 Settlement

The steps below describe the algorithm when an asset owner wants to exchange tokens in *D3ledger* with other participants of the system to their tokens atomically so that the asset owner can perform settlements for diversification of owned securities for hedging purposes.

- Create a settlement request by specifying a counterparty and amount
- This request is stored in pending transaction storage. This pending transaction is propagated in notary network
- Another D3 client checks their list of pending exchanges and accepts or declines it
- A transaction signed by both parties is sent for processing by notaries
- A corresponding batch transaction is stored in the ledger and both parties see it in their accounts

5 CONCLUSION

D3ledger network is simple, yet effective financial infrastructure solution for institutional investors. This service increases liquidity of assets by featuring horizontal scalability across jurisdictions that have agents connected to *D3ledger* network through *D3ledger* nodes. The underlying technology for asset tokenization supports debt, equity, and derivative securities by using Hyperledger Iroha API. Delivery-versus-payments are easy to perform and can be extended to any realization of escrow payments via Iroha's transaction batch. Two-way peg, that is used for tokenization of security tokens, currently supports Bitcoin and Ethereum networks, and can potentially support other chains, featuring multi-signature accounts. The code for *D3ledger* is open-source and is available at https://github.com/d3ledger/.

6 REFERENCES

[1] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009.

[2] Nick Szabo. Formalizing and securing relationships on public networks. First Monday – Peer-reviewed Journal on the Internet, 2(9), September 1997.

[3] etherscan.io. Etherscan token tracker. URL: https://etherscan.io/tokens, 2019.

[4] etherscan.io. Etherscan erc-721 token tracker page. URL: https://etherscan.io/tokens-nft, 2019.

[5] coinmarketcap.com. Capitalization of bitcoin. URL: https://coinmarketcap.com/currencies/bitcoin/, Apr 2019.

[6] Marshall W. van Alstyne. Why bitcoin has value. Commun. ACM, 57(5):30–32, 2014.

[7] Daniel Shane. A crypto exchange may have lost 145 million usd after its ceo suddenly died. CNN. URL: https://edition.cnn.com/2019/02/05/tech/quadriga-gerald-cotten-cryptocurrency/index.html, 2019.

[8] Singapore Exchange, Monetary Authority of Singapore, and Deloitte. Delivery versus payment on distributed ledger technologies project ubin. 2018.

[9] Bank of Canada, TMX Group, Payments Canada, Accenture, and R3. Jasper phase iii: Securities settlement using distributed ledger technology. 2018.

[10] International Securities Services Association ISSA. Infrastructure for crypto-assets: A review by infrastructure providers. 2018.

[11] Artem Duvanov. D3ledger blog: On vertical and horizontal blockchains. D3ledger.com, URL: https://d3ledger.com/verticalvshorizontal; 2019:

[12] Linux Foundation. Hyperledger iroha. URL: https://www.hyperledger.org/projects/iroha, 2019.

[13] Fedor Muratov, Andrei Lebedev, Nikolai Iushkevich, Bulat Nasrulin, and Makoto Takemiya. Yac: Bft consensus algorithm for blockchain. 2018.

[14] Hyperledger Iroha Documentation. Api documentation. URL: https://iroha.readthedocs.io/, 2019.

[15] Mari Eagar. What is the difference between decentralized and distributed systems? Medium. URL: https://medium.com/distributed-economy/what-is-the-difference-between-decentralized-and-distributedsystems-f4190a5c6462, Nov 2017.

[16] Demers Alan and Greene Dan. Epidemic Algorithms for Replicated Database Maintenance. Association For Computing Machinery, proceedings of the sixth annual acm symposium on principles of distributed computing edition, 1987.

[17] Mondovisione.com. Soramitsu announces iroha: A proposal to the linux foundation's hyperledger. http://www.mondovisione.com/media-and-resources/news/soramitsu-announces-iroha-a-proposal-to-thelinux-foundations-hyperledger-pr/, 2016.

[18] Investopedia. Delivery versus payment (dvp): What guarantees do you have? URL: https://www.investopedia.com/terms/d/dvp.asp, 2019.

[19] Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timón, and Pieter Wuille. Enabling blockchain innovations with pegged sidechains. URL: https://blockstream.com/sidechains.pdf, page 72, 2014.

[20] Johnny Dilley, Andrew Poelstra, Jonathan Wilkins, Marta Piekarska, Ben Gorlick, and Mark Friedenbach. Strong federations: An interoperable blockchain solution to centralized third party risks. CoRR, abs/1612.05491, 2016.

OTS 2019
SODOBNE INFORMACIJSKE TEHNOLOGIJE IN STORITVE
M. Heričko in K. Kous (ur.)

Univerzitetna založba
Univerze v Mariboru

UPORABA PLATFORME IOTA ZA ZBIRANJE IOT PODATKOV

VID KERŠIČ, MUHAMED TURKANOVIĆ

Povzetek: *Tehnologija veriženja blokov (ang. blockchain) kot del tehnologije porazdeljene knjige (ang. distributed ledger technology - DLT) ima prednosti kakor tudi slabosti. Poglavitna slabost je razširljivost, saj je tehnologija neprimerna za sočasno procesiranje velike količine podatkov. Omenjeno pomanjkljivost naslavljajo nove generacije platform DLT. Primer takšne je platforma IOTA, ki je osredotočena na domeno Interneta stvari in z inovativnim pristopom naslovi problem razširljivosti. V prispevku bomo predstavili podrobnosti platforme IOTA in njene poglavitne prednosti. Izpostavili bomo spremenjeno podatkovno strukturo, ki ne temelji na med seboj povezanih skupnih transakcij - blokih, temveč na usmerjenem acikličnem grafu transakcij, imenovan Tangle. Prav tako bomo predstavili strukturo porazdeljenega omrežja in algoritem porazdeljenega soglasja na katerem temelji. Osredotočili se bomo na predstavitev uporabe platforme na primeru zbiranja in sledenja senzorskih podatkov s pomočjo protokola Masked Authenticated Messaging (MAM). Opisali bomo različne načine uporabe slednjega protokola in njihove praktične primere. Zaključili bomo z analizo prednosti in slabosti, ki jih platforma ponuja, in pregledom trenutnega stanja razvoja IOTA omrežja.*

Ključne besede: *IOTA, DLT, IOT, senzorski podatki, MAM.*

NASLOVA AVTORJEV: Vid Keršič, študent, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Blockchain Lab:UM, Maribor, Slovenija, e-pošta: vid.kersic@student.um.si. dr. Muhamed Turkanović, docent, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Blockchain Lab:UM, Maribor, Slovenija, e-pošta: muhamed.turkanovic@um.si.

https://doi.org/10.18690/978-961-286-282-4.5 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 UVOD

Tehnologija veriženja blokov (ang. blockchain) kot del tehnologije porazdeljene knjige (ang. distributed ledger technology - DLT) je od svojega nastanka leta 2008 pokazala številne prednosti kakor tudi slabosti [1]. Zaradi slednjih so se razvile nove generacije platform DLT, ki jih poskušajo na inovativen način odpraviti in pri tem ne vplivati negativno na prednosti tehnologije. Ena izmed poglavitnih slabosti je razširljivost, saj je tehnologija neprimerna za sočasno procesiranje velike količine podatkov [2]. Omenjena slabost preprečuje uporabo tehnologije v primerih, kjer uporabniki omrežja pošiljajo veliko število transakcij v zelo kratkem časovnem intervalu. Omenjen primer se pojavi v domeni Interneta stvari (Internet of things - IOT), za katero je značilna velika količina podatkov, generirana iz najrazličnejših virov (naprav in senzorjev) v kratkih časovnih intervalih. Primer platforme, temelječe na DLT, ki rešuje omenjen problem, je IOTA, ki z inovativnim pristopom naslovi problem razširljivosti.

V prispevku bomo predstavili platformo IOTA. Osredotočili se bomo na posebnosti le te, izpostavili poglavitne prednosti v primerjavi z drugimi DLT platformami in predstavili uporabo platforme za primer zbiranja in sledenja senzorskih podatkov s pomočjo protokola Masked Authenticated Messaging (MAM) [3]. Opisali bomo različne načine uporabe slednjega protokola in njihove praktične primere. Zaključili bomo z analizo prednosti in slabosti, ki jih platforma ponuja, in pregledom trenutnega stanja razvoja IOTA omrežja.

2 PLATFORMA IOTA

IOTA je platforma, ki je bila namensko razvita za domeno IOT in za podporo pri komunikaciji med elektronskimi napravami (ang. machine to machine – M2M). Njeni avtorji David Sønstebø, Sergey Ivancheglo, Dominik Schiener in Serguei Popov izpostavljajo, da se bo število IOT naprav do leta 2020 povečalo na 20.4 milijarde, pri čemer bo nujno potreben protokol, ki bo omogočal njihovo medsebojno komunikacijo in hkrati samostojno izmenjavo denarnih oz. materialnih vrednosti [4]. Platforma hkrati podpira tudi običajno plačevanje s kriptovaluto, podobno kot ostale platforme, temelječe na tehnologiji veriženja blokov.

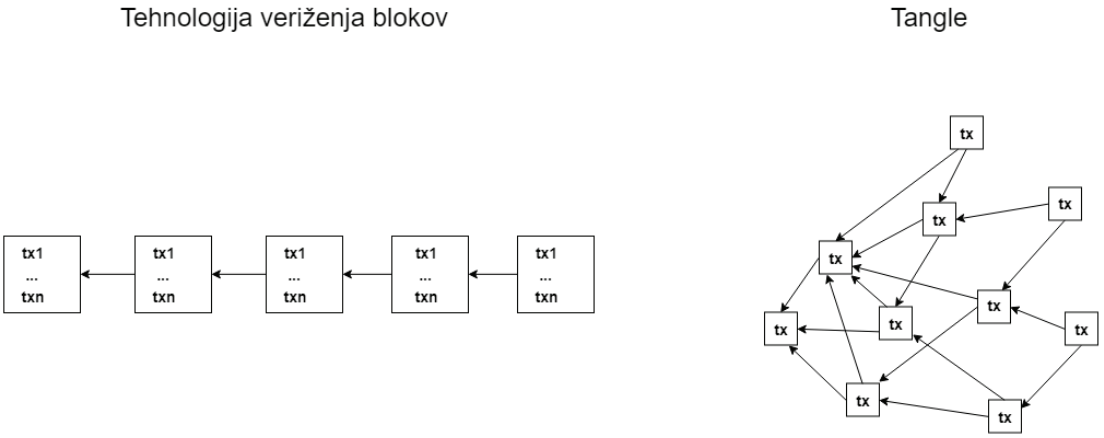
2.1 Arhitektura

Arhitektura omrežja IOTA je v veliko pogledih podobna ostalim omrežjem verig blokov. Omrežje je decentralizirano in sestavljeno iz medsebojno povezanih vozlišč, imenovanih IOTA Reference Implementation - IRI [5]. Ta vozlišča so poljubno povezana med seboj, pri čemer si skupaj delijo javno porazdeljeno glavno knjigo transakcij (ang. ledger). Prav tako preverjajo veljavnost transakcij, izvajajo algoritem izbire transakcij za potrjevanje (razloženo kasneje) in oddajajo nove ter prejete transakcije sosednjim vozliščem. Trenutno so v zagonu tri globalna omrežja: glavno (ang. mainnet), razvijalsko (ang. devnet) in testno (ang. spamnet) omrežje [6]. Bistvena razlika, ki loči platformo IOTA od drugih, je struktura porazdeljene glavne knjige transakcij. Medtem ko je ta pri tehnologiji veriženja blokov strukturirana iz zaporedja med seboj povezanih blokov transakcij, je pri platformi IOTA strukturirana v obliki usmerjenega acikličnega grafa (ang. directed acyclic graph - DAG) transakcij imenovanem Tangle [7]. Slika 1 predstavlja slikovito primerjavo med omenjenima podatkovnima strukturama.

Transakcije so v DAG predstavljene kot vozlišča povezanega grafa. Med njimi so usmerjene povezave, ki potekajo od transakcije, ki je na novo dodana v graf (glavno knjigo), do dveh že shranjenih transakcij, s čemer ju nova transakcija potrjuje [7]. Iz tega sledi usmerjenost grafa. Pomemben pojem pri Tangle-u je tudi neposredno in posredno sklicevanje transakcij, saj igra pomembno vlogo pri mehanizmu porazdeljenega soglasja omrežja [7]. Transakcija se neposredno sklicuje na drugo transakcijo, če obstaja neposredna usmerjena povezava do nje. Po drugi strani se transakcija posredno sklicuje na drugo transakcijo, če obstaja usmerjena pot do nje preko drugih transakcij. Zaradi nespremenljivosti že obdelanih transakcij se pri sklicevanju ne more zgoditi cikel, iz česar sledi acikličnost grafa.

Zaradi specifične podatkovne strukture (DAG), omrežje IOTA ne vsebuje t. i. klasičnih rudarjev, ki bi potrjevali skupke oz. t. i. bloke transakcij, kot jih klasična omrežja tehnologij veriženja blokov. Porazdeljeno soglasje omrežja IOTA temelji na tem, da vsaka nova transakcija potrdi dve prejšnji transakciji, kar pomeni, da se neposredno na le-te tudi sklicuje [7]. V izogib potencialnim zlorabam omrežja, vključuje platforma IOTA prav tako algoritem dokaza o delu (ang. Proof-of-Work - PoW), ki je sicer manj potraten kot v ostalih klasičnih

protokolih in omrežjih verig blokov, vendar se uporablja zgolj zaradi preprečitev nezaželenih transakcij (ang. spam), saj je za vsako transakcijo potrebno izvesti določen proces dokaza o delu [8]. Omenjeni algoritem PoW se izvaja lokalno pri uporabniku, ki želi izvesti transakcijo, pri čemer se lahko opcijsko izvajanje prenese na vozlišče omrežja [9].



Slika 1: Struktura porazdeljene glavne knjige pri tehnologiji veriženja blokov in platformi IOTA [10].

Pred opisom algoritma izbire transakcij za potrjevanje, je potrebno še definirati dva pojma: (1) teža (TeTr) in (2) kumulativna teža transakcije (KTeTr) [7]. TeTr je številčna vrednost, ki je proporcionalna količini dela, ki jo je uporabnik opravil z algoritmom PoW, z namenom izdajanja nove transakcije. KTeTr določene transakcije je vsota TeTr in vseh TeTr transakcij, ki se neposredno ali posredno sklicujejo nanjo.

Uporabljen algoritem izbire transakcij se imenuje Markov chain Monte Carlo (MCMC), ki temelji na verjetnostni porazdelitvi [11]. Najpomembnejši parameter algoritma je KTeTr. Algoritem se prične pri zadnji prelomnici (posebna vrsta transakcije, razložena v poglavju 2.3) in se nadaljuje v smeri novejših transakcij [12]. Pri vsaki transakciji se mora algoritem odločiti, katera bo naslednja transakcija v sprehodu oz. v katero smer bo nadaljeval. To se odloči glede na verjetnostno porazdelitev, ki je odvisna od KTeTr transakcij [7]. Ta postopek oz. sprehod po grafu se ponavlja do robnih novih transakcij (ang. tips). Algoritem teži k temu, da so na koncu izbrane novejše transakcije, saj se tako omrežje lažje in učinkoviteje širi. Uporaba omenjenega algoritma ni eksplicitno zahtevana, saj lahko upravljalec vozlišča izbere poljuben algoritem, vendar je zaradi omenjenih razlogov njegova uporaba priporočena [13].

Posebnost platforme je tudi uporaba ternarnega številskega sistema [14]. Za razliko od binarnega sistema, kjer je osnovna enota bit, ki zaseda dve različni vrednosti (0 ali 1), je pri tem osnovna enota t. i. trit, ki zaseda tri različne vrednosti in sicer -1, 0 in 1 (uravnotežen ternarni sistem). Trije triti so enaki enemu t. i. trajtu, ki lahko ima 27 različnih vrednosti. Teh 27 vrednosti je predstavljenih z veliki črkami angleške abecede (A - Z) in številko 9. Vsi podatki v transakciji so predstavljeni v tem številskem sistemu[1].

Vsak uporabnik, ki želi uporabljati omrežje, torej pošiljati transakcije, potrebuje zasebni ključ oz. v primeru platforme IOTA je to t. i. seme (ang. seed) [15]. To je sestavljeno iz 81 trajtov, kar pomeni, da obstaja 3^{81} različnih semen. Vsakemu uporabniku pripada unikatno seme in če tega uporabnik izgubi, izgubi tudi vse kriptožetone vezane na to seme. Iz semena so generirani tudi vsi zasebni ključi, ki se uporabljajo za potrebe digitalnega podpisovanja transakcij in javni naslovi, kateri so uporabljani za prejemanje transakcij. Javni naslovi so tudi velikosti 81 trajtov. Pri semenu in javnih naslovih je velikokrat na koncu dodanih 9 trajtov, ki se uporabljajo izključno za preverjanje pravilnosti (ang. checksum).

Transakcije so povezane v snope (ang. bundle), ki vsebujejo eno ali več transakcij in vsakemu pripada unikatna zgostitev (ang. hash) [16]. Združevanje transakcij v snope ima pomembno vlogo v protokolu MAM in pri prenosu kriptožetonov. Pri slednjem ločimo tri vrste transakcij: vhodne, izhodne in meta transakcije. V vhodnih transakcijah so navedene vrednosti in naslovi pošiljatelja, v izhodnih transakcijah vrednosti in naslovi prejemnika ter v meta transakcijah digitalni podpis pošiljatelja. Vsaka transakcija ima enako strukturo. Vsebuje

[1] Primer naslova IOTA:
ZIM9QPENUCRAGYOXOAJHDSRABOUMIBFJNZBPLU9UXCIMLMYUCHQXZQIQURNRZCIJZPXFBTWKHLLRTBM9CKSHUMWZXW

zgostitev, ki se izračuna iz podatkov naslova, TeTr, oznako (ang. tag), časovno značko (ang. timestamp), indeks transakcije v snopu, zadnji indeks (število transakcij v snopu), zgostitev snopa, enkratnik (ang. nonce), ki predstavlja rezultat algoritma PoW in sporočilo. Vsa polja imajo omejeno velikost. Za nas je pomembno polje sporočilo, ki ima velikost omejeno na 2187 trajtov. Za vsako polje velja, da je zapolnjeno s trajtom 9, v primeru, ko je vrednost manjša od velikosti polja. Posebnost meta transakcij je, da niso namenjane prenosu kriptožetonov, kar pomeni, da obstajata dva tipa transakcij: (1) transakcije prenosa vrednosti in (2) brez-vrednostne transakcije [16]. Pri slednjih je vrednost v polju *vrednost* enaka 0 in se uporabljajo zgolj za prenos sporočila, ki se nahaja v polju sporočilu. Takšne transakcije se lahko uporabljajo za pošiljanje poljubnih podatkov (npr. pošiljanje senzorskih podatkov v protokolu MAM). V primeru vhodnih transakcij se v polje sporočilo zapiše digitalni podpis pošiljatelja. Ker ima polje omejeno velikost in jo digitalni podpis preseže, se le-ta nadaljuje v naslednjih meta transakcijah istega snopa.

2.2 Masked Authenicated Messaging (MAM)

Brez-vrednostne transakcije najdejo svojo uporabnost pri pošiljanju surovih podatkov brez finančne vrednosti. Kot v ostalih omrežjih verig blokov, se transakcija pošlje na naslov, ki je v njej naveden. Vendar v domeni IOT, kjer naprave oz. senzorji pošiljajo podatke na določen časovni interval (npr. vsako sekundo) in so si ti podatki med seboj povezani, omenjene transakcije niso zelo učinkovite, saj ne moremo iz njih dobiti sosledja oz. zaporedja transakcij. Velikokrat je tudi zaželeno, da so ti tokovi podatkov zašifrirani in namenjeni le določenim osebam. V ta namen se v sklopu platforme IOTA uporablja protokol MAM.

Protokol MAM je komunikacijski protokol namenjen verižnemu pošiljanju podatkov, pri čemer so sporočila digitalno podpisana s strani pošiljatelja in po potrebi tudi šifrirana [3]. Protokol deluje decentralizirano brez posrednika in hkrati zagotavlja integriteto ter zasebnost sporočil. V ozadju uporablja običajne brez-vrednostne transakcije, v katerih je v polju *sporočilo,* shranjena posebna podatkovna struktura, ki vsebuje podatke s pomočjo katerih poteka sledenje verigi sporočil MAM. Tako kot običajne transakcije so tudi ta sporočila poslana na določene naslove IOTA. Ker so vse transakcije na Tangle-u dostopne in na voljo za pogled že preden so potrjene, lahko sporočila MAM in njihove vsebovane podatke pridobimo takoj. Velikost sporočila, ki ga je moč poslati je neomejena, vendar je za večje velikosti potrebnih več transakcij, zaradi česar posledično poteka branje toka sporočil zelo počasneje.

Protokol MAM ponuja tri načine uporabe: javni, zasebni in omejeni [3]. Načini se med seboj razlikujejo le v načinu šifriranja naslovov in podatkov. Pred opisom vsakega načina je potrebno definirati podatkovno strukturo sporočil MAM.

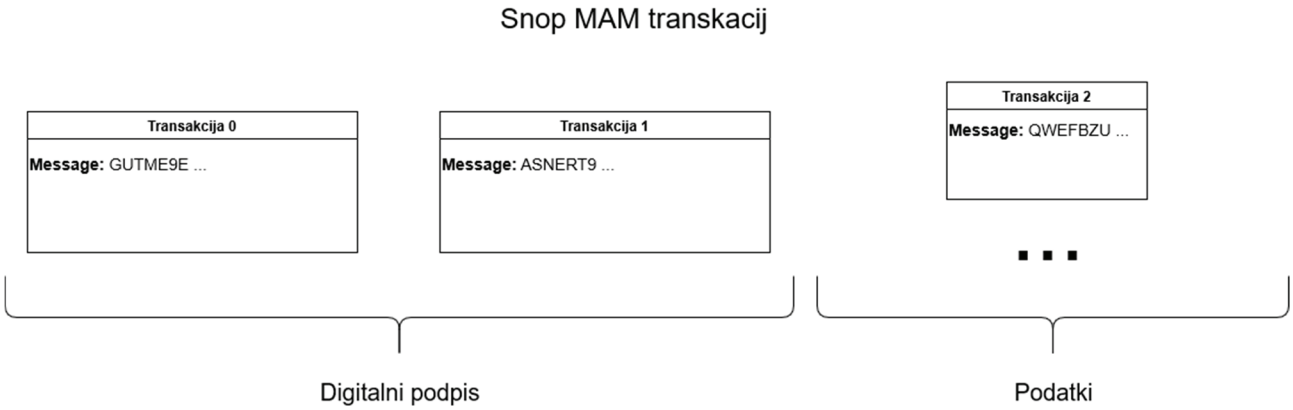
Protokol MAM uporablja za podpisovanje sporočil Merklova drevesa (ang. Merkle tree based signature scheme) [17]. Koren drevesa je uporabljen kot identifikator kanala in vsako sporočilo vsebuje tudi koren naslednjega drevesa. To pomeni, da se toku podatkov sledi preko korenov Merklovih dreves. Ker se v sporočilu nahaja le naslednji koren, lahko toku podatkov sledimo le naprej, medtem ko nazaj ni mogoče. Koren drevesa je pridobljen z digitalnim podpisom lastnika sporočila in uporabo zgoščevalnih funkcij. Pri branju toka se prvo preveri lastnikov podpis. Če je le-ta veljaven, se sporočilo dešifrira, sicer se zavrže. Koren drevesa je med drugim uporabljen tudi za šifriranje podatkov.

Objekt sporočila MAM lahko primerjamo s sintakso JSON (JavaScript Object Notation) [18]. Vsak objekt vsebuje štiri ključe: *stanje*, *koristni tovor* (ang. payload), *koren* in *naslov* [19]. Pod ključem *stanje* so shranjeni stranski ključ, naslednji koren, število sporočil oz. listov drevesa, način uporabe in ostali podatki. Vsebino (*koristni tovor*) pridobimo tako, da se dejanski podatki sporočila in nekateri podatki o Merklovem drevesu zašifrirajo s korenom drevesa. Pod ključem *koren* se nahaja koren Merklovega drevesa in pod ključem *naslov,* naslov IOTA. Kot pri običajnih transakcijah so tudi tukaj vsi podatki kodirani v ternarnem sistemu.

Poenostavljena struktura transakcij MAM je sledeča: na naslov, definiran v objektu MAM, se pošlje snop transakcij. Prvi dve transakciji v snopu predstavljala lastnikov digitalni podpis. Za tem sledi še vsaj ena transakcija (v primeru večje količine podatkov jih je lahko več). V teh transakcijah je v polju *sporočilo* zapisan *koristni tovor* objekta MAM. Slika 2 prikazuje snop MAM transakcij.

Pri javnem načinu uporabe imajo vsi udeleženci omrežja dostop do sporočil. Za branje toka sporočil je potreben le naslov IOTA, ki je v tem primeru enak korenu drevesa. Če poznamo koren drevesa, lahko dešifriramo sporočilo MAM in preberemo podatke. Ta način uporabe je primeren, ko so podatki javno dostopni, torej na voljo vsem uporabnikom omrežja.

Zasebni način uporabe se uporablja, kadar želi imeti samo lastnik sporočil dostop do teh. Pri tem načinu je naslov IOTA enak zgostitvi korena drevesa. To pomeni, da za dešifriranje ni dovolj zgolj poznavanje naslova, ampak je potrebno tudi poznati koren. Tega pozna le avtor sporočila oz. lastnik toka podatkov.



Slika 2: Snop MAM transakcij [20].

Pri omejenem načinu uporabe imajo dostop do sporočil vsi udeleženci omrežja, s katerimi lastnik sporočil deli stranski ključ. Naslov se pridobi enako kot pri zasebnem načinu, le da je sporočilo zašifrirano še s stranskim ključem. Ta način je uporaben, ko želimo omejiti dostop do toka podatkov. Tistim, katerim želimo dovoliti dostop, se posredujeta stranski ključ in koren drevesa. Posledično je z zamenjavo stranskega ključa možno tudi spremeniti dostop do podatkov.

2.3 Prednosti platforme IOTA

Platforma ima zaradi svoje inovativne strukture štiri poglavitne prednosti in sicer: (1) razširljivost, (2) decentralizacija, (3) transakcije brez pristojbine in (4) zaščita pred kvantnim računalništvom [21]. Omenjene prednosti imajo še posebej veliko vlogo v domeni IOT.

Zmožnost razširljivosti omrežja se ocenjuje s povečanjem števila transakcij. Medtem, ko večina omrežij verig blokov s povečanjem števila transakcij, postane počasnejših, je pri platformi IOTA ravno nasprotno in sicer zaradi uporabljenega algoritma porazdeljenega soglasja. Porazdeljeno soglasje omrežja, kot je že omenjeno v prejšnjem poglavju, temelji na tem, da vsaka novo izvedena transakcija odobri dve prejšnji transakciji. Iz tega sledi, da večje število novih transakcij lahko potrdi večje število obstoječih transakcij. Teoretično gledano je tako omrežje IOTA neskončno razširljivo, kar posledično predstavlja primerno rešitev za okolje IOT [22].

Platforma IOTA je za razliko od omrežij tehnologije veriženja blokov veliko bolj decentralizirana, saj vsak udeleženec omrežja sodeluje pri odobravanju transakcij in ne zgolj rudarji oz. potrjevalci blokov[2]. Platforma je relativno mlada in ker število »poštenih« transakcij ni dovolj veliko, da bi bilo omrežje zavarovano pred štiriintrideset procentnim napadom, je v omrežje vključeno posebno koordinatorsko vozlišče [8] [23]. To vozlišče je pod nadzorom fundacije IOTA in na določen časovni interval (ena minuta) izda povsem običajne transakcije imenovane prelomnice (ang. milestones), ki prav tako odobrijo dve obstoječi transakciji. Pravilo omrežja je, da je transakcija dokončno potrjena, če se prelomnica neposredno ali posredno sklicuje nanjo, kar posledično pomeni, da platforma IOTA vsaj za zdaj še ni popolnoma decentralizirana. Potrebno je še dodati, da je implementacija tega vozlišča odprtokodna in morajo tudi te transakcije biti veljavne, da bodo kasneje potrjene s strani transakcij preostalih vozlišč [24].

Tretja prednost sledi iz uporabljenega algoritma porazdeljenega soglasja. Ker vsak uporabnik odobri obstoječe transakcije in s tem pripomore k aktivnosti omrežja, ni potrebe po pristojbini transakcij. Z vsako transakcijo lahko pošljemo poljubno količino kriptovalute ali surove podatke brez pristojbine. Ta prednost omogoča obilico novih priložnosti in primerov uporabe. Najbolj očiten primer uporabe je pošiljanje senzorskih podatkov, ki je predstavljen v naslednjem poglavju.

Čeprav je kvantno računalništvo še vedno v zgodnji fazi razvoja, je potrebno imeti v mislih, da so naprave IOT izdelane tako, da delujejo mnogo let brez dodatnih posegov in popravkov v strojni in/ali programski opremi,

[2] Vseh 2,779,530,283,000,000 kriptožetonov IOTA je bilo ustvarjenih na začetku.

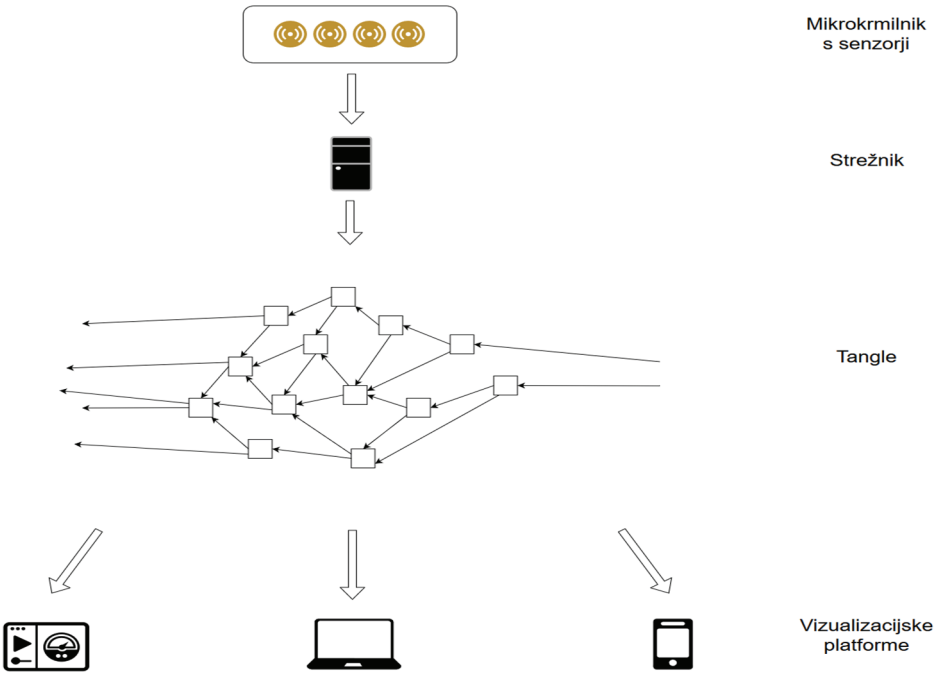
kar pomeni, da je potrebno zavarovati protokol tudi za prihodnost. Zato se v platformi za potrebe identifikacije pri digitalnem podpisu uporablja Winternitz One-Time Signature Scheme, ki bo po mnenju strokovnjakov tudi varen način šifriranja v dobi kvantnih računalnikov [25].

3 REŠITEV ZA ZBIRANJE IOT PODATKOV

Število naprav IOT se vsako leto povečuje in zanimanje za to domeno zelo narašča. Posledično tako narašča tudi število konceptualnih idej uporabe, kot tudi implementiranih projektov na to aktualno temo. Zaradi tega smo se odločili uporabiti platformo IOTA z namenom zbiranja realno-časovnih podatkov naprav IOT. Zanimalo nas je, kako zrela je omenjena tehnologija za omenjen praktičen primer uporabe. Cilj rešitve je bil pošiljati številne senzorske odčitke v Tangle in hkrati podatke iz Tangle-a prebirati ter jih vizualizirati. Ker nas je zanimala tudi učinkovitost pošiljanja in pridobivanja podatkov, smo spreminjali določene parametre in sicer vrsto senzorjev, dostopnost omrežja (omrežje za razvijalce in zasebno) in način vizualizacije.

3.1 Arhitektura

Arhitektura rešitve je modularna, pri čemer posamezen modul temelji na primerni tehnologiji in je hkrati popolnoma samostojen. Visokonivojska arhitektura, prikazana na Slika 3, je sestavljena iz različnih senzorjev, strežnika, omrežja IOTA oz. Tangle-a in vizualizacijske platforme. Naloga senzorjev je odčitavanje številnih fizikalnih lastnosti in posredovanje vrednosti le teh na strežnik. Slednji s pomočjo protokola MAM objavlja podatke v omrežje IOTA oz. Tangle. Vpisani podatki se nato iz omrežja IOTA preberejo in vizualizirajo na različnih vizualizacijskih napravah. V nadaljevanju sledi podrobnejši opis posameznih modulov.



Slika 3: Arhitektura rešitve za zbiranje podatkov IOT, temelječa na platformi IOTA.

3.2 Senzorji

Za odčitavanje in zbiranje senzorskih podatkov smo uporabili dva različna pristopa: (1) uporaba mikrokrmilnika in (2) uporaba Raspberry Pi [26]. Uporabili smo mikrokrmilnik ESP32, ki ga poganja operacijski sistem FreeRTOS in ponuja podporo za brezžično komunikacijo [27]. Mikrokrmilnik ESP32 izvaja pet osnovnih nalog, ki se izvajajo paralelno:

- prebiranje podatkov iz senzorjev s pomočjo vodila I²C in pošiljanje pridobljenih podatkov v vrsto sporočil (ang. message queue), ki je uporabljena za komunikacijo med nitmi,

- poslušanje sporočilne vrste in pošiljanje prebranih podatkov v medpomnilnik namenjen prikazu podatkov,
- poslušanje sporočilne vrste in pošiljanje podatke na vnaprej definirane teme protokola Message Queuing Telemetry Transport (MQTT) [28] z uporabo brezžične povezave WLAN,
- vzdrževanje MQTT povezave in
- prebiranje podatkov iz medpomnilnika in prikaz podatkov na ekranu.

V sklopu pilotne rešitve smo uporabili štiri senzorje, ki so odčitavali šest različnih fizikalnih lastnosti. V Tabela 1 so navedeni uporabljeni senzorji in pri vsakem namen uporabe.

Tabela 1: Uporabljeni senzorji in namen uporabe vsakega senzorja.

Senzor	**Namen uporabe**
CCS811	Ogljikov dioksid (CO_2, VOC (ang. volatile organic compound))
BMP280	Pritisk
Si7021	Vlaga, temperatura
MAX44009	Svetloba

Slabost uporabe mikrokrmilnika ESP32 je majhna procesorska moč. Sam mikrokrmilnik ni sposoben ustvarjati transakcij, izvajati algoritma dokaza o delu in nato še pošiljati transakcije v omrežje. V ta namen je uporabljen strežnik, ki služi kot posrednik med mikrokrmilnikom in omrežjem IOTA oz. Tangle-om.

Alternativa mikrokrmilniku je uporaba Raspberry Pi-ja. Uporabili smo model 3 B+. Prednost uporabe slednjega je neposredna priključitev senzorjev, pridobivanje odčitkov in sposobnost pošiljanja transakcij v omrežje IOTA.

Na Raspberry Pi smo namestili operacijski sistem Raspbian in izvajalno okolje Node.js. Za slednjega smo se odločili, ker je trenutno edina zrela implementacija protokola MAM le v programskem jeziku Javascript [18]. Knjižnica se imenuje mam.client.js. Na Raspberry Pi smo naložili Node.js skripto, ki je vsakih 15 sekund pošiljala podatke v Tangle. Podatki so bili predstavljeni v JSON formatu, kar je razvidno na Slika 4. Transakcije so bile uspešno poslane v omrežje v roku petih sekund (odvisno od zasedenosti uporabljenega vozlišča). Ker je za branje podatkov potrebno poznavanje korena toka podatkov, smo le-tega shranili v podatkovno bazo Firebase [29].

Slika 4: Struktura podatkov v transakcijah protokola MAM [30].

3.3 Strežnik

Mikrokrmilnik pošilja vrednosti senzorskih odčitkov s pomočjo protokola MQTT na strežnik. Naloga strežnika je branje podatkov iz senzorjev preko omenjenega protokola in pošiljanje le-teh na Tangle. Prednost takšnega postopka je zmogljivost, saj so strežniki veliko bolj zmogljivi kot Raspberry Pi, kar pomeni, da pošiljanje transakcij poteka hitreje in sicer do tri sekunde hitreje.

Za samo izvedbo rešitve je potrebno izbrati programski jezik, ki nudi podporo protokoloma MQTT in MAM. Kot je že omenjeno, je to trenutno le Javascript, zaradi česar smo se tudi tukaj odločili za Node.js. Drugi uveljavljeni programski jeziki in njihove knjižnice za IOTA trenutno še ne podpirajo protokol MAM (tj. C, C++, Java, Python in Go).

3.4 Omrežje IOTA

Kot je bilo omenjeno v poglavju 2.1, so na voljo tri globalna javna omrežja. Za našo rešitev smo uporabljali omrežje za razvijalce. Za uporabo javnega omrežja je potrebno pridobiti IP naslov in vrata enega izmed vozlišč omrežja.

Ker morajo vozlišča opravljati veliko opravil, se pri uporabi omrežja za razvijalce lahko zgodi, da so vozlišča preprosto prezasedena (lahko tudi neodzivna) in pošiljanje transakcij zato poteka dlje časa oz. sploh ni uspešno. Zaradi tega smo se odločili vzpostaviti lastno zasebno omrežje.

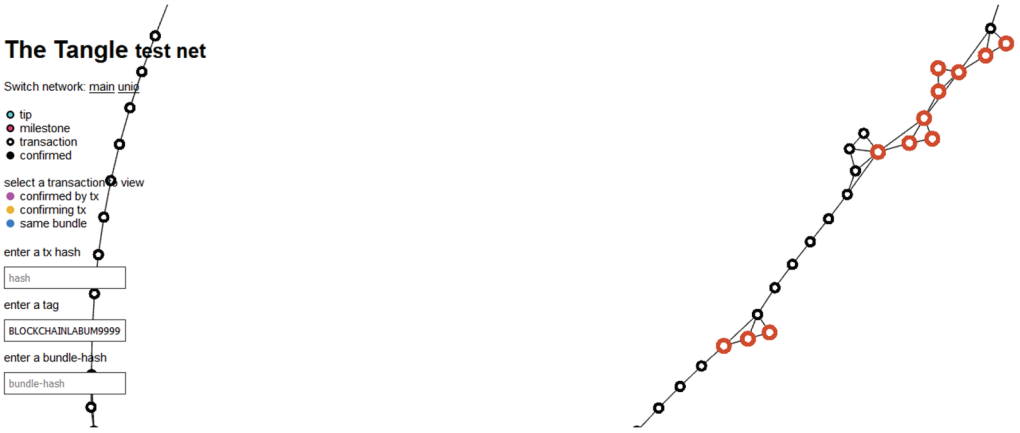
Omrežje smo vzpostavili po navodilih, ki so objavljena na uradni shrambi platforme IOTA (ang. repository) na spletni strani Github [31]. Za vzpostavitev je potreben strežnik Linux, za katerega je priporočenih vsaj osem GB delovnega pomnilnika in štiri-jedrni procesor. Na strežniku smo vzpostavili eno vozlišče omrežja in hkrati tudi lastnega koordinatorja omrežja, ki je za delovanje nujno potreben.

Pri zasebnem omrežju po pričakovanjih ni bilo težav zaradi prezasedenosti vozlišča, zaradi česar bi moral biti povprečen čas pošiljanja transakcij manjši. Težava nastopi zaradi premajhnega števila transakcij, saj je na javnem omrežju veliko večje število prihajajočih novih transakcij kot na zasebnem. Zaradi tega je bilo potrebno zraven pošiljanja senzorskih podatkov, še dodatno poslati nekaj navadnih transakcij za pohitritev omrežja.

3.5 Vizualizacijske platforme

En izmed glavnih ciljev senzorskih podatkov naprav IOT je njihova obdelava in vizualizacija. Tega smo se lotili na dva načina: prikaz samih podatkov senzorjev in prikaz transakcij v acikličnem grafu transakcij. Medtem, ko je razlog za prvega očiten, je za drugega prikaz razvoja Tangle-a in povezanost naših transakcij oz. snopa transakcij med seboj.

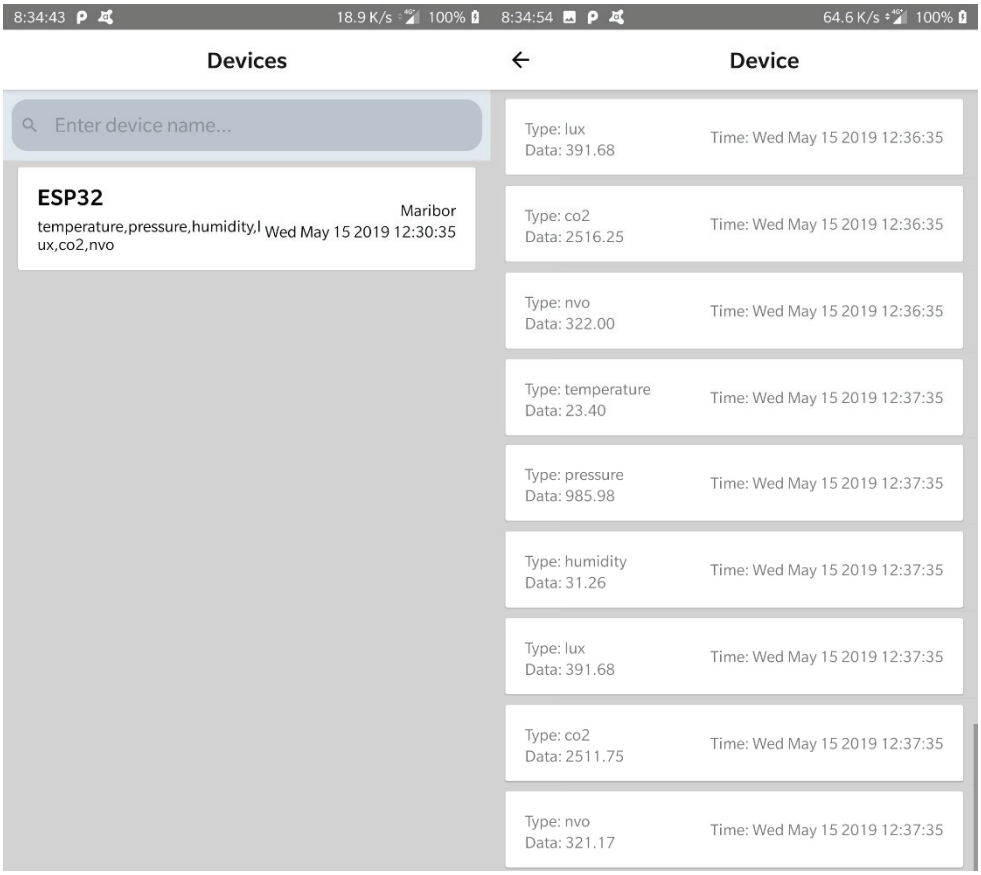
Za prikaz transakcij v Tangle-u smo uporabili spletno aplikacijo tangle.glumb.de [32]. Aplikacija prikazuje graf transakcij, kjer so transakcije predstavljene kot vozlišča. Povezave predstavljajo sklicevanje transakcij. Ob kliku na vozlišče se prikaže zgostitev in oznaka transakcije ter zgostitev snopa. Glavna funkcionalnost aplikacije, ki smo jo uporabljali, je iskanje transakcij po njihovi oznaki. Vsem transakcijam MAM smo določili oznako BLOCKCHAINLABUM999999999999 in nato na omenjeni aplikaciji izvedli iskanje po oznaki. Vse najdene transakcije se obarvajo rdeče. Slika 5 prikazuje transakcije MAM – snop treh ali več transakcij s podatki senzorjev, ki imajo omenjeno oznako BLOCKCHAINLABUM999999999999.



Slika 5: Prikaz transakcij MAM v Tangle-u, kjer so rdeče obarvane transakcije z oznako BLOCKCHAINLABUM.

Prikaz samih podatkov je v fazi testiranja potekal s pomočjo ukazne vrstice. Po uspešni implementaciji pošiljanja podatkov, smo začeli z implementacijo mobilne aplikacije. Ker je potrebna knjižnica za prebiranje podatkov iz Tangle-a na voljo le za programski jezik Javascript (mam.client.js), smo bili omejeni z izbiro ogrodja za izdelavo mobilnih aplikacij. Odločili smo se za uporabo React Native.

Cilj mobilne aplikacije je realno-časovni prikaz senzorskih podatkov. Aplikacija je povezana s podatkovno bazo Firebase, iz katere pridobiva informacije o senzorjih oz. Raspberry Pi-ju, ki pošiljajo podatke, in korene tokov podatkov. Za vsak tok podatkov se nato od tega korena naprej preberejo vse transakcije MAM, ki se dešifrirajo in prikažejo na zaslonu. Ker branje podatkov traja nekaj časa, smo vsaki dve uri v podatkovno bazo dodali nov koren.



Slika 6: Mobilna aplikacija (React Native), ki prikazuje sprotno odčitavanje senzorjev.

4 DISKUSIJA

Zaradi svoje edinstvene zgradbe in načina delovanja je s platformo IOTA moč pošiljati transakcije, ki vsebujejo poljubne podatke (transakcije brez vrednosti). Platforma omogoča hkrati uporabo namenskih protokolov, kot je MAM, ki omogoča intervalno pošiljanje poljubno povezanih podatkov v omrežje IOTA in s tem v Tangle. Trenutna pomanjkljivost protokola MAM je ta, da ga je možno zgolj uporabljati s programskim jezikom Javascript. Platforma je še v zgodnji fazi razvoja in v prihodnosti bo zelo verjetno prišlo tudi do večjih sprememb, kakor tudi do uvedbe knjižnic za druge programske jezike. Slabost samega protokola MAM je zaporedno branje toka podatkov. Čeprav to poteka relativno hitro, ni možno prebrati podatkov oz. transakcij določenega časovnega intervala.

Pilotno rešitev smo prvo povezali z javnim omrežjem IOTA, saj je povezovanje nanj zelo preprosto. Zaključili smo, da javno omrežje ni dovolj odzivno v času povečanega števila transakcij, predvsem zaradi še trenutno premajhnega števila vključenih vozlišč. Menimo, da je vzrok temu, da upravljalci vozlišč nimajo finančnih spodbud kot v primeru omrežij verig blokov, kjer ti večinoma sodelujejo pri rudarjenju in imajo s tem možnost prislužiti si nagrado v obliki novo izdanih kriptožetonov. Zaradi omenjenega problema smo se odločili vzpostaviti zasebno omrežje IOTA. Shramba za zasebna omrežja IOTA je dokaj nova in v času vzpostavljanja še ni bila dobro dokumentirana. Po vzpostavitvi je uporaba zasebnega omrežja ekvivalentna javnemu.

Med razvojem pilotne rešitve je bilo ugotovljeno, da knjižnica mam.client.js ne nudi možnosti sledenja transakcijam po oznakah. Zaradi želje po sledenju transakcijam izbranih senzorjev, smo bili primorani posodobiti omenjeno knjižnico na način, da smo jo nadgradili z možnostjo določanja poljubnih oznak za transakcije MAM (npr. BLOCKCHAINLABUM999999999999). Po testiranju smo dodatno funkcionalnost oddali na uradno shrambo platforme IOTA, kjer je bila v nekaj dneh dodana v uradno knjižnico.

Prav tako smo med razvojem prišli do zaključka, da je trenutno še zmeraj težava s samim razvojem končnih aplikacij (npr. mobilnih), ki bi bile povezane s platformo IOTA – predvsem zaradi trenutne omejitve na programski jezik Javascript. Za namen preverjanja kompatibilnosti in testiranja smo preizkusili Ionic 4, Ionic 1 in React Native. Ionic 4 s Typescript-om ni kompatibilen z omenjeno knjižnico. Vključitev knjižnice v Ionic 1 je možna, vendar je to seveda zastarelo ogrodje. Izbira React Native se je izkazala, kot najboljša izbira, vendar je za uspešno uporabo potrebno izvesti zagon aplikacije v načinu za razvijalce.

Iz predstavljenega je razvidno, da je platforma IOTA še v zgodnji fazi razvoja. Knjižice in ostale komponente potrebujejo še veliko dodatkov in optimizacije za enostavnejšo in učinkovitejšo uporabo. Problem trenutnih knjižnic, vozlišč omrežja (IRI), uporabe ternarnega sistema za kodiranje podatkov in nekaterih ostalih procesov je njihova fizična zahtevnost in energijska potratnost. Naprave, kot so senzorji IOT, trenutno še nimajo dovolj zmogljive strojne naprave za hitro ustvarjanje in pošiljanje transakcij. Današnja strojna oprema temelji na binarnem sistemu, zaradi česar je potrebno podatke kodirane v ternarnem sistemu tekom procesa pretvarjati v binarni, kar vzame dodaten čas. Na tem področju se sicer razvijajo različni prototipi strojne opreme, ki delujejo v ternarnem sistemu, in bodo v primeru platforme IOTA učinkovitejši [33]. Vozlišča omrežja pri večji zasedenosti delujejo počasneje in sicer zaradi procesov, ki so zahtevni in tudi potekajo sočasno. Zaradi tega se je začelo razvijati in tudi nastalo testno omrežje, ki deluje na vozliščih imenovanih IOTA Controlled agenT (ICT) [34]. Prednost teh vozliščih je modularna zgradba, kjer samo vozlišče podpira le najosnovnejše pošiljanje in prejemanje transakcij ter ostala nujno potrebna opravila za delovanje. Vse ostale dodatne funkcionalnost so dodane z moduli imenovanimi IOTA eXtension Interface (IXI) [34]. Ta vozlišča je mogoče namestiti tudi na Raspberry Pi (vozlišča IRI je sicer tudi možno, ampak zaradi zahtev po boljši strojni opremi ni smiselno), kjer delujejo brezhibno. Vendar slednje testno omrežje je še v zgodnejši fazi razvoja.

Naslednji korak v razvoju omrežja je odstranitev koordinatorja omrežja, kar poteka v sklopu projekta Coordicide [35]. Z odstranitvijo slednjega bo prišlo do večjih sprememb v omrežju, saj ne bo več t. i. transakcij prelomnic in posledično se bo na drug način določilo, kdaj je transakcija dokončno potrjena. Kot pri ostalih tekočih projektih, tudi pri tem ni točno dorečeno, kaj vse se bo spremenilo in kako, vendar prihaja več idej s strani fundacije IOTA.

5 ZAKLJUČEK

V prispevku smo predstavili platformo IOTA, ki predstavlja eno od novih platform porazdeljene glavne knjige. Predstavili smo njene poglavitne prednosti in potencialne slabosti. Osredotočili smo se na arhitekturo platforme, kakor tudi na podatkovno strukturo Tangle in razlike s klasičnimi omrežji verig blokov. Z namenom preverjanja uporabnosti platforme, smo zasnovali in načrtovali pilotno rešitev za domeno IOT, ki temelji na platformi IOTA. Cilj pilotne rešitve je bilo zajemanje senzorskih odčitkov in shranjevanje vrednosti le teh v omrežje IOTA in kasneje dostop in prebiranje senzorskih podatkov z mobilno aplikacijo, ki bi hkrati senzorske podatke primerno vizualizirala. Z namenom testiranja smo ob implementaciji hkrati izbirali različne pristope, kot je uporaba zasebnega ali javnega omrežja IOTA, uporaba različnih ogrodij za razvoj mobilnih aplikacij, uporaba različnih načinov za odčitavanje senzorskih vrednosti, itd.

Zaradi želje po sledenju senzorskih odčitkov v Tangle-u, smo uporabili namenski IOTA protokol MAM, ki pa do takrat privzeto ni omogočal povezovanja in iskanja povezanih podatkov po oznakah. V ta namen smo knjižico za protokol MAM nadgradili in omogočili določanje poljubnih oznak, s pomočjo katerih smo dosegli željeni cilj. Predlagana nadgradnja knjižnice je postala tudi uradni del le te.

Končna pilotna rešitev tako zajema celovito rešitev za domeno IOT, ki je dokazala uporabnost platforme IOTA za izbrano domeno in s tem prednost pred klasičnimi platformami verig blokov. Kljub temu pa je potrebno izpostaviti, da je platforma še v razvoju in da je še veliko prostora za izboljšanje in razvoj.

V nadaljevanju načrtujemo izvajanje testov zmogljivosti pilotne rešitve, kakor tudi analizo varnosti komponent platforme IOTA.

6 LITERATURA

[1] S. Nakamoto, „Bitcoin: A Peer-to-Peer Electronic Cash System“. [Na spletu]. Dostopno: https://bitcoin.org/bitcoin.pdf. [Dostopano: 27-maj-2019].

[2] S. Kim, Y. Kwon, in S. Cho, „A Survey of Scalability Solutions on Blockchain“, v *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, 2018, str. 1204–1207.

[3] „Introducing Masked Authenticated Messaging – IOTA“. [Na spletu]. Dostopno: https://blog.iota.org/introducing-masked-authenticated-messaging-e55c1822d50e. [Dostopano: 09-maj-2019].

[4] I. Foundation, „The Next Generation of Distributed Ledger Technology | IOTA“, 2018. [Na spletu]. Dostopno: https://www.iota.org/. [Dostopano: 25-apr-2019].

[5] „iotaledger/iri: IOTA Reference Implementation“. [Na spletu]. Dostopno: https://github.com/iotaledger/iri. [Dostopano: 09-maj-2019].

[6] „IOTA networks | References | Getting Started | IOTA Documentation“. [Na spletu]. Dostopno: https://docs.iota.org/docs/getting-started/0.1/references/iota-networks?q=network&highlights=network. [Dostopano: 23-maj-2019].

[7] S. Popov, „The Tangle“, 2018. [Na spletu]. Dostopno: https://assets.ctfassets.net/r1dr6vzfxhev/2t4uxvsIqk0EUau6g2sw0g/45eae33637ca92f85dd9f4a3a218e1ec/iota1_4_3.pdf. [Dostopano: 27-maj-2019].

[8] Q. Bramas, „The Stability and the Security of the Tangle“, 04-apr-2018. [Na spletu]. Dostopno: https://hal.archives-ouvertes.fr/hal-01716111v2. [Dostopano: 27-maj-2019].

[9] „Proof of work | Concepts | The Tangle | IOTA Documentation“. [Na spletu]. Dostopno: https://docs.iota.org/docs/the-tangle/0.1/concepts/proof-of-work. [Dostopano: 24-maj-2019].

[10] „Blockchain: IOTA Vs Bitcoin technology differences“. [Na spletu]. Dostopno: https://www.appsbee.com/blog/blockchain-iota-vs-bitcoin-technology-differences/. [Dostopano: 23-maj-2019].

[11] D. van Ravenzwaaij, P. Cassey, in S. D. Brown, „A simple introduction to Markov Chain Monte–Carlo sampling“, *Psychon. Bull. Rev.*, let. 25, št. 1, str. 143–154, feb. 2018.

[12] „Tip selection | Concepts | The Tangle | IOTA Documentation“. [Na spletu]. Dostopno: https://docs.iota.org/docs/the-tangle/0.1/concepts/tip-selection. [Dostopano: 23-maj-2019].

[13] „Incentives in the Tangle | Concepts | The Tangle | IOTA Documentation“. [Na spletu]. Dostopno: https://docs.iota.org/docs/the-tangle/0.1/concepts/incentives-in-the-tangle. [Dostopano: 24-maj-2019].

[14] „Trinary | Concepts | IOTA Basics | IOTA Documentation“. [Na spletu]. Dostopno: https://docs.iota.org/docs/iota-basics/0.1/concepts/trinary?q=trinary&highlights=trinari. [Dostopano: 23-maj-2019].
[15] „What is a seed? | Introduction | Getting Started | IOTA Documentation“. [Na spletu]. Dostopno: https://docs.iota.org/docs/getting-started/0.1/introduction/what-is-a-seed. [Dostopano: 23-maj-2019].
[16] „Bundles and transactions | Concepts | IOTA Basics | IOTA Documentation“. [Na spletu]. Dostopno: https://docs.iota.org/docs/iota-basics/0.1/concepts/bundles-and-transactions. [Dostopano: 23-maj-2019].
[17] R. C. Merkle, „A Certified Digital Signature“, v *Advances in Cryptology — CRYPTO' 89 Proceedings*, New York, NY: Springer New York, 1989, str. 218–238.
[18] „iotaledger/mam.client.js: Masked Authentication Messaging wrapper for Javascript (Browser and Node)“. [Na spletu]. Dostopno: https://github.com/iotaledger/mam.client.js/. [Dostopano: 09-maj-2019].
[19] „IOTA tutorial 19: Masked Authenticated Messaging - YouTube“. [Na spletu]. Dostopno: https://www.youtube.com/watch?v=Nnwn_o_ZBFU. [Dostopano: 25-maj-2019].
[20] „IOTA: MAM Eloquently Explained – Coinmonks – Medium“. [Na spletu]. Dostopno: https://medium.com/coinmonks/iota-mam-eloquently-explained-d7505863b413. [Dostopano: 25-maj-2019].
[21] N. Yuva in I. Kirbas, „Directed Acyclic Graph Based on Crypto Currency Application Example : IOTA“, *Conf. Int. Conf. Data Sci. Appl. ICONDATA 2018, Yalova*, št. December, str. 92–99, 2018.
[22] B. Ku´smierzku´smierz, P. Staupe, in A. Gal, „Extracting Tangle Properties in Continuous Time via Large-Scale Simulations“, 2018. [Na spletu]. Dostopno: https://assets.ctfassets.net/r1dr6vzfxhev/4T4IAlxk9ym0eWco0UoQIQ/90094e746745b89253eb3636b4ad1597/Extracting_Tangle_Properties_in_Continuous_Time_via_Large_Scale_Simulations_V2.pdf. [Dostopano: 27-maj-2019].
[23] „The Coordinator | Concepts | The Tangle | IOTA Documentation“. [Na spletu]. Dostopno: https://docs.iota.org/docs/the-tangle/0.1/concepts/the-coordinator?q=coordinator&highlights=coordinator%27. [Dostopano: 25-maj-2019].
[24] „iotaledger/compass“. [Na spletu]. Dostopno: https://github.com/iotaledger/compass. [Dostopano: 25-apr-2019].
[25] J. Buchmann, E. Dahmen, S. Ereth, A. Hülsing, in M. Rückert, „On the security of the Winternitz one-time signature scheme“, v *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, let. 6737 LNCS, Springer, Berlin, Heidelberg, 2011, str. 363–378.
[26] „Raspberry Pi — Teach, Learn, and Make with Raspberry Pi“. [Na spletu]. Dostopno: https://www.raspberrypi.org/. [Dostopano: 09-maj-2019].
[27] „ESP32 Overview | Espressif Systems“. [Na spletu]. Dostopno: https://www.espressif.com/en/products/hardware/esp32/overview. [Dostopano: 14-maj-2019].
[28] U. Hunkeler, H. L. Truong, in A. Stanford-Clark, „MQTT-S — A publish/subscribe protocol for Wireless Sensor Networks“, v *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE '08)*, 2008, str. 791–798.
[29] „Firebase“. [Na spletu]. Dostopno: https://firebase.google.com/. [Dostopano: 25-maj-2019].
[30] „Devnet Public MAM decoder - TheTangle.org“. [Na spletu]. Dostopno: https://devnet.thetangle.org/mam/WAC9LYKMHEWYMSPXUZEGSQLPXXEICNGSWW9JRDHHYDFIMHNBEQP9MOIRDLPTBMA9BNMJBHJ9NNDDXVTMV. [Dostopano: 16-maj-2019].
[31] „compass/docs/private_tangle at master · iotaledger/compass“. [Na spletu]. Dostopno: https://github.com/iotaledger/compass/tree/master/docs/private_tangle. [Dostopano: 09-maj-2019].
[32] „the IOTA tangle“. [Na spletu]. Dostopno: http://tangle.glumb.de/. [Dostopano: 09-maj-2019].
[33] S. B. H. Hassine, M. Jemai, in B. Ouni, „Design and FPGA implementation of ternary hardware IP core for square root function“, v *2017 International Conference on Engineering & MIS (ICEMIS)*, 2017, str. 1–6.
[34] „CHAT.ixi—Using Ict for Permissionless Chat on the IOTA Tangle“. [Na spletu]. Dostopno: https://blog.iota.org/chat-ixi-using-ict-for-permissionless-chat-on-the-iota-tangle-59ce6c5b95fb. [Dostopano: 24-maj-2019].
[35] „Coordinator. Part 1: The Path to Coordicide – IOTA“. [Na spletu]. Dostopno: https://blog.iota.org/coordinator-part-1-the-path-to-coordicide-ee4148a8db08. [Dostopano: 14-maj-2019].

OTS 2019
SODOBNE INFORMACIJSKE TEHNOLOGIJE IN STORITVE
M. Heričko in K. Kous (ur.)

Univerzitetna založba
Univerze v Mariboru

RAZVOJ DECENTRALIZIRANIH APLIKACIJ: RAZVOJ IGRE NA SREČO Z GENERATORJEM PSEVDO-NAKLJUČNIH ŠTEVIL

JURE TRILAR, ANDREJ KOS, EMILIJA STOJMENOVA DUH

Povzetek: *Blokovne verige (ang. Blockchain) zagotavljajo nepotvorljivo revizijsko sled različnih sredstev v obliki kriptožetonov, pametnih pogodb in podatkov v njih, ob tem pa ponujajo učinkovite preživitvene in varnostne mehanizme tudi v primerih velikih varnostnih ali infrastrukturnih motenj omrežja. To odstrani potrebo po zaupanju centralni avtoriteti in uporabnik se lahko zanese na avtonomno delovanje sistema, ki temelji na naboru matematičnih pravil. Primer bolj aktualnih uporab blokovnih verig je distribuirano izvajanje računalniških algoritmov, ki so zapisani v obliki pametnih pogodb. Na tem področju po odprtosti in razširjenosti prednjači platforma Ethereum, s pomočjo katere izdelujemo t.i. decentralizirane aplikacije (dApp). Kljub temu, da Ethereum okolje za izvajanje ponuja Touringovo celovitost, pa obstajajo določene zadrege, ki izhajajo iz konceptualne zasnove t.i. največjega in najpočasnejšega računalnika na svetu, kjer se programska koda izvaja simultano na več deset tisoč vozliščih, ki praviloma vrnejo isti izid v razdobju nekaj minut. Ob tem se pojavlja vprašanje, kako zagotoviti naključnost (ang. randomness) v takšni distribuirani platformi, kjer morajo vsi akterji praviloma izračunati isti rezultat. Naključnost generiranja števil je bistvena predvsem za igre na srečo; platforma Ethereum pa po zasnovi tega ne more zagotoviti. V članku najprej raziščemo splošne omejitve platforme pri shranjevanju podatkov v pametnih pogodbah na decentraliziranih sistemih ter obravnavamo tipična varnostna tveganja in luknje v pametnih pogodbah. Nadalje opišemo razvoj in scenarije delovanje igre na srečo, ki temelji na izbranih pravilih in zagotavlja poštenost prek zaledja razvitega na platformi Ethereum. Začelje decentralizirane aplikacije je iz aktualnih spletnih tehnologij Node, Vue in Web3.js. Pri načrtovanju igre je upoštevan pristop, ki vključuje tudi uporabnikove potrebe, predvsem z vidika težavnosti razumevanja in uporabe blokovnih verig, ter temeljnih motivacijskih mehanizmov, kar bi omogočilo večjo privlačnost za pogostejšo uporabo. Članek se zaključi z analizo pridobljenih izkušenj glede razvoja konkretne decentralizirane aplikacije in uporabe te tehnologije ter razmislekom o primernosti in zrelosti te tehnologije ter o ovirah, ki jih bo treba v prihodnje preseči.*

Ključne besede: *blokovne verige, decentralizirane aplikacije, Ethereum, generiranje naključnih števil, spletne igre na srečo*

NASLOVI AVTORJEV: Jure Trilar, raziskovalec, Univerza v Ljubljani, Fakulteta za elektrotehniko, Ljubljana, Slovenija, e-pošta: jure.trilar@fe.uni-lj.si. dr. Andrej Kos, redni profesor, Univerza v Ljubljani, Fakulteta za elektrotehniko, Ljubljana, Slovenija, e-pošta: andrej.kos@fe.uni-lj.si. dr. Emilija Stojmenova Duh, docentka, Univerza v Ljubljani, Fakulteta za elektrotehniko, Ljubljana, Slovenija, e-pošta: emilija.stojmenova@fe.uni-lj.si.

https://doi.org/10.18690/978-961-286-282-4.6 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 UVOD

Ob kratkovidnem navdušenju nad tehnologijo blokovnih verig (angl. blockchain) je precej posameznikov in organizacij, ki poskušajo najti načine, kako bi blokovne verige uporabili v obstoječih procesih ali razvili nove. Izkušnje kažejo, da večina teh poskusov služi zgolj kot opozorilo, da je potrebno za vsak primer uporabe na specifičnih domenskih področjih posebej dobro razmisliti, ali s to novo tehnologijo dejansko prispevamo k dodatni uporabni vrednosti.

Blokovne verige se večkrat omenja kot inovacija, ki rešuje problem decentraliziranega zaupanja, zato ocenjujemo, da bi bila navezava z spletnimi igrami na srečo, kljub določenim omejitvam, dobra izbira za izvedbo učinkovitega prikaza izbranega uporabniškega scenarija.

Pogosto smo bili v vlogi, ko bi s pomočjo blokovnih verig, želeli reševati velike družbene probleme ali izboljševali kompleksne procese v panogah, kot so bančništvo, energetika, avtomobilska industrija, turizem,... Tokrat pa smo se omejili na, pravzaprav preprosto, vendar pa ne nepomembno in povsem praktično demonstracijo reševanja enega osnovnih konceptualnih zadreg v procesiranju na blokovnih verigah, oz. natančneje, težave pri generiranje naključnih števil na platformi Ethereum.

2 BLOKOVNE VERIGE

Tehnologije blokovnih verig imajo širok spekter uporabnosti, ki, poleg zanimivih tehničnih rešitev, vključujejo tudi pričakovanja, da bodo postale ena bolj disruptivnih tehnologij v poslovnih procesih. V preteklem času so rešitve, povezane s temi tehnologijami, že doživele širšo prepoznavnost, kar je bilo povezano z medijskimi objavami povezanimi z špekulacijami glede cen teh novih, prepoznavnih, vendar pogosto nerazumljenih tehnologij.

Bitcoin, ki ga je izumil anonimni avtor pod psevdonimom Satoshi Nakamoto leta 2008, je bil prvi javni prikaz tehnologije in konceptov blokovnih verig. Nakamoto je poleg znanega članka, *Bitcoin: A Peer-to-Peer Electronic Cash System*, izdal tudi prvo verzijo delujoče kode, ki je povzročila veliko zanimanje v kriptografski in razvijalski skupnosti. Rešitev služi, kot prva javna infrastruktura za kriptovaluto Bitcoin, ki je reševala problem dvojnega trošenja (angl. double-spending), brez centralne avtoritete, ki bi bila zadolžena za nadzor. Zagotavlja nepotvorljivo, javno dostopno bilanco (angl. ledger), kamor je možno zapisati, ne samo finančne, temveč tudi razne druge transakcije za širšo uporabo [1]. Če ilustriramo, prešli smo iz tradicionalne uporabe računalnika, kjer je bilo moč neko podatkovno entiteto kopirati večkrat, in jo obenem obdržati pri sebi, do uporabe, kjer edinstveno entiteto lahko prenesemo drugam ter je, zaradi tega, nimamo več na voljo pri sebi, kar bolj pristno posnema »naravni« svet izven digitalnim aparatov.

Zasnova Bitcoina je navdihnila razvoj cele vrste novih platform na različnih področjih uporabe. Večina platform blokovnih verig, podobno kot prvi vzor za blokovne verige - Bitcoin, obljublja odprto, javno dostopno, distribuirano bilanco, ki omogoča transakcijo med različnimi akterji na zanesljiv in preverljiv način. Dostop, spremljanje stanja in izvedbo transakcij omogočata privatni ključ in njegov asimetrično kriptiran derivat v obliki javnega ključa. Splošne definicije, kaj so blokovne verige in čemu so namenjene, vključujejo navedbe, da so blokovne verige naraščajoč nabor podatkov, ki so združeni v bloke in med sabo povezani z kriptografskimi funkcijami [2]. Vsak blok vsebuje kriptografski povzetek vsebine prejšnjega bloka, časovni žig in podatke o transakcijah, ki so večkrat predstavljeni s povzetki Merklovih korenov (angl. Merkle tree root hash). Za učinkovito distribuiranje podatkov skrbi P2P omrežje, ki kolektivno skrbi za skladnost oz. konsenz glede vsebine transakcij v komunikaciji med vozlišči pri potrjevanju blokov [3]. Kljub temu, da večkrat zasledimo to besedo, konsenz, pa praviloma ne gre za pravi konsenz, ki vključuje razumevanje in strinjanje z določenimi vnosi, temveč gre v originalni izvedbi (*proof of work* pri Bitcoin-u) za loterijo, ki skrbi za to, da blok potrdi naključno izbran rudar (angl. miner) oz. rudarska zadruga (angl. pool), ki si obeta nagrado za pravilno ugotovljen povzetek aktualnega bloka z določenim številom ničel spredaj, ki predstavljajo težavnosti, ki je neposredno povezana z računsko močjo - na ta način omrežje poskrbi za svojo samoohranitev. Obstajajo seveda tudi drugi pristopi (*proof of stake*, *proof of authority*, dBTF,…), ki poizkušajo vzpostaviti mehanizme, ki bi ponujali dovolj zanesljivosti in motivacije za sodelovanje pri uporabi ter vzdrževanju takih omrežja, vendar za splošno razumevanje, kaj so to blokovne verige, ne bomo osvetlili vseh platformskih projektov, saj vsaka ponuja svoj pristop oz. konkurenčno prednost, ki jih loči od ostalih.

Negotovost, ki obdaja uporabo teh tehnologij, je rezultat tisočerih poizkusov in projektov, ki so ostali na robu pred-produkcijskega razvoja. Zrelost in uporabnost tehnologije blokovnih verig, poleg tehničnih dejavnikov, določajo predvsem dejavniki, ki so povezani z poslovnimi procesi in uporabniško izkušnjo [4]. Splošne omejitve, na katere ljudje naletijo pri uporabi blokovnih verig zajemajo [5]:

- *Blokovne verige ne pokrivajo vseh potreb in želja.* Za enkrat še velja pravilo, da so blokovne verige ujete med tri konkurenčne-si cilje: hitrost, nizki stroški obratovanja in dovolj velika decentralizacija. Ponazorimo: hitre in dovolj decentralizirane verige rezultirajo v višjih stroških shranjevanja in prenosov, ki se jim je težko izogniti tudi z bolj sofisticiranimi prijemi. Tako sledi tudi, da hitre in poceni verige potrjujejo transakcije na manjšem številu znanih vozlišč, vendar pa postavlja pod vprašaj zaupanje v tako omrežje, saj lahko ponudnik/razvijalec na omrežje lažje vpliva.
- *Blokovne verige ne predvidevajo pomoči strankam.* Ob tem, ko se je povprečni uporabnik navadil na možnost pridobitve pozabljenega gesla spletnih storitev s strani ponudnika, je težko narediti korak nazaj. Zaradi svoje ireverzibilne narave, kar je obenem tudi ključna prednost, pri izgubi zasebnih ključev nihče ne more pomagati. Izguba sredstev zaradi minornih napak in nepozornosti je za uporabnike zastrašujoča.
- *Blokovne verige dodajajo »trenje« v že optimizirane procese.* Uporaba decentraliziranih aplikacij v tem trenutku je vse prej kot intuitivna in lahka. Nepoučeni uporabniki, ki so danes sicer že vešči izvajanja plačil na spletu, se soočajo z bojaznijo kaj se zgodi z njihovimi sredstvi in, ali bodo za svoje plačilo z kriptovaluto res prejeli naročeno blago. To v kombinaciji z počasnostjo in provizijami sigurno ne vzbuja zaupanja v splošni javnosti. Uporabniki so v današnjem času navajeni hitrih mobilnih aplikacij in takojšnje zadovoljitve potreb in za to že obstajajo rešitve, ki so bile optimizirane v desetletjih razvoja, kot v primeru plačila z kreditno kartico, povezano z mobilno aplikacijo, kjer se transakcija izvede v manj kot dveh sekundah [6].
- *Pomanjkanje nadzora in regulacije povzroča negotovost v poslovnih procesih.* Z razmahom spornih poslovnih praks, propadlih projektov in povečano magnitudo nereguliranega zbiranja denarja prek t.i. ICO postopkov, se je povečala potreba po pravni zaščiti udeležencev. Zakonodajne in vladne inštitucije zaenkat sicer sledijo novim inovativnim tehnologijam in procesom, vendar težko ocenijo stanje in ukrepajo zoper posamezne akterje in aplikacije v primeru neupoštevanja varnostnih zahtev in poslovnih modelov, ki niso v skladu z zakonodajo [7].
- *Kompleksnost tehnologije ustvarja splošen vtis, da te tehnologije niso dovolj uporabne.* Pojmi enkripcije, decentralizacije in porazdeljenega konsenza, kot temeljev blokovnih verig, so pogosto preveč odtujeni od povprečnega občana, da bi v njih videl uporabno vrednost. Brez širšega privzemanja tehnologije v družbi in podpore inštitucij je težko pričakovati povečano potrebo po uporabi blokovnih verig.

Potreben je razmislek o temeljnih problemih, ki jih želimo rešiti s to tehnologijo, s posebnim ozirom na omejitve, ki ne izvirajo iz tehničnih pristopov k načrtovanju te tehnologije, temveč iz uporabniških ter konceptualnih zadreg in so verjetno bolj pomembne, kar se tiče uspeha in uporabnosti v splošni javnosti. Tehnologije blokovnih verig niso univerzalen odgovor na probleme v družbi in ekonomiji, dobro pa rešujejo specifične scenarije, ki jih prej ni bilo mogoče nazvati z etabliranimi pristopi.

Sklenemo, glavna konceptualna značilnost vseh platform blokovnih verig je nespremenljivost podatkov, zapisanih v seriji blokov, v matematičnih funkcijah (npr. DAG – usmerjeni neciklični grafi) ali podobno. Ta nespremenljivost podatkov v povezavi z inovativnimi motivacijskimi mehanizmi za podpiranje infrastrukture predstavlja osnovo za privlačne rešitve, ki vnašajo v svet računalništva nove razburljive koncepte, povezane predvsem z transakcijskimi procesi. Ostale skupne značilnosti so:

- *Decentraliziranost baze podatkov.* Ob nedosegljivosti enega ali nekaj od podpornikov infrastrukture, sistem še vedno deluje brez težav.
- *Varnost podatkov.* Podatki so sočasno zapisani v več vozliščih, redundantnost podatkov je maksimalna.
- *Odstranitev potrebe po zaupanju centralni avtoriteti*. Več partnerjev, ki imajo v svojih vozliščih v vsakem trenutku pri sebi zadnjo aktualno verzijo podatkov, ne potrebuje osrednjega potrjevalca transakcij, temveč se zanašajo na avtonomno delovanje sistem, ki temelji na matematičnih primitivih.

- *Preverljivost in nezmožnost ponarejanja podatkov.* Vnosi so za nazaj nespremenljivi in ustvarjajo nepotvorljivo, praviloma javno dostopno, revizijsko sled.
- *Mehanizmi za konsenz.* Če pride do sočasno nasprotujočih si vnosov, mehanizmi za konsenz (praviloma naključno izbrano vozlišče) poskrbijo za razrešitev konfliktov, kar vpliva na celovito integriteto podatkov.

3 DECENTRALIZIRANE APLIKACIJE

Najbolj pomemben koncept, po mnenju avtorjev, ki se je uveljavil na nekaterih aktualnejših platformah blokovnih verig je decentralizirano računalništvo, ki prej omenjene prednosti nadgradi še z možnostjo izvajanja serije ukazov oz. algoritmov na zanesljivi distribuirani infrastrukturi.

Za napredek transakcijskih platform, ki so na začetku temeljile na kodi in mehanizmih, ki jih je uveljavil Bitcoin, je počasen razvoj postal problem. Razvijalci so se znašli pred razpotjem, ali razširiti obstoječ nabor funkcij, kar bi bilo kompleksno in časovno zelo potratno, ali razviti popolnoma novo platformo. Ta izziv je prepoznal mladi Kanadčan ruskega rodu, Vitalik Buterin, in z novimi pristopi izumil platformo za decentralizirano izvajanje ukazov – Ethereum. Za Ethereum rečemo, da je največji, vendar najbolj počasen virtualni računalnik na svetu z hitrostjo obratovanja vezano na generacijo novega bloka.

Pred izumom Ethereuma so bile aplikacije, ki so komunicirale z Bitcoin omrežjem, omejene na ozek nabor operacij, saj so bili Bitcoin in posnemovalci razviti izrecno za *peer-to-peer* digitalne valute. Osrednja inovacija, Ethereum virtualni stroj (EVM) je programska oprema, ki deluje na temelju odprtega Ethereum omrežja in omogoča programsko celovitost po Turingu. EVM je temelj razvoja decentraliziranih aplikacij. Namesto, da bi morali za vsak primer uporabe narediti svojo blokovno verigo, lahko na podlagi Ethereum omrežja in kratkih programčkov, t.i. pametnih pogodb (angl. smart contracts), ustvarimo namensko rešitev za decentralizirane aplikacije, ki lahko vsebuje tudi lastne žetone za transakcije brez posrednikov. Ti žetoni imajo lahko različne vloge, od splošnih tipa ERC 20 do nosilcev edinstvenih sredstev ERC 721 ter drugih naprednih, npr. ERC 223, 777 in 820.

Popularni izraz za kratka programska navodila, ki se izvajajo na izbrani platformi blokovnih verig, so pametne pogodbe – ki v resnici niso niti pametne, saj gre za »ne-posebno-pameten« niz programskih navodil, niti niso pogodbe *per se*, saj avtomatizem in nepremenljivost, kot temeljni lastnosti pametnih pogodb, lahko posegata v zakonsko načelo prostega urejanja obligacijskih razmerij. Kljub številnim prednostim, decentralizirane aplikacije seveda niso vedno brezhibne in brez napak, kakor se je že večkrat izkazalo. Pametne pogodbe so varne le toliko, kot so ljudje, ki so jih napisali, za to usposobljeni. Napake in hrošči v knjižnicah se lahko izkoristijo za različne napade, kar vodi do nezaželenih posledic ter izgube vseh sredstev. Ob tem, zaradi konceptualne zasnove, ni učinkovitih načinov, kako preprečiti tekoči napad ali povrniti prejšnje stanje – razen v izredno redkih primerih, ko gre za spremembe infrastrukturne jedrne kode in njene zamenjave na vozliščih, kjer se procesirajo pametne pogodbe, za kar pa je potrebno pridobiti večino razvijalske skupnosti – t.i. *hard-fork*.

Pametne pogodbe v praksi predstavljajo jedro zaledja (angl. backend) v smislu shranjevanja in procesiranja podatkov za t.i. decentralizirane aplikacije (angl. dApp). Za razvoj začelja, prednje maske (angl. frontend) se uporabljajo uveljavljene spletne tehnologije, ki se praviloma procesirajo tam, kjer je prezentacija vmesnika, pri uporabniku (angl. client-side).

Kriteriji za izbiro primerne infrastrukture, kjer bi razvili prototipa so enaki, kot v podobnih situacijah pri razvoju decentraliziranih aplikacijah s katerimi so se že srečali v preteklosti, in vključujejo [8]:

- zadosten *dostop do razvojnih virov*, velikost in odzivnost aktivne razvijalske skupnosti, dostopnost dokumentacije in dobri primeri obstoječih izdelkov, na osnovi česar lahko minimiziramo možnost, da se razvoj ustavi, ker temeljne infrastrukturne rešitve niso dobro dokumentirane ali morebitne težave še niso bile obravnavane, ter se izognemo kritičnim tehničnim oviram, ki nastanejo v primeru napak na jedrni platformi, in težavam z vzpostavitvijo lastne informacijske hrbtenice, ki bi podpirala to platformo;
- *možnost prototipiranja in testiranja* širokega nabora uporabniških scenarijev, česar ne omogočajo vse platforme, saj so nekatere med njimi specializirane zgolj za specifične tehnične in poslovne pogoje;

- *možnost prenosa realiziranih konceptov* in uporabne vrednosti na druge platforme, ko bodo le-te bolj zrele za produkcijsko uporabo;
- *uporaba javnega okolja*, pri čemer se za shranjevanje podatkov in procesne potrebe integracije lahko zanašajo na javno infrastrukturo, ki je ni potrebno vzdrževati.

Tudi tokrat je rezultat izbire še vedno enak – izbrali smo najbolj razširjeno , ki ima dovolj veliko razvijalsko skupnost in dovolj kvalitetne razvijalske dokumentacije: Ethereum.

Protokol za varen prenos podatkov v tehnologiji blokovnih verig na omrežju Ethereum vključuje [8]:

- Sorazmerno hitro pridobitev podatkov iz platforme Ethereum, ki praviloma ne presega 300 milisekund;
- manjšo hitrost zapisovanja in potrjevanja zapisov, ki praviloma zahteva od 3 do 5 minut;
- konstantnejše generiranje blokov v primerjavi z omrežjem Bitcoin;
- uporabo različnih razvojnih okolij in knjižnic, kot so Truffle, Remix Web IDE itd.;
- procesiranje programske logike v izvajalnem okolju Ethereum Virtual Machine [9], ki omogoča visoko decentralizacijo na 27.500 vozliščih po vsem svetu;
- razvoj pametnih pogodb v namenskem jeziku Solidity [9] ali Vyper, ki temelji na jeziku Python;
- integracijo s pametnimi pogodbami iz spletnih aplikacij na osnovi aplikacijskega programskega vmesnika Ethereum JavaScript API (Web3.js) [10], kar predstavlja široko dostopno razvojno okolje;
- lastno sredstvo za potrjevanje transakcij v obliki kriptovalue Ether v obliki Gas [11], kar predstavlja mikro enote primarne valute in se nanaša na obremenitev »največjega in najpočasnejšega« računalnika za svetu, pri čemer so v praksi stroški transakcij manjši kot provizije za transakcije Bitcoina;
- transformacijo iz energetsko potratnega protokola nagrajevanja PoW (Proof-of-Work) v protokol PoS (Proof-of-Stake) [12, 13];
- razvoj v različnih razvojno-testnih omrežjih (Ropsten, Rinkeby, Kovan ali RPC po meri), v katerih se za testiranje ne porablja prava vrednost Ethra, kot je to na glavni hrbtenici (main-net).

4 NAKLJUČNOST NA DECENTRALIZIRANI PLATFORMI ETHEREUM

EVM deluje na način, da se ista koda izvede sočasno na več vozliščih, vedno zgolj po klicu funkcij v pametni pogodbi. Za zagotavljanje integritete mora biti rezultat izhoda funkcije isti. Obstajajo pa primeri, ko za rezultat potrebujemo naključno število in v takih situacijah naletimo na težavo, kako podati naključno število, ki je rezultat sinhronega procesiranja [14]. Pravilno generiranje naključnih števil je zelo pomembno predvsem pri igrah na srečo in v kolikor to ni naključno in se da nanjo vplivati oz. vnaprej oceniti rezultat, je pristransko in problematično ter izzove vsaj nezaupanje udeležencev v igri.

Potrebno se je zavedati, da je prava naključnosti (v računalništvu) zelo obširna in težavna tema. Obstaja več vrst naključnosti, ki jih lahko nazovemo z pravilnim inženiringom računalniških čipov, glede na potrebe in stopnjo šuma in zadovoljive rezultate. Da bi generirali pravo naključno številko, bi morali izmeriti neko fizično stanje naključja izven računalniškega sistema. Ker pa v praksi ne potrebujemo meriti entropije, npr. v obliki naključnega razpada atoma, ki ga po kvantni teoriji ne moremo napovedati, se zadovoljimo z računalniško generiranimi aproksimacijami [15]. To vsekakor zagotavlja zadostno stopnjo naključnosti za večino primerov uporab na svetu. Namesto merjenja entropije se uporablja t.i. zrno (seed) ali več njih, nato pomočjo raznih preslikav, asimetričnih transformacij predstavimo na videz naključen rezultat – uporabimo t.i. psevdo-naključnost.

Za potrebe prototipa se bomo zadovoljili z deterministčnim generiranjem števil, na katero ne bo mogoče neposredno vplivati z zunanjimi dejanji. Pred tem smo preverili, kakšne so možne kritične napake pri generiranju naključnih števil na platformi Ethereum, ki bi napadalcem omogočila vplivanje na rezultat. Reutov [14] in sodelavci so analizirali 3.649 pametnih pogodb iz različnih virov, od teh je bilo 43 pogodb ranljivih z vidika ustvarjanja psevdo-naključnih števil, saj so za zrno izbrali vire, ki jih je mogoče določiti že prej: na podlagi raznih podatkov o trenutnem bloku in stanju omrežja, na podlagi povzetkov prejšnjih blokov, na podlagi kombinacije podatkov zapisanih v blokih in na videz zasebnih podatkov ter drugih psevdo-naključnih generatorjev, kjer pa je mogoče z izračunom vnaprej določiti izid. Podrobnosti nazornih primerih so na voljo

v strokovnem članku Reutova *Predicting Random Numbers in Ethereum Smart Contracts* [14], v tem prispevku jih ne potrebujemo detajlno izpostavljati. Izpostavili pa bi predlagane rešitve, ki jih lahko uporabimo za lasten prototip:

- *Uporaba zunanjega oraklja* (angl. oracle). Zaupanja vreden zunanji vir, ki se ne nahaja na platformi Ethereum, preko API zahtevka ali posrednika v obliki javne, za to namenjene pogodbe, prosimo za generiranje naključnega števila. Zadržki so očitni, gre za centralizirano zunanjo storitev, kjer je potrebno zaupati ponudniku. Različni ponudniki (Oraclize, BTCRelay, Randao) vključujejo različne vire entropije, vsi pa ponujajo neko stopnjo kompleksnosti, ki jo je težko simulirati [15]. Nekatere storitve so plačljive.
- *Uporaba niza namenskih pametnih pogodb* (algoritem Signidice). Algoritem, ki temelji na zasebnih kriptografskih podpisih v proces generiranja psevdo-naključnega števila, vključuje dve strani, uporabnika in ponudnika. Obe strani v tem primeru, po tem ko so naložena sredstva za stavo, uporabita svoj privatni ključ, zato gre moč z veliko gotovostjo trditi, da nimata motivacije za goljufanje, saj bi to vodilo v izgubo sredstev na računu. Ta pristop je bilo sicer, pred nadgradnjo Ethereum omrežja (Metropolis hardfork), z strani ponudnika možno zlorabiti.
- *Pristop objavi-razkrij* (angl. commit-reveal approach). Gre za dvostopenjski način, kjer v prvi fazi vsi udeleženi prispevajo svoje kriptografske skrivnosti v pametno pogodbo, kar omeji vpliv enega udeleženca na rezultat. V drugi fazi pa pametna pogodba na podlagi združenih kriptografskih skrivnosti (poljuben tekstovni niz, uporabniško ime, javni ključ ali podobno) ter dodatka povzetka (angl. blockhash) pravkar generiranega bloka, izgradi psevdo-naključno število, ki ga nato uporabimo neposredno v decentralizirani aplikaciji. Ta pristop ponuja tri vire entropije: uporabnikov vnos, ponudnikov vnos, infrastrukturni žig. Motivacije za goljufijo neposredno (uporabnik in ponudnik) ali posredno (rudar) udeleženih v procesu se med seboj izključujejo, tudi kadar gre v resnici za isto osebo v deljenih vlogah (npr. uporabnik in rudar).

Za naš prototip smo izbrali tretji pristop, dvostopenjski način, ki omogoča, da generiranje psevdo-naključnih števil vodimo iz ene pametne pogodbe, brez dodatnih tveganj in kompleksnosti. V nadaljevanju (Programska koda 1), pri razvoju zalednega dela prototipa decentralizirane aplikacije, bomo prikazali pametno pogodbo v relevantnem delu, ki se nanaša na psevdo-naključno generiranje števila za potrebe igre na srečo.

5 UPORABNIŠKA IZKUŠNJA SPLETNE IGRE NA SREČO

Za prikaz uporabe psevdo-naključnega generiranja števila smo si izbrali primer uporabe, kjer je to še kako ključno – spletno igro na srečo. Naše temeljno videnje je, da je pri načrtovanju vsakršnih digitalnih rešitev, tudi prototipov, potrebno upoštevati vsaj osnovne poglede uporabniške izkušnje. Ob uporabi na uporabnika usmerjenega pristopa (angl. user-centred design) načrtujemo rešitve, kjer izboljšujemo tehnične vidike, ki se skladajo z potrebami in željami dejanskih uporabnikov.

Analiza področja motivacije za igranje spletnih iger [16-19], nas navaja na robustni konceptualni okvir, ki kaže na podobno strukturo iz generičnih faktorjev, ki se nanašajo na sledeče dimenzije:

- *Dosežki pri igranju*. Ta dimenzija vključuje generične faktorje, kot so: napredek pri igranju, nezatikajoča-se tekmovalna mehanika igranja.
- *Družabnost* se nanaša na možnost neposrednega tekmovanja in medsebojne komunikacije med igralci.
- *Pristna izkušnja* omogoča igralcu na videz nasprotujoča se faktorja: na eni strani nove izkušnje, na drugi strani pa okolje z znanimi zgodbovnimi elementi.
- *Zaupanje*, ki se reflektira v lojalnosti uporabe. Zaupanje vključuje različne faktorje, od predvidljivosti uporabniškega vmesnika in zanesljivosti delovanja igre, do ugleda ustvarjalca oz. ponudnika igre ter jasnega sporočila.

Za povečano motivacijo za sodelovanje v spletnih igrah na srečo pomembnih več teh kritičnih dimenzij. Ocenjujemo, da je najbolj pomemben kriterij v teh primerih je četrta komponenta - upravičevanje zaupanja. Sistematično testiranje modelov motivacije in zaupanja za sodelovanje v spletnih igrah na srečo je prikazalo več dimenzij, na katere so pozorni uporabniki pri komponenti zaupanja v spletnem igralništvu [16]:

- Informativna vsebina, *razumljivost*. Uporabniki, ki igrajo igre na srečo, morajo poznati način delovanja igre, da lahko bolj ali manj uspešno ocenijo tveganje, ki nastaja ob stavi.
- *Odnosi z uporabniki*. Ta vidik se nanaša na vidne kontakte in vtis, da je poslovanje pod okriljem znane osebe ali organizacije, ki nosi pravno odgovornost v primeru težav in napak.
- Lastnosti uporabniškega vmesnika in *privlačna mehanika igre*. Predvidljivost, jasen namen kaj igra od uporabnika pričakuje.
- *Odnos uporabnikov do iger na srečo*. Pomemben filter za sodelovanje v igrah na srečo je seveda temeljni odnos določenega dela populacije do tega področja. Faktorjev za uporabo takih rešitev ne moremo učinkovito meriti na populaciji, ki takih igre ne uporablja ali ima do le-teh odklonilen odnos.

Na podlagi obravnavanih dimenzij (dosežki, družabnost, pristna izkušnja in zaupanje), ki so pomembne za motivacijo za igranje spletnih iger v splošnem, in tudi v specifičnem primeru spletnih iger na srečo, smo v nadaljevanju izbrali in opisali funkcionalnosti prototipa, ki podpirajo te vidike.

6 ZASNOVA SPLETNE IGRE ICO SIMULATOR

Z prototipom smo, v povezavi z razmislekom o uporabniški izkušnji, želeli prikazati prijeme pri generiranju naključnih števil z decentralizirano aplikacijo v obliki igre na srečo. Poskušali smo upoštevati ključne generične komponente spletnih iger na srečo, vendar smo se soočili še z enim posebnim izzivom – z načrtovanjem uporabniške izkušnje, kjer je ta, zaradi visokega praga znanja, ki je potrebno za sodelovanje, izjemno težavna.

Pri načrtovanju zgodbovnega jedra in načina grafične predstavitve se tako soočimo z dilemo – ali poskušati izboljšati uporabniško izkušnjo za splošne množice (kar predstavlja že osnovni problem okolja blokovnih verig) ali izboljšati uporabniško izkušnjo za ožji segment uporabnikov, ki že poznajo specifike uporabe blokovnih verig [20-22]. Glede na prej navedene zadrege glede uporabe blokovnih verig v širši javnosti in pogovorom z strokovnjaki in obstoječimi uporabniki decentraliziranih aplikacij smo sklenili uporabniško izkušnjo prilagoditi za ožji segment uporabnikov, ki tovrstne rešitve že poznajo, prepoznavajo njihovo uporabnost in povezane vsebinske elemente.

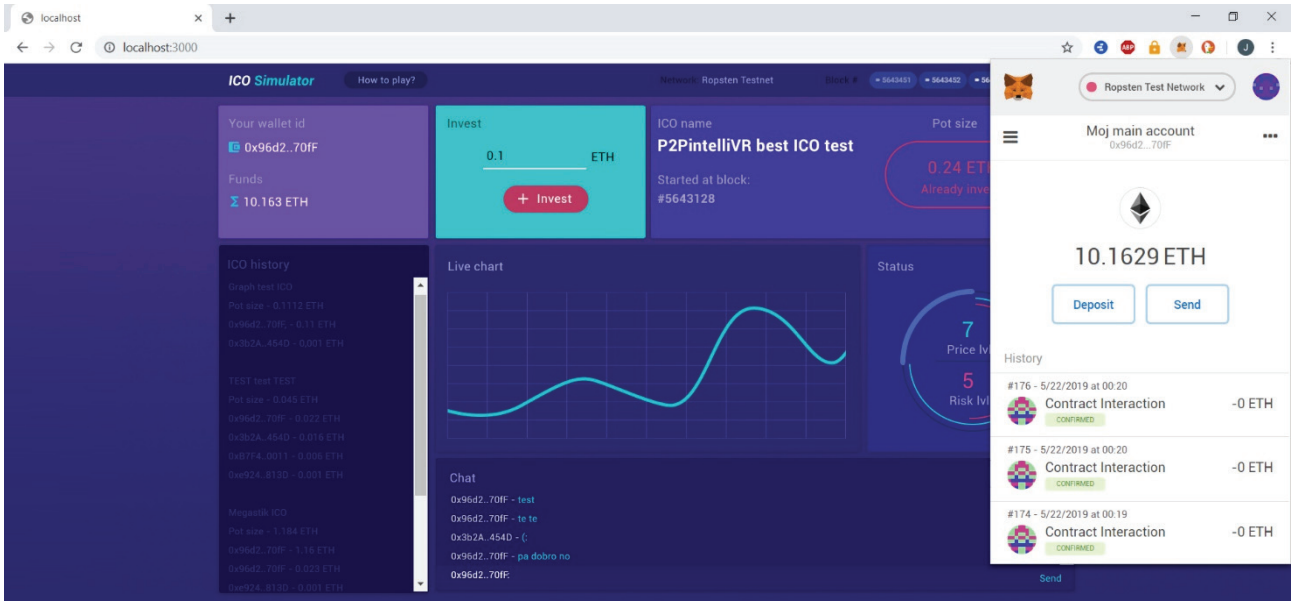
Razvpitost poslovnega modela zbiranja kapitala v kriptovalutah, t.i. ICO (angl. Initial Coin Offering) skupinskega financiranja je konceptualna podlaga za mehanizem igre, ki bi bila na ta način lahko na nek način znana in zanimiva za uporabnike, ki sicer že poznajo okolje in uporabo blokovnih verig.

Spletno igro na srečo v obliki decentralizirane aplikacije smo poimenovali ICO Simulator. Gre za zabaven, rahlo parodičen destilat dogajanja na področju ICO zbiranja denarja preko kriptovalut in nadaljnjih aktivnosti, kjer je bil, kljub pričakovanjem navadnih vlagateljev, da bo to inovativna tehnologija, glavni katalizator sprememb v cenah promocijski ustroj, ki je z manipulacijo in različnimi psihološki pritiski krojil usodo vloženega denarja.

Z ozirom na pomembne komponente spletnih iger s poudarkom na vzpodbujanja zaupanja smo elemente uporabniškega vmesnika razdelili na posamezne sklope:

Elementi uporabniškega vmesnika (Slika 1):

- Vzpodbujanje zaupanja v storitev:
  - stanje omrežja in generiranja blokov,
  - identifikacija uporabnikove denarnice in stanje sredstev v povezani denarnici,
  - zgodovina krogov in dobitkov.
- Družabnost:
  - komunikacija med prisotnimi igralci,
- Dosežki pri igranju z vizualno stimulacijo:
  - prikaz sredstev v stavi trenutnega kroga,
  - animiran graf, ki se giblje v skladu z trenutno stopnjo.
- Pristna izkušnja preko poznanih vsebinskih konceptov:
  - besedila in aktivnosti so preneseni iz različnih scenarijev iz ICO področja,
  - uporabniško generirano ime in podatki trenutnega kroga.



Slika 1: Spletni vmesnik spletne igre na srečo

Potek igre smo načrtovali, kot iskanje vzporednic v dogajanju množice ICO postopkov, skozi oči nepoučenega zunanjega opazovalca, in na ta način začrtali potek od začetka prodaje navideznih deležev, do propada projekta. Praviloma gre ob tem za kovanje dobičkov zgodnjih investitorjev na plečih kasnejših, ki so pripravljeni za sredstva oz. deleže plačati več.

Mehanizem igre:

1. Krog se začne z praznim stanjem sredstev
2. Igralec, ki prvi vplača sredstva ima pravico, da poimenuje svojo ICO krog.
3. Vsakič, oz. ob naslednjem generiranju bloka, ko katerikoli igralec vplača sredstva, se stopnja oz. krog poveča za 1.
4. Ob generiranju bloka z vplačilom se izžreba polno število od 0 do 9, to je 10 možnih rezultatov. Če je rezultat od 1 do 9, se igra nadaljuje in čaka nove »investitorje«. Če pa je rezultat 0, ICO naredi t.i. »exit scam« oz. vrednost pade na 0; s tem se igra zaključi.
5. Sredstva se po določenem ključu razdelijo med sodelujoče, največjo nagrade dobivajo tisti, ki so prvi »investirali« v ICO. Tisti, ki so prišli zadnji pa izgubijo vložek.
6. Igra se ustavi in ustvari se nov krog, ki čaka na prvo vplačilo.

Možnost, da se igra zaključi v posamezni dani stopnji je tako 1/10, vendar na nek način igralcu vizualno sporočamo kontinuiteto in zveznost stopenj. Igralca, kot pri vseh igrah na srečo, želimo odtujiti od pravilne ocene verjetnosti [23], da bo prišel do zanj ugodnega rezultata. V resnici iščemo verjetnost, da se igra ne bo zaključila prekmalu, vsaj ne dokler ni dovolj novih »investitorjem«, ki investirajo za nami. Te verjetnosti so temelj za pravilno oceno ključa po katerem se delijo vplačana sredstva ob zaključku igre – to se zgodi, ko psevdo-naključni generator števil v pametni pogodbi iz nabora 0..9 vrne ničlo.

Pri tem se zavedamo vsaj dveh skrajnih taktik, ki se jih poslužujejo igralci: (1) prvi oz. čimprej začeti igro in »investirati« dovolj, da znesek predstavlja dovolj velik delež pobranih sredstev, kar koristi ob končnem poračunu po določenem ključu; (2) investirati deleže v kasnejših korakih, da si zagotovimo končanje igre, vendar po tem, ko investirajo tudi drugi *večji tepci* (po principu »greater fool«). Rezultat bi bila dinamična igra, ki na videz osnovni princip nadgradi z sistematičnimi prijemi bolj izkušenih igralcev. Razumevanje take igre nam lahko, koristi pri povečavanju verjetnosti za dober delež vplačanih sredstev, vendar pa nam ga ne zagotavlja.

V prilagoditve modela bi lahko vpeljali tudi dodatna pravila, na primer najmanjši znesek »investicije« ali pa provizijo ponudnika – vendar za potrebe tega prototipa in boljše razumljivosti modela to ni potrebno. Dinamika igre in uporabniški vmesnik bosta predmet nadaljnjega testiranja, z namenom pridobitve dodatnih dopolnitev v mehaniki igre ali interakciji z igralci.

7 TEHNIČNA IZVEDBA PROTOTIPA

Prototip spletne igre na srečo ICO Simulator je implementiran kot decentralizirana aplikacija in je sestavljen, kot tipične take aplikacije, iz ospredja, ki lahko deluje na poljubni lokaciji, ter ob povezavi z spletom, z vmesnikom Metamask v brskalniku, komunicira z zalednim delom, ki skrbi za procesiranje in shranjevanje podatkov na blokovni verigi Ethereum.

7.1 Začelje – uporabniški del decentralizirane aplikacije

Uporabniški vmesnik (Slika 1) je narejen z osnovnimi spletnimi gradniki za strukturo (HTML5) in prikaz (CSS3). Izgled vmesnika je izboljšan z knjižnico Materialise CSS. Za dinamično vsebino skrbi VueJS z pomočjo knjižnice jQuery, ki podpira funkcionalnosti za animiranega grafe. Aplikacija je vpeta v ogrodje Express, ki se izvaja na mini strežniku Node.js za lokalno testiranje. Za namen javne objave smo vzpostavili odlagališče Github, ki je povezano s platformo HerokuApp, ki poskrbi za dostavo uporabniškega vmesnika preko javnega spletnega naslova v uporabnikov brskalnik. Vsi uporabniki imajo v igri enakovredno vlogo. Uporabniki praviloma uporabljajo spletni vtičnik v brskalniku Metamask za identifikacijo in prenos sredstev iz denarnice v pametno pogodbo aplikacije, za kar se da optimalno izkušnjo.

7.2 Zaledje – generiranje naključnega rezultata, procesiranje in shranjevanje podatkov

Operacije zaledja so v celoti podprte z namensko pametno pogodbo napisano v programskem jeziku Solidity v vmesniku Remix IDE in zaenkrat domuje na testnem omrežju Ropsten. V primeru prijave na glavno omrežje Ethereum (mainnet) nas aplikacija opozori, saj tam konkretna pametna pogodba ni objavljena, ne deluje in zato ne dopušča izgube sredstev. Administrator lahko ureja in objavi pametno pogodbo z lastniškimi pravicami vezanimi na njegovo zasebno denarnico, kar je zapisano v pametni pogodbi. Pogodba je objavljena na unikatnem javnem naslovu. Po tem ko pridobimo njen unikaten naslov in vmesnik ABI (Application Binary Interface) jo lahko povežemo z uporabniškim delom aplikacije. Vnosov, ki so shranjeni v pametno pogodbo ne moremo več spreminjati, v primeru popravkov moramo postopek objave ponoviti in ponovno definirati ABI v kodi ospredja.

Generiranje psevdo-naključnega števila od 0..9 poteka tako (Programska koda 1), da, ob novem vplačilu uporabnika, za zrna vzamemo javne naslove vseh udeleženih igralcev, po vrsti njihovih vplačil, ter jih začinimo s povzetkom zadnjega generiranega bloka. Vse to zgostimo s pomočjo transformacije niza znakov (*string*) v števila (*uint*), nato izračunamo vrednost od 0..9 ter vrnemo rezultat v uporabniški del, ki ustrezno prilagodi vmesnik, stopnja vizulne stimulacije se poveča, v primeru pa, da je rezultat enako 0, razdelimo vsa vplačila po izbranem ključu, ter podatke o dobitkih zapišemo v zgodovino dobitkov.

Programska koda 1: Izsek pametne pogodbe, ki se nanaša na psevdo-naključno generiranje števil. V tem prispevku ni celotne pametne pogodba, ki je predmet iterativnih izboljšav vezanih na mehaniko igre.

```
pragma solidity >=0.4.20;
contract ICOsimulator {
  ..//..
  struct CurrentRoundInvestor { /* new for every round */
    bytes32[] stageEntered; /* enetered stage */
    uint balanceOf; /* investment in entered stage */
  }
  mapping (address => CurrentRoundInvestor ) public users; /* current users mapping*/
  address[] private usersList; /* currentusers list for iterations */
  ..//..

  function GetRandomNumber() public view returns (uint256 result){ //  from CurrentRoundInvestor
addresses string to int
    for (uint i=0; i<usersList.length; i++) {
      string memory usersListString = new string(usersListString + usersList[i]);
    }
  uint256 ConcatenateAddressesInt = CompoundAddresses(usersListString);
```

```
  uint max = 9; // 0..max
  uint256 factor = ConcatenateAddressesInt * 100 / max;
  uint256 lastBlockNumber = block.number - 1;
  uint256 hashVal = uint256(blockhash(lastBlockNumber));
  return uint256((uint256(hashVal) / factor)) % max;
  }
  ..//..
}
```

8 ZAKLJUČEK

Popolnoma varne implementacije psevdo-naključnega generiranja števil na platformi Ethereum ostajajo redke, saj blokovne verige ne ponujajo veliko možnosti za entropijo, kar pogosto vodi v napake razvijalcev na tem področju. Za praktične izvedbe so predlagani pristopi zadostni, vendar je potrebno pred tem analizirati scenarije uporabe decentralizirane aplikacije in možnosti vplivanja na rezultate iz strani uporabnika, ponudnika ter podpornikov infrasktrukture.

V tem članku smo prikazali postopek razvoja decentralizirane aplikacije v obliki spletne igre na srečo, od osnovnega definiranja cilja, kako narediti optimalen generator psevdo-naključnih števil, do konceptualne zasnove mehanike igre in načrtovanja uporabniške izkušnje, do same tehnične izvedbe in kodiranja ter objave na spletu.

Sama izkušnja razvoja pametnih pogodb, v okolju Ethereum, se v preteklem letu v resnici ni bistveno spremenila. Razen lastnega širjenja znanja jezika in kompleksnejših struktur ter dogodkov, ni na voljo novih orodij ali radikalno boljših pristopov. Medtem je razvoj v drugih, konkurenčnih, platformah (npr. Hyperledger Fabric) opazneje napredoval. Vsekakor je opaziti popularizacijo tovrstnih rešitev v razvoju, saj vedno več amaterskih in profesionalnih programerjev spoznava tovrstne decentralizirane rešitve, vendar, po oceni avtorjev, uspešne produktivizacije ni bilo zaznati oz. gre še vedno predvsem za nišne rešitve ali dobre prototipe. To seveda ne pomeni, da razvoju decentraliziranih aplikacij na Ethereumu in drugih platformah ni potrebno namenjati pozornosti, prav nasprotno, glede na to, da so decentralizirane zaupanja vredne rešitve še v začetni fazi razvoja in popularizacije, moramo v tem prostoru, kjer se pojavljajo dvomi in določeni izzivi, s posebno zbranostjo spremljati njihov napredek.

9 LITERATURA

[1] TAPSCOTT Don & TAPSCOTT Alex, Blockchain revolution, 2016, https://blockgeeks.com/guides/what-is-blockchain-technology/, obiskano 6.5.2019.

[2] Wikipedia: Blockchain, en.wikipedia.org/wiki/Blockchain, obiskano 1.3.2019.

[3] BALIGA Arati, Understanding Blockchain Consensus Models, Persistent Systems, White Paper, 2017, https://www.persistent.com/wp-content/uploads/2018/02/wp-understanding-blockchain-consensus-models.pdf, obiskano 6.4.2019.

[4] PAUNOVIC Goran, The New Frontier: Decoding User Experience For Blockchain Technology, 2018, https://www.forbes.com/sites/forbesagencycouncil/2018/06/19/the-new-frontier-decoding-user-experience-for-blockchain-technology/#3e2a1c363c44, obiskano 25.4.2019.

[5] FRISBY Adam, Why blockchain isnt't ready for the primetime, Sinespace, 2018, https://venturebeat-com.cdn.ampproject.org/c/s/venturebeat.com/2018/03/11/why-blockchain-isnt-ready-for-primetime/amp/, obiskano: 6.4.2019

[6] MARR Bernard, The 5 big problems with blockchain everyone should be aware of, Forbes, 2018, https://www.forbes.com/sites/bernardmarr/2018/02/19/the-5-big-problems-with-blockchain-everyone-should-be-aware-of/#5adf0c661670, obiskano 24.3.2019.

[7] AUNE Rune, KRELLENSTEIN Adam, O'HARA Maureen, SLAMA Ouziel, Footprints on a Blockchain: Trading and Information Leakage in Distributed Ledgers. The Journal of Trading,12(3), 5-13, 2017.

[8] BREGAR, Andrej, KAFOL, Ciril, TRILAR, Jure, NOSAN, Matej. Razvoj programskih rešitev veriženja blokov za energetsko poslovno domeno. V: HERIČKO, Marjan (ur.), KOUS, Katja (ur.). Sodobne informacijske tehnologije in storitve: OTS 2018 : zbornik triindvajsete konference, Maribor, 19. in 20. junij 2018. 1. izd. Maribor: Univerzitetna založba Univerze: Fakulteta za elektrotehniko, računalništvo in informatiko. 2018, str. 20-36.

[9] PRUSTY Narayan, Building Blockchain Projects: Building Decentralized Blockchain Applications with Ethereum and Solidity, Pack Publishing, 2017.

[10] Ethereum JavaScript API, github.com/ethereum/web3.js, obiskano 13.3.2019.

[11] What is the »Gas« in Ethereum?, www.cryptocompare.com/coins/guides/what-is-the-gas-in-ethereum/, obiskano 13.3.2019.

[12] Proof of Work vs Proof of Stake – What is POW & POS mining?, bitcoinexchangeguide.com/proof-of-work-vs-proof-of-stake-mining, obiskano 13.3.2019.

[13] Proof of Stake FAQ, github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ, obiskano 14. 3. 2019.

[14] REUTOV Arseny, Predictiong Random Numbers in Ethereum Smart Contracts, 2018, https://blog.positive.com/predicting-random-numbers-in-ethereum-smart-contracts-e5358c6b8620, obiskano 21.4.2019.

[15] HOFFMAN Chriss, How Computers Generate Random Numbers, 2017, https://www.howtogeek.com/183051/htg-explains-how-computers-generate-random-numbers/, obiskano 21.4.2019.

[16] SHELAT Bhiru, EGGER Florian, What makes people trust online gambling sites?, Extended Abstracts on Human Factors in Computing Systems, CHI 02, 2002.

[17] CHOI Dongseng, KIM Jinwoo, Why People Continue to Play Online Games: In Search of Critical Design Factors to Increase Customer Loyalty to Online Contents, CyberPsychology & Behavior,7(1), 11-24, 2004.

[18] YEE Nick, Motivations for Play in Online Games, CyberPsychology & Behavior,9(6), 772-775, 2006.

[19] JAMES Richard, O'MALLEY Claire, TUNNEY Richard, Understanding the psychology of mobile gambling: A behavioural synthesis, British Journal of Psychology,108(3), 608-625, 2016.

[20] SNEIDER David, How We Solved Blockchain Application User Experience: The Benefits of the Blockchain with the Usability of Web 2.0, 2019, https://hackernoon.com/how-we-solved-blockchain-application-user-experience-1b422acb1346, obiskano 15.3.2019.

[21] PALAYER Guillaume,19 Ways to Cheat at dApp Design, 2018, https://medium.com/betoken/19-ways-to-cheat-at-dapp-design-11bb4266bf7c, obiskano 15.3.2019.

[22] TSE Chris, Rethinking the Web 3.0 Experience: »Decentralization« isn't enough, 2018, https://medium.com/cardstack/rethinking-the-web-3-0-experience-9b5fe508aa77, obiskano 15.3.2019.

[23] GRADIŠNIK, Branko: Igre: Volčje in ovčje: Priročnik za hazarderje in proti njim, DZS, 1993.

OTS 2019
SODOBNE INFORMACIJSKE TEHNOLOGIJE IN STORITVE
M. Heričko in K. Kous (ur.)

Univerzitetna založba
Univerze v Mariboru

ORKESTRACIJA GRUČ APLIKACIJ Z UPORABO DOCKER SWARM NA ROBNEM OBLAKU

URBAN ZALETEL, MARKO DERGANC, ROK PETRIČ

Povzetek: *Trend razvoja programske opreme so že nekaj časa mikroservisi [1]. Za pakiranje teh mikroservisov je najbolj razširjen vsebnik Docker. S pojavom orkestratorskih orodij kot so Rancher, Kubernetes in Docker Swarm se je trend nagnil v smer, kjer želimo imeti čim več naših komponent znotraj teh orodij, saj omogočajo hitro in ponovljivo avtomatsko dostavo, pri čemer je enostavno vračanje v prejšnje stanje. Večina uporablja Kubernetes kot plačljivo verzijo v oblaku znano kot GKE (Google Kubernetes Engine), ki je po mnenju nekaterih tudi edina zadovoljiva implementacija orkestratorja Kubernetes. V poslovnih okoljih smo večkrat omejeni z uporabo javnega oblaka, kljub temu pa bi radi uporabljali enaka orodja, ki so na voljo v oblaku za razvoj sodobne programske opreme. V nasprotju z Googlom smo pogosto omejeni tudi z zmogljivostjo strojne opreme ter njenim številom. V prispevku bomo predstavili odprtokodno knjižnico, ki smo jo razvili v Iskratelu in omogoča postavitev gruč in aplikacij s stanjem na Docker Swarmu. Predstavljene bodo ključne značilnosti knjižnice, kako jo lahko uporabimo in razširimo sami, hkrati pa bomo nekaj besed namenili tudi Docker Swarmu, orkestratorskem orodju, ki se na malih postavitvah izkaže z enostavnostjo in minimalnimi sistemskimi zahtevami. Razlogi za uporabo »lažjih« orkestratorskih orodij so na robnem oblaku (Edge cloud) še toliko bolj izraziti, saj imamo namesto gruče desetih in več strežnikov, po navadi na voljo le tri strežnike, kamor želimo namestiti čim več aplikacij.*

Ključne besede: *Docker Swarm, orkestracija, vsebnik, gruča aplikacij, robni oblak, Raft protokol*

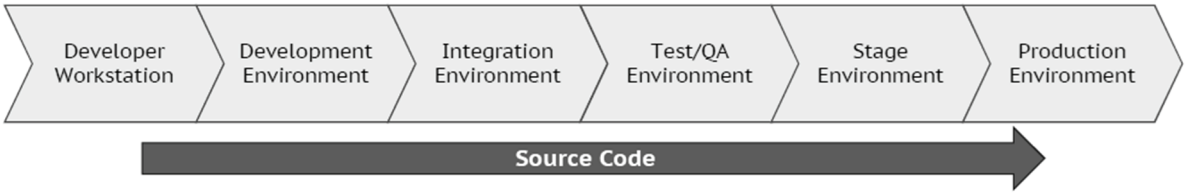
NASLOVI AVTORJEV: Urban Zaletel, IskraTEL d.o.o., Kranj, Slovenija, e-pošta: zaletel @iskratel.si. Rok Petrič, IskraTEL d.o.o, Kranj, Slovenija, e-pošta: r.petric@iskratel.si. Marko Derganc, IskraTEL d.o.o., Kranj, Slovenija, e-pošta: derganc@iskratel.si.

https://doi.org/10.18690/978-961-286-282-4.7 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 UVOD

Ko so marca leta 2013 pri podjetju Docker inc. izdali istoimensko odprtokodno rešitev za pakiranje, prenosljivost in poganjanje aplikacij, so pomembno vplivali na razvoj programske opreme. Lahko govorimo o tem, da so spremenili način, kako razvijamo programsko opremo ter tudi hitrost razvoja. Z vsebniki je tudi mikroservisna arhitektura dobila velik zagon, saj smo z njimi pridobili enostavno pakiranje aplikacij z vsemi odvisnostmi. Vse hitrejši razvoj aplikacij pa vpliva tudi na vse razvijalce programske opreme, saj imajo možnost izbrati programski jezik, kjer bodo najhitreje rešili problem ne glede na izvajalsko okolje. Tako imamo namesto enega izvajalnega okolja, več različnih. Idealno je, da imamo izvajalska okolja ločena med seboj, kar je pred prihodom vsebnikov pomenilo veliko število virtualnih strežnikov in konfiguracijsko nočno moro za sistemske in DevOps inženirje, ki so morali vzdrževati tak sistem. Vsebnikov so se zato najbolj razveselili DevOps inženirji, saj so bili prav oni odgovorni za nameščanje aplikacij v produkcijo. Z vsebniki smo dobili izolativnost med aplikacijami in njihovimi odvisnostmi. Hkrati pa vsebniki prinesejo zelo malo režijske porabe CPU in RAM, saj si vsi delijo vire strežnika nad katerem tečejo.

Aplikacijski nivo in odvisnosti smo tako z vsebniki ločili od infrastrukture na kateri tečejo. Z uporabo vsebnikov so se odvisnostne matrike omejile samo na kompatibilnostno matriko aplikacij in ne kompatibilnostno matriko aplikacij in njihovih odvisnosti. Z razmahom mikroservisov in njihovim hitrim povečevanjem pa se je hitro pokazala težava upravljanja posameznih vsebnikov in njihovo nadzorovanje.



Slika 1. Aplikacijo zapakiramo v vsebnik in jo pošljemo po cevovodu - poteka enako na vsakem koraku ne glede na infrastrukturo na kateri teče

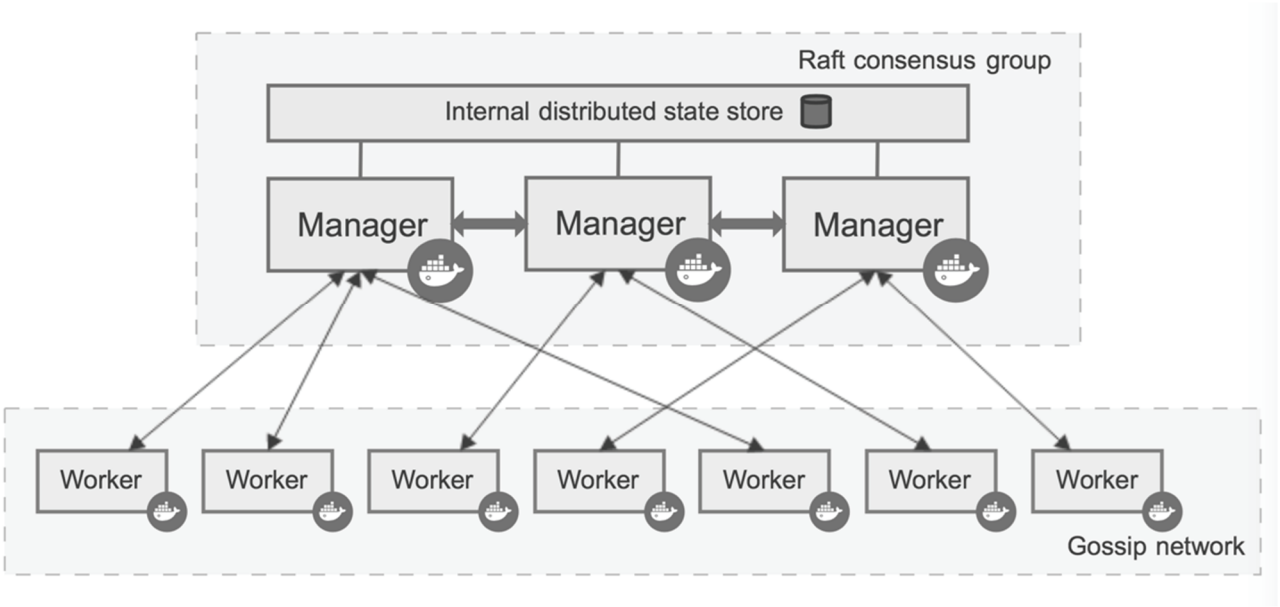
Kubernetes in Docker Swarm, ki sta v krstni različici nastala dobro leto po izdaji vsebnikov Docker, sta naslovila problem upravljanja vsebnikov in še pospešila popularizacijo vsebnikov za pakiranje aplikacij.

Orkestracijska orodja za upravljanje življenjskega cikla vsebnikov kot sta Docker Swarm in Kubernetes, sta v ekosistemu vsebnikov povsem izpodrinila orodja za konfiguracijo aplikacij, kot so Chef, Ansible in Puppet, ki smo jih poznali pri klasičnem nameščanju aplikacij.

2 DOCKER SWARM

Na začetku je bilo veliko primerjav Docker Swarma in Kubernetesa [2]. Večina je bila navdušena nad enostavnostjo Docker Swarma in odprtostjo Kubernetesa. Prevladala je odprtost, podpora za modularnost in povsem odprtokodni projekt Kubernetes. Glede na Google Trend analize je februarja 2019 popularnost Kubernetesa in iskanja ključne besede v povprečju zadnjih 12 mesecev dosegla 75%, medtem ko je popularnost Docker Swarma in iskanja po spletu ključnih besed povezanih z njim, le 5%. Usoda Docker Swarma je negotova in nejasna, saj je podjetje Docker na svoji platformi Docker EE [3] podprlo oba orkestratorja, kar vliva še dodatno nezaupanje do uporabe Docker Swarma. Medtem se Kubernetes razvija s svetlobno hitrostjo, v projekt je do aprila 2019 prispevalo že 2112 kontributorjev.

Roj (ang. swarm) je skupina virtualnih ali fizičnih strežnikov na katerih teče Docker povezanih v gručo. Strežnik povezan v gručo imenujemo vozlišče (ang. node). Swarm nadzorniki (ang. swarm managers) so edina vozlišča v gruči, ki lahko izvršijo Docker ukaze ali avtorizirajo vozlišča, da se pridružijo roju kot delavci (ang. workers). Nadzorniki, lahko uporabljajo različne strategije pri poganjanju vsebnikov, kot npr. »najbolj prazen node«, kjer vsebnike namesti na strežnike, ki so najmanj obremenjeni oz. v primeru »global«, kjer namesti vsebnike z eno instanco na vsako vozlišče v gruči.



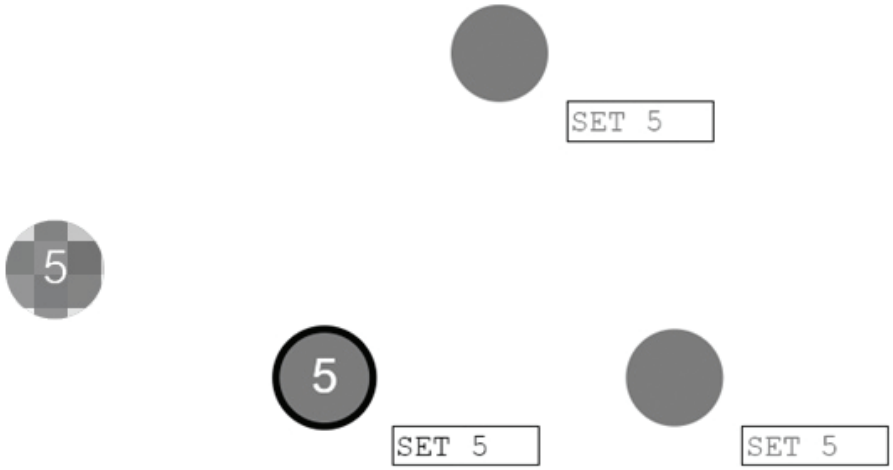
Slika 2. Arhitektura Docker Swarm.

2.1 Raft protokol

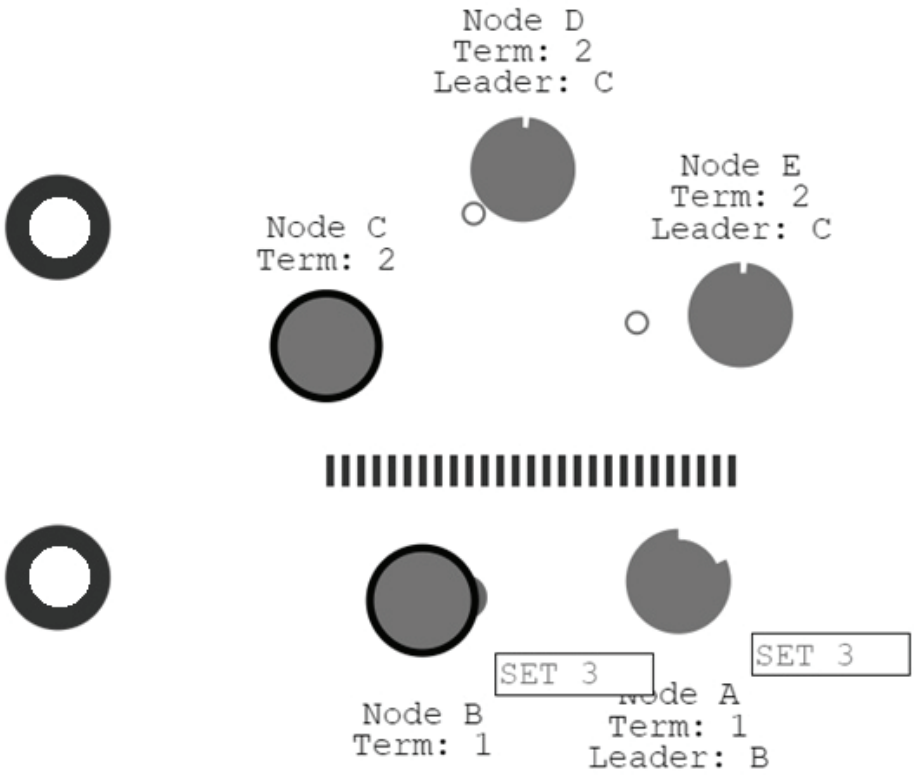
Docker Swarm gruča uporablja algoritem soglasja Raft (ang. Raft consensus algorithm) [4]. Raft je protokol, ki implementira distribuirano soglasje, to pomeni, da se v distribuiranih sistemih več strežnikov strinja z vodjo, ki je izvoljen. Vozlišče ima lahko 3 stanja: sledilec, kandidat in vodja. (ang. follower, candidate, leader). Vsa vozlišča v gruči začnejo v stanju sledilec in če v gruči ne slišijo vodje, potem lahko sledilec postane kandidat. V protokolu sta pomembni dve časovni omejitvi (ang. timeout). Prva je volitvena časovna omejitev (ang. election timeout). To je čas, da sledilec (ang. follower) postane kandidant, ta vrednost je naključno generirana in znaša med 150 in 300ms. Po pretečeni volitveni omejitvi, kandidat štarta nov volitveni krog, glasuje zase in zahteva glasove od ostalih vozlišč v gruči. Če kandidat dobi večino glasov, potem postane vodja. Temu procesu rečemo elekcija vodje. (ang. leader election). Vodja periodično pošilja sporočilo dodajte vnose (ang. append entries) svojim sledilcem. Sledilci odgovorijo na vsako sporočilo in v primeru, da sledilec preneha dobivati sporočila se zažene nova elekcija vodje.

V primeru štirih vozlišč v gruči, kjer dva vozlišča hkrati postaneta kandidata, se lahko zgodi cepljeno glasovanje, ko obe vozlišči dobita po en glas od sledilcev. Ker noben od njiju ne dobi večine, nihče ne postane vodja in ponovno se zažene cikel glasovanja.

Vodja je tisto vozlišče, ki beleži vse akcije v gruči, npr. dodajanje vozlišča, odstranjevanje, kreacija servisa... Roj poskrbi, da je beležka vodje replicirana na vseh nadzornikih, da v primeru, ko vodja postane nedosegljiv, katerikoli nadzornik lahko prevzame vlogo vodje. Beležka (ang. log) je kriptirana. Komunikacija med vozlišči poteka pa po TLS protokolu. Raft algoritem tolerira (N-1)/2 izgub in zahteva večino v kvorumu (N/2)+1 članov, da se dogovorijo o vrednostih predlaganih za gručo. To pomeni, da se v gruči petih nadzornikov, tolerira izguba dveh. V primeru, da tri vozlišča niso več na voljo, potem sistem ne more več obdelovati zahtev in akcij. Obstoječe naloge se sicer nadaljujejo, vendar pa planer (ang. scheduler) ne more uravnovesiti nalog.



Slika 3. Kockasko vozlišče (klient), da zahtevo po zapisu številu 5. Šele ko večina v gruči potrdi zapis, se vrednost tudi potrdi (ang. commit) v vodji. Vodja nato obvesti sledilce (ang. followers), da je zapisal vpis. Gruča je sedaj soglasna o stanju sistema.



Slika 4. Swarm s petimi vozlišči v gruči (modra krogci), zaradi omrežne nedoseljivosti razpade na dve coni (A in B, ter C, D in E). Ker imamo dve coni, dobimo tudi dve vodji. Ko klienta (črna krogca z belo piko) želita posodobiti vrednost (en število 3, drug število 8) vsak v svoji coni, uspe samo enemu v coni, ki ima večino, saj ko se omrežna dosegljivost vzpostavi nazaj, se vrednost 3, ki jo je nastavil klient v coni, ki ni imela večine ne replicira, saj se njen log ni zapisal (ang. commit).

2.2 Problem aplikacij v gruči v Docker Swarmu

Vsebniki so pisani na kožo aplikacijam brez stanj. V svoji osnovi sta orkestratorja Docker Swarm in Kubernetes idealna prav slednjim. Ker pa so vsebniki zelo prikladen način za pakiranje in distribucijo aplikacij, orkestratorja Docker Swarm in Kubernetes pa za upravljanje z našimi aplikacijami, seveda želimo vse naše servise preseliti v to okolje. [5]

Kubernetes je že v zgodnjih verzijah ponujal več načinov namestitev: replikacijski niz, namestitev in niz s stanjem (ang. replica set, deployment, stateful set), kasneje pa je z razširitvijo z operatorji (ang. operators) dobil vsa orodja za poganjanje aplikacij, ki potrebujejo operativno znanje. Čeprav mnogi primerjajo niz s stanjem z operatorji, gre povsem za dve različne stvari. Niz s stanjem je zasnovani tako, da omogočajo podporo v Kubernetes za aplikacije, ki zahtevajo, da jim gruča daje »stalna sredstva«, kot so statični IP-ji in shramba. Aplikacije, ki potrebujejo ta model s stanjem, pa še vedno lahko potrebujejo operaterjevo avtomatizacijo, npr. za opozarjanje in ukrepanje ob napaki, varnostni kopiji ali ponovni konfiguraciji.

Torej, kaj točno so operatorji v Kubernetesu? Operatorji so aplikacije, ki upravljajo druge aplikacije, s tesnim povezovanjem s Kubernetes API-jem. Omogočajo nam, da v njih vgradimo svoje lastno "operativno znanje" o aplikaciji in izvedemo avtomatizirana dejanja pri upravljanju teh aplikacij.

V Docker Swarmu nimamo tako širokega nabora namestitev naših aplikacij in ravno tu tiči problem v upravljanju aplikacij, ki potrebujejo operativno znanje.

2.3 Razlogi za uporabo Docker Swarm

Čeprav se zavedamo, da je Kubernetes zmagovalec v dvoboju orkestratorkih orodij za vsebnike in na spletu najdemo precej člankov z naslovom [6]: »Docker Swarm je mrtev, naj živi Kubernetes«, je v nekaterih primerih uporaba Docker Swarma še vedno smotrna. V primeru, da sami upravljate z infrastrukturo in kjer nimate na voljo plačljivega servisa v oblaku, kjer bi samo zakupili Kubernetes gručo, kot na primer EKS ali GCE, se hitro pokaže kompleksnost namestitve Kubernetes gruče v HA načinu in upravljanju.

2.3.1 *Robni oblak*

Robni oblak je oblak, ki ima računsko zmogljivost bližje omrežnemu prometu oz. bližje končnemu uporabniku. Robni oblak je odgovor na zahtevo po manjši latenci in izjemno veliki pretočnosti. Robni oblak postavlja vire torej tam, kjer promet. Ravno zaradi so lokacije precej pogostejše in manjše v primerjavi s centralnim oblakom. Robni oblak lahko deluje povsem avtonomno, seveda pa ima lahko interakcijo z večjim centralnim oblakom.

Na robnem oblaku smo zelo omejeni s strojnimi zmogljivostmi, saj imamo tipično na voljo le 1-3 strežnike, kjer mora biti programska oprema za delovanje oblaka, kot na primer Openstack. Le-ta je nameščena v primeru robnega oblaka v virtualnih strežnikih ali vsebnikih in optimizirana za manjše število strežnikov, ki jih bo nadzorovala v tem oblaku. Ravno zaradi potrebe po optimizaciji, enostavnosti in majhnemu številu strežnikov je uporaba Docker Swarma precej smotrna. V bližnji prihodnosti pa se uspe spremeniti tudi to, in sicer s projektom, ki ga razvijajo pri Rancher Labs - K3s [7] ali s projektom KubeEdge.

3 DOCKER BATON

3.1 Razlog za nastanek

Izpostavila se je potreba po dodatni orkestraciji nekaterih obstoječih aplikacij, ki smo jih morali »dockerizirat«. Problem je v tem, da Docker po zagonu vsebnika, aplikacije ne konfigurira več. Primer: aplikacija, ki uporablja Infinispan gručo, mora poznati IP naslove vseh članov gruče. Sam Swarm tega ne omogoča, Kubernetes pa ni primeren za naš robni oblak.

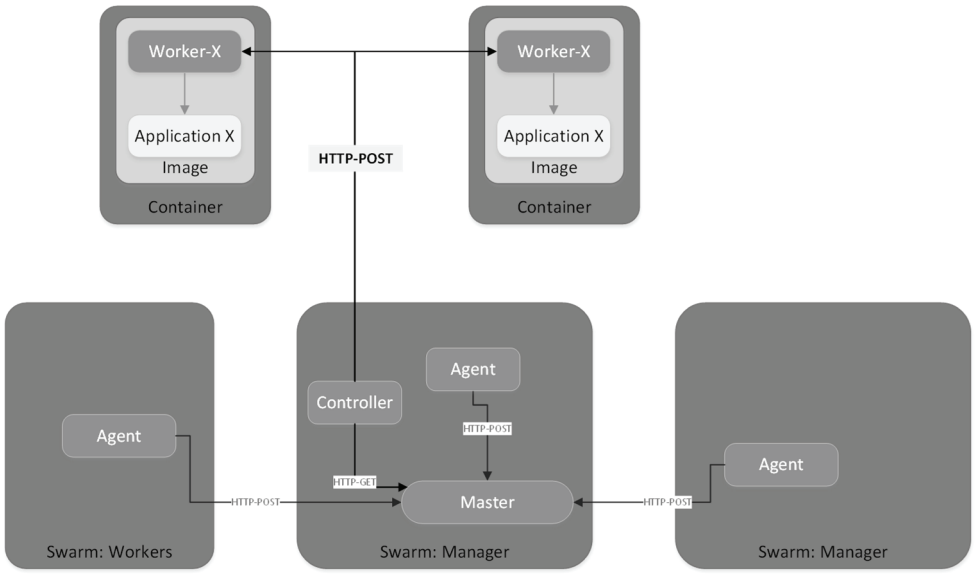
Skratka, iskali smo možnost, kako konfigurirati, zagnati, prekonfigurirati in ustaviti aplikacijo v vsebniku, ko za to nastanejo pogoji, ki pa so zunaj Swarmove orkestracijske domene.

Docker Engine ima API, ki omogoča pridobivanje podatkov, ki jih potrebujemo za dodatno orkestracijo neke aplikacije. Morali smo rešiti problem, kako pridobiti te podatke in jih v primernem trenutku poslati v vsebnik, sam vsebnik pa potem izvede vse potrebne akcije. Tako je nastal Docker Baton.

3.2 Arhitektura

Docker Baton sestavljajo sledeče komponente:

- poveljnik (ang. master)
- agent (ang. agent)
- krmilnik (ang. controller)
- delavec (ang. worker)



Vse komponente, rezen delavca (ang. worker), so nameščene v Swarm kot Docker-Baton Stack. Delavec mora biti vrinjen v sliko (ang. image) specifične aplikacije.

Komunikacija med komponentami poteka preko generičnega Docker-Baton API-ja.

Razen delavca, so vse ostale komponente Docker Baton-a generične. Poveljnik in krmilnik tečeta vsak v eni instanci na enemu izmed Swarm manager vozlišč. Agent teče globalno – ena instanca na Swarm vozlišče.

3.3 Delovanje

Docker Baton deluje v vsaki Swarm gruči v Linux okolju. Konfiguracija ni potrebna. Začasen izpad ene ali več komponent ne povzroči nobenih problemov. Vsi podatki so samo v RAM-u, izguba podatkov ne predstavlja problema, saj se podatki slej ko prej obnovijo. Komponente izmenjujemo podatke preko http REST API v JSON obliki in sicer v lastnem omrežju (DockerBatonNetwork).

Ker obstaja samo en poveljnik v stacku, je dosegljiv preko imena (hostName=master)

3.3.1 *Poveljnik*

Poveljnik (ang. master) je centralna komponenta za podatke Docker okolja. Postavi se v eni repliki na swarm nadzornem vozlišču. Njegove naloge:

- Preko DockerEngine REST-a neprestano (vsakih 10 ms) zahteva podatke o Swarm okolju (a.k.a. node ls, stack ls, service ls, stack ps)
- Preko svojega REST-a sprejema »docker container inspect« podatke o vsebnikih
- Ažurira in integrira zbrane podatke
- Odstranjuje podatke, ki so starejši od deset sekund
- Za dostop do trenutnih podatkov ima svoj http REST Server

3.3.2 *Agent*

Postavi se globalno: eno vozlišče, en agent. Njegova dela:

- neprestano preko Docker Engine REST-a zahteva podatke o vsebnikih, ki tečejo na »njegovem« vozlišču (a.k.a. docker container inspect)
- Pridobljene podatke takoj pošlje poveljniku, v kolikor le-ta ni dosegljiv jih zavrže

3.3.3 *Krmilnik:*

Postavi se v eni repliki. Ima pa samo dve nalogi:

- Neprestano zahteva od komponente poveljnik podatke, ki so relevantni za dodatno orkestracijo neke aplikacije
- Ko detektira, da je prišlo do spremembe podatkov, podatke pošlje v relevantne vsebnike (http POST, JSON body) po generičnemu algoritmu.

Algoritem je enostaven in robusten:

- vsi vsebniki kontrolirane aplikacije morajo biti v stanju RUN
- vsi IP naslovi morajo biti znani
- kopije podatkov se pošilja vsebnikom v sekvenci, novejši vsebniki so na vrsti prej
- podatke sestavljajo generični parametri docker okolja kontrolirane aplikacije
- naslednji vsebnik je na vrsti za tem, ko trenutni vsebnik odgovori z 200 OK
- če odgovor ni 200 OK, se sekvenca prekine (seveda se proces ponovi, pri naslednji spremembi podatkov; t.j. docker okolja)

3.3.4 *Delavec*

V sliko neke aplikacije namestimo za to aplikacijo specifičnega delavca in sicer tako, da delavec zaganja to aplikacijo in ne več Docker - Docker zažene delavca. S tem ima delavec kontrolo nad aplikacijo: lahko jo zažene, ustavi, prisilno ustavi in konfigurira. Konfiguracija je specifična za aplikacijo.

Delavec ima sledeče generične procese:

- spletni strežnik
- servis, ki upravlja aplikacijo
- servis, ki preverjanje zdravje aplikacije

Procesi zapisujejo in berejo sledeča stanja:

- stanje aplikacije, možna stanja: ustavljen, štartan, v teku, napaka (ang. stopped, started, run, error)
- zdravje aplikacije, možna stanja: neznano in zdrav (ang. unknown, healthy)

Obe stanji spremljajo vsi procesi, ki morajo ob spremembi stanja ustrezno odreagirati. Npr, Web strežnik na zahtevo odgovori z 200 OK, ko zazna stanje v teku. Če pa zazna stanje napaka, odgovori z 400 Bad Request. Stanje napaka lahko zazna samo v enem primeru: ko je start aplikacije neuspešen zaradi napake, ki jo povzroči hrošč (ang. bug) – predviden Linux Shell ukaz se ne more izvršiti.

Ker, in to je pomembno, krmilnik nadaljuje s procesom orkestracije samo, če je odgovor 200 OK, sicer prekine proces.

Ni statusa nezdrav (ang. unhealthy). Če se zdrava aplikacija ne odziva več servisu za preverjanje zdravja, potem servis uniči vsebnik. Delavec, ki teče v terminalu vsebnika, naredi non-zero exit; za ostalo poskrbi Swarm.

3.4 Jezik Go – zakaj

Razlog, zakaj je bil izbran jezik Go je ta, da primernejšega jezika za implementacijo takšne rešitve ni. In sicer zato:

- Odprtokodni Docker Go SDK
- Docker Engine REST API uporablja JSON in UNIX socket
- V vsaki Docker Baton komponenti hkrati poteka več procesov, ki si med seboj delijo podatke
- Prevedena koda nima nobenih odvisnosti od drugih knjižnic
- Delavec »paše« v vse slike
- Go je enostaven jezik, ki pa je vsaj tako hiter kot C

3.5 Compose File

docker-baton-stack.yml

```
services:
  agent:
    image: …/agent
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    networks:
      - master_network
    deploy:
      mode: global
      placement:
        constraints: [node.platform.os == linux]

  master:
    image: …/master
    ports:
      - "18080:18080"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    networks:
      - master_network
    deploy:
      mode: replicated
      replicas: 1
      placement:
        constraints: [node.role == manager]

networks:
  master_network:
    driver: overlay
    attachable: true
```

3.6 Opensource na Githubu

Projekt bo objavljen tudi Github profilu: https://github.com/iskratel, kjer se bodo dobila tudi navodila za uporabo projekta in možne razširitve.

4 ZAKLJUČEK

Kljub prevladi Kubernetesa med orkestracijskimi orodji za vsebnike je v enostavnih primerih še vedno smotrno uporabiti Docker Swarm - predvsem zaradi zelo nizke krivulje učenja. Hkrati s knjižnico Docker Baton dobimo vse možnosti poganjanja aplikacij v gruči, ki zahtevajo operativno znanje. Ob morebitnem prehodu na Kubernetes se lahko poslovno logiko uporabi v operatorjih in sicer se poslovno logiko delavca lahko prepiše v funkcijo uskladiti (ang. reconcile), ki je v operatorjih.

5 LITERATURA

[1] Mag. Andrej KRAJNC, Urban ZALETEL, mag. Bojan ŠTOK, dr. Ciril PETR "Orkestracija mikroservisov z uporabo Kubernetes", Sodobne tehnologije in storitve OTS 2017: Zbornik dvaindvajsete konference, Maribor, Fakulteta za elektrotehniko, računalništvo in informatiko, Inštitut za informatiko

[2] https://rancher.com/blog/2019/kuberntes-versus-docker-swarm/, Podrobnosti in razlike med Docker Swarm in Kubernetes orkestacijo, obiskano 29. 4. 2019.

[3] https://blog.docker.com/2018/05/integrating-kubernetes-docker-enterprise-edition-2-0-top-10-questions-docker-virtual-event/, Informacije Docker Enterprise Edition, obiskano 29. 4. 2019.

[4] http://thesecretlivesofdata.com/raft, Informacije o Raft protokolu, obiskano 5.5.2019

[5] https://coreos.com/blog/introducing-operators.html, Informacije o operatorjih v Kubernetesu, obiskano 10.52019

[6] https://medium.com/@reactiveops/docker-swarm-is-dead-long-live-kubernetes-2d0db0609e09, Informacije o dvoboju Kubernetes in Docker Swarm, obiskano 10.5.2019

[7] https://k3s.io/, Informacija o produktu K3s, obiskano 10.5.2019

OTS 2019
SODOBNE INFORMACIJSKE TEHNOLOGIJE IN STORITVE
M. Heričko in K. Kous (ur.)

Univerzitetna založba
Univerze v Mariboru

VZPOSTAVITEV KONZORCIJSKEGA OMREŽJA ETHEREUM

BLAŽ PODGORELEC, PATRIK REK, MIHA STREHAR, MUHAMED TURKANOVIĆ

Povzetek: *Javna omrežja verig blokov prispevajo k robustnosti in varnosti, po drugi strani pa so povezana z upočasnjenim delovanjem in transakcijskimi stroški. Omenjeni slabosti sta tudi eni od razlogov, da se tehnologija veriženja blokov še ni uveljavila v večjem obsegu. Ena od možnih rešitev je vzpostavitev zasebnega oz. konzorcijskega omrežja verig blokov, ki se lahko poljubno konfigurira in se s tem doseže učinkovitejše okolje za poganjanje končnih decentraliziranih aplikacij. Med pomembnejšimi nastavitvami je vsekakor izbira porazdeljenega algoritma soglasja, ki ne zahteva procesa rudarjenja in omogoča izvedbo transakcij brez potrebe po plačevanju pristojbin. V prispevku se bomo osredotočili na tehnični del vzpostavitve konzorcijskega omrežja Ethereum. Predstavili bomo ključne korake za uspešno izvedbo tega in podali nekaj dobrih praks za takšno vzpostavitev. Predstavitev bo temeljila na uspešno vzpostavljenem mednarodnem konzorcijskem omrežju za platformo EduCTX, ki je operativna in združuje tako Ethereum kot IPFS del, katerega vzpostavitev bomo prav tako predstavili.*

Ključne besede: *veriga blokov, konzorcijsko omrežje, Ethereum, IPFS, Eductx*

NASLOVI AVTORJEV: Blaž Podgorelec, doktorski študent, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Blockchain Lab:UM, Maribor, Slovenija, e-pošta: blaz.podgorelec@um.si. Patrik Rek, podiplomski študent, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Blockchain Lab:UM, Maribor, Slovenija, e-pošta: patrik.rek@um.si. Miha Strehar, tehnični sodelavec, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Blockchain Lab:UM, Maribor, Slovenija, e-pošta: miha.strehar@um.si. Muhamed Turkanović, docent, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Blockchain Lab:UM, Maribor, Slovenija, e-pošta: muhamed.turkanovic@um.si.

https://doi.org/10.18690/978-961-286-282-4.8 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 UVOD

Javno omrežje verig blokov, temelječe na platformi Ethereum velja za robustno in varno, kar je posledično zagotovljeno s porazdeljenostjo, decentraliziranostjo in velikostjo javnega omrežja, ki je trenutno sestavljeno iz približno devet tisoč med seboj povezanih vozlišč [1].

Decentraliziranost omrežja je zagotovljena s tem, da temelji na algoritmu dokaza o delu (ang. proof-of-work) kot mehanizmu porazdeljenega soglasja, ki posledično zagotavlja enotno verigo blokov skozi celotno omrežje [2] [3]. Algoritem dokaz o delu zagotavlja, da so si vsa pridružena vozlišča popolnoma enakovredna, saj je lahko katerokoli od teh naslednje vozlišče, ki bo zbralo še nepotrjene transakcije v nov skupek transakcij imenovan blok, katerega bodo ostala vozlišča potrdila in replicirala. Brez mehanizma porazdeljenega soglasja, bi vozlišča v decentraliziranem in porazdeljenem okolju ločeno potrjevala in zbirala transakcije v bloke, pri čemer bi omrežje posledično imelo številne verige s podobnimi, vendar različnimi bloki, tj. vsebinami. Mehanizem porazdeljenega soglasja posledično zagotavlja porazdeljeno, vendar popolnoma enako verigo blokov, tj. vsebino.

Čeprav so si vozlišča enakovredna, saj veljajo za vse enaka pravila igre, je verjetnost, da bo vozlišče nov potrjevalec, odvisno predvsem od računske moči, ki jo vozlišča vložijo v proces zaključevanja novega bloka. Nov blok se lahko zaključi le s primerno izračunanim enkratnim kriptografskim številom (ang. nonce), ki bo rezultiral v vnaprej definirani strukturi zgostitve (ang. hash) vsebine bloka. Posledica takšnemu mehanizmu je neizmerno velika poraba energije na nivoju celotnega omrežja, ki je trenutno ekvivalentna porabi električne energije celotne Irske [4].

Razen tega, da platforme tehnologij veriženja blokov vključujejo številne kriptografske mehanizme, je varnost javnih omrežij z vsakim dodatnim vozliščem toliko večja. Razlog temu je dejstvo, da se veriga blokov replicira med vsa vozlišča, pri čemer so bloki med seboj povezani, saj se vsak nov blok sklicuje na prejšnji na način, da ima v svoji strukturi shranjeno zgostitev vsebine prejšnjega bloka. Posledično bi moral potencialni napadalec spremeniti verigo pri več kot 51 procentih vozlišč omrežja, pri čemer pa lahko verigo uspešno spremeni samo tako, da na novo pripravi vsebino blokov za vse povezane bloke, ki so nastali za blokom, ki ga želi napadalec spremeniti.

Zasnova omrežja in protokola kot je opisano zgoraj, zagotavlja torej robustnost in varnost, pri čemer pa hkrati tudi počasnejše delovanje oziroma procesiranje transakcij, saj trenutno omrežje Ethereum omogoča obdelavo zgolj med 7 in 15 transakcij na sekundo [5]. Hkrati povzroča javno omrežje Ethereum nestanovitne transakcijske stroške, plačilo katerih je za uspešno izvedbo transakcij znotraj javnega omrežja verig blokov nujno potrebno [5]. Slabost javnega omrežja verig blokov Ethereum se je najbolj odražala v času pojava spletne igre v obliki decentralizirane aplikacije imenovane Cryptokitties, ki je zaradi svoje začetne popularnosti ohromila celotno delovanje javnega omrežja, saj to ni bilo zmožno procesirati tako velike količine transakcij [6].

Potrebno je poudariti, da se razvijalci in raziskovalci omenjene platforme trenutno aktivno osredotočajo na razvoj novih mehanizmov[1], cilj katerih je zmožnost hitrejšega procesiranja transakcij ter posledično stabilizacija transakcijskih stroškov [7].

Omenjene negativne lastnosti javnih omrežij je tudi eden od poglavitnih razlogov, da se tehnologija veriženja blokov v gospodarstvu ne uporablja v večjem obsegu, kar tudi velja za platformo Ethereum. Zraven hitrejšega procesiranja transakcij in stabilnejših ali ničelnih stroškov pri izvedbi le teh so v gospodarstvu zahtevane tudi druge lastnosti, kot so zasebnost procesiranih podatkov, omejen dostop do podatkov, doseganje takojšnje končnosti transakcije po njeni vključitvi v blok, brez da bi pri tem obstajala možnost, da blok, ki tako transakcijo vključuje, ni del veljavne verige blokov, ipd.

Ena izmed možnih rešitev je vzpostavitev zasebnega oz. konzorcijskega omrežja verig blokov, zato bomo v tem prispevku le to predstavili, pri čemer bomo izpostavili nekatere razlike med različnimi tipi omrežij verig blokov, prednosti in slabosti le teh ter primere uporab različnih konzorcijskih omrežij v praksi.

[1] https://github.com/ethereum/eth2.0-specs

2 OMREŽJA VERIG BLOKOV

Poslovni procesi, ki jih lahko implementiramo znotraj omrežja verig blokov so zelo različni [8] [9]. Med njimi obstajajo tudi razlike glede na vrsto omrežja verig blokov v katero jih je primerno umestiti.

Glede na zasnovo lahko omrežja verig blokov razdelimo v dve kategoriji in sicer:

- **OMREŽJA BREZ DOVOLJENJA (ang. permissionless)**

  Najbolj razširjen in znan tip omrežja verig blokov, katerega poglavitna lastnost je, da lahko kdorkoli ter kadarkoli glede na svoje potrebe postane enakopraven del takšnega omrežja. Tako lahko kot enakopraven deležnik omrežja bodisi sodeluje pri potrjevanju transakcij ali le te zgolj pošilja v omrežje. Zaradi teh lastnosti, imajo takšna omrežja prednost v decentralizaciji ter posledično v teoretično večji odpornosti na morebitno cenzuro ali pristranskost [10] [8].

- **OMREŽJA Z DOVOLJENJEM (ang. permissioned)**

  V določenih poslovnih procesih je nesprejemljivo javno deljenje podatkov, ki se obdelujejo v sklopu transakcij javnih omrežij verig blokov. Primerno omejitvi je mogoče uporabiti t. i. omrežja verig blokov z dovoljenjem, med katere lahko uvrstimo popolnoma zasebna kot tudi konzorcijska omrežja verig blokov. Zasebna omrežja verig blokov lahko smatramo kot centralizirana omrežja, saj so omrežja pod nadzorom ene organizacije. Za razliko od zasebnih omrežij verig blokov, so konzorcijska omrežja delno decentralizirana, saj deležnike, ki so enakopravni del omrežja, predstavlja več različnih organizacij. V takšnih omrežjih so deležniki in njihove vloge vnaprej določene, tj. možnosti zapisovanja in prebiranja transakcij. Prav tako je v okvirju takšnih omrežij možna konfiguracija določenih nastavitev med katere sodijo pravila za doseganje porazdeljenega soglasja v omrežju, kot tudi velikost posameznega bloka. Takšne prilagoditve je možno izvesti glede na posamezen primer uporabe in s tem doseči učinkovitejše delovanje omrežja [10] [8].

Tehnologija veriženja blokov se je v svojih začetnih fazah preboja in množičnega sprejetja, večinoma uporabljala kot okolje, ki je nudilo podporo različnim javnim elektronskim plačilnim sistemom. Organizacije, ki primarno spadajo v t. i. sektor finančne tehnologije (ang. fintech), sicer še vedno prednjačijo pri raziskovanju in vključevanju tehnologije veriženja blokov v svoje poslovne procese, vendar se je glede na zadnje poročilo globalnega svetovalnega podjetja Deloitte trend prevesil tudi na druga področja, kot so mediji, telekomunikacije, vladne ustanove in zdravstvo [11].

Potrebno je izpostaviti dejstvo, da za določene poslovne procese zadostuje uporaba splošnih porazdeljenih podatkovnih baz. Kljub temu pa lahko pravilna vpeljava tehnologije veriženja blokov, predstavlja odločilno dodano vrednost pred konkurenco in sicer za organizacije, ki sodelujejo v večjih omrežjih, pri čemer opravljajo občutljive poslovne transakcije in si praviloma ne zaupajo stoodstotno. Z vpeljavo tehnologije veriženja blokov lahko dosežemo manjše stroške pri procesu vzpostavitve zaupanja ter usklajevanja pri sodelovanju znotraj poslovnih procesih. Med drugim odpira uporaba tehnologije veriženja blokov možnosti razvoja novih poslovnih procesov, ki temeljijo na digitalnih sredstvih prihodnosti. Zaradi svojih lastnosti tehnologija prav tako omogoča izboljšanje odgovornosti in poslovne učinkovitosti znotraj poslovnega omrežja [12].

Kot je bilo predhodno že omenjeno, uporaba javnih omrežij verig blokov v večini primerov, zaradi že opisane problematike upočasnjenega delovanja ter nestanovitnih transakcijskih stroškov za izvedbo transakcij kot tudi problematike omejevanja dostopa, zasebnosti podatkov ter končnosti opravljenih transakcij, organizacijam ne omogoča zadostne dodane vrednosti, pri vključitvi le te v svoje poslovne procese.

Iz prej omenjenih razlogov se bomo v nadaljevanju osredotočili na predstavitev gradnje konzorcijskega omrežja verig blokov, temelječega na platformi Ethereum. Z uporabo konzorcijskega omrežja Ethereum, je mogoče tudi organizacijam, ki jim uporaba javnega omrežja verig blokov ne zadostuje, ponuditi na tehnologiji veriženja blokov temelječe okolje, ki bo ob morebitni identifikaciji primernih poslovnih procesov, služi kot učinkovito podporo okolje [12].

3 KONZORCIJSKO OMREŽJE ETHEREUM

V nadaljevanju se bomo osredotočili na predstavitev ključnih gradnikov in funkcionalnosti konzorcijskega omrežja verig blokov, ki temelji na platformi Ethereum. Predstavili bomo tudi nekatere platforme s katerimi je možno vzpostaviti konzorcijsko omrežje Ethereum na način tehnologija veriženja blokov kot storitev (ang. Blockchain as a Service).

3.1 Odjemalci

Protokol Ethereum je implementiran s pomočjo različnih odjemalcev, ki se med seboj razlikujejo, glede na programske jezike s katerimi so implementirani. Na voljo je več kot deset različic implementacij[2], med katerimi sta trenutno najbolj uporabljena odjemalca Geth[3], ki je implementiran s programskim jezikom Go ter Parity[4], implementacija katerega temelji na programskem jeziku Rust. Izbira odjemalca, ki ob zagonu predstavlja tudi vozlišče omrežja verige blokov Ethereum, je odvisna predvsem od različnih potreb okolja, v katerem bodo le ti uporabljeni. Pri izbira le tega, ima najpomembnejšo vlogo algoritem porazdeljenega soglasja, saj različni odjemalci podpirajo različne algoritme porazdeljenega soglasja za namene konzorcijske uporabe. Velika večina odjemalcev implementira in s tem omogoča uporabo algoritma »dokaz o delu« (ang. Proof-of-Work - PoW), ki pa ni primeren za uporabo v konzorcijskem omrežju. V omrežju, ki ga ponavadi sestavljajo vnaprej znani deležniki in so njihove vloge vnaprej določene, je tako bolj smiselna izbira algoritma »dokaz o avtoriteti« (ang. Proof-of-Authority - PoA).

V naslednjih poglavjih bomo opisali vlogo odjemalcev konzorcijskega omrežja kot tudi posamezne mehanizme oz. gradnike, katere le ti izvajajo.

3.2 Doseganje učinkovitosti delovanja

Z namenom ustvarjanja konzorcijskega okolja, ki temelji na tehnologiji veriženja blokov, je potrebna pravilna konfiguracija posameznih odjemalcev, ki kot celota tvorijo konzorcijsko omrežje verige blokov. V tem poglavju bomo poudarili tri gradnike oz. mehanizme, katerih delovanje je mogoče za ta namen tudi prilagoditi.

Algoritmi za porazdeljeno doseganje soglasja

Ker se v prispevku osredotočamo na gradnjo konzorcijskega in ne javnega omrežja verig blokov, je namesto algoritma PoW, ki se za doseganje porazdeljenega soglasja uporablja v javnem omrežju Ethereum, bolj primerno uporabiti algoritem PoA. Strnjena primerjava posameznih algoritmov je prikazana v Tabela 1.

Algoritem, ki temelji na dokazovanju avtoritete, od vozlišč ne zahteva reševanja t. i. matematične uganke, temveč izrecno definira vozlišča, ki posedujejo dovoljenje, s katerim jim je omogočeno, da znotraj omrežja ustvarjajo, kot tudi potrjujejo veljavnost novo ustvarjenih blokov [13] [14].

Tabela 1: Primerjava lastnosti algoritmov doseganja porazdeljenega soglasja [15].

	Dokaz o delu	**Dokaz o avtoriteti**
Hitrost delovanja	Počasnejše Delno deterministično	Hitrejše Deterministično - nastavljivo
Energetski vpliv	Potratno	Učinkovito
Okolje izvajanje konsenza	Strojna oprema	Programska oprema
Končnost transakcij	Verjetnostna	Absolutna
Stroški vzdrževanja	Visoki	Brez stroškov

Izbira dokaza o avtoriteti kot algoritma za doseganje porazdeljenega soglasja, v primerjavi z algoritmom dokaza o delu, v primeru konzorcijskega omrežja velja za varnejšo, saj so povezave v omrežju, vzpostavljene

[2] https://github.com/ConsenSys/ethereum-developer-tools-list#ethereum-clients
[3] https://github.com/ethereum/go-ethereum
[4] https://github.com/paritytech/parity-ethereum

z zaupanja vrednimi deležniki, kar pomeni, da potencialnim napadalcem ne omogoča prerazporeditve transakcij v omrežju, že samo s samo vzpostavitvijo povezave z omrežjem [13].

Velikost bloka

Število transakcij, ki so lahko izvedene in potrjene znotraj enega bloka (tj. skupek transakcij), se v primeru protokola Ethereum razlikuje od bloka do bloka, pri čemer je znotraj protokola definirana zgornja meja. Število transakcij v bloku je namreč odvisno od skupne količine enot t. i. goriva, ki ga te v okviru transakcije skupaj porabijo za izvedbo določenih sprememb določenega stanja v omrežju [3]. Zgornja meja enot goriva na blok oz. najvišja količina enot goriva je določena znotraj nastavitev vsakega posameznega odjemalca in se lahko po potrebi tudi prilagaja. S povečevanjem zgornje meje enot goriva, ki jih lahko posamezna avtoriteta potrdi znotraj bloka, posledično določa število transakcij, ki jih lahko omrežje v določenem času izvede [13]. Na ta način se lahko konzorcijsko omrežje verig blokov dodatno pohitri.

Hitrost potrjevanja blokov

Ker je v konzorcijskem omrežju verig blokov, ki uporablja algoritem porazdeljenega soglasja PoA, generiranje bloka transakcij vnaprej definirano, je pri tem mogoče definirati tudi čas po katerem se v omrežju generira nov blok. Takšne nastavitve v primeru uporabe algoritma PoW niso možne, saj je potrjevanje novega bloka odvisno od vozlišč in njihovega reševanja t. i. matematične uganke. Kljub temu je v takšnem primeru čas generiranja novega bloka mogoče napovedati, saj se zahtevnost matematične uganke dinamično prilagaja, z namenom zagotavljanja dovolj hitrega oz. počasnega generiranja novega bloka. V primeru konzorcijskega omrežja ob uporabi algoritma PoA, definiranje časa generiranja novega bloka, vpliva na hitrost kot tudi potrjevanje sprememb, izvedenih s transakcijami, vsebovanimi v vsakem posameznem bloku.

Plačevanje pristojbin

V okvirju platforme Ethereum je za izvajanje operacij znotraj omrežja, potrebno plačilo določene vsote pristojbin v kriptovaluti Ether. Skupna vsota pristojbin zbranimi z izvedenimi operacijami, ki so v obliki transakcij, zbrane v bloku, se kot nagrada za opravljeno delo prenese na naslov odjemalca, ki je ta blok uspešno generiral. Namen obračunavanja pristojbin je preprečevanje nepotrebnega obremenjevanja omrežja verig blokov. Obračunavanje pristojbin za izvedbo operacij je smiselno v javnem omrežju verig blokov, kjer se lahko vsaka entiteta pridruži omrežju in na njem izvaja transakcije. V primeru vzpostavitve konzorcijskega omrežja, kjer so vnaprej znani deležniki, ki bodo omrežje sestavljali, je takšno obračunavanje manj smiselno. Zato lahko ob konfiguraciji vozlišč, ki bodo v omrežju nastopala v vlogi podpisovalca (opisano v poglavju 3.3), določimo spodnjo mejo pristojbin, ki jo bo vozlišče ob generiranju novega bloka tudi sprejelo. Vrednost spodnje meje je zato smiselno nastaviti na nič, saj v tem primeru deležnikom, ki bodo v omrežju izvajali transakcije, ni treba posredovati kriptovalute. V nasprotnem primeru je ob konfiguraciji konzorcijskega omrežja, treba definirati naslove odjemalcev in količino kriptovalute, ki ga le ti prejmejo na svoj račun. V tem primeru je odgovornost odjemalcev, da z ustrezno distribucijo kriptovalute, omogočijo ostalim deležnikom izvajanje transakcij znotraj omrežja.

3.3 Arhitektura omrežja

V omrežju verig blokov platforme Ethereum, ki temelji na algoritmu PoA, obstajajo različni tipi vozlišč, pri čemer je v konzorcijskem omrežju smiselno imeti ločene vsaj naslednje tri tipe:

- podpisovalci (ang. signer node) – predstavljajo vozlišča, katerih naloga je generiranje novih blokov v omrežje verige blokov. V primeru, ko vozlišče ni tisto, ki je generiralo nov blok, je njihova naloga preverjanje pravilnosti izvedenih transakcije znotraj predlaganega bloka. Vozlišče v okvirju konzorcijskega omrežja lahko postane podpisovalec, če ga z glasovanjem za to nalogo potrdi 51 % vseh vozlišč v omrežju, ki so v tistem trenutku v vlogi podpisovalca transakcij. Prav tako je mogoče, vozlišča, ki bodo v omrežju verige blokov, delovala v vlogi podpisovalca, definirati znotraj prvega bloka verige (ang. genesis block). Omenjeno vozlišče, definirano znotraj prvega bloka verige poimenujemo pečatno vozlišče (ang. sealer) in ima v omrežju popolnoma enako nalogo kot vozlišča poimenovana podpisovalci.
- arhivska vozlišča (ang. archive nodes) – naloga takšnih vozlišč je zgolj in samo hranjenje celotne zgodovine podatkov o verigi blokov, kar zajema vse zapise o spremembah stanja znotraj omrežja verige blokov, kot tudi podatke shranjene znotraj le tega; in

- vozlišča z izpostavljenim aplikacijskim programskim vmesnikom – namenjena so povezovanju in interakciji končnih uporabnikov z omrežjem verige blokov. Z namenom uporabniku prijazne izkušnje, takšna interakcija običajno poteka s pomočjo namenskih decentraliziranih aplikacij. Aplikacijski programski vmesnik lahko izpostavljajo tudi vozlišča, ki imajo v omrežju verige blokov nalogo podpisovalca, vendar je zaradi zagotavljanje večje varnosti samega omrežja, bolj smiselno ločevanje le teh dveh tipov vozlišč. Vozlišča, ki izpostavljajo aplikacijski programski vmesnik namreč predstavljajo neposredni stik omrežja verige blokov z zunanjimi entitetami, kar posledično pomeni, večjo možnost za kakršnekoli varnostne incidente. V primeru, da pride do takšnega varnostnega incidenta, je najslabši možni scenarij izpad vozlišča iz omrežja, kar prav zaradi striktnega ločevanja vozlišč, na delovanje omrežja verige blokov, nima nobenega vpliva.

Primer takšne arhitekture je prikazan na sliki 1, ki ponazarja arhitekturo omrežja EduCTX, kjer vsaka univerza vključuje specifične tipe vozlišč.

3.4 Shramba podatkov

Ne gleda na rabo javnega ali konzorcijskega omrežja verig blokov, je uporaba le tega za shranjevanje večje količina podatkov ali datotek neprimerna. V ta namen je ob načrtovanju poslovnih procesov, ki temeljijo na tehnologiji veriženja blokov, vedno več v uporabi hibridni sistem, ki vključuje porazdeljene datotečne sisteme, kot sta IPFS ali Swarm. Ta se uporabljata za shranjevanje večjih količin podatkov, medtem ko se omrežja verig blokov uporabljajo za potrjevanje celovitosti in sledljivosti shranjenih datotek v omenjene porazdeljene datotečne sisteme.

Omenjeni hibridni način ni edini, saj se lahko namesto porazdeljenega datotečnega sistema uporabi preprost centraliziran datotečni sistem ali javna oblačna storitev, pri čemer se v takšnem primeru izgubi koncept porazdeljenosti in decentraliziranosti, ki je tudi eden od poglavitnih razlogov uporabe tehnologije veriženja blokov.

V konzorcijskih omrežij verig blokov je tako smiselna hkratna uporaba vozlišč, ki poganjajo odjemalce za protokol in omrežje Ethereum in hkrati odjemalce za porazdeljen datotečni sistem IPFS ali Swarm. Le ti so lahko nameščeni na istih vozliščih ali na ločenih vozliščih istega skrbnika. S tem konzorcijsko omrežje omogoča porazdeljeno shranjevanje večje količine podatkov, pri čemer zagotovi lastna pravila upravlja le teh in s tem posledično dogovorjeno stopnjo decentraliziranosti med vpletenimi deležniki in njihovimi vozlišči. Poudariti je potrebno tudi, da so vključena omrežja porazdeljenih datotečnih sistemov lahko, enako kot omrežja verig blokov, javna ali zasebna oz. konzorcijska, pri čemer so prednosti in slabosti le teh podobne tistim, ki veljajo za takšne tipe omrežij verig blokov.

IPFS je porazdeljen datotečni sistem, zasnovan na povezovanju računalnikov z enakim sistemom datotek. Nadzira lokacije datotek na omrežju in skrbi za premike datotek. Namesto naslavljanja strežnikov imamo naslavljanje datotek po njihovi zgostitvi. Tako iz IPFS omrežja pridobimo natančno lokacijo datoteke in nato prenesemo datoteko na svoje vozlišče. Zaradi tega imajo datoteke, ki so bolj povpraševane, več kopij in so bolj porazdeljene. Tako je manjša verjetnost po izgubi datotek. Na omrežje IPFS se povezujemo s pomočjo klicev HTTP na vozlišče ali s pomočjo JavaScript knjižnice js-ipfs [16].

Swarm je tako kot IPFS porazdeljen datotečni sistem, razvit in vzdrževan s strani fundacije Ethereum. Zaradi tega je neposredno povezan z verigo blokov Ethereum. Je omrežje, ki deluje po principu »vsak z vsakim«. Tako kot IPFS, je sistem, kjer se ne naslavlja strežnikov, ampak zgostitev datoteke. Za razliko od IPFS, je Swarm manj razvit, saj je še v začetnih fazah razvoja. Razlika je tudi v tem, da na Swarm lahko datoteke naložimo, medtem ko na IPFS zgolj objavimo pot do datotek, ki se nahajajo na našem disku. [17].

Porazdeljene datotečne sisteme lahko uporabimo na način, da nanje naložimo večje datoteke, medtem ko na verigo blokov zapišemo zgolj zgostitve. S tem zmanjšamo velikost transakcij v verigah blokov.

3.5 Blockchain as a service

Veriga blokov kot storitev (ang. Blockchain-as-a-Service) je storitev, ki razvijalcem omogoča učinkovitejši razvoj aplikacij, ki temeljijo na tehnologiji veriženja blokov. Omogoča razvoj, gostovanje in uporabo decentraliziranih aplikacij, pametnih pogodb in njihovo nalaganje na verigo blokov, medtem ko ponudnik storitve upravlja vsa potrebna opravila in aktivnosti za vzdrževanje agilnosti in operativnosti infrastrukture

[15] [16]. Vodilna ponudnika verig blokov kot storitve sta Microsoft s svojim Azure Blockchain in ConsenSys s Kaleido-om.

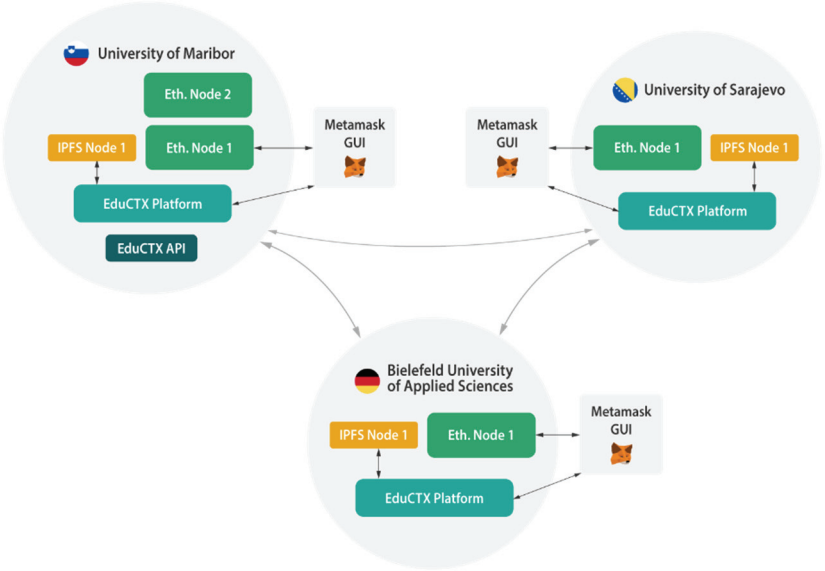
Kaleido je celovita platforma za poenostavitev namestitve in upravljanja konzorcijskih verig blokov. Omogoča gradnjo konzorcijskega omrežja verige blokov, ki temelji na protokolu Ethereum. Uporabniki lahko do omrežja dostopajo s pomočjo aplikacijskega programskega vmesnika, kar omogoča uporabo v decentralizirani aplikaciji. Ponuja tudi preprosto nastavljanje pravic dostopa partnerskim institucijam. S pomočjo namenske tržnice vtičnikov je mogoča tudi preprosta uporaba različnih naprednih orodij, kar močno zmanjša količino potrebnega programiranja. Omogočajo nenehno podporo in s tem posledično varno omrežje [20].

Tudi **Azure Blockchain** je celovita platforma za upravljanje konzorcijskih verig blokov. Omogoča preprosto poganjanje upravljanih omrežij verig blokov, možnost razširljivosti in ponuja upravljanje omrežja brez naporov, kar omogoča razvijalcem osredotočanje na logiko aplikacij. Celotno upravljanje poteka preko nadzorne plošče, kjer se nastavljajo tudi pravice in pravila za uporabnike in člane konzorcija. Azure prinaša platformo, ki je osredotočena na varnost in preprosto uporabo [21].

4 PRIMER VZPOSTAVITVE KONZORCIJSKEGA OMREŽJA

Kot primer vzpostavitve konzorcijskega omrežja Ethereum bomo predstavili omrežje EduCTX. EduCTX je decentralizirana digitalna platforma, temelječa na tehnologiji veriženja blokov, pametnih pogodbah protokola Ethereum, kriptografiji in decentraliziranih aplikacijah. Namenjena je dodeljevanju, shranjevanju, upravljanju, deljenju, prikazovanju in preverjanju digitalnih mikro-certifikatov (diplome, pohvale, certifikati itd.). Certifikati služijo kot dokaz o pridobljenem znanju in spretnostih, kjer so izdajalci certifikatov tudi vozlišča omrežja. V primeru platforme EduCTX so deležniki omrežja in s tem vozlišča le tega sodelujoče izobraževalne ustanove. Platforma predstavlja učinkovito, preprosto in varno rešitev za decentralizirano in porazdeljeno upravljanje in preverjanje mikro-certifikatov.

Trenutno zraven Univerze v Mariboru, ki predstavlja začetno organizacijo, v projektu EduCTX sodelujeta še Univerza v Sarajevu in Univerza uporabnih znanosti Bielefeld, kot je prikazano na sliki 1. V Mariboru se nahajata dve Ethereum vozlišči, ki delujeta kot podpisovalca, ob tem pa ima eno tudi izpostavljene aplikacijske programske vmesnike za neposredno uporabo s strani decentralizirane aplikacije. Ob tem se v Mariboru nahaja tudi glavno vozlišče IPFS vozlišče, ki deluje kot povezovalno vozlišče. Preostali vključeni organizaciji imata svoji vozlišči Ethereum in IPFS, pri čemer vozlišče Ethereum opravlja obe prej omenjeni vlogi. S tem dosežemo popolno decentraliziranost in porazdeljenost, saj se uporabniki drugih institucij povezujejo na svoja vozlišča in imajo celotno verigo blokov tudi v primeru izpada katerega od vozlišč. Mariborsko vozlišče, ki nima izpostavljenih aplikacijskih programskih vmesnikov, ima tako tudi funkcijo arhivskega vozlišča, saj je najbolj stabilno, ker je zaščiteno pred neposrednimi napadi.



Slika 1: Trenutna arhitektura omrežja EduCTX.

4.1 Odjemalci omrežja

Konzorcijsko omrežje EduCTX, temelji na platformi Ethereum in se sestoji iz odjemalcev implementacije imenovane Parity, katera je zraven implementacije imenovane Geth največkrat uporabljena implementacija odjemalca znotraj omrežij verig blokov Ethereum [1]. V začetnem bloku verige (ang. genesis block) je mogoče definirati naslove odjemalcev, ki bodo v omrežju opravljali vlogo potrjevalcev. Na primeru začetnega bloka omrežja EduCTX zapisanega v izseku kode 1, lahko vidimo, da je uporabljena posebna vrsta definiranja odjemalcev z vlogo potrjevalca. Račune odjemalcev, ki bodo v omrežju opravljali vlogo potrjevalca, smo namreč definirali s pomočjo za to namenske pametne pogodbe nameščene v omrežju verige blokov. Znotraj takšne pametne pogodbe so tako definirani naslovi odjemalcev z vlogo potrjevalca kot tudi funkcija, ki kot argument sprejeme naslov novega odjemalca, ki bo v omrežje dodan kot potrjevalec in je lahko prožena s strani tistega, ki je pametno pogodbo namestil v omrežje verige blokov. Posebnost takšnega načina definiranja potrjevalcev v omrežju je namreč, da za dodajanje novega odjemalca z vlogo potrjevalca ni potrebe po glasovanju s strani ostalih odjemalcev v omrežju, niti pri tem ni potrebe po spreminjanju prvotnega bloka verige, ki lahko privede do razvejitve verige blokov (ang. fork). Kot je še vidno iz prvega bloka je kot potrjevalec do bloka 650 definiran določen odjemalec[5], kar se s 650. generiranim blokom spremeni, saj od tega trenutka prične veljati seznam potrjevalcev definiran znotraj namenske pametne pogodbe[6].

Znotraj začetnega bloka je z namenom konfiguracije konzorcijskega omrežja prav tako smiselno definirati čas generiranja novih blokov ter zgornjo mejo goriva, ki je lahko porabljena v posameznem, novo ustvarjenem bloku, saj lahko z ustrezno konfiguracijo teh parametrov dosežem učinkovito delovanje konzorcijskega omrežja verige blokov, kot je opisano v poglavju 3.2.

Za uspešno vzpostavitev konzorcijskega omrežja Ethereum, ki temelji na implementaciji odjemalca Parity, je potrebna tudi nastavitvena datoteka, ki je prikazana v izseku kode 2. V njej so najprej zapisane nastavitve odjemalca Parity, kot je imenska pot do datoteke z začetnim blokom ter imenska pot do lokacije shrambe celotne verige blokov. Prav tako je potrebna nastavitev omrežnih vrat na katerih deluje odjemalec in datoteke, kjer so zapisani naslovi enode vseh vozlišč, na katera se vozlišče povezuje.

V primeru, da se odločimo odjemalca konfigurirati tako, da bo le ta omogočal povezovanje končnih uporabnikov do omrežja verig blokov je smiselno znotraj razdelka [rpc] definirati dostop do aplikacijskih vmesnikov, ki komunicirajo s pomočjo protokola JSON-RPC.

Ob prvem zagonu odjemalca Parity je potrebno ustvariti tudi račun, čigar geslo zapišemo v datoteko, imensko pot katere definiramo v razdelku [account]. V razdelek [mining] se zapiše podpisovalca, kar je naslov računa, ki je ustvarjen ob zagonu odjemalca. Zadnji razdelek služi definiranju imenske poti lokacije datoteke za shranjevanje dnevniških zapisov.

Ob pridružitvi novega odjemalca v omrežje, mora skrbnik le tega po definiranju vseh nastavitev posredovati glavnemu vozlišču svoj naslov enode, katerega mora upravitelj začetnega vozlišča dodati na seznam vozlišč omrežja, definiran v razdelku [network]. Zaradi večje zmogljivosti celotnega omrežja je omenjen naslov prav tako smiselno posredovati vsem ostalim vozliščem, ki se bodo tako povezovala na vsa ostala vozlišča. V primeru konzorcijskega omrežja EduCTX, omrežje trenutno sestavljajo štiri vozlišča, kjer ima vsako vozlišče popolnoma replicirano stanje verigo blokov.

4.2 IPFS

Kot v poglavju 3.4 predstavljeno, je v konzorcijskih omrežjih, ki temeljijo na tehnologiji veriženja blokov smiselna hibridna arhitektura, ki ob omrežju verig blokov hkrati vpeljuje porazdeljene datotečne shrambe. Tako je zasnovana tudi arhitektura platforme EduCTX, ki uporablja konzorcijsko omrežje porazdeljene datotečne shrambe IPFS za shranjevanje šifriranih datotek mikro-certifikatov.

Nastavitev zasebnega oz. konzorcijskega omrežja IPFS se začne z začetno namestitvijo in konfiguracijo običajnega odjemalca IPFS. Za tem je potrebno generiranje ključa ipfs swarm, s pomočjo orodja ipfs-swarm-key-gen[7]. Omenjeni ključ služi kot ključ omrežja in ga potrebujejo vsi deležniki omrežja, zaradi česa se shrani

[5] Naslov: 0x1234123412341234123412341234123412341234
[6] Naslov: 0x4321432143214321432143214321432143214321
[7] https://github.com/Kubuxu/go-ipfs-swarm-key-gen

v datoteko, ki se mora nahajati v enaki mapi kot vse datoteke odjemalca IPFS. Ko je generiran in shranjen ključ swarm, se lahko požene ukaz za zagon vozlišča[8]. ID vozlišča dobimo ob inicializaciji odjemalca IPFS, IP naslov pa je naslov glavnega vozlišča. Če imamo več glavnih vozlišč, lahko drug ukaz ponovimo za vsako vozlišče. Pred zagonom lahko v konifguracijski datoteki, ki se nahaja v mapi IPFS spremenimo še nastavitve glede omrežnih vrat z namenom, da se ne prekrivajo z drugimi storitvami na strežniku. Postopek je univerzalen za vsa vozlišča, z izjemo, da je swarm ključ potrebno generirati le enkrat.

Izsek kode 1: Začetni blok EduCTX omrežja

```
{
    "name": "EduCTX",
    "engine": {
        "authorityRound": {
            "params": {
                "stepDuration": "5",
                "validators" : {"multi": {
                      "0": {"list":["0x1234123412341234123412341234123412341234"]},
                      "650": {"contract":"0x4321432143214321432143214321432143214321"}
                      }}}},
    "params": {
        "gasLimitBoundDivisor": "0x400",
        "maximumExtraDataSize": "0x20",
        "minGasLimit": "0x7A1200",
        "networkID" : "0x1234",
        "eip155Transition": 0,
        "validateChainIdTransition": 0,
        "eip140Transition": 0,
        "eip211Transition": 0,
        "eip214Transition": 0,
        "eip658Transition": 0,
              "eip150Transition": "0x0",
              "eip160Transition": "0x0",
              "eip161abcTransition": "0x0",
              "eip161dTransition": "0x0",
              "eip98Transition": "0x7fffffffffffff",
              "maxCodeSize": 24576,
              "maxCodeSizeTransition": "0x0"
    },
    "genesis": {
        "seal": {
            "authorityRound": {
                "step": "0x0",
                "signature":
"0x0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000"}
        },
        "difficulty": "0x20000",
        "gasLimit": "0x7A1200"
```

[8] ipfs bootstrap rm --all in ipfs bootstrap add /ip4/[IP naslov]/tcp/4001/ipfs/[ID vozlišča]

Izsek kode 2: Nastavitvena datoteka Parity.

```
[parity]
chain = "EduCTX.json"
base_path = "C:/EduCTX/Data/"

[network]
port = 30300
#reserved_peers = "C:/EduCTX/EduCTX_Network.txt"

[rpc]
disable = true
port = 8080
interface = "all"
hosts = ["all"]
cors = ["all"]
apis = ["web3", "eth", "net"]

[websockets]
disable = true
port = 8081
interface = "all"
hosts = ["all"]

[account]
#password = ["EduCTX.pwds"]

[mining]
#engine_signer = ""
reseal_on_txs = "none"
force_sealing = true

[misc]
log_file = "C:/EduCTX/Data/EduCTX.log"
```

5 ZAKLJUČEK

Uspešna vzpostavitev učinkovitega konzorcijskega omrežja verig blokov Ethereum zahteva upoštevanje številnih faktorjev. Pomembnejši med slednjimi so ustrezna konfiguracija in določitev algoritma za porazdeljeno doseganje soglasja, zgornje meje velikosti posameznega bloka ter intervala hitrosti generiranja novih blokov. Ob tem je za doseganje ne samo učinkovitega, temveč tudi varnega konzorcijskega omrežja, treba upoštevati priporočila, pri snovanju arhitekture omrežja, ki vključuje ustrezno določanje različnih tipov vozlišč znotraj tega. Vzpostavitev učinkovitega in varnega konzorcijskega omrežja verig blokov Ethereum smo v prispevku prikazali na primeru uspešno vzpostavljenega mednarodnega omrežja platforme EduCTX. S pravilno izvedbo konfiguracije je tako mogoče doseči hitrejše procesiranje transakcij s stabilnejšimi ali celo ničelnimi stroški izvedbe transakcij, doseganje takojšnje končnosti le teh, kot tudi zasebnost procesiranih podatkov, kar lahko pripomore k ustvarjanju primernega okolja, ki je lahko uporabljeno v gospodarstvu in temelji na tehnologiji veriženja blokov.

6 LITERATURA

[1] „Ethereum nodes“. [Na spletu]. Dostopno: https://www.ethernodes.org/network/1. [Dostopano: 17-maj-2019].

[2] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, in S. Capkun, „On the Security and Performance of Proof of Work Blockchains“, *Proc. 2016 ACM SIGSAC Conf. Comput. Commun. Secur. - CCS'16*, str. 3–16, 2016.

[3] G. Wood, „Ethereum: A secure decentralised generalised transaction ledger“, *Ethereum Proj. yellow Pap.*, let. 151, str. 1–32, 2014.

[4] K. J. O'Dwyer in D. Malone, „Bitcoin mining and its energy footprint“, 2014.

[5] F. M. Benčić in I. P. Žarko, „Distributed ledger technology: blockchain compared to directed acyclic graph“, v *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018, str. 1569–1570.

[6] O. Kharif, „Cryptokitties mania overwhelms ethereum network's processing“. [Na spletu]. Dostopno: https://www.bloomberg.com/news/articles/2017-12-04/cryptokitties-quickly-becomes-most-widely-used-ethereum-app. [Dostopano: 10-maj-2019].

[7] M. Bez, G. Fornari, in T. Vardanega, „The Scalability Challenge of Ethereum: An Initial Quantitative Analysis“, v *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2019, str. 167–16709.

[8] K. Wüst in A. Gervais, „Do you need a Blockchain?“, v *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, 2018, str. 45–54.

[9] M. Crosby, P. Pattanayak, S. Verma, in V. Kalyanaraman, „Blockchain technology: Beyond bitcoin“, *Appl. Innov.*, let. 2, št. 6–10, str. 71, 2016.

[10] Z. Zheng, S. Xie, H. Dai, X. Chen, in H. Wang, „An overview of blockchain technology: Architecture, consensus, and future trends“, v *2017 IEEE International Congress on Big Data (BigData Congress)*, 2017, str. 557–564.

[11] Deloitte, „Deloitte's 2019 Global Blockchain Survey“. [Na spletu]. Dostopno: https://www2.deloitte.com/content/dam/insights/us/articles/2019-global-blockchain-survey/DI_2019-global-blockchain-survey.pdf. [Dostopano: 10-maj-2019].

[12] Consensys, „11 Ways Ethereum Can Benefit Enterprise“. [Na spletu]. Dostopno: https://media.consensys.net/11-ways-ethereum-can-benefit-enterprise-aac6d798a9fb. [Dostopano: 05-maj-2019].

[13] P. T. Documentation, „Proof-of-Authority Chains - Wiki“. [Na spletu]. Dostopno: https://wiki.parity.io/Proof-of-Authority-Chains. [Dostopano: 26-apr-2019].

[14] S. De Angelis, L. Aniello, R. Baldoni, F. Lombardi, A. Margheri, in V. Sassone, „Pbft vs proof-of-authority: applying the cap theorem to permissioned blockchain“, 2018.

[15] G. Wood, „Blockchains: What and Why“. [Na spletu]. Dostopno: https://www.slideshare.net/gavofyork/blockchain-what-and-why. [Dostopano: 03-maj-2019].

[16] Protocol Labs, „IPFS Documentation“. [Na spletu]. Dostopno: https://docs.ipfs.io/. [Dostopano: 05-maj-2019].

[17] „Swarm documentation“. [Na spletu]. Dostopno: https://swarm-guide.readthedocs.io/en/latest/. [Dostopano: 05-maj-2019].

[18] Q. Lu, X. Xu, Y. Liu, in W. Zhang, „Design Pattern as a Service for Blockchain Applications“, v *2018 IEEE International Conference on Data Mining Workshops (ICDMW)*, 2018, str. 128–135.

[19] J. Frankenfield, „Blockchain-as-a-Service (BaaS)“. [Na spletu]. Dostopno: https://www.investopedia.com/terms/b/blockchainasaservice-baas.asp. [Dostopano: 14-maj-2019].

[20] „Kaleido: Enterprise Blockchain as a Service & Blockchain Cloud“. [Na spletu]. Dostopno: https://kaleido.io/. [Dostopano: 12-maj-2019].

[21] Micosoft, „Ethereum on Azure“, 2019. [Na spletu]. Dostopno: https://azuremarketplace.microsoft.com/sl-si/marketplace/apps/microsoft-azure-blockchain.azure-blockchain-ethereum. [Dostopano: 12-maj-2019].

OTS 2019
SODOBNE INFORMACIJSKE TEHNOLOGIJE IN STORITVE
M. Heričko in K. Kous (ur.)

Univerzitetna založba
Univerze v Mariboru

MOŽNOSTI IN IZZIVI UPORABE PROTOKOLA IPFS

AIDA KAMIŠALIĆ, MUHAMED TURKANOVIĆ

Povzetek: *Sistem za porazdeljeno in decentralizirano shranjevanje datotek IPFS (ang. Interplanetary File System) temelji na protokolu in omrežju vrstnikov (P2P) ter služi varnemu shranjevanju in deljenju vsebin. Vsako vozlišče v takšnem omrežju lokalno hrani zbirko datotek, za povezovanje in pridobitev datotek iz omrežja pa uporablja zgostitev (ang. hash) vsebine. Čeprav ni prvotno temu namenjen, je protokol IPFS pridobil na priljubljenosti, predvsem zaradi uporabe v sklopu decentraliziranih aplikacij (dApp), ki temeljijo na tehnologiji veriženja blokov (ang. blockchain). V prispevku PRIMERJAMO IPFS s konkurenčnimi sistemi kot so Swarm, Storj in Sia. Uporaba sistema IPFS se kaže kot primeren pristop za naslavljanje izziva omejene velikosti podatkov, ki jih lahko pri izvedbi transakcije shranimo v verigo blokov.*

Ključne besede: *protokol IPFS, porazdeljena shramba, decentralizirana shramba, tehnologija veriženja blokov, omrežje vrstnikov (P2P)*

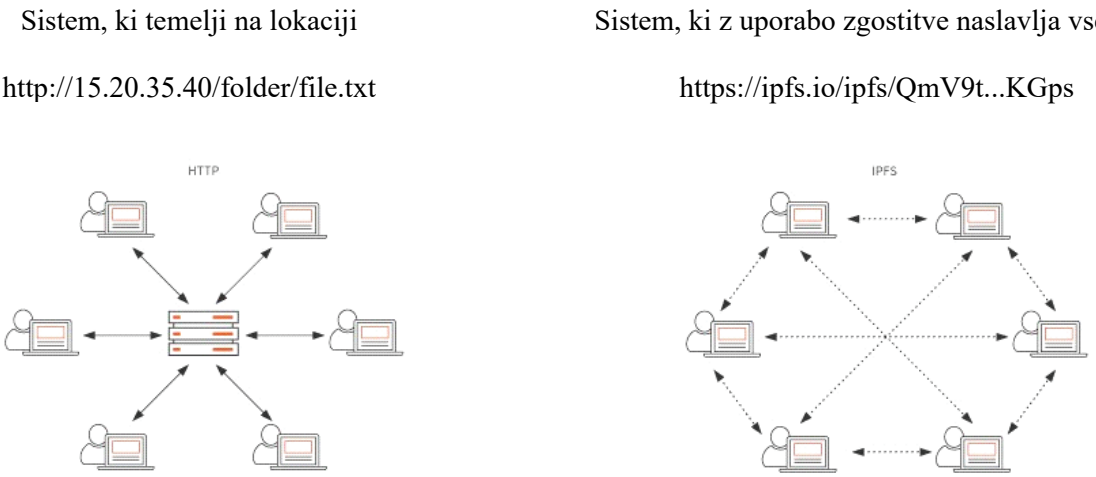
NASLOVA AVTORJEV: dr. Aida Kamišalić, docentka, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Blockchain Lab:UM, Maribor, Slovenija, e-pošta: aida.kamisalic@um.si. dr. Muhamed Turkanović, docent, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Blockchain Lab:UM, Maribor, Slovenija, e-pošta: muhamed.turkanovic@um.si.

https://doi.org/10.18690/978-961-286-282-4.9 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 UVOD

Ob razvoju tehnologije veriženja blokov in večanju popularnosti le te je moč opaziti povečano zanimanje za koncept decentralizacije. Obstoječe paradigme, temelječe na principu strežnik – odjemalec, so se med drugim pokazale dovzetne za cenzuriranje in monopol ponudnikov storitev. Izvirna ideja interneta je bila postavitev skupnega nevtralnega globalnega omrežja, kjer vsi deležniki enakovredno sodelujejo v dobro družbe. Splet 3.0 ponovno oživlja to idejo – decentraliziran splet odporen na cenzure, brez kritične točke odpovedi in možnost sodelovanja med akterji, ki si privzeto ne zaupajo [1]. Med prvimi uspešnimi sistemi, ki so temeljili na principu porazdeljenih aplikacij, so bila (1) omrežja vrstnikov (ang. peer-to-peer, P2P) Gnutella, namenjena iskanju in deljenju datotek [2], (2) BitTorrent – sistem P2P za deljenje vsebin [3], (3) Tribler – razširitev protokola BitTorrent z lastnostmi kot sta integriran iskalnik in video na zahtevo [4] in (4) Freenet – omrežje vrstnikov s porazdeljeno podatkovno shrambo, odporno na cenzuriranje [5].

Uspešne ideje obstoječih sistemov P2P, kot so porazdeljena zgoščevalna tabela (ang. distributed hash tables), izmenjava blokov (ang. block exchanges), sistem za nadzor različic (ang. version control systems) za sledenje vsebini in lastno potrjevanje imenskega prostora (ang. self-certified namespace), so združene v sistemu za porazdeljeno in decentralizirano shranjevanje datotek IPFS (ang. Interplanetary File System). Temelji na protokolu in omrežju P2P ter služi varnemu shranjevanju in deljenju vsebin brez omejevanja velikosti shranjenih podatkov. Vsako vozlišče v takšnem omrežju lokalno hrani zbirko datotek. Večje datoteke so razdrobljene na manjše dele in shranjene na različnih vozliščih. Sistem IPFS za povezovanje in pridobitev datoteke iz omrežja ne uporablja naslova strežnika (vozlišča omrežja), temveč zgostitev vsebine (ang. hash) datoteke, kot je razvidno iz Slike 1. Posledično se ob zahtevi po datoteki le ta prenese iz najbližjega vozlišča, pri čemer uporabnikom ni potrebno poznavanje naslovov vozlišč omrežja IPFS.



Slika 1: Prikaz sistemov, ki temeljijo na lokaciji in sistemov, ki naslavljajo vsebino (povzeto po [6])

V nadaljevanju bomo podrobno predstavili arhitekturo in podrobnosti delovanja protokola IPFS. Prednosti in slabosti takšnega sistema za porazdeljeno shranjevanje datotek in primerjava z drugimi sistemi P2P so izpostavljeni v tretjem poglavju. Možnosti in primeri uporabe ter izzivi ob uporabi z ali brez tehnologije veriženja blokov so predstavljeni v četrtem poglavju. Zaključek je podan v petem poglavju.

2 ARHITEKTURA

Protokol IPFS je sestavljen iz več podprotokolov, ki so odgovorni za naslednje funkcionalnosti: (1) identiteto – identifikacijo vozlišč vrstnikov; (2) omrežje – upravljanje povezav med vrstniki; (3) usmerjanje (ang. routing) – iskanje vrstnikov in shranjenih objektov; (4) izmenjavo – upravljanje porazdelitve blokov; (5) objekte – vsebinsko naslavljanje nespremenljivih objektov in povezav; (6) datoteke – nadzorovanje različic in (7) poimenovanje (ang. naming) – spremenljivo poimenovanje trajnih objektov. Našteti protokoli niso neodvisni, temveč so integrirani z namenom izkoriščanja kombiniranih lastnosti [7].

2.1 Identitete (ang. Identities)

Protokol za upravljanje identitet skrbi za ustvarjanje identitet vozlišč in njihovo overjanje. Vozlišča so identificirana z uporabo kriptografske zgostitve javnega ključa, ki se beleži kot NodeId. Vozlišča hranijo svoje javne in zasebne ključe. Uporabniki sicer lahko ob vsakem zagonu ustvarijo novo vozliščno identiteto, vendar s tem ne morejo izkoristiti prednosti, ki so jih v omrežju nabrali. Namreč, vsako vozlišče za ohranjanje identitete dobi spodbudo. Ob prvi povezavi vozlišča izmenjajo javne ključe in preverijo, ali se zgostitev kateregakoli javnega ključa ujema z NodeId. Če ujemanje obstaja, se povezava ohrani, sicer se povezava prekine [7].

2.2 Omrežje (ang. Network)

Protokol IPFS upravlja povezave med vozlišči in pri tem uporablja različne mrežne protokole. Za prenos podatkov največkrat uporablja protokola WebRTC Data Channels in uTP [8]. Zanesljivost je zagotovljena z uporabo uTP LEDBAT (ang. Low Extra Delay Background Transport) ali SCTP (ang. Stream Control Transmission Protocol). Za povezovanje uporablja IPFS traverzne tehnike ICE NAT (ang. Interactive Connectivity Establishment Network Address Translation). Opcijsko lahko preverja celovitost sporočil z uporabo zgostitve ter avtentičnost sporočil z uporabo HMAC (ang. Hash Message Authentication Code) in pošiljateljevega javnega ključa [7].

2.3 Usmerjanje (ang. Routing)

Protokol IPFS skrbi za vzdrževanje informacij, potrebnih za lociranje konkretnih vozlišč – vrstnikov, ki imajo iskan objekt. Delovanje je prednastavljeno na uporabo porazdeljene neurejene zgoščevalne tabele (ang. Distributed Sloppy Hash Table - DSHT), lahko pa se uporabi tudi drugačen princip glede na potrebe uporabnikov [7]. Porazdeljena zgoščevalna tabela se uporablja za usklajevanje in upravljanje metapodatkov o sistemu P2P. Uporabljena je implementacija Kademlia [9] tako kot v sistemu BitTorrent, kateri je izboljšana razširljivost in zmogljivost z implementacijo, ki jo prinaša Coral [10]. Med drugim omogoči vozliščem iskanje datotek ne glede na njihovo priljubljenost in se izogne vročim vstopnim točkam (ang. hotspots) v indeksni infrastrukturi [7], [11].

2.4 Izmenjava (ang. Exchange)

Izmenjava podatkov temelji na protokolu BitSwap [7], ki upravlja učinkovito razdeljevanje blokov ob spodbujanju replikacije podatkov. Vrstniki imajo seznam blokov, ki jih želijo pridobiti, in seznam tistih blokov, ki jih lahko ponudijo. Protokol BitSwap ne zahteva prenosa vseh zahtevanih blokov od enega vrstnika. Lahko se pridobivajo bloki iz različnih vozlišč in iz popolnoma nepovezanih datotek. Omogoča izkoriščanje fizične bližine za prevzem iskane vsebine v majhnih delih. BitSwap temelji na sporočilnem protokolu, kjer vsa sporočila vsebujejo seznam želja. Ko vrstnik prejme sporočilo z zahtevami, se lahko odloči, ali pošlje zahtevane bloke ali zavrne zahtevo, kar je opredeljeno s strategijo kdaj in komu se pošiljajo podatki. V primeru, da bloke prejme, pošiljatelj zahteve posodobi listo želja in sporoči vsem vrstnikom, da prejetih blokov več ne potrebuje.

2.5 Objekti (ang. Objects)

Protokol temelji na Merklovem usmerjenem acikličnem grafu (ang. Directed Aciclic Graph - DAG) [7], ki se uporablja za predstavitev alternativnih podatkovnih struktur, kot so hierarhije datotek in komunikacijski sistemi. Vsebina, ki jo shranjujemo na sistem IPFS, je razdeljena na manjše dele, t.j. objekte velikosti 256 Kb. Za vsak objekt je pridobljena zgostitev vsebine, pri čemer so objekti kombinirani v hierarhično podatkovno strukturo Merklovega usmerjenega acikličnega grafa. Vgrajene so povezave do objektov, ki so predstavljeni z zgostitvami. Povezave do objektov lahko same overjajo objekte. Naslavljanje vsebine, odpornost proti spremembam in odstranitev dvojnih vsebin so lastnosti, ki jih protokolu IPFS zagotavlja Merklov usmerjen aciklični graf. Ta predstavlja temeljno podatkovno strukturo za podatkovni model IPLD (ang. Interplanetary Linked Data) [12].

2.6 Datoteke (ang. Files)

IPFS omogoča verzioniranje datotek v Merklovem usmerjenem acikličnem grafu. Model verzioniranja je podoben modelu Git [13]. Vsebuje naslednje objekte [7]:

- Zbirko binarnih podatkov (ang. blob), ki predstavlja blok podatkov spremenljive velikosti in nima povezav.
- Seznam, ki predstavlja veliko datoteko ali nepodvojeno datoteko, ki je sestavljena iz več sestavljenih zbirk binarnih podatkov. Seznami vsebujejo urejeno zaporedje zbirk binarnih podatkov ali seznamov objektov.
- Drevo predstavlja direktorij nazivov zgostitev. Zgostitve lahko referencirajo zbirke binarnih podatkov, sezname, druga drevesa ali potrditve.
- Potrditev predstavlja posnetek v zgodovini verzij kateregakoli objekta in lahko referencira katerikoli tip objekta.

2.7 Poimenovanje (ang. Naming)

IPFS uporablja sistem za lastno potrjevanje poimenovanja (ang. Self-certifying File System) [14]. Ponuja način za pridobivanje spremenjenih objektov z uporabo enakega naslova. Omogoča vzpostavitev lastnega potrjevanja imen, ki so spremenljiva. Vozlišča so implicitno overjena preko svojih imen, saj vključujejo odtis javnega ključa strežnika. Temelji na kriptografsko dodeljenem globalnem imenskem prostoru. Vsak uporabnik dobi spremenljiv imenski prostor, ki ima oznako IPNS (InterPlanetary Name Space) [7]. Tako je preprosto postavljena ločnica med spremenljivo (ipns) in nespremenljivo (ipfs) vsebino. Najprej se objekt objavi kot nespremenljiv objekt IPFS, nato se njegova zgostitev objavi na sistemu za usmerjanje kot vrednost metapodatka. IPNS skrbi za kreiranje zaupanja vrednih verig. IPFS uporablja različne tehnike (npr. povezave vrstnikov, storitve okrajšav imen, izgovorljivi identifikatorji, uporaba DNS TXT zapisov) za povečanje uporabniške prijaznosti ob uporabi imen, ki so vrednosti zgostitev.

3 PREDNOSTI IN SLABOSTI SISTEMA IPFS

Nekatere prednosti IPFS so neposredno vezane na prednosti, ki izhajajo iz uporabe porazdeljenih in decentraliziranih sistemov, kot so: (1) ni kritične točke odpovedi, (2) ni cenzure vsebin in (3) pohitreno pridobivanje vsebin zaradi možnosti izkoriščanja fizične bližine [8][6]. Določene prednosti pa so rezultat uspešnega kombiniranja lastnosti različnih sistemov P2P [1]: (1) uporaba zgostitve za naslavljanje vsebine, zagotavlja istovetnost vsebine ne glede na njeno lokacijo, (2) vozlišča so medsebojno neodvisna in ni potrebe po zaupanju med njimi, (3) ni omejitve prostora za shranjevanje – po potrebi je možno kapacitete povečati, (4) vsebina je dostopna v omrežju tako dolgo, dokler je nekomu dovolj v interesu, da jo ohranja dostopno, (5) omogoča hitrejše pošiljanje ali deljenje datotek in (6) zagotavlja poceni komunikacijo med vozlišči.

Nektere potencialne slabosti sistema IPFS izhajajo iz dejanskega principa delovanja, ki temelji na soudeležbi in sodelovanju. Če nihče nima vsebine, ki se nahaja na naslovu (zgostitvi), vsebina ne bo dosegljiva. Ni vgrajenega mehanizma, ki zagotavlja dostopnost vsebine. Vsebina ni nujno trajno shranjena v omrežju, saj lahko vozlišča poljubno brišejo datoteke z namenom varčevanja prostora. Istočasno pa vsebina ne more biti popolnoma odstranjena iz omrežja IPFS tako dolgo, dokler jo vsaj eno vozlišče ohranja dostopno, ne glede na to, ali je ta oseba prvotni avtor vsebine [8].

Omenjene izzive je naslovila javna platforma tehnologije veriženja blokov, imenovana FileCoin [16], ki predvideva finančno spodbudo za hranjenje velikih količin podatkov ter za zagotavljanje dostopnosti vsebine. FileCoin predvideva uporabo dveh mehanizmov soglasja – dokaz o prostoru in času (ang. Proof-of-Spacetime) in dokaz o replikaciji (ang. Proof-of-Replication).

Replikacija v sistemu IPFS se izvaja v trenutku iskanja vsebine – ne podpira popolne replikacije. Posledično pomeni, da se vsebine, ki niso zelo iskane, nahajajo na manjšem številu vozlišč, kar vodi v počasno pridobivanje takšnih vsebin. Dodatni izziv je rezultat dejstva, da je vsebina, ki se objavi na IPFS, v osnovi javno dostopna. IPFS ne ponuja vgrajenega mehanizma za shranjevanje zasebnih podatkov. Za ohranitev zasebnosti shranjene vsebine je potrebno pred shranjevanjem izvesti šifriranje le-te.

3.1 Primerjava z drugimi sistemi P2P

Osnovne lastnosti, ki jih zagotavljajo vse decentralizirane shrambe P2P, so transparentnost, odpornost na napake in razširljivost. Nekatere specifične lastnosti za štiri porazdeljene sisteme P2P (IPFS, Swarm, Storj in Sia) smo povzeli v Tabeli 1.

Tabela 1: Primerjava lastnosti sistemov za porazdeljeno shranjevanje v omrežje vrstnikov

	IPFS	**Swarm**	**Storj**	**Sia**
Naslavljanje vsebine	Zgostitev	Zgostitev	Zgostitev	Zgostitev
Velikost bloka	256 Kb	≤ 4 Kb	≤ 4 Kb	40 Mb
Shramba	Zagotovilo ni vgrajeno Možno z uporabo FileCoin	Zagotovilo vgrajeno	Zagotovilo vgrajeno	Zagotovilo vgrajeno
Gostovanje	Nizko nivojski protokol - projekti se gradijo na infrastrukturi IPFS	V oblaku	V oblaku	V oblaku
Spodbuda	Ni vgrajena Možna uporaba FileCoin	Vgrajena	Vgrajena	Vgrajena
Kriptovaluta	Ni	Swarm	Storj token	Siacoin
Mehanizem soglasja	Ni vgrajen FileCoin uporablja Dokaz o prostoru in času in Dokaz o replikaciji	Dokaz o skrbništvu	Dokaz o delu	Dokaz o shrambi
Zasebnost	Ni vgrajenega mehanizma šifriranja Možna nastavitev zasebnega omrežja	Šifriranje podatkov vgrajeno	Šifriranje podatkov vgrajeno	Šifriranje podatkov vgrajeno
Replikacija	Lokalna replikacija – v trenutku povpraševanja po vsebini	Lokalna replikacija in izbrisna koda	Paritetno črepinjenje (ang. parity shards) in izbrisna koda	Replikacija na osnovi kode Reed-Solomon in izbrisna koda
Tehnologija veriženja blokov	Ni neposredne povezave	Temelji na plaformi Ethereum	Ethereum	Pametne pogodbe

Swarm je porazdeljena platforma za shranjevanje in deljenje vsebin, ki predstavlja lastni temeljni sloj sklada Ethereum Web 3 [17]. Njegova primarna naloga je zadostno decentralizirana in redundanta shramba javnih zapisov platforme Ethereum – shranjevanje in porazdelitev kode in podatkov porazdeljenih aplikacij (dApp) kot tudi podatkov, shranjenih na verigo blokov [1]. Kot del sklada Ethereum omogoča preprosto uporabo pametnih pogodb. Bolj je primeren za shranjevanje delčkov podatkov (maksimalne velikosti 4 Kb), kar omogoča majhne zakasnitve. Temelji na nadomeščanju podatkovnega sloja s slojem shrambe [15]. Vse informacije so shranjene v nespremenljivi shrambi delčkov, ki uporablja vsebino za naslavljanje. Ima integrirano spodbudo za shranjevanje vsebine, za razliko od sistema IPFS, ki tega v osnovi ne podpira. Posledično ima Swarm integrirano zagotovilo shranjevanja podatkov, med tem ko IPFS ne [1]. Prav zaradi potrebe po finančni spodbudi za shranjevanje velikih količin podatkov, predvideva IPFS uporabo omrežja FileCoin. Swarm vključuje spodbude za dolgoročno shranjevanje vsebine, ki ni zelo iskana kot tudi spodbude za zelo iskano vsebino. Vsaka zahteva drugačen pristop glede nagrad oziroma kazni. Vključuje princip dokaza o skrbništvu (ang. Proof-of-Custody), kar omogoča obstoj dokazila o shranjeni vsebini, brez potrebe po

prenosu celotne vsebine in brez zahteve po razkritju dejanske vsebine. Swarm se uporablja kot storitev, namenjena gostovanju v oblaku (ang. cloud hosting), medtem ko se IPFS uporablja za projekte, ki bodo zgrajeni na lastni infrastrukturi. IPFS prepušča razvijalcem iskanje dejanskih pomnilniških kapacitet [1]. Glede zrelosti kode, razširljivosti, sprejetja, skupnosti in interakcije s skupnostjo razvijalcev prednjači IPFS pred Swarm-om. Medtem ko IPFS cilja na postavitev protokola in je bolj splošen, nižje nivojski projekt, je Swarm vezan na vizijo Ethreum Web 3 z večjim poudarkom na odpornosti na cenzure [1]. Posledično je glavna prednost sistema Swarm možnost neposredne integracije z omrežjem Ethereum.

Storj je odprtokodni projekt zagotavljanja decentralizirane datotečne shrambe, ki temelji na omrežju Ethereum [18], [19]. Vključuje lastno kriptovaluto (Storj token). Uporablja šifriranje podatkov, deljenje datotek in zgoščevalno tabelo za shranjevanje datotek v omrežju P2P. Omrežje temelji na mehanizmu soglasja dokaz o delu (ang. Proof-of-Work). Predvideva tudi profitni del (Storj Labs), ki oddaja v najem omrežje in zaračunava njegovo uporabo [19]. Gre za nekoliko bolj centraliziran model, ki je neposredna konkurenca Dropbox-u in Google Drive-u.

Sia je decentralizirana oblačna platforma za shranjevanje podatkov. Platforma povezuje gostitelje, ki imajo neiskoriščen pomnilniški prostor z najemniki, ki potrebujejo prostor za shranjevanje podatkov. Uporablja šifriranje podatkov, pametne pogodbe in napredno metodo redundance, ki temelji na shemi Reed-Solomon [20]. Metoda deluje po principu 10 od 30, kar pomeni shranjevanje vsebine na 30 vozlišč po celem svetu, pri čemer je potrebnih zgolj 10 vozlišč, da je vsebina primerno zavarovana in vedno dostopna [21]. Sia vključuje lastno kriptovaluto – Siacoin, s katero je zagotovljena finančna spodbuda za gostitelje. Uporabniki za najem shrambe uporabijo Sia žetone, gostitelji pa položijo Sia žetone kot odškodnino v primeru, da bi njihova vozlišča bila nedostopna.

3.2 Sinergija protokola IPFS in tehnologije veriženja blokov

Tehnologija veriženja blokov omogoča vzpostavitev porazdeljenega in decentraliziranega okolja ali shrambe P2P, v katerem so podatki in transakcije zaradi uporabe kriptografskih mehanizmov varni, celoviti in zaupanja vredni. Ker je okolje hkrati decentralizirano in ni pod nadzorom osrednje avtoritete, se upravljanje porazdeli enakomerno med vse deležnike (vozlišča). Vsaka dokončana transakcija se na preverljiv in trajen način hrani v javno dostopni glavni knjigi (ang. ledger) v obliki verige blokov [22]. Ena izmed pomanjkljivosti tehnologije veriženja blokov je omejena velikost podatkov, ki jih lahko pri izvedbi transakcije shranimo v verigo blokov. Tehnologija veriženja blokov se največkrat povezuje s kriptovalutami, njen potencial pa se kaže tudi izven te domene uporabe, saj lahko pozitivne lastnosti tehnologije izkoristimo tudi pri razvoju inovativnih storitev na drugih področjih, npr. v zdravstvu, zavarovalništvu, izobraževanju itd. Poslovni procesi, ki se izvajajo znotraj omenjenih domen, ponavadi zahtevajo obdelavo večje količine podatkov in datotek kot jih lahko učinkovito shranjujemo znotraj obstoječih omrežij verig blokov. Rešitev se kaže v uporabi decentraliziranih in porazdeljenih shramb P2P, ki ne omejujejo velikosti shranjenih podatkov. Podatki se naslovijo in shranijo v decentralizirano in porazdeljeno omrežje, medtem ko se nespremenljive in trajne zgostitve vsebine, poslane Znotraj transakcije, shranijo v omrežje verige blokov. Tako se zaščiti vsebina, ne da bi podatke dejansko hranili na verigi blokov.

4 PRIMERI UPORABE SISTEMA IPFS

V nadaljevanju bomo predstavili možnosti uporabe sistema IPFS. Lahko se uporablja kot:

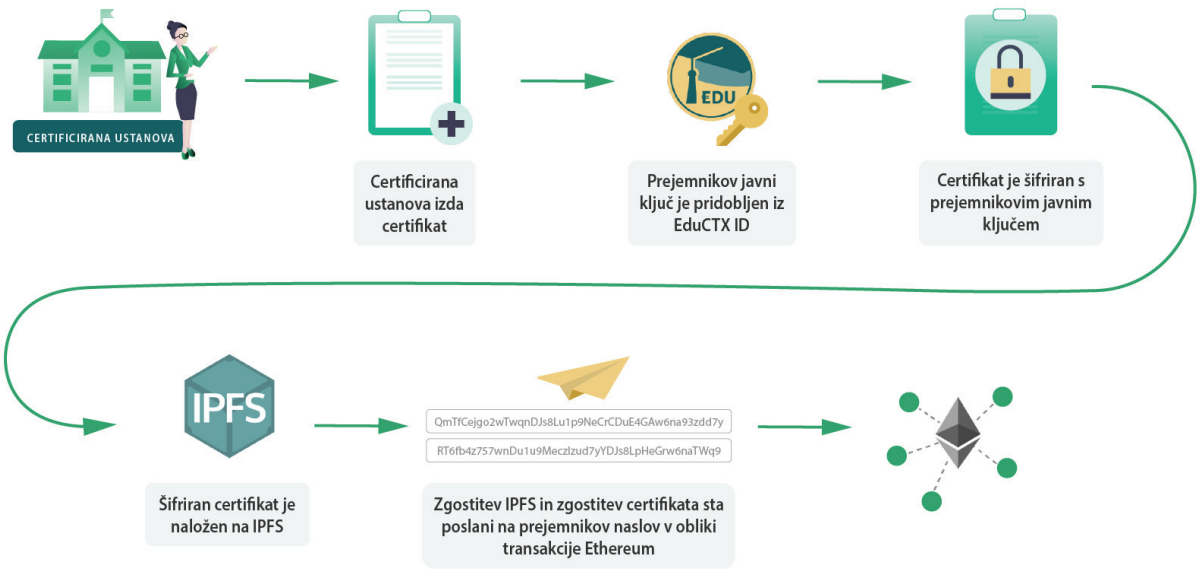
- nameščen globalni datotečni sistem z /ipfs in /ipns,
- osebna sinhronizarcijska mapa, ki samodejno izvaja verzioniranje,
- objavljanje in varnostno beleženje vseh zapisov,
- sistem za deljenje šifriranih datotek ali podatkov,
- upravljalec verzioniranja za vso programsko opremo,
- korenski datotečni sistem navideznih računalnikov (ang. virtual machine),
- zagonski datotečni sistem navideznih računalnikov,
- podatkovna baza, kjer lahko aplikacije pišejo neposredno v podatkovni model Merklovega usmerjenega acikličnega grafa in imajo dostopno verzioniranje, predpomnenje in porazdelitev,

- povezana in šifrirana komunikacijska platforma,
- neokrnjeno omrežje za porazdelitev,
- omrežje za verzioniranje in dostavo vsebin (ang. content delivery network - CDN) za velike datoteke,
- šifrirano omrežje za dostavo vsebin,
- spletno omrežje za dostavo vsebin, nova platforma za shranjevanje aplikacij (zelo primerna za decentralizirane aplikacije),
- nov trajni splet, kjer so povezave vedno dostopne in ne zastarajo,
- porazdeljena shramba, ki rešuje problem razširljivosti tehnologije veriženja blokov [1].

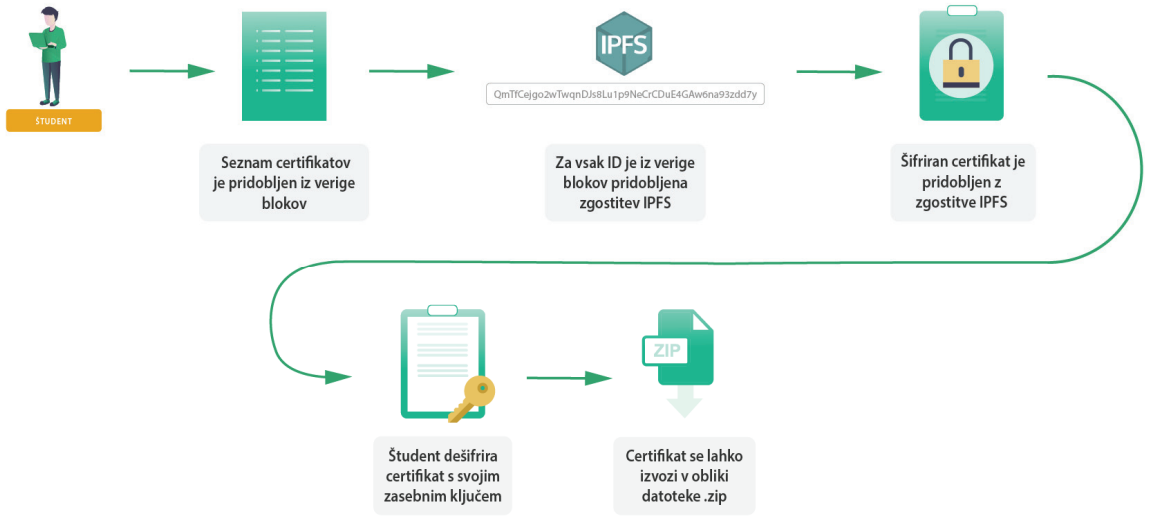
Slednja uporaba predvideva naslavljanje velike količine podatkov s sistemom IPFS. Nespremenljive, trajne povezave na IPFS pa pošlje v transakcijo na verigo blokov. Tako je vsebina označena s časovnim žigom in zagotovljena nespremenljivost, čeprav vsebina ni shranjena v verigi blokov.

4.1 Projekt EduCTX

Uporabnost sistema IPFS bomo predstavili na pilotnem projektu EduCTX, ki kombinira tehnologijo veriženja blokov in porazdeljeno shrambo IPFS. EduCTX je decentralizirana platforma za upravljanje digitalnih mikro certifikatov [22]. Temelji na tehnologiji veriženja blokov. Žetoni in transakcije ECTX predstavljajo zaupanja vreden in transparenten dokaz o pridobljenih veščinah in znanjih posameznikov v obliki digitalnih mikro-certifikatov. V projektu sta uspešno združeni tehnologija veriženja blokov in sistem IPFS. Da poskrbimo za zasebnost podatkov, ki bodo shranjeni v omrežju IPFS, je potrebno najprej datoteko, ki predstavlja digitalni mikro certifikat, šifrirati. Šifrirana datoteka je nato naložena na IPFS, medtem ko je zgostitev šifrirane datoteke, ki predstavlja naslov IPFS, skupaj z zgostitvijo mikro certifikata (ki bo uporabljena za validacijo dokumenta) shranjena v verigo blokov. Tako je sistem IPFS uporabljen kot zasebno oziroma konzorcijsko omrežje. Postopek izdaje mikro-certifikata je predstavljen na Sliki 2, medtem ko je postopek izvoza mikro certifikatov predstavljen na Sliki 3.



Slika 2: Postopek izdaje certifikata na platformi EduCTX



Slika 3: Postopek izvoza certifikata na platformi EduCTX

5 ZAKLJUČEK

IPFS je decentralizirana in globalno porazdeljena shramba P2P. Podpira usmerjanje in iskanje vsebine in vozlišč, ki to vsebino imajo. Vozlišča so implicitno overjena preko imen, saj vključujejo odtis javnega ključa strežnika. Merklov usmerjen aciklični graf omogoča trajno shranjevanje unikatno identificiranih podatkov. Omogoča dostop do shranjenih objektov preko njihovih zgostitev ter skrbi za preverjanje celovitosti in povezave na druge objekte. Porazdeljena zgoščevalna tabela omogoča objavo objektov na varen in popolnoma porazdeljen način. Kdorkoli lahko objavi objekt tako, da preprosto doda svoj ključ v porazdeljeno zgoščevalno tabelo, sebe doda kot vrstnika in deli z drugimi vrstniki pot do objekta. Objekti so nespremenljivi, saj nove različice objektov rezultirajo v drugačni zgostitvi, kar posledično pomeni, da so to novi objekti. Sledenje verzijam pa je naloga dodatnega verzioniranja objektov, ki omogoča dostope s pomočjo spremenljivih nazivov.

Čeprav ni prvotno temu namenjen, je protokol IPFS pridobil na priljubljenosti, predvsem zaradi uporabe v sklopu decentraliziranih aplikacij (dApp), ki temeljijo na tehnologiji veriženja blokov. IPFS se kaže kot primerna rešitev za omejeno velikost podatkov, ki jih lahko pri opravljanju transakcije shranimo v verigo blokov. Datoteke s podatki se ob uporabi sistema IPFS shranijo v decentralizirano in porazdeljeno omrežje, medtem ko se zgostitve vsebine shranijo v omrežje verig blokov, ki delujejo kot kazalec na varno shranjeno večjo količino podatkov. Na ta način se vsebina zaščiti, ne da bi podatke dejansko hranili na verigi blokov.

Glavne prednosti protokola IPFS so porazdeljeno shranjevanje brez omejitve velikosti shranjenih objektov, zagotovilo istovetnosti vsebine ne glede na njeno lokacijo, neobstoj kritične točke odpovedi in neobstoj zahteve za zaupanjem med vozlišči. Glavne slabosti protokola so neobstoj vgrajenega zagotovila shranjevanja podatkov in dostopnosti le-teh, ter mehanizma za shranjevanje zasebnih podatkov. Z rešitvijo EduCTX smo pokazali, da je možna učinkovita uporaba sistema IPFS za dopolnitev fukncionalnosti, ki jih ponuja tehnologija veriženja blokov. V prihodnosti bomo izvedli celovito primerjavo učinkovitosti uporabe različnih sistemov za porazdeljeno shranjevanje P2P.

6 LITERATURA

[1] JANKOV Tonino "Decentralized Storage and Publication with IPFS and Swarm", 2018.
[2] RIPEANU Matei, IAMNITCHI Adriana, FOSTER Ian "Mapping the Gnutella Network", IEEE Internet Computing, letnik 6, številka 1, februar 2002, str. 50-57.
[3] www.bittorrent.com/, BitTorrent, obiskano 21. 5. 2019.
[4] www.tribler.org, Tribler, obiskano 21.5.2019.
[5] https://freenetproject.org/, Freenet, obiskano 21. 5. 2019.
[6] https://www.maxcdn.com/one/visual-glossary/interplanetary-file-system/, Visual Glossary, Stackpath, obiskano 20.5.2019.
[7] BENET Juan "IPFS – Content Addressed, Versioned, P2P File System", 2014, https://arxiv.org/abs/1407.3561v1.
[8] https://docs.ipfs.io/, IPFS Documentation, obiskano 21. 5. 2019.
[9] MAYMOUNKOV Petar, MAZIERES David "Kademlia: A peer-to-peer information system based on the XOR metric", International Workshop on Peer-to-Peer Systems, IPTPS 2001, Lecture Notes in Computer Science 2429, str. 53-65.
[10] FREEDMAN Michael J., MAZIERES David "Sloppy hashing and self-organizing clusters", International Workshop on Peer-to-Peer Systems, IPTPS 2003, Lecture Notes in Computer Science 2735, str. 45-55.
[11] ZHANG Hao, WEN Yonggang, XIE Haiyong, YU Nenghai "A Survey on Distributed Hash Table (DHT): Theory, Platforms, and Applications", Springer Publishing Company, Incorporated, 2013.
[12] https://ipld.io/, IPLD Documentation, obiskano 26.5.2019.
[13] https://git-scm.com/doc, GIT Documentation, obiskano 25. 5. 2019.
[14] MAZIERES David "Self-certifying File System", doktorat znanosti, MIT, junij 2000.
[15] VERMA Atul Kumar, GARG Arpit "IPFS and SWARM: Future of decentralized storage system", International Journal of Engineering Research in Computer Science and Engineering, letnik 4, številka 11, november 2017, str. 14-17.
[16] PROTOCOL LABS "Filecoin: A Decentralized Storage Network", Whitepaper, 19. julij 2017.
[17] https://swarm-guide.readthedocs.io, Swarm Documentation, obiskano 25. 5. 2019.
[18] https://github.com/storj/storj/wiki, Storj Documentation, obiskano 25. 5. 2019.
[19] STORJ LABS "Storj: A Decentralized Cloud Storage Network Framework", Whitepaper V3, 30. oktober 2018.
[20] REED I, SOLOMON G "Polynomial Codes Over Certain Finite Fields", J. Soc. Ind. Appl. Math., letnik 9, številka 2, junij 1960, str. 300-304.
[21] https://sia.tech/docs/, Sia Documentation, obiskano 25. 5. 2019.
[22] TURKANOVIĆ Muhamed, HÖLBL Marko, KOŠIČ Kristjan, HERIČKO Marjan, KAMIŠALIĆ Aida EduCTX: "A blockchain-based higher education credit platform", IEEE Access, številka 6, januar 2018, str. 5112-5127.

OTS 2019
SODOBNE INFORMACIJSKE TEHNOLOGIJE IN STORITVE
M. Heričko in K. Kous (ur.)

Univerzitetna založba
Univerze v Mariboru

PLATFORMA GEMALOGIC FLEXIBILITY ZA ZDRUŽEVANJE DEA NAMEŠČENIH V RTP

TOMAŽ BUH, ERVIN PLANINC, PRIMOŽ BOGATAJ, SIMON MIHEVC,
SERGEJ ANŽELJ, MIROSLAV BERANIČ, GORAN ČAPELNIK

Povzetek: *Referat predstavlja platformo GemaLogic Flexibility, ki je bila razvita za namen združevanja dizel električnih agregatov (v nadaljevanju DEA), ki so nameščeni v razdelilnih transformatorskih postajah (v nadaljevanju RTP) sistemskega operaterja prenosnega omrežja ELES. Družba ELES, d.o.o. ima za potrebe napajanja nujne lastne rabe (v nadaljevanju NLR) v svojih RTP nameščene dizel električne agregate, kateri so uporabljeni v primeru izpada primarnega napajanja iz omrežja. Stalna topla pripravljenost in zadostna kapaciteta goriva DEA omogočata koriščenje tudi zadruge namene, kot na primer zagotavljanje sistemske storitve ročne rezerve za povrnitev frekvence (v nadaljevanju rRPF), pri tem pa zagotavljanje napajanja NLR ne sme biti oslabljeno ali ogroženo. V družbi ELES, d.o.o. so se tako odločili, da DEA uporabijo za vključevanje v elektroenergetski sistem (v nadaljevanju EES) za potrebe rRPF. DEA bodo predvidoma proženi v takšni meri, da ne bodo več potrebni redni mesečni zagonski testi. Uporabljen je bil OpenADR protokol, ki je zasnovan za potrebe prejemanja in izvajanja funkcionalnosti upravljanja odjema električne energije ter omogoča standardizirano izmenjavo podatkov ter temelji na predhodnih OASIS standardih.*
Referat prav tako podaja pregled razvitih funkcionalnosti za potrebe integracije zalednih procesnih in informacijskih sistemov v podjetju ELES, kot je za potrebe vodenja integracija s SCADA EMS sistemom z uporabo ICCP TASE.2 protokola, sistemom za upravljanje s sredstvi MAXIMO, sistemom obveščanja in alarmiranja EnterpriseAlert, sistemom za zajem števčnih obračunskih podatkov Advance in dizel električnimi agregati DEA.

Ključne besede: *OpenADR, SCADA EMS, ICCP TASE.2, GemaLogic, sistemske storitve, ročna rezerva za povrnitev frekvence, dizel električni agregat, razdelilna transformatorska postaja, platforma, sistemski operater prenosnega omrežja, izravnava sistema, informacijski sistem, informacijska rešitev*

NASLOVI AVTORJEV: Tomaž Buh, Solvera Lynx d.o.o., Ljubljana, Slovenija, e-pošta: tomaz.buh@solvera-lynx.com. Mag. Ervin Planinc, Eles d.o.o., Ljubljana, Slovenija, ervin.planinc@eles.si. Primož Bogataj, Eles d.o.o., Ljubljana, Slovenija, primoz.bogataj@eles.si. Simon Mihevc, Solvera Lynx d.o.o., Ljubljana, Slovenija, e-pošta: simon.mihevc@solvera-lynx.com. Sergej Anželj, Bintegra d.o.o., Hoče, Slovenija, e-pošta: sergej.anzelj@bintegra.com. Miroslav Beranič, Bintegra d.o.o., Hoče, Slovenija, e-pošta: miroslav.beranic@bintegra.com. Goran Čapelnik, Bintegra d.o.o., Slovenija, e-pošta: goran.capelnik@bintegra.com.

https://doi.org/10.18690/978-961-286-282-4.10 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 UVOD

Družba ELES, d.o.o. ima za potrebe napajanja porabnikov nujne lastne rabe (v nadaljevanju NLR) v svojih razdelilnih transformatorskih postajah (v nadaljevanju RTP) na voljo dizel električne agregate (v nadaljevanju DEA), ki se uporabljajo za zagotavljanje napajanja NLR. Kapaciteta goriva DEA poleg zagotavljanja NLR omogoča tudi zagotavljanje sistemskih storitev (v nadaljevanju SR) za čas porabe goriva do te mere, da NLR ni ogrožena. Zaradi neizkoriščenosti DEA, trenutno se le-ti zaganjajo zgolj enkrat mesečno za potrebe periodičnih mesečnih testov delovanja (zgolj zagon DEA, brez testa sinhronizacije na omrežje), se je ELES odločil, da se DEA uporabi za vključitev DEA v nudenje podpori ročne rezerve za povrnitev frekvence (v nadaljevanju rRPF). Za potrebe vključevanja DEA v elektro-energetski sistem (v nadaljevanju EES) je bila razvita programska rešitev GemaLogic Flexibility (v nadaljevanju DEA platforma). DEA so sposobni hitrega zagona in otočnega napajanja porabnikov NLR v primeru izpada primarnega vira napajanja NLR RTP-ja. Po ponovni vzpostavitvi omrežja se napajanje ponovno preklopi na zunanje omrežje in DEA se zaustavi. Skupna moč DEA, ki so primerni za vključitev v rRPF je 4,8 MW.

V obseg sistemskih storitev, ki jih izvaja ELES, spada tudi zagotavljanje ročne rezerve za povrnitev frekvence. DEA, ki so nameščeni v RTP spadajo med rezervne vire napajanja in zagotavljajo napajanje NLR v času, ko napajanje iz glavnega ali pomožnega vira ni možno. DEA se zaženejo samodejno, nekaj sekund po izpadu omrežja in morajo zagotavljati vsaj 24 urno avtonomijo rezervnega napajanja NLR.

Vzpostavljanje SR z DEA je projekt združevanja 15 daljinsko vodenih malih proizvodnih virov električne energije ($P_n => 1MW$ t.i. mikro elektrarne), ki je bil izveden tako, da primarna funkcija DEA, tj. zagotavljanje NLR, ni nikoli ogrožena.
V referatu je predstavljen razvoj in integracija DEA platforme na zaledne informacijske sisteme. Na nivoju med DEA platformo in integracijskim vodilom je bil uporabljen protokol OpenADR, ki je zasnovan za potrebe prejemanja in izvajanja funkcionalnosti upravljanja odjema električne energije ter omogoča standardizirano izmenjavo podatkov ter temelji na predhodnih OASIS standardih. Referat prav tako podaja pregled razvitih funkcionalnosti za potrebe integracije zalednih procesnih in informacijskih sistemov v podjetju ELES, kot je za potrebe vodenja integracija s SCADA EMS sistemom z uporabo ICCP TASE.2 protokola, sistemom za upravljanje s sredstvi, sistemom obveščanja in alarmiranja, sistemom za zajem števčnih obračunskih podatkov in DEA z uporabo komunikacijske povezovalne platforme.

2 FUNKCIONALNOST DEA

DEA platforma v sistem SR za potrebe rRPF vključuje DEA, ki se vključujejo v distribucijsko omrežje ali pa preko transformatorja lastne rabe RTP. Vključevanje DEA v distribucijsko omrežje mora biti izvedeno skladno z zahtevami Sistemskih obratovalnih navodil za distribucijsko omrežje električne energije – SONDO [1], zahteve Priloge 5 – SONDO, ki definira Pravila za priključevanje in obratovanje elektrarn električne moči do 10 MW [4] in za vključevanje DEA v prenosno omrežje skladno z navodili in pravili, ki so zapisana v Sistemskih obratovalnih navodilih za prenosno omrežje električne energije SONPO [6]. Ker so nazivne moči DEA večje od dejanske moči NLR posameznega objekta, kjer so DEA priključeni, se lahko DEA uporabi za potrebe SR. DEA lahko brez težav zadostijo zahtevam zagotavljanja SR rRPF, kjer je glavni pogoj aktivacija zahtevane moči najkasneje v 15 minutah od podane zahteve za aktivacijo. Za potrebe SR se lahko porabi delež dizel goriva, ki predstavlja višek goriva, ki jo DEA potrebuje za zagotavljanje NLR. DEA bodo za potrebe SR obratovali paralelno z omrežjem (omrežno obratovanje), kjer bodo zagotavljali regulacijo delovne moči. V omrežnem obratovanju bo operater nastavil želeno delovno moč (P) in čas trajanja aktivacije. Krmilnik na strani DEA zagotovi, da ob konstantnem številu vrtljajev dizel motorja prilagodi porabo goriva na način, da motor na gredi proizvede količino zahtevane moči. Krmilnik zagotovi vklop, sinhronizacijo in zaščito DEA.

2.1 Priključevanje in obratovanje DEA v distribucijskem omrežju

DEA so priključeni na distribucijsko omrežje zato morajo le-ti obratovati paralelno z distribucijskim omrežjem (t. i. omrežno obratovanje) ter hkrati upoštevati pravila, ki veljajo tudi za vse ostale elektrarne, ki obratujejo v distribucijskem omrežju in so zapisana v navodilih SONDO. Nekateri DEA so priključeni na transformator lastne rabe, kar pomeni, da je tako potrebno v tem primeru upoštevati zahteve SONPO.

Na nivoju DEA lokalna procesna avtomatika zagotavlja regulacijo delovne in jalove moči tako, da lahko DEA stabilno obratuje v distribucijskem omrežju. Lokalna procesna avtomatika skrbi tudi za zaščito DEA (pod-napetostna, nad-napetostna, nad-tokovna, kratkostična, pod-frekvenčna in nad-frekvenčna), ki je nastavljena tako, da loči DEA od omrežja v vsakem primeru, ko je v enem ali drugem omrežju okvara ali drugo nenormalno obratovalno stanje. Obenem pa lokalna procesna avtomatika zagotavlja tudi otočno obratovanje DEA (NLR), kar ima prioriteto pred omrežnim obratovanjem (SR).
Zagotovljeno je bilo tudi, da se je pred vsako priključitvijo DEA, ki so pred vključevanjem v DEA platformo obratovali izključno otočno za obratovanje paralelno z omrežjem opravilo preverbo obratovalnih stanj s katerimi se ugotovi možnosti in morebitne ovire pri zagonu DEA za potrebe SR.

2.2 Avtomatizacija, daljinsko vodenje, monitoring in merjenje

DEA so priključeni na distribucijsko omrežje zato morajo le-ti obratovati paralelno z distribucijskim omrežjem Posamezni DEA so opremljeni s krmilnikom Deep Sea Electronics DSE 8620 oziroma podobnim krmilnikom, ki zagotavlja obratovalna stanja DEA. Na nivoju komunikacije med DEA platformo in krmilnikom je uporabljen ModbusTCP protokol.

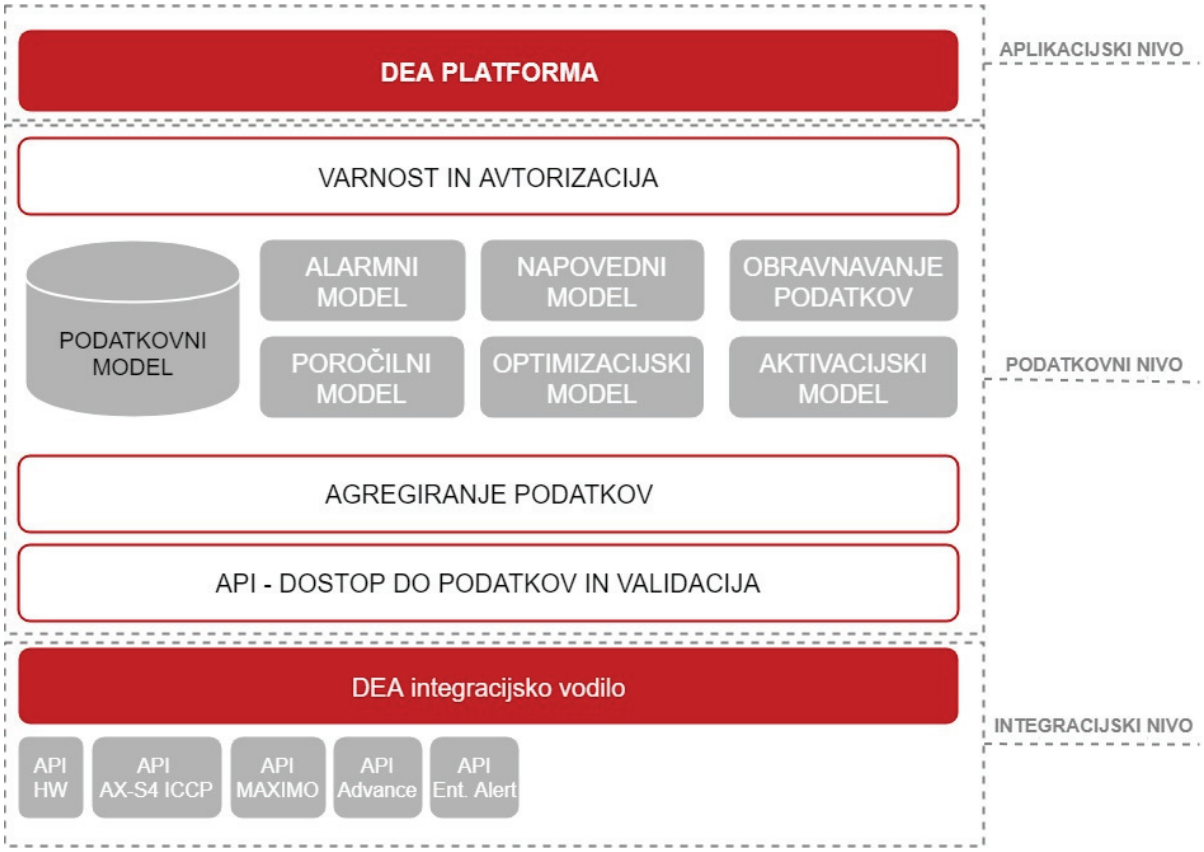
Na nivoju RTP sta vgrajena dva obračunska števca električne energije, kjer eden meri prevzeto/oddano električno energijo iz omrežja (P3) in drugi prevzeto/oddano električno energijo lastne rabe RTP, kjer je nameščen tudi DEA (P2) in tako meri proizvedeno električno energijo DEA. Glede na to se načina nadzora in obračuna nudene terciarne rezerve obračunava glede na stanje števca P2. Sistemski števec omogoča minutno merjenje v razredu točnosti 0,2. Za zajem merilnih podatkov je uporabljen sistem za zajem obračunskih merilnih podatkov HES Advance, kjer je na nivoju med obračunskim števcem in HES Advance poteka zajem podatkov poteka skladno s DLMS protokolom preko TCP/IP komunikacije.

3 ZASNOVA DEA PLATFORME

Funkcionalnost DEA platforme je namenjena združevanju DEA nameščenih v RTP, izvajanje funkcij daljinskega vklopa in izklopa le-teh s strani SCADA EMS, posredovanje alarmov v primeru napak vsem odgovornim osebam in izmenjave obratovalnih podatkov ter nerazpoložljivosti v in iz sistema za upravljanje s sredstvi. Prav tako je DEA platforma namenjena izvajanju funkcionalnosti monitoringa DEA, pregleda poročil, obračuna izvedenih aktivacij in enostavnega vključevanja dodatnih DEA, kjer avtomatiziran proces poskrbi za izvedbo celotne konfiguracije vseh integracijskih točk in vmesnikov, da dodan DEA služi celotni funkcionalnosti DEA platforme in nudenja podpore SR za potrebe zagotavljanja rRPF. Zaradi velikega števila integracijskih točk tako DEA platformo delimo na tri glavne nivoje, kot so integracijski nivo, podatkovni in aplikacijski nivo, kar tudi prikazuje Slika 1, ki predstavlja konceptualno zasnovo DEA platforme. Integracijski nivo (DEA integracijsko vodilo) DEA platformo povezuje z vsemi zalednimi informacijskimi sistemi, kot so:

- KepServerEX za povezavo z dizel električnimi agregati (DEA)
- AX-S4 ICCP (SCADA EMS),
- HES Advance,
- MAXIMO in
- Enterpreise Alert.

Aplikacijski nivo vključuje grafični vmesnik, ki preko integracijskega vodila omogoča pregled obratovalnih podatkov, razpršenih virov, virov, opozoril, preteklih aktivacij, opozoril, alarmov in drugih z merilnimi mesti povezanimi stanji ter statusi. Funkcionalni sklop podatkovnega nivoja omogoča funkcionalnosti izvajanja zahtevanih procesov aktivacije DEA s strani SCADA EMS.



Slika 1: Konceptualna zasnova DEA platforme

3.1 Arhitektura DEA platforme

Zasnova DEA platforme sestoji iz treh glavnih komponent in zunanjih aplikacijskih vmesnikov (API) za konfiguracijo naprav v povezovalni platformi KepServerEX, alarmiranje in registracijo VEN/VTN naprav preko OpenADR protokola. Glavne tri komponente DEA platforme tako predstavljajo:

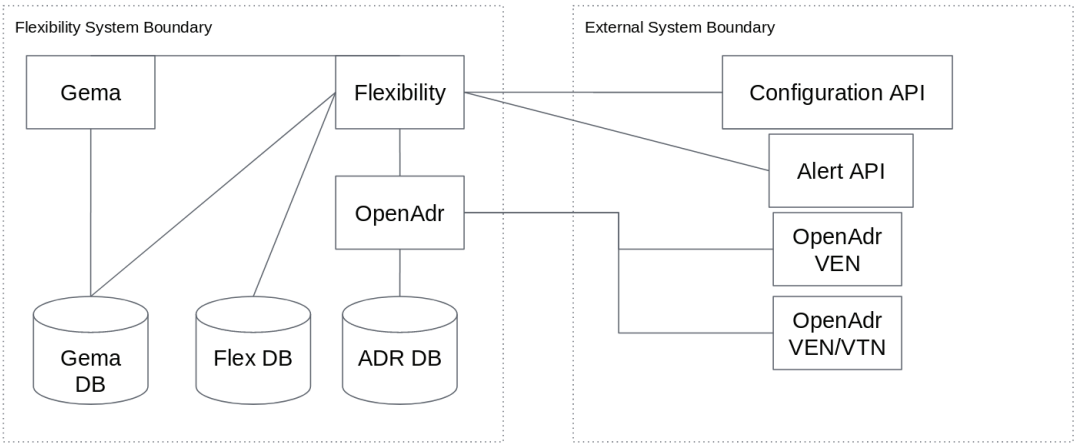
1. modul Gema,
2. modul Flexibility in
3. modul OpenADR.

Modul Gema sestoji iz:

- podatkovnega vodila GemaDB (uporabljeni sta metodi *»querry«* in *»insertion«*),
- opravila, ki so potrebni za izvajanje procesov, kot so agregacije, samodejne konfiguracije, izmenjave MetaData poročil, itn. (ang.: *»Job System«*),
- upravljanja uporabniških računov in uporabniških pravic (ang.: *»User and Privilege Management«*),
- procesiranja, upravljanja in shranjevanja podatkov (ang.: *»Process, data Handling and Storage«*),
- meta podatkov (ang.: *»MetaData – Data Sources«*),
- procesnih podatkov in pregleda zgodovinskih podatkov (ang.: *»Process data and history overwiev«*) pregleda urejanja alarmov,
- pregleda sproženih alarmov,
- geolokacijskega pregleda.
- preglednih plošč,
- shranjenih analiz,
- splošnega pregleda,
- ročnega vnosa,
- stanja,
- arhiva stanj,

- gradnikov pregledne plošče, kot so številčni prikaz, aktivacijski zahtevek, aktivacija v teku, razpoložljivost, nadzor razpršenih virov, sproženi alarmi, geolokacijski pregled, splošni pregled, stanje, arhiv stanj, tabela zadnjih vrednosti, tortni diagram ter drugi.

Slika 2 predstavlja arhitekturo programske rešitve GemaLogic z dodatnimi internimi moduli Flexibility, OpenADR, podatkovnimi bazami in zunanjimi aplikacijskimi vmesniki.

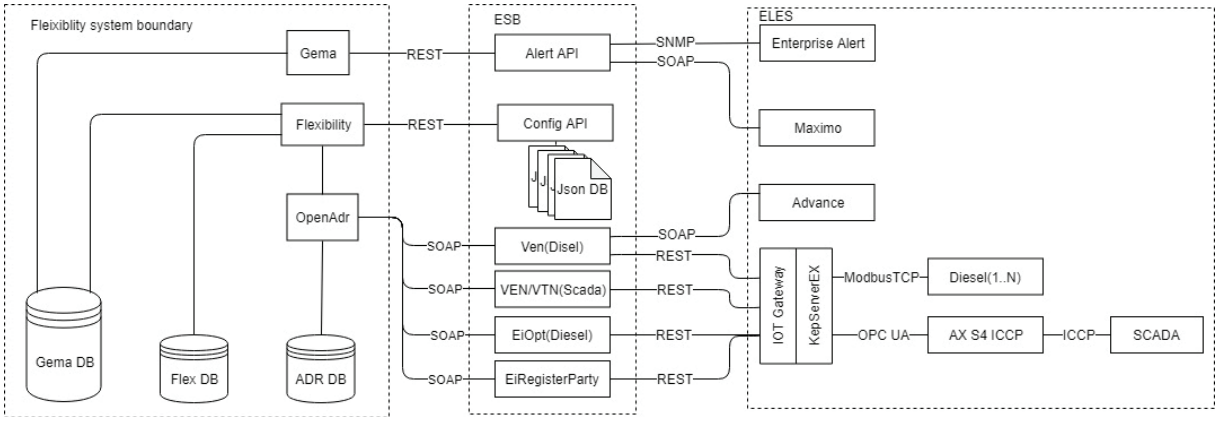


Slika 2: Arhitektura DEA platforme

Modul Flexibility predstavlja samostojni modul oziroma spletno aplikacijo, ki vsebuje funkcionalnosti za upravljanje z razpršenimi viri za zajem obratovalnih podatkov, kot tudi upravljanje le-teh in sestoji iz:

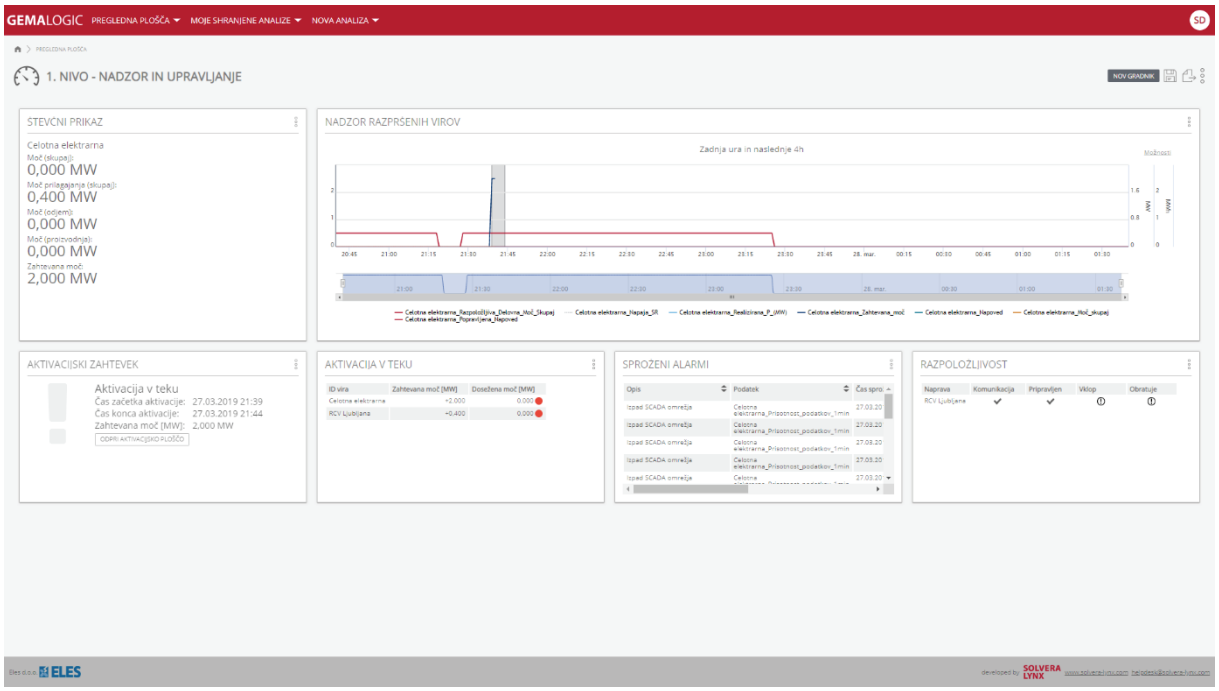
- podatkovnega vodila FlexDB,
- daljinskega upravljanja (ang.: *»Remote Control for power reduction or increasing«*),
- avtomatske zunanje konfiguracije podatkovnih virov posameznega DEA, geolokacij in alarmov,
- avtomatske zunanje konfiguracije uporabniških računov in njihovih pravic za dostop do analiz in orodij DEA platforme,
- pregleda obračuna aktivacij
- pregleda aktivacij v teku
- orodja lastniki,
- pregleda poročilo o aktivacijah,
- pregleda nadzor razpršenih virov,
- orodja razpršeni viri, kjer uporabnik dodaja nove vire in na podlagi katerega se nato izvede avtomatska zunanja konfiguracija in
- podprtih virov:
    - virtualne elektrarne in
    - dizel električnega agregata

Modul OpenADR prav tako predstavlja samostojni modul oziroma spletno aplikacijo, ki omogoča podporo OpenADR protokolu, ki omogoča registracijo t.i. VEN (ang.: *»Virtual End Node«*), kot tudi VTN (ang.: *»Virtual Top Node«*) naprav in izmenjavo MetaData podatkov oz. t.i. *»Reportov«*. OpenADR je zasnovan na podatkovnem vodilu ADRDB. Slika 3 predstavlja arhitekturo modula Flexibility, kot tudi OpenDAR z integracijskimi točkami na strani SOPO ELES.



Slika 3: Arhitektura modula Flexibility

Slika 4 predstavlja primer pregledne plošče in vsebuje pet osnovnih gradnikov, kot so »Števčni prikaz«, »Nadzor razpršenih virov«, »«Aktivacijski zahtevek«, »Aktivacije v teku«, »Sproženi alarmi« in »Razpoložljivost«. Slika 4 predstavljata izgled pregledne plošče v času v času aktivacije.



Slika 4: Primer pregledne plošče v času med aktivacijo

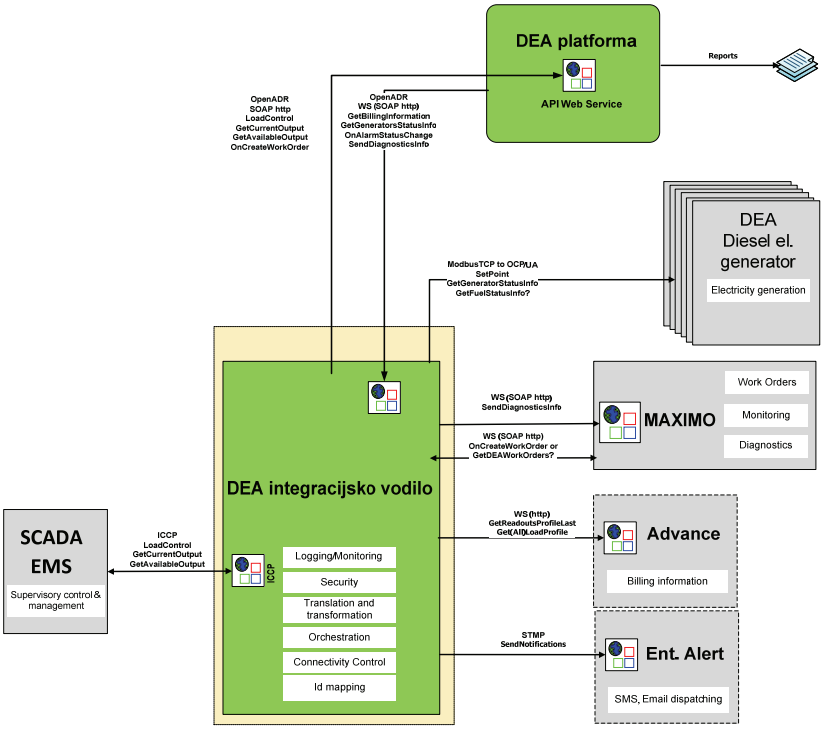
4 VMESNIKI IN INTEGRACIJSKE TOČKE

Višje nivojska slika rešitve z vsemi identificiranimi sistemi/komponentami in njihovimi interakcijami je prikazana na Slika 5. Ker je bilo v fazi analize identificirano precejšnje število integracijskih točk je bilo uporabljeno namensko integracijsko vodilo, katerega logika je implementirana v namenskem integracijskem sloju, ločena od aplikativne in poslovne logike sistemov. Na ta način je bila dosežena višja stopnja neodvisnosti in enkapsulacije posameznih sistemov ter večja fleksibilnost in učinkovitost DEA platforme, kot tudi posledično nižji stroški vzdrževanja, kar je pomembna prednost z integracijami v stilu vsak z vsakim (ang. *»Point to Point Integrations«*).

Na nivoju DEA platforme in DEA integracijskega vodila so bile vzpostavljene naslednje funkcionalnosti, funkcije in integracije:

- podatkovni vmesnik DEA platforma (OpenADR in OASIS):
- omogoča izmenjavo vseh podatkov do DEA integracijskega vodila, ki je nadalje povezan z vsemi navedenimi zalednimi informacijskimi sistemi,

- podatkovni vmesnik AX-S4 for ICCP (ICCP to OPC UA):
- Omogoča izvajanje funkcionalnosti aktivacij ter posredovanja skupnih statusnih informacij DEA platforme,
- Podatkovni vmesnik KepServer EX – DEA (Modbus TCP to WS):
- Omogoča zajem obratovalnih podatkov in posredovanje zahtevkov za aktivacijo, ki zaženejo oziroma zaustavijo DEA za potrebe SR,
- podatkovni vmesnik Advance (WS):
- omogoča zajem merilnih podatkov sistemskega števca, ki meri proizvedeno električno energijo za posamezen razpršen vir,
- podatkovni vmesnik Maximo (WS):
- posredovanje alarmov in merilnih podatkov v sistem za upravljanje z razpršenimi viri,
- podatkovni vmesnik EnterpriseAlert (SNMP TRAP):
- posredovanje alarmov v sistem obveščanja.



Slika 5: Vmesniki in integracijske točke DEA platforme

4.1 Podatkovni vmesnik API DEA platforma (OpenADR).

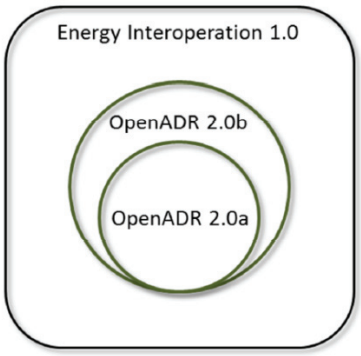
Za potrebe povezovanja DEA integracijskega vodila in DEA platforme se uporabi OpenADR standard. OpenADR je zasnovan za potrebe prejemanja in izvajanja funkcionalnosti prilagajanja odjema ter omogoča enotno izmenjavo podatkov. Namen OpenADR je poenotiti področje izmenjave podatkov na vseh nivojih energetskih dejavnosti, kot so to na eni strani na primer sistemski operaterji ali drugi neodvisni izvajalci energetskih dejavnosti ter na drugi strani končni uporabniki omrežja. Namen OpenADR podatkovnega modela je integracija z najrazličnejšimi informacijskimi sistemi, kot so npr. t.i. BEMS, HEMS, FEMS, xEMS, SCADA, MDMS, itn. Gre za odprti standard, ki vsakomur omogoča sodelovanje pri prilagajanju odjema.

OpenADR je del širšega neprofitnega združenja OASIS, ki omogoča razvoj, zbliževanje in implementacijo odprtokodnih standardov za namen svetovne uporabe. OASIS definira tudi Energy Interoperation[1] standard, ki opisuje informacijski in komunikacijski model za potrebe poenotenja na področju energetike temelječ na OASIS SOA[2] referenčnem modelu in XML modelov za potrebe interoperabilnosti ter standardne izmenjave:

[1] OASIS Energy Interoperation, vir: http://docs.oasis-open.org/energyinterop/ei/v1.0/csd01/energyinterop-v1.0-csd01.html
[2] OASIS SOA, vir: http://docs.oasis-open.org/energyinterop/ei/v1.0/csd01/energyinterop-v1.0-csd01.html

- dinamičnih cenovnih signalov,
- zanesljivostih signalov,
- kritičnih signalov,
- signalov povpraševanja in ponudb,
- napovednih modelov in nadzora razpršenih virov,
  kar še kako poenostavi izmenjave informacij in podatkov na dereguliranem energetskem trgu, ki:
    - omogoča učinkovitejši odziv v primeru pojava kritičnih in zanesljivostih signalov,
- omogoča prevzem izziva zniževanja stroškov za električno energijo s prilagajanjem odjema ali upravljanjem z energijo,
- omogoča trgovanje z energijo,
- podpira direktno povezavo med proizvajalci in končnimi uporabniki omrežja,
- zagotavljajo združevanje naštetih funkcionalnosti prilagajanja ali upravljanja odjema.

Slika 6 prikazuje povezavo med podatkovnim modelom Energy Interoperation and OpenADR.



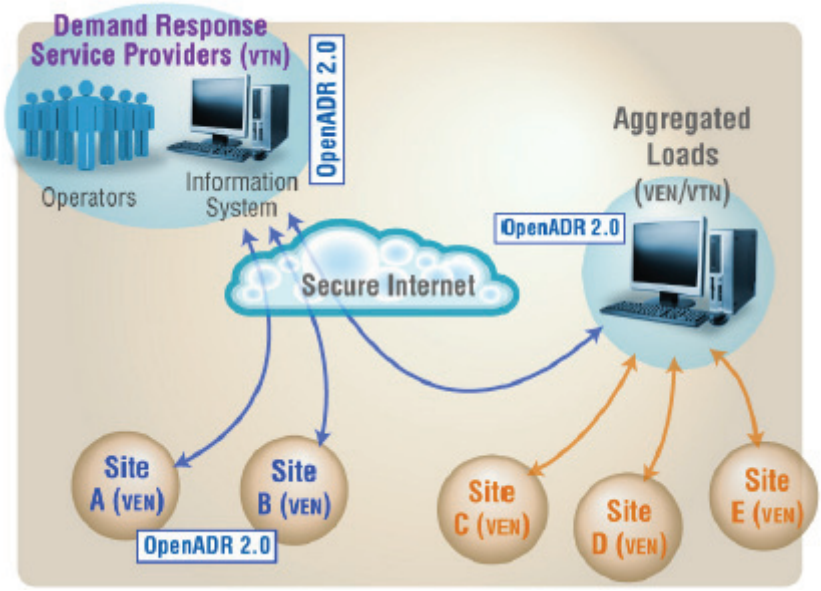
Slika 6: Povezava standardov Energy Interoperation 1.0 in OpenADR

OpenADR definira, da mora biti sistem zasnovan tako, da omogoča funkcionalnosti izvajanja aktivacije v danem trenutku ali skladno z zahtevano časovnim razporedom, registracijo le-teh, načrtovanje dogodkov in posredovanje poročil. Prav tako mora omogočati opravljati spremembe predhodno poslanih informacij.

Slika 7: OASIS SOA model

Področje uporabe interoperabilnosti torej povezuje vse ključne akterje na dereguliranem energetskem trgu, kar kot primer prikazuje Slika 7. Arhitektura omrežja OpenADR predstavlja Slika 8, kjer lahko vidimo, da se pojavljajo tri poimenovanja posameznih akterjev znotraj OpenADR in sicer:

- VEN (anj.: Virtual End Node)
- VTN (anj.: Virtual Top Node) in
- VEN/VTN (anj.: Virtual End Node/Virtual Top Node).



Slika 8: Primer možne uporabe OpenADR

DEA platforma predvideva uporabo OpenADR podatkovnega modela na področju povezave DEA platforme z DEA integracijskim vodilom, kar pomeni, da tako DEA platforma predstavlja kombinacijo VEN/VTN modela, kjer pa mora biti prav tako tudi DEA integracijsko vodilo zasnovano na enak način saj VEN predstavlja šele končni razpršeni vir.

4.2 Podatkovni vmesnik SCADA EMS

Za potrebe povezovanja SCADA EMS z DEA platformo je bil uporabljen klient AX-S4 for ICCP. Klient AX-S4 ICCP omogoča povezovanje s SCADA EMS na strani ELES-a ter izmenjavo podatkov in krmilnih signalov aktivacije SR. Klient AX-S4 ICCP pa se na drugi strani preko OPC UA protokola povezuje na klient KepServerEX, ki omogoča podporo OPC UA protokolu. Vsi navedeni klienti so tako del DEA integracijskega vodila, kjer pa je implementiran OpenADR protokol in se tako le-ta v določenem trenutku obnaša, kot VEN oziroma, kot VTN. Komunikacija poteka dvosmerno. Frekvenca osveževanja podatkov je 1 minuta. Podatkovni vmesnik SCADA EMS vsebuje t.i. *»Outbound«* (izhodne) in *»Inbound«* (vhodne) podatke, ki so naslednji:

- Outbound DEA platforme na nivoju AX-S4 ICCP vmesnika:
    - seštevek trenutne razpoložljive moči SR v MW vseh proizvodnih virov (DEA) za zadnji 1 min interval;
    - ažurirana napoved razpoložljive skupne energije SR v MWh vseh proizvodnih virov (DEA) za 24 ur v naprej na 15 minutni resoluciji;
    - status stanja preklopke za izbiro aktivacije:
        - Vsi razpoložljivi DEA == 1;
        - Želena moč v MW == 0;
    - status stanja preklopke za izbiro trajanja aktivacije:
        - Do preklica == 1;
        - Trajanje v minutah == 0;
    - status stanja aktivacije:
        - v mirovanju;
        - prejeta zahteva za delovanje v SR:
        - ni mogoče poslati zahteve za aktivacijo:
        - DEA se zaganja:
        - Težave pri doseganju polne moči:
        - Aktiven;
        - DEA se zaustavlja;

  - realizirana moč P v MW;
  - status stanja DEA razpoložljiv za SR:
    - DA ==1;
    - NE ==0;
- Inbound DEA plaftorme na nivoju API AX-S4 ICCP vmesnika – ob aktivaciji SR:
  - Komanda za spremembo preklopke za izbiro aktivacije:
    - Vsi razpoložljivi == 1;
    - Želena moč v MW == 0;
  - Komanda (Setpoint) za nastavitev želene moči aktivacije (MW);
  - Komanda za spremembo preklopke za izbiro trajanja aktivacije:
    - Do preklica == 1;
    - Trajanje v minutah == 0;
  - Komanda (Setpoint) čas začetka aktivacije (hh:mm:ss);
  - Komanda (Setpoint) za nastavitev trajanja aktivacije (min);
  - Komanda za zagon in zaustavitev DEA:
    - Aktiviraj SR == 1;
    - Zaustavi SR == 0;

4.3 Podatkovni vmesnik DEA – KepServerEX

Za povezavo DEA v DEA integracijsko vodilo je bil uporabljen klient KepServerEX, ki na nivoju komunikacije z DEA omogoča podporo ModbusTCP protokolu ter na zgornjem nivoju t.i. *»IoT Gateway«*. IoT Gateway omogoča podporo uporabe spletnih storitev (*»WebServices«*) za zunanjo konfiguracijo naprav in izmenjavo podatkov. Funkcionalnost zunanje konfiguracije in integracije klienta KepServerEX je bila implementirana na strani DEA integracijskega vodila, kjer je bil za izmenjavo podatkov z DEA platformo uporabljen OpenADR protokol za zunanjo konfiguracijo naprave pa aplikacijski vmesnik skladno s specifikacijo, ki je že podprt na strani IoT Gateway modula klienta KepServerEX. S strani DEA se tako v DEA platformo prenašajo naslednje informacije oziroma signali:

- daljinsko upravljanje posamezne enote,
- razpoložljivost DEA enote (pripravljena, avtomatsko in brez napak),
- status delovanja DEA enote,
- delovna moč DEA enote (A+, A-),
- jalova moč DEA enote,
- avtonomija obratovanja DEA na prednastavljeni moči v minutah in
- drugi obratovalni podatki, kot so napetosti in tokovi tako na nivoju generatorja, kot tudi omrežja.

Na strani DEA so nameščeni krmilniki DSE tip 8620 ali podobni, kateri imajo prednastavljene modbus registre skladno s koncepti projekta [7]. Komunikacija poteka dvosmerno. Frekvenca osveževanja podatkov je 1 minuta.

4.4 Podatkovni vmesnik HES Advance

Za zajem obračunskih merilnih podatkov je bil uporabljen WS sistema za zajem obračunskih merilnih podatkov HES Advance. Komunikacija z HES Advance poteka enosmerno proti DEA integracijskemu vodilu. Obračunski merilni podatki se v HES Advance zajemajo vsakih 15 minut z 1 minutnim časovnim intervalom. Poleg DEA je vgrajen sistemski števec P2, ki predstavlja obračunsko merilno mesto. DEA integracijsko vodilo uporablja t.i. WS metodo *»GetRedoutProfileLast«*, ki omogoča zajem obračunskih podatkov za preteklo časovno obdobje. DEA integracijsko vodilo ima implementirano rešitev, kjer pridobi informacijo zadnjih zajetih obračunskih merilnih podatkov in tako ob naslednjem klicu pridobi obračunske podatke od zadnjega zajetega podatkov do zadnjega razpoložljivega podatka v HES Advance. Identifikator, ki predstavlja merilno mesto iz katerega želimo pridobiti obračunske merilne podatke predstavlja naziv merilnega mesta in ga uporabnik vnese ob konfiguraciji DEA v DEA platformi.

4.5 Podatkovni vmesnik Maximo

Maximo predstavlja sistem za upravljanje s sredstvi s katerim DEA platforma izmenjuje obratovalne podatke, kot tudi alarme na podlagi katerih Maximo kreira nalog za delo, ki je podlaga za odpravo napake na strani DEA. Komunikacija med DEA platformo in Maximo poteka dvosmerno, kar pomeni, da v primeru, da je bil v Maximo kreiran nalog za delo se le-ta podatek prenese tudi v DEA platformo, kjer DEA pridobi status nerazpoložljivosti oziroma se le-temu do preklica naloga za delo zabeleži višina razpoložljive moči, ki je enaka 0 MW. Za izmenjavo podatkov z Maximo je uporabljen WS.

4.6 Podatkovni vmesnik Enterprise Alert

V primeru nastanka alarma DEA platforma posreduje podatke v sistem za alarmiranje, ki se nahaja na strani SOPO ELES – Enterprise Alert. Na nivoju DEA integracijskega vodila je implementiran sporočilni strežnik preko katerega DEA platforma pošilja alarme z uporabo metode SNMP TRAP. V Enterprise Alert se tako ob nastopu alarma pošljejo podatki o identifikaciji razpršenega vira, kjer je prišlo do alarma, tip alarma in opis, čas nastopa alarma ter odgovorno osebo. Enterprise Alert skrbi, da se sporočilo, ki je bilo poslano s strani DEA platforme pošlje na nastavljeno telefonsko številko odgovorne osebe, kot SMS sporočilo. Parametre, kot so telefonska številka, odgovorna oseba in alarme lahko nastavi skrbnik DEA platforme.

5 PILOTNI PRIMER UPORABE AKTIVACIJE SR

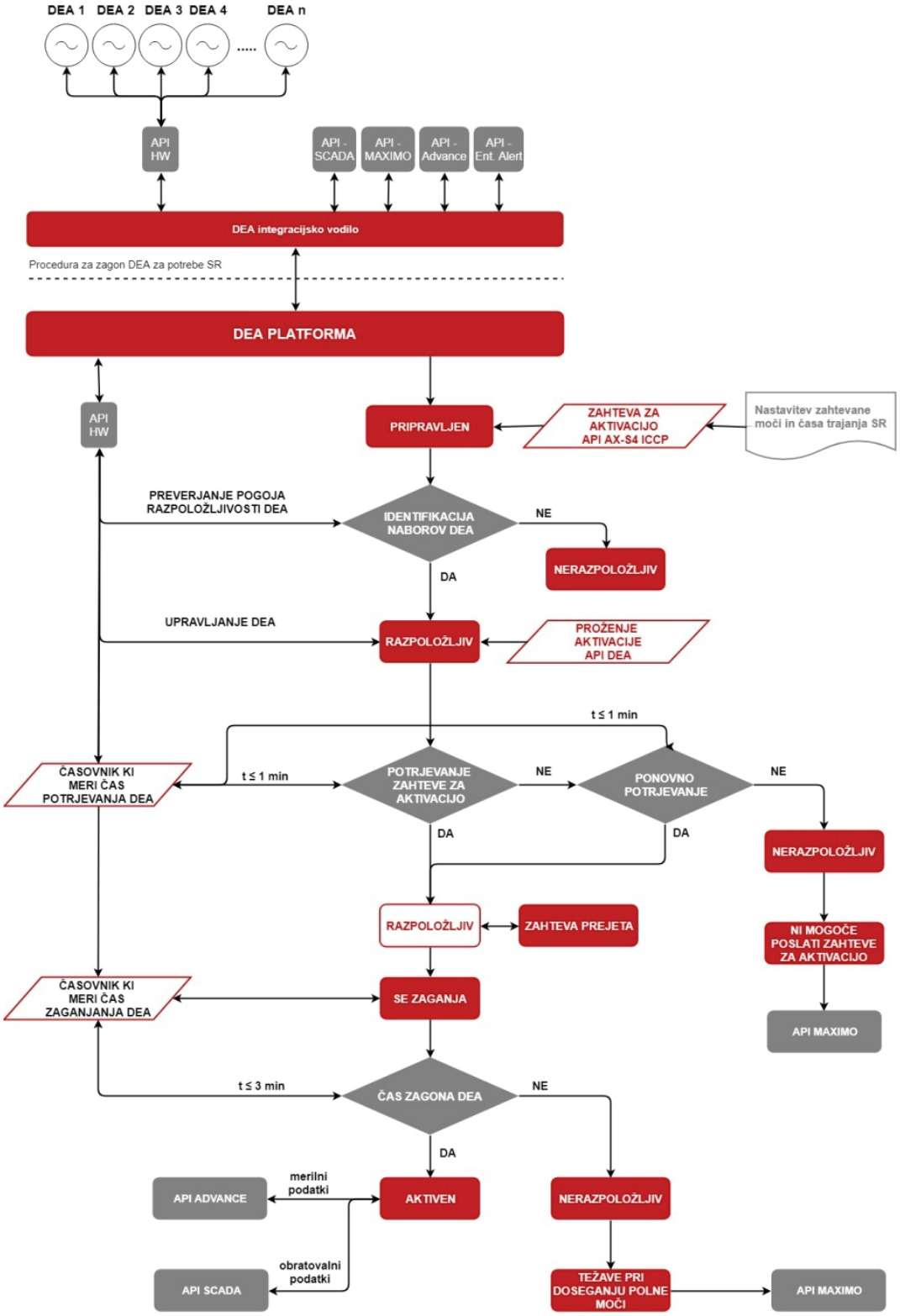
Testiranje DEA platforme je potekalo v dveh nivojih in sicer na testiranje v simulacijskem okolju s strani podjetja Solvera Lynx in v realnem okolju, kot pilotni primer skupaj s predstavniki podjetja Solvera Lynx, Bintegra in ELES. Pilotni primer se je izvajal na strani podjetja ELES z vključevanjem DEA na lokaciji Hajdrihove ulice v Ljubljani. Celotna procedura testiranja in pilotnega primera v realnem okolju je potekala skladno z opisom v nadaljevanju in, kot to prikazuje blokovni diagram na Slika 9 za primer aktivacije in na Slika 10.

Zahtevo za aktivacijo SR izvede operater v republiškem centru vodenja ELES (v nadaljevanju operater), kjer je bila v SCADA EMS ustvarjena dodatna shema, ki prikazuje obratovalne podatke, stanja preklopk, podatke o razpoložljivi moči, energiji in avtonomiji, ki operaterju nudi možnost aktivacije s pritiskom na gumb aktiviraj SR. Operater vnese zahtevane parametre v SCADA EMS in izvede komando za zagon DEA, kjer nato sledi:

- DEA platforma prejme zahtevo za aktivacijo, kjer nato optimizacijski model avtomatsko izvede postopek identifikacije možnih naborov DEA med vsemi tistimi, ki imajo status aktiven izhod na krmilniku **»DEA razpoložljiv za SR«**
- Sproži se aktivacija razpoložljivih virov od katerih se nato čaka potrditev izhoda krmilnika **»Prejeta zahteva za delovanje v SR«.**
- Če je potrditev zahteve po pretečenem času 1 minute uspešna DEA platforma nadaljuje z zagonom DEA, sicer se prekine. Vsem tistim, ki niso potrdili zahteve za delovanje v SR se aktivacija ponovno posreduje. V kolikor tudi tokrat zahteva ni bila potrjena po pretečenem času 1 minute DEA spremeni status razpoložljivosti iz **»DEA razpoložljiv za SR«** v **»Nerazpoložljiv«**, kjer statusu pripišemo vzrok **»Ni mogoče poslati zahteve za aktivacijo«**. Skrbniku agregata se posreduje obvestilo o nerazpoložljivosti agregata.
- Tistim DEA, ki so potrdili zahtevo za aktivacijo:
    - se status **»DEA razpoložljiv za SR«** spremeni v status **»DEA se zaganja«**, kar preverimo z aktivnim statusom na izhodu krmilnika **»DEA se zaganja«**,
    - začne teči **»Časovnik«**, ki meri čas do zagona na polni moči,
    - če v času 3 minut DEA ne doseže polne moči se status **»DEA se zaganja«** spremeni v status **»Težave pri doseganju polne moči«**,
    - DEA, ki dosežejo polno moč se jim status **»DEA se zaganja«** spremeni v status **»Aktiven«** s pripisano trenutno močjo DEA glede na nazivno moč in pripadajočimi obratovalnimi podatki.
- Opazuje se potek aktivacije in ustreznost odziva proizvedene moči in energije na nivoju celotne DEA platforme na minutni časovni resoluciji. DEA platforma sproti podaja informacijo o ustreznosti proizvedene

moči, zato obstaja možnost popolnega pregleda nad trenutno situacijo, napovedjo proizvedene moči in pričakovano proizvedeno močjo.

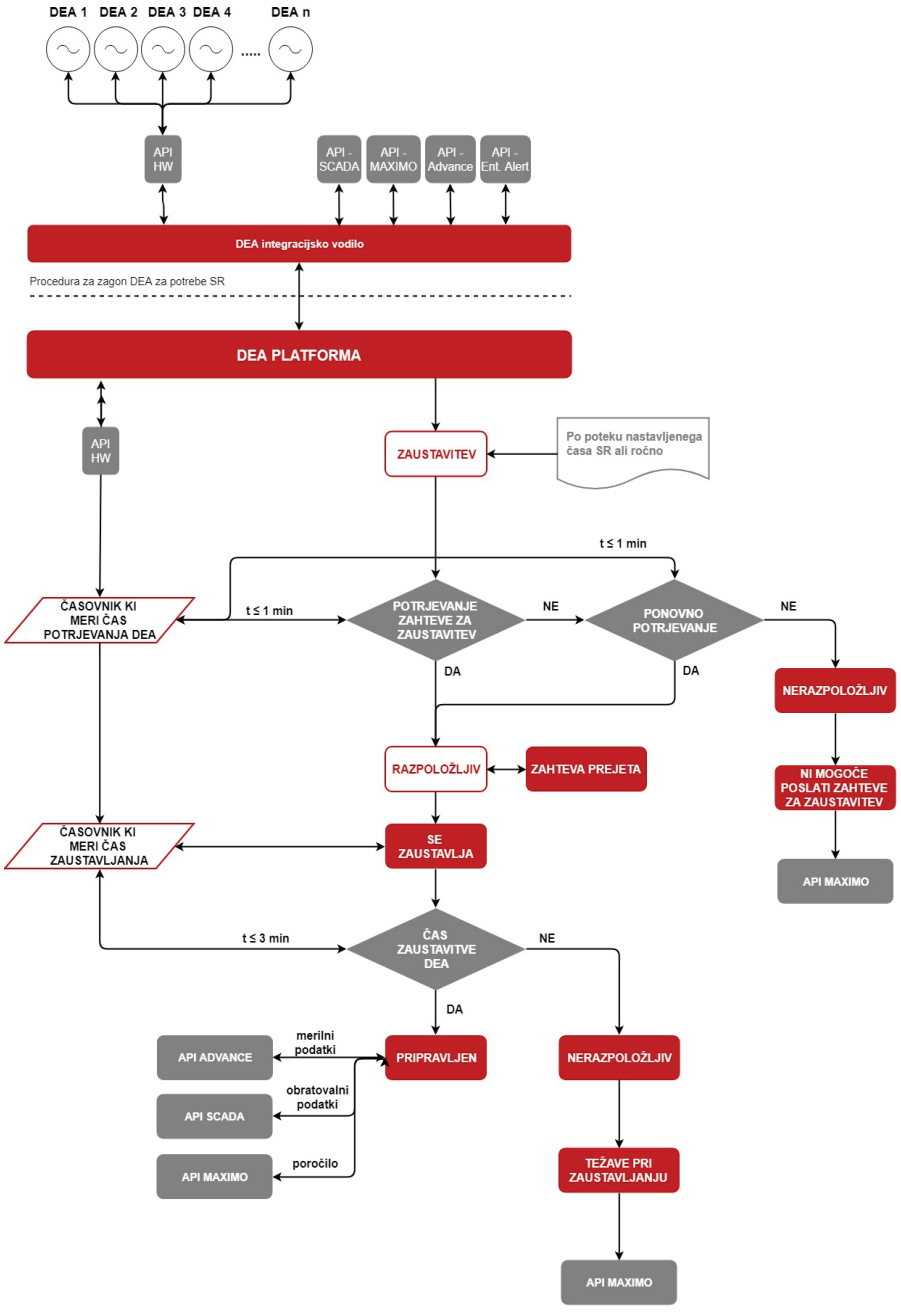
- Po končani aktivaciji je na voljo takojšnjo informacijo o tem, ali je bila aktivacija uspešna ali ne, in kakšen je bil odziv posameznih DEA do nivoja posameznih razpršenih virov.



Slika 9: Proces aktivacije DEA

V DEA platformi so na voljo tudi zgodovinski pregledi vseh aktivacij, do nivoja odziva posameznih razpršenih virov. Na ta način je zagotovljena transparentnost in sledljivost delovanja DEA, vključno z analizo delovanja in odziva DEA. Zahteva za zaustavitev SR se lahko izvede »**Avtomatsko**« po poteku časa trajanja ali pa »**Ročno**« v trenutku, ko operater izvede »**Zaustavitev SR**«. Le-ta se izvede po naslednjem postopku:

- DEA platforma prejme zahtevo za zaustavitev SR,
- Vsem »**Aktivnim**« DEA, DEA platforma pošlje komando za »**Zaustavitev**« in počakamo na potrditev.
- Za tiste DEA, ki so potrdili prejem komande, naredimo naslednje:
  - spremeni se jim trenutni status iz »**Aktiven**« v »**DEA se zaustavlja**«,
  - začne teči časovnik, ki meri čas do zaustavitve DEA, ki je implementiran na krmilniku,
  - če se DEA v predvidenem času ne zaustavi, dobi status »**Težave pri zaustavitvi**«,
  - po zaustavitvi se preveri status na izhodu krmilnika »**DEA razpoložljiv za SR**«, ki je sedaj zopet aktiven. DEA je tako razpoložljiv za ponovno aktivacijo SR.



Slika 10: Proces zaustavitve

6 ZAKLJUČEK

Tekom projekta je bila razvita DEA platforma, ki podaja integracijo zalednih informacijskih sistemov z izrabo spletnih storitev *»WebServices«* in namenskih protokolov, kot je OpenADR protokol [22], ki jasno definira protokol komunikacije na nivoju spodnjega in zgornjega nivoja. Prav tako je bil uporabljen nov pristop implementacije programskih rešitev prilagajanja odjema, kjer prav zaradi modularne zasnove brez velikih sprememb izvedemo integracijo z zalednimi informacijskimi sistemi. Nov pristop predstavlja tudi implementirano orodje za konfiguracijo razpršenih virov DEA platformi, ki z vnosom zgolj najbolj pomembnih podatkov s strani uporabnika, samodejno kreira nov razpršeni vir, ob tem pa avtomatiziran proces poskrbi za izgradnjo povezav od razpršenega vira do DEA platforme. Avtomatiziran proces poskrbi, da je ob kreiranju razpršenega vira le-ta enostavno vključen v izbrano konfiguracijo združenih virov in se tako lahko v nekaj minutah že uporabi v modelu nudenja podpore rRPF.

DEA platforma predstavlja rešitev za potrebe zagotavljanja SR ročne rezerve za povrnitev frekvence, kot podpora enoti za vodenje odjema in razpršene proizvodnje, ki lahko na zahtevo zagotovi vklop vključenih DEA. Razpršeni DEA se tako obnašajo, kot enoten, združen vir in zagotavlja 100% zahtevano moč najkasneje v petnajstih (15) minutah od sprožene aktivacije. Zakasnilni čas stikalne manipulacije paralelnega vključevanja DEA v omrežje predstavlja tri (3) minute, kar pomeni da so DEA lahko na nazivni moči v času treh (3) minut.

Zagotovljeno je, da napajanje NLR v nobenem primeru ni ogroženo, torej niti v času izvajanja del niti v rednem obratovanju (razen vzdrževanja). Števčni podatki so zajeti in obdelani na način, da je mogoče izvesti obračun oziroma povezava proizvedene električne energije za nabavo goriva in uveljavljanje vračila trošarine. Omogočeno je tudi zbiranje in hramba podatkov DEA, njihova obdelava in posredovanje v nadaljnjo uporabo.

DEA platforma je zasnovana tako, da se lahko v prihodnje izvede nadgradnja algoritma za delitev moči po posameznih DEA, ki bo omogočala tudi prilagajanje zahtevane moči posameznega DEA skladno z naknadno definirano specifikacijo naročnika in bo vplivala na spremembo optimizacijskega algoritma.

DEA platforma zagotavlja naslednje mehanizme:

- vodenja,
- preklopne avtomatike,
- beleženja podatkov za analizo in diagnostiko DEA:
    - nazivni podatki,
    - obratovalni podatki motorja,
    - stanje goriva,
    - obratovalni podatki generatorja,
    - obratovalni podatki ločilnega mesta,
- zajema obratovalnih meritev in
- informacij o rednih in izrednih vzdrževalnih posegih.

Razvit je bil tudi standardni model vključevanja razpršenih virov, kjer je bil le-ta trenutno uporabljen za potrebe DEA v prihodnje pa bi ga lahko enostavno prenesli tudi na druge tipe razpršenih virov. Z vzpostavljenimi povezavami DEA platforme z zalednimi informacijskimi sistemi in povezovalnimi platformami, DEA platforma predstavlja možnost za enostavno nadgradnjo z dodatnimi tipi razpršenih virov, kot so na primer baterijski sistemi, sončne elektrarne, vetrne elektrarne, male hidro elektrarne in drugi razpršeni viri električne energije, kot tudi nadgradnjo nudenju podpore sistemskih storitev avtomatske rezerve za povrnitev frekvence (aRPF).

7 LITERATURA

[1] Spletna stran SODO, »Sistemska obratovalna navodila distribucijskega omrežja«. Dosegljivo: http://www.sodo.si/_files/361/SONDO%202011.pdf. [Obiskano: 25. 4. 2018]

[2] Spletna stran SODO, »Priloga 5 – Navodila za priključevanje in obratovanje elektrarn inštalirane električne moči do 10MW«.

[3] Dosegljivo: https://www.sodo.si/_files/366/SONDO%202011%20Priloga%205.pdf [Obiskano: 25. 4. 2018]

[4] Sistemska obratovalna navodila za prenosno omrežje električne energije, Uradni list RS, št. 49/2007 z dne 4. 6. 2007.

[5] Študija številka 2250, »Koncept vključitve dizel električnih agregatov v sistemsko rezervo«, Elektro inštitut Milan Vidmar, Ljubljana, april 2015.

[6] Lawrence Berkeley National Laboratory Akuacom, »Open automated demand response communications specification (version 1.0)«, California, April 2009.

[7] Dosegljivo: https://drrc.lbl.gov/sites/default/files/cec-500-2009-063.pdf. [Obiskano: 23. 5. 2018].

[8] OASIS Committe, »Energy Market Information Exchange (EMIX) Version 1.0«, Januar 2012. Dosegljivo: http://docs.oasis-open.org/emix/emix/v1.0/emix-v1.0.html. [Obiskano: 23. 5. 2018].

[9] OASIS Committe, »Energy Interoperation Version 1.0«, November 2010. Dosegljivo: http://docs.oasis-open.org/energyinterop/ei/v1.0/csd01/energyinterop-v1.0-csd01.html. [Obiskano: 23. 5. 2018].

[10] OASIS Committe, »Reference Architecture Foundation for Service Oriented Architecture Version 1.0«, December 2012. Dosegljivo: http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/cs01/soa-ra-v1.0-cs01.html. [Obiskano: 23. 5. 2018].

[11] UCAIug OpenSG OpenADR Task Force OpenSG, »OpenADR 1.0 System Requirements Specification«, september 2010 . Dosegljivo: http://osgug.ucaiug.org/sgsystems/OpenADR/Shared%20Documents/SRS/OpenSG%20OpenADR%201.0%20SRS%20v1.0.pdf. [Obiskano: 23.5.2018].

[12] OASIS Commite, »WS-Calendar Version 1.0«, julij 2011. Dosegljivo: http://docs.oasis-open.org/ws-calendar/ws-calendar-spec/v1.0/cs01/ws-calendar-spec-v1.0-cs01.html. [Obiskano: 23.5.2018].

[13] openADR Alliance, »OpenADR 2.0a Profile Specification v 1.1«, december 2011. Dosegljivo: OpenADR 2 0a Profile Specification_v1.0. [Obiskano: 23.5.2018].

OTS 2019
SODOBNE INFORMACIJSKE TEHNOLOGIJE IN STORITVE
M. Heričko in K. Kous (ur.)

Univerzitetna založba
Univerze v Mariboru

»POLITIKA« KIBERNETSKE VARNOSTI

BOŠTJAN KEŽMAH

Povzetek: *Predpisi na področju kibernetske varnosti in elektronskega poslovanja se množijo tako na ravni EU kot na ravni posameznih držav članic. Reguliranje je z vidika varstva potrošnikov in posameznikov pomembno, hkrati pa lahko vodi do neenake obravnave ponudnikov storitev informacijske družbe v posameznih državah članicah. Kako lahko dodatne omejitve v posameznih državah članicah vplivajo na nadaljnjo digitalno transformacijo predstavljamo na primeru ureditve elektronskega zajema lastnoročnega podpisa v Sloveniji. Podobno kot omejitve v predpisih lahko tudi organi, ki imajo posebne pristojnosti po posameznih predpisih, s svojim razumevanjem predpisov bistveno vplivajo na nadaljnjo informatizacijo postopkov, kar predstavljamo na primeru ponudbe elektronskih posojil.*

Ključne besede: *kibernetska varnost, elektronsko poslovanje, GDPR, SISBON, ZVOP-1*

NASLOV AVTORJA: dr. Boštjan Kežmah, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, 2000 Maribor, Slovenija, e-pošta: bostjan.kezmah@um.si.

https://doi.org/10.18690/978-961-286-282-4.11 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 UVOD

Kibernetska varnost se kot podmnožica informacijske varnosti neločljivo prepleta z zagotavljanjem skladnosti s predpisi. Zaradi vse večjega pomena informacijske tehnologije v poslovnih procesih tako EU kot države članice sprejemajo vse več predpisov, ki se nanašajo neposredno na kibernetsko varnost.

Med zadnjimi pomembnejšimi predpisi sta Zakon o informacijski varnosti in Splošna uredba o varstvu podatkov.

Novi predpisi lahko predstavljajo tudi priložnost za optimizacije in nadaljnjo digitalno transformacijo poslovanja, če le niso preveč omejujoči.

V nadaljevanju bomo analizirali dva primera predpisov na področju elektronskega poslovanja, pri katerih je bistveno omejena nadaljnja digitalna transformacija podjetij.

2 ELEKTRONSKI ZAJEM LASTNOROČNEGA PODPISA

Vse več podjetij poskuša z digitalno transformacijo uvesti brezpapirno poslovanje. Ker je poslovanje podjetja neločljivo povezano tudi s strankami, mora digitalna transformacija doseči tudi postopke, ki so povezani s poslovanjem s strankami.

Eden pomembnejših procesov v poslovanju je sklenitev pogodbe s stranko. Pri poslovanju med podjetji je to praviloma enostavno, ker je večina podjetij že zaradi poslovanja s Finančno upravo in zaradi izdaje elektronskih računov opremljena vsaj z enim kvalificiranim digitalnim potrdilom.

Zaplete se pri poslovanju s fizičnimi osebami. Fizične osebe zaradi preteklih odločitev države niso sistematično opremljene s tehnologijo, ki bi omogočala enostavno elektronsko poslovanje na daljavo oziroma ponujanje storitev informacijske družbe, katerih sestavni del je zanesljiva identifikacija stranke na daljavo ali elektronsko podpisovanje na daljavo s tehnologijo in postopkom, po katerem je elektronski podpis enakovreden lastnoročnemu podpisu. Še posebej pri pogodbah večje vrednosti praviloma podjetja ne želijo sprejeti tveganja, da bodo morala naknadno na sodišču dokazovati, da je elektronski podpis enakovreden lastnoročnemu, zato stremijo k uporabi tehnologije, ki je kot lastnoročni podpis varovana z zakonom.

Država trenutno še ne izdaja osebnih izkaznic, ki bi temeljile na pametni kartici in imele v ta namen nameščeno digitalno potrdilo. Vsak državljan ima sicer kartico zdravstvenega zavarovanja, ki ima vgrajene vse potrebne tehnologije, vendar izdajatelj digitalnih potrdil ni opravil postopkov preverjanja, ki bi omogočali vpis overitelja v register ponudnikov storitev zaupanja po Uredbi eIDAS. To javno navaja tudi Zavod za zdravstveno zavarovanje kot izdajatelj kartic in digitalnih potrdil, ki se opredeljuje kot »zaprt sistem zasebne infrastrukture javnih ključev (ZZZS-PKI) za potrebe delovanja sistema kartice zdravstvenega zavarovanja« [1].

2.1 Elektronski podpis ob osebni navzočnosti stranke

Posedovanje tehnologij in podatkov, ki so potrebni za elektronsko identifikacijo in elektronsko podpisovanje na daljavo je pomembno zato, ker je te tehnologije mogoče uporabiti tudi za identifikacijo in elektronsko podpisovanje ob fizični navzočnosti stranke.

Kadar ustrezna tehnologija stranki v času fizičnega obiska ponudnika ni dostopna, lahko stranka podpiše pogodbo le z lastnoročnim podpisom. Če se želi ponudnik izogniti papirnemu poslovanju, mora lastnoročni podpis zajeti na elektronski način, pri tem pa lahko le zajame sliko lastnoročnega podpisa ali pa zbere tudi spremljajoče biometrične podatke lastnoročnega podpisa.

Splošna uredba o varstvu podatkov (v nadaljevanju GDPR) v 1. odst. 9. člena prepoveduje uporabo biometričnih podatkov za namen edinstvene identifikacije posameznika, vendar predvideva izjeme po 2. odst. istega člena [2]. Tudi, če bi bilo po GDPR dovoljeno ob lastnoročnem podpisu zajeti biometrične podatke, tega ne dopusti Zakon o varstvu osebnih podatkov (v nadaljevanju ZVOP-1), ki v 1. odst. 80. člena v zasebnem sektorju omejuje biometrijske ukrepe tako, da jih lahko podjetje kot upravljavec izvaja le nad svojimi zaposlenimi [3].

S tem je bistveno zmanjšana dokazna vrednost zajetega lastnoročnega podpisa, saj je treba poiskati druge rešitve, ki zagotavljajo enolično povezavo podpisanega dokumenta s posameznim elektronsko zajetim

lastnoročnim podpisom. Trenutno ne poznamo rešitve z zanesljivim zagotovilom podpisniku, da tako zajetega lastnoročnega podpisa ni mogoče preseliti oziroma prekopirati na drugi dokument. S tem pa ni izpolnjen eden od bistvenih pogojev za varen elektronski podpis, kot ga določa Zakon o elektronskem poslovanju in elektronskem podpisu (v nadaljevanju ZEPEP) in sicer da je elektronski podpis povezan s podatki, na katere se nanaša, tako da je opazna vsaka kasnejša sprememba teh podatkov ali povezave z njimi [4]. Ker je po 15. členu ZEPEP le varen elektronski podpis, overjen s kvalificiranim potrdilom po zakonu enakovreden lastnoročnemu podpisu, s tako obliko poslovanja podjetje prevzema tveganje dokazovanja avtentičnosti lastnoročnega podpisa v povezavi s podpisanim dokumentom.

Ker mnoga podjetja niso pripravljena prevzeti tveganja dokazovanja, to bistveno omejuje nadaljnjo digitalno transformacijo podjetij pri poslovanju s fizičnimi osebami.

Skladno z 2. odst. 1. člena ZEPEP se sicer lahko stranke dogovorijo za drugačne pogoje elektronskega poslovanja, vendar samo v primerih, ko gre za znano število pogodbenih strank, torej v zaprtih sistemih [4]. Povedano drugače, drugačna pravila elektronskega poslovanja lahko vpeljemo šele po tem, ko smo s stranko že sklenili pogodbo, ki pa je po teh pogojih ne moremo skleniti v elektronski obliki.

Kadar se podjetje odloči za lastno rešitev elektronskega podpisovanja, je bistveno, da pri tem upošteva določbe ZEPEP glede definicije varnega elektronskega podpisa in izpolni vse s tem povezane pogoje, saj bo tako lažje dokazovalo enakovrednost zajetega elektronskega podpisa lastnoročnemu.

Ob tem izpostavljamo nenavadno strogo omejitev uporabe biometrije v elektronskih postopkih, saj nenazadnje vsako podjetje, ki sklene pogodbo v fizični obliki, pravzaprav zajema in shranjuje biometrične podatke o lastnoročnem podpisu (iz deformacije papirja lahko strokovnjaki grafologi razberejo množico biometričnih podatkov in na tej podlagi ugotavljajo avtentičnost lastnoročnih podpisov), vendar v analogni, fizični obliki.

3 SAMODEJNO DELOVANJE INFORMACIJSKEGA SISTEMA

Področje zajema lastnoročnega podpisa ni edino regulirano področje na katerem lahko opazimo, da zakonodajalec meni, da so ročni, fizično izvedeni postopki bolj varni od enakovrednih elektronskih.

Oglejmo si to na primeru bank.

3.1 Samodejno preverjanje podatkov v SISBON

Temeljni predpis, ki določa pogoje za dajanje posojil je Zakon o potrošniških kreditih (v nadaljevanju ZPotK-2) [5]. Banka je dolžna preveriti kreditno sposobnost posameznika, zato mora pridobiti podatke iz Centralnega kreditnega registra skladno z Zakonom o centralnem kreditnem registru (ZCKR) [6] in Pravili sistema izmenjave informacij o zadolženosti fizičnih oseb – SISBON [7].

Skladno s 4. odstavkom 19. člena ZCKR lahko do podatkov v sistemu izmenjave informacij pri članih sistema in vključenih dajalcih kreditov dostopajo le osebe, ki so pooblaščene za dostop do zaupnih podatkov zaradi izvajanja nalog pri ocenjevanju kreditnih tveganj člana sistema oziroma vključenega dajalca kreditov v zvezi s sklepanjem ali izvajanjem kreditnih poslov [6].

V 25. tč. 2. odst. ZCKR so opredeljeni »upravljavci drugih zbirk podatkov«, ki so »državni organi, nosilci javnih pooblastil ali druge pooblaščene osebe« [6]. Že iz definicije »upravljavcev drugih zbirk podatkov« sledi, da pojem »pooblaščene osebe« po ZCKR nima univerzalnega pomena »osebe v fizični obliki« in torej ne pomeni nujno ne fizične osebe ne »osebe v fizični obliki«, temveč se lahko nanaša tudi na drugo vrsto oseb, nenazadnje tudi pravno osebo.

Glede na to, da tudi ZVOP-1 v 3. tč. 1. odst. 6. člena določa, da je »obdelava osebnih podatkov delovanje ali niz delovanj, ki so avtomatizirano obdelani« [3], tudi po ZVOP-1 ni podlage za razlago, da se »pooblaščena oseba« nanaša izključno na »osebo v fizični obliki«, saj je obdelava osebnih podatkov dopustna tako na ročni kot avtomatizirani način.

Ne glede na to je Informacijski pooblaščenec 2. 12. 2016 v zadevi številka 0712-1/2016/2449 zavzel odklonilno stališče glede samodejnega povezovanja informacijskega sistema banke z informacijskim sistemom SISBON in sicer iz razloga, ker gre v tem primeru za »povezovanje zbirk podatkov« in ker v primeru, če do sistema SISBON dostopa informacijski sistem banke slednji ne predstavlja »pooblaščene osebe«.

1. odst. 4. člena ZCKR določa, da je SISBON »elektronski sistem za izmenjavo podatkov in informacij« [6]. ZEPEP v 1. točki 2. člena določa, da je elektronsko sporočilo niz podatkov, ki so poslani ali prejeti na elektronski način, kar vključuje predvsem elektronsko izmenjavo podatkov in elektronsko pošto [4]. Ker je SISBON skladno z ZCKR, kot je navedeno zgoraj, »elektronski sistem za izmenjavo podatkov«, gre za sistem izmenjave sporočil, za katerega se uporablja ZEPEP.

Vpogled v SISBON se tehnično izvaja tako, da »pooblaščena oseba«, ki izvaja vpogled, pošlje elektronsko sporočilo informacijskemu sistemu SISBON, ki vrne zahtevane podatke. Pri tem gre torej tako glede na pravne podlage kot glede na dejansko tehnično izvedbo za elektronsko izmenjavo elektronskih sporočil po ZEPEP.

Skladno s 6. točko 1. odst. 2. člena ZEPEP je pošiljatelj elektronskega sporočila oseba, ki je sama poslala elektronsko sporočilo ali pa je bilo sporočilo poslano v njenem imenu in v skladu z njeno voljo; posrednik elektronskega sporočila se ne šteje za pošiljatelja tega elektronskega sporočila [4].

Skladno s 3. alinejo 1. odst. 5. člena ZEPEP velja, da elektronsko sporočilo izvira od pošiljatelja, če ga pošlje informacijski sistem, ki ga upravlja pošiljatelj sam, ali kdo drug po njegovem nalogu, da deluje samodejno [4].

Glede na določbe ZEPEP je torej pošiljatelj elektronskega sporočila »oseba« tudi v primeru, če ga pošlje informacijski sistem, ki ga upravlja pošiljatelj sam, ali kdo drug po njegovem nalogu, da deluje samodejno.

Skladno s predpisi na področju elektronskega poslovanja zato ni nobene pravne podlage, da v imenu »osebe v fizični obliki« sporočil ne more samodejno v njenem imenu pošiljati informacijski sistem, zato tudi ni podlage, da bi določbe ZCKR razlagali drugače, kot določa specialni predpis na področju elektronskega poslovanja, saj ZCKR ni niti izključil elektronskega poslovanja, niti določil svojih, posebnih pravil za področje elektronskega poslovanja.

Pravila v 2. odst. 8. člena dopuščajo uporabo »strežniškega potrdila« za uporabo aplikacijskega vmesnika [7], kar pomeni, da je že upravljavec v svojih navodilih predvidel neposredno tehnično povezovanje med lastnimi aplikacijami uporabnika in sistemom SISBON (in ne samo ročni vpogled z uporabo spletnega vmesnika). Pri tem se mora ne glede na to, da poizvedbo izvaja aplikacija, pri poizvedbi upoštevati enoznačni način identificiranja pooblaščene osebe v sistemu SISBON (ID pooblaščene osebe), kar je nujno za vodenje revizijske sledi na način, kot to določa 6. odst. 8. člena [7], saj sicer ne bi bilo mogoče ugotoviti kdo je bil uporabnik, ki je obdeloval osebne podatke.

Kljub temu, da 8. člen Pravil izrecno dopušča neposredno izmenjavo podatkov med aplikacijami [7], nikjer ne omejuje uporabe izključno na »osebe v fizični obliki« ali na proženje poizvedb, ki se izvajajo izključno ročno. Nasprotno, uporaba strežniškega potrdila izrecno predvideva pošiljanje poizvedb tudi drugim aplikacijam (ne samo »osebam v fizični obliki«), ker se pravila nanašajo na vmesnike med aplikacijami pa je prepuščeno lastniku aplikacije, da določi način delovanja svoje aplikacije, ki pošilja poizvedbe v SISBON, ki bi bil samo kot primer lahko izveden z ročnim ali samodejnim proženjem, s takojšnjim poizvedovanjem ali z zbiranjem večjega števila zahtev in naknadnim pošiljanjem poizvedb, vse odvisno od zahtev poslovnih procesov in informacijskega sistema, v katerem se SISBON uporablja.

V kolikor je dovoljeno le ročno pošiljanje poizvedb s strani »osebe v fizični obliki«, potem je vprašljiva tudi zakonitost obstoječe ureditve aplikacijskega dostopa v Pravilih sistema izmenjave informacij o zadolženosti fizičnih oseb – SISBON.

Iz zadnjih sprememb Pravil sistema izmenjave informacij o zadolženosti fizičnih oseb – SISBON je prav tako razvidno, da ni namen omejevati uporabe SISBON izključno na fizično obliko poslovanja. Skladno s 3. odst. 18. člena je upravljavec SISBON predvidel, da lahko dajalec kreditov potrdi istovetnost posameznika tudi brez osebne navzočnosti, »če je posameznik izkazal interes za sklenitev kreditnega posla z uporabo sredstev varne elektronske komunikacije, ki so dogovorjena s članom ali vključenim dajalcem kreditov in vključujejo osebne varnostne elemente visoke ravni zanesljivosti, ki jih za namene enolične identifikacije posamezniku dodeli član ali vključeni dajalec kreditov (kot kombinacija črk, številk ali znakov) ali drug izdajatelj sredstev elektronske identifikacije s sedežem v Republiki Sloveniji v skladu s predpisi, ki urejajo elektronsko identifikacijo« [7].

Glede na to, da je torej mogoče elektronsko preveriti istovetnost posameznika na primer z uporabo sredstev avtentikacije, ki so vgrajena v elektronsko banko, ni smiselna omejitev, da bi morala fizična oseba, pooblaščena za dostop do SISBON, ročno sprožiti postopek pridobivanja podatkov iz SISBON. To bi namreč pomenilo, da posameznik sicer lahko v nekaterih primerih izkaže interes za sklenitev kreditnega posla v celoti

na elektronski način, dajalec kreditov pa mora kljub temu posel obvezno zaključiti ročno zaradi proženja »ročne« poizvedbe v SISBON, to pa onemogoča takojšnje preverjanje izpolnjevanja pogojev za odobritev kredita. Kadar ima banka jasno določene kriterije za odobritev, ki jih je mogoče preveriti elektronsko in je postopek v banki že v celoti informatiziran, ročno proženje ne predstavlja nobene dodane vrednosti, saj dodaja le »ročen klik na gumb« v informacijskem sistemu, ki pa še vedno celoten postopek opravi samodejno.

4 SKLEP

Kljub dobrim namenom zakonodajalcev se lahko praktična uporaba predpisov, ki se nanašajo tudi na kibernetsko varnost, izrodi. V ostrih pogojih konkurence je še posebej pomembno, da organi, ki imajo pristojnosti v zvezi s predpisi, spoznajo, da so podjetja izpostavljena globalnemu trgu tudi, če tega ne želijo.

Primer na področju bančništva je banka N26, ki deluje v celoti elektronsko. Slovenska podjetja, v tem primeru banke, bodo lahko zagotovila svojo konkurenčnost le, če bodo lahko delovala v enakih ali vsaj primerljivih pogojih, kot veljajo za druge ponudnike. Zato je v sklopu sprejemanja domačih predpisov in njihove razlage ključno, da se ozremo po tem kakšne dodatne omejitve sprejemajo druge članice EU, saj lahko le v tem primeru govorimo o skupnem digitalnem trgu EU, ki razen prostega pretoka storitev zagotavlja enake pogoje poslovanja tudi ponudnikom storitev informacijske družbe.

5 LITERATURA

[1] https://partner.zzzs.si/wps/portal/portali/aizv/e-poslovanje/kartice_in_citalniki/overitelj_digitalnih_potrdil_zzzs_ca/, Overitelj digitalnih potrdil Zavoda za zdravstveno zavarovanje Slovenije, obiskano 27. 5. 2019.

[2] UREDBA (EU) 2016/679 EVROPSKEGA PARLAMENTA IN SVETA z dne 27. aprila 2016 o varstvu posameznikov pri obdelavi osebnih podatkov in o prostem pretoku takih podatkov ter o razveljavitvi Direktive 95/46/ES (Splošna uredba o varstvu podatkov)

[3] Zakon o varstvu osebnih podatkov (ZVOP-1), Uradni list RS, št. 94/07

[4] Zakon o elektronskem poslovanju in elektronskem podpisu (ZEPEP), Uradni list RS, št. 98/04

[5] Zakon o potrošniških kreditih (ZPotK-2), Uradni list RS, št. 77/16

[6] Zakon o centralnem kreditnem registru (ZCKR), Uradni list RS, št. 77/16

[7] Pravila sistema izmenjave informacij o zadolženosti fizičnih oseb – SISBON, Uradni list RS, št. 65/17, 6/18 in 68/18

OTS 2019
SODOBNE INFORMACIJSKE TEHNOLOGIJE IN STORITVE
M. Heričko in K. Kous (ur.)

Univerzitetna založba
Univerze v Mariboru

ZKP (ZERO-KNOWLEDGE PROOF) POD DROBNOGLEDOM

BLAŽ PODGORELEC, MUHAMED TURKANOVIĆ

Povzetek: *Področje kibernetske varnosti iz leta v leto postaja vse bolj pomembno. Vzrok temu je vsekakor iskati v vedno večji vseprisotnosti IKT, kakor tudi globalni porasti kibernetskih napadov. Zaskrbljenost je slutiti na političnem, kot tudi na osebni ravni, saj se v času, ko kot posamezniki postopoma izgubljamo možnost nadzorovanja zasebnosti, vse bolj zavedamo pomena le-te. Večina mehanizmov, ki so uporabljeni z namenom doseganja kibernetske varnosti, temelji na kriptografiji. Simetrična kriptografija, kriptografija javnih ključev in kriptografske zgoščevalne funkcije so tako širši javnosti, kot tudi IT strokovnjakom že dodobra poznani, kar še ne pomeni, da so tudi edini mehanizmi, s katerimi je mogoče zagotavljati oz. povečati stopnjo kibernetske varnosti. Primer so protokoli ZKP (Zero-Knowledge Proof), s pomočjo katerih je omogočeno uspešno dokazovanje določenega dejstva, brez potrebe, da o dejstvu samem izdamo kakršno koli dodatno informacijo. V prispevku bomo opisali protokol ZKP in izpostavili kje ga je mogoče smiselno vpeljati v delovanje kateregakoli sistema, ki zahteva javno verifikacijo potencialno zasebnih informacij.*

Ključne besede: *zasebnost, zaupanja vredno računalništvo, tehnologija veriženja blokov, kriptografija, zero-knowledge*

NASLOVA AVTORJEV: Blaž Podgorelec, doktorski študent, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Blockchain Lab:UM, Maribor, Slovenija, e-pošta: blaz.podgorelec@um.si. dr. Muhamed Turkanović, docent, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Blockchain Lab:UM, Maribor, Slovenija, e-pošta: muhamed.turkanovic@um.si.

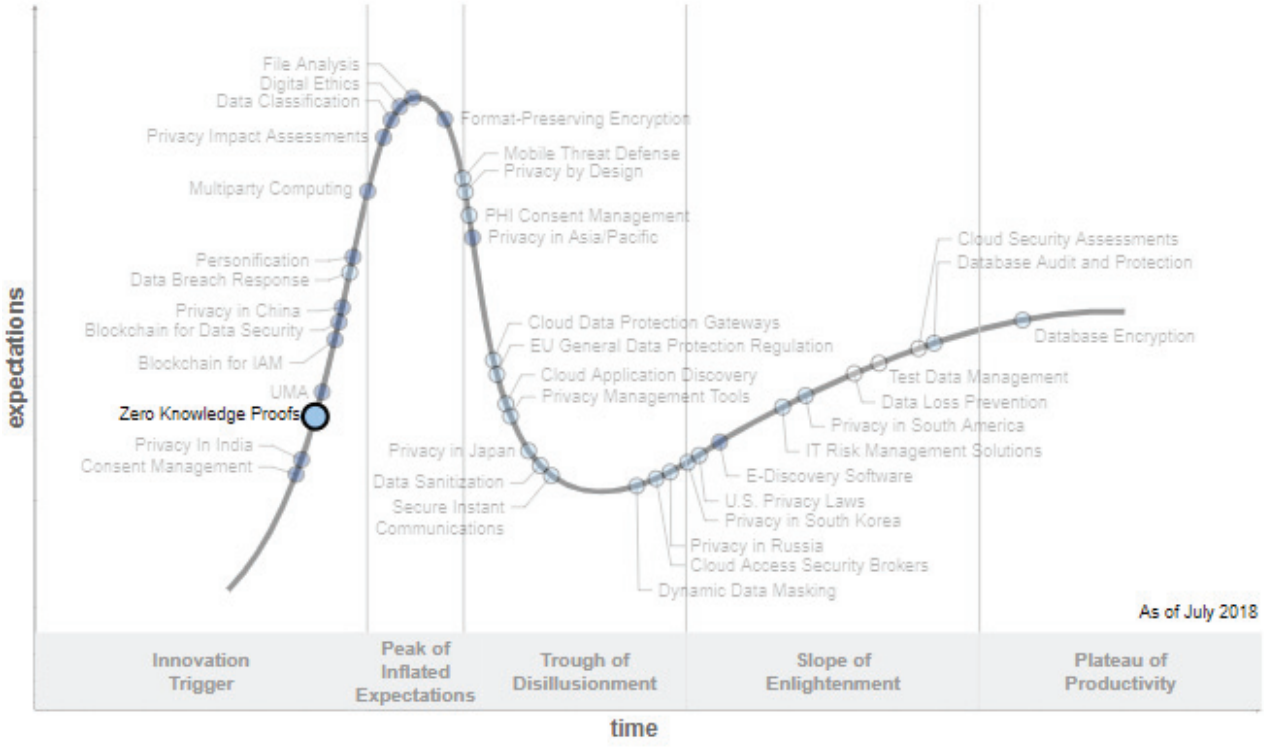
https://doi.org/10.18690/978-961-286-282-4.12 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 UVOD

Kibernetska varnost vključuje številne mehanizme, pri čemer je večini osnova kriptografija. Večina IT strokovnjakov je s temi mehanizmi seznanjena in v procesu razvoja IKT rešitev tudi aktivno uporablja kriptografske metode, kot so simetrična kriptografija, kriptografija javnih ključev, kriptografske zgoščevalne funkcije itd. Zaradi vedno večje vpetosti slednjih v vsakdanje življenje, se je potrebno zavedati tudi določenih nevarnosti, ki smo jim kot posamezniki pri tem lahko izpostavljeni. V potrditev kaže število kibernetskih napadov oz. varnostnih incidentov povezanih z rešitvami IKT, ki iz leta v leto naraščajo [1]. Zaskrbljenost je slutiti na politični kot na individualni ravni [2], saj je vedno več kibernetskih napadov povezanih z zasebnostjo končnih uporabnikov ali s strateško pomembnimi podatki in procesi.

Odgovor na trenutno situacijo je moč opaziti na različnih nivojih, saj je vedno več procesov in storitev povezanih z varnostjo IKT. Kot primer lahko izpostavimo Evropsko unijo, ki trenutno veliko sredstev namenja raziskavam na področju kibernetske varnosti [3]. Eden od pomembnih faktorjev raziskovanja na tem področju, je proučevanje možnosti uporabe kriptografskega protokol imenovanega Zero-Knowledge-Proof (v nadaljevanju ZKP).

Čeprav je protokol ZKP prvič predstavljen že leta 1985 [4], zasledimo šele leta 2017 prvič omembo le tega v sklopu enega izmed Gartnerjevih poročil, tj. Gartnerjev graf navdušenja za tehnologijo veriženja blokov (ang. Hype Cycle for Blockchain Technologies) [5]. Že leta 2018 se isti znajde tudi na Gartnerjevem grafu navdušenja za področje zasebnosti (ang. Hype Cycle for Privacy) in sicer kot tehnologija v fazi zasnove koncepta, kar proži veliko zanimanje javnosti (Slika 1).



Slika 1: Gartnerjev graf navdušenja za področje zasebnosti [6].

Protokol ZKP omogoča dokazovanje poznavanja določenega dejstva, brez potrebe po razkrivanju samega dejstva ali kakršnekoli dodatne informacije. Poenostavljeno izhaja posebnost protokola iz dejstva, da je zelo preprosto dokazati nekomu poznavanje ključnega dejstva, če se le to v dokaz enostavno razkrije, pri čemer je izziv dokazati isto in samo dejstvo ne razkriti.

V nadaljevanju bomo predstavili osnovne koncepte delovanja protokola in posamezne sisteme, ki v svoje delovanje vključujejo protokol ZKP. Podali bomo primere uporabe protokola na področju tehnologije veriženja blokov in možne uporabe le tega pri razvoju drugih IKT rešitev.

2 PREDSTAVITEV PROTOKOLA

Delovanje protokola je mogoče opisati kot interakcijo med dvema deležnikoma:

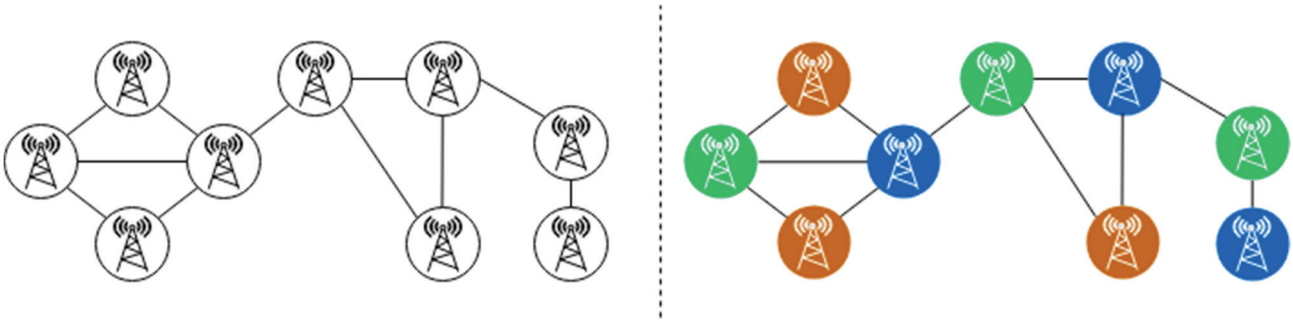
- Dokazovalec (ang. Prover)
- Preverjevalec (ang. Verifier)

Cilj dokazovalca je preverjevalca prepričati, da je določeno dejstvo obstaja in je resnično. Posebnost pri tem je, da v procesu dokazovanja, dokazovalec uporablja dokaze, ki o samem dejstvu ne razkrivajo nobene druge informacije, kot samo informacijo o tem, da dejstvo obstaja in da je resnično.

Proces dokazovanja dejstva se izvaja z izmenjavo sporočil (preverjevalec izzove dokazovalca, ta mu na izziv odgovori) med opredeljenima deležnikoma, zato protokol ZKP prištevamo med interaktivne dokazovalne sisteme (ang. interactive proof system). V postopku preverjanja resničnosti vsakega posameznega dokaza o resničnosti dejstva preverjevalec uporablja naključno izbiro izzivov, s čimer z vsako novo interakcijo povečuje verjetnost, da je določeno dejstvo mogoče sprejeti za resnično [7]. Zaradi lažjega razumevanje uporabnosti protokola ZKP je v nadaljevanju delovanje protokola predstavljeno na konkretnem primeru, na katerem tudi temelji eden izmed prvih praktičnih implementacij protokola, tj. Zerocash [8] [9] [10].

2.1 Postavitev oddajnikov telekomunikacijskega omrežja

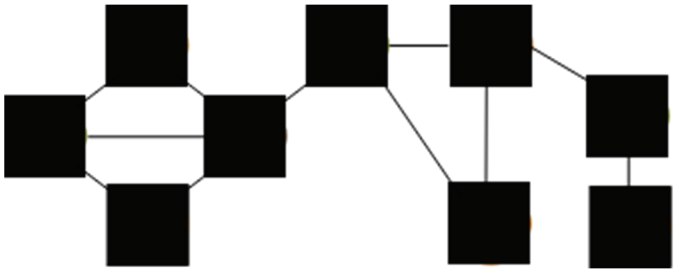
Uporabnost protokola ZKP bomo predstavili na problemu postavitve oddajnikov telekomunikacijskega omrežja. V omenjenem scenariju predstavlja preverjevalca telekomunikacijsko podjetje, ki si želi postaviti novo komunikacijsko omrežje. Primer sheme in potencialne veljavne rešitev je v obliki grafa, predstavljen na sliki 2. Komunikacijsko omrežje se sestoji iz oddajnikov, pri čemer povezave med njimi predstavljajo lokacije, kjer se signala oddajnikov prekrivata, kar lahko povzroči motnje v omrežju. Omrežje lahko sestavlja tri vrste različno konfiguriranih oddajnikov (predstavljeno s tremi različnimi barvami), ki v primeru, da so pravilno umeščeni v omrežje, ne povzročajo motenj signala svojih sosednjih – povezanih oddajnikov.



Slika 2: Shema problema in potencialne rešitve postavitve oddajnikov telekomunikacijskega mobilnega omrežja [8].

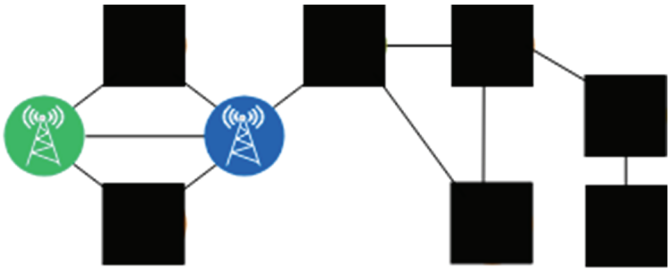
Telekomunikacijsko podjetje omenjeno problematiko primerne postavitve oddajnikov, iz različnih razlogov, ne more rešiti samostojno. Za reševanje problematike zaprosi zunanjega izvajalca. Težava nastane, ko telekomunikacijsko podjetje ni pripravljeno izvesti plačila, ne da bi se pred tem prepričalo, ali je rešitev za njihovo težavo res pravilna, ob enem pa izvajalec ni pripravljen razkriti svoje rešitve, dokler le ta ne bo plačana. Podjetji se tako znajdeta na mrtvi točki, ki je rešljiva z uporabo protokola ZKP, pri čemer v okvirju predstavitve protokola ZKP izvajalec nastopa v vlogi dokazovalca in telekomunikacijsko podjetje v vlogi preverjevalca.

Za namen testiranja telekomunikacijsko podjetje pripravi shemo komunikacijskega omrežja (Slika 3) za katero potrebujejo pomoč pri postavitvi primerno konfiguriranih oddajnikov. Po predaji sheme, izvede izvajalec primerne izračune in pripravi rešitev, ki omenjene oddajnike predstavljene na shemi obarva z eno izmed treh različnih barv na način, da so oddajniki medsebojno postavljeni tako, da med njimi ne prihaja do motenj signala, pri čemer vsaka od treh barv predstavlja različno konfiguracijo oddajnika.



Slika 3: Zakrita shema rešitve postavitve oddajnikov telekomunikacijskega mobilnega omrežja.

Interes izvajalca je dokazati, da je predstavljena rešitev pravilna, brez da bi pri tem razkrili celotno rešitev, zato telekomunikacijskemu podjetju dovolijo naključno izbiro dveh medsebojno povezanih oddajnikov, katerih konfiguracija (z barvo) bo telekomunikacijskemu podjetju tudi razkrita. Slika 4 prikazuje razkriti konfiguraciji dveh oddajnikov, ki sta del celotne sheme mobilnega telekomunikacijskega omrežja.

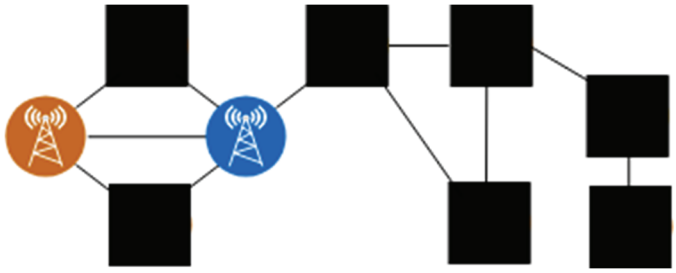


Slika 4: Razkriti konfiguraciji oddajnikov.

S tem ko je izvajalec razkril del svoje rešitve, se lahko telekomunikacijsko prepriča, ali je razkrit del rešitve veljaven in ali ustreza zahtevam. V primeru, ko sta oddajnika enake konfiguracije (barve), je rešitev glede na podane zahteve napačna. Medsebojno povezana oddajnika z različnima konfiguracijama (barvama) nakazujeta, da je podana rešitev pravilna. Kljub razkritju dveh konfiguracij oddajnikov, telekomunikacijsko podjetje s tem ne zbere dovolj informacij, da bi s tem lahko sestavilo ali izračunalo postavitev preostalih oddajnikov.

Po eni iteraciji naključne izbire dveh oddajnikov, telekomunikacijsko podjetje še vedno ni prepričano, da je celotna rešitve sheme telekomunikacijskega omrežja pravilna, zato izvajalcu predlaga, da postopek ponovita. Preden telekomunikacijsko podjetje izbere dva nova naključna sosednja oddajnika, katerih konfiguracija bo razkrita, izvajalec naključno premeša postavitev konfiguracij oddajnikov v celotni shemi mobilnega komunikacijskega omrežja. Ko telekomunikacijsko podjetje v novi iteraciji izbire dveh povezanih oddajnikov, izbere isto ležeče oddajnike kot v prejšnji iteraciji, s tem ne pridobi nujno vedno enako predlagano rešitev konfiguracije. Primer druge iteracije je prikazan na sliki 5. Takšen postopek posledično rezultira v dejstvo, da v primeru, če telekomunikacijsko podjetje beleži konfiguracije oddajnikov po vsaki iteraciji, kljub temu ne bo nikoli zmožno sestaviti oz. pridobiti toliko informacij, da bi lahko samostojno pripravil končno in s tem pravilno shemo konfiguracij oddajnikov telekomunikacijskega omrežja.

S ponavljanjem postopka pred vsako iteracijo in s povečevanjem števila le teh, je telekomunikacijsko podjetje lahko vedno bolj prepričano, da je izvajalec res pripravil shemo telekomunikacijskega omrežja, ki sledi vsem predhodnim zahtevam o postavitvi le tega in lahko postopek ponavlja tako dolgo, dokler verjetnost, da je predstavljena rešitev s strani izvajalca napačna, ne postane praktično zanemarljiva.



Slika 5: Razkriti konfiguraciji oddajnikov, v eni izmed nadaljnjih iteracij.

Primer predstavlja način kako lahko s pomočjo principov protokola ZKP, izvajalec telekomunikacijskemu podjetju dokaže, da je določeno dejstvo resnično, tj. dokaže svojo sposobnost pravilne postavitve oddajnikov v shemi telekomunikacijskega omrežja, brez da bi pri tem izdalo kakršnokoli drugo informacijo o dejstvu samem. Zaradi naključnosti pri generiranju postavitve oddajnikov v vsaki iteraciji, telekomunikacijsko podjetje o samem dejstvu – celotni rešitvi pravilne postavitve oddajnikov, ne izve ničesar kot samo to, ali je del postavitve in s tem dokaz pravilen ali ne.

2.2 LASTNOSTI

V tem poglavju bomo definirali lastnosti, ki veljajo za protokol ZKP.

Medtem ko o dejstvu samem protokoli ZKP ne izdajo nikakršne dodatne informacije (ang. zero knowledge), mora dejstvo, ki ga želimo dokazati s takšnim protokolom, temeljiti na problemu razreda kompleksnosti NP [11] [7]. V teoriji kompleksnosti takšno dejstvo definiramo kot niz težav, ki so rešljive z odločanjem (odgovor da ali ne) v polinomskem času z ne deterministčnim Turingovim strojem [7]. Ena izmed poglavitnih lastnosti takšnih problemov, ki spadajo v razred kompleksnosti NP je, da je njihovo reševanje zahtevno in terja večjo računsko moč. Nasprotno pa velja za preverjanje rešitev takšnih problemov, saj jih je zelo lahko preveriti, brez potrebe po posedovanju velike količine računske moči [11].

Protokol, ki ustreza zahtevam ZKP, mora zadostiti naslednje tri lastnosti, ki jih bomo opisali s pomočjo primera iz prejšnjega poglavja:

- Popolnost (ang. Completness)

  Vsako resnično dejstvo mora o svoji veljavnosti posedovati prepričljiv dokaz [12]. V navezavi s predhodno podanim primerom postavitve oddajnikov, to pomeni, da če je izvajalec resnično sposoben rešiti shemo telekomunikacijskega omrežja, potem bo s časom o tem prepričal tudi telekomunikacijsko podjetje [8].

- Trdnost (ang. Soundness)

  Nobeno napačno dejstvo, ne more posredovati prepričljivega dokaza o svoji resničnosti [12]. Izvajalec lahko prepriča telekomunikacijsko podjetje o pravilni postavitvi oddajnikov samo v primeru, če je rešitev res pravilna. Vključitev interakcije in naključnosti v protokol dokazovanja telekomunikacijskemu podjetju omogoča, veliko verjetnost ugotavljanja morebitne napačne postavitve oddajnikov [8].

- Ničelno znanje (ang. Zero-knowledgeness)

  Dokaz o dejstvu ne razkriva nikakršnega dodatnega znanja, kot samo to, da je le to resnično [12]. Telekomunikacijsko podjetje med celotnim postopkom dokazovanja ne pridobi nikakršne dodatne informacije o sami končni postavitvi preostalih oddajnikov v celotni shemi telekomunikacijskega omrežja [8].

3 PROTOKOLI ZKP

Vpeljava interakcije, ki poteka med dokazovalcem in preverjevalcem je eden izmed ključnih aspektov uspešnega delovanja protokola ZKP. Za uspešno izvedbo dokazovanja poznavanja določenega dejstva, sta tako morala dokazovalec in preverjevalec izmenjati večje število sporočil, ki so preverjevalca prepričala, da dokazovalec določeno dejstvo resnično pozna. Preverjevalec zmeraj sam določi v katerem trenutku je verjetnost, da dokazovalec res pozna dejstvo za njega dovolj velika in prepričljiva.

Leta 1988 so znanstveniki razvili prvi teoretični model protokola temelječega na ZKP, s katerim je mogoče dosegati predhodno opisane lastnosti, z razliko, da se iz procesa dokazovanja določenega dejstva, odstrani

oziroma zmanjša število potrebnih interakcij (število sporočil), ki si jih morata dokazovalec in preverjevalec pri tem izmenjati [13] [14]. Obdobje razvoja in s tem prehoda iz teoretičnih modelov do dejanskih implementacij protokolov, ki temeljijo na principih ZKP in pri tem ne vključujejo interakcije med dokazovalcem in preverjevalcem, se je intenzivneje začelo z razvojem različnih protokolov tehnologije veriženja blokov (ang. blockchain). Vpeljava principov protokola ZKP v platforme, temelječe na tehnologiji veriženja blokov, omogoča popolnoma anonimno izvajanje transakcij znotraj omrežja verige blokov. Trenutno obstaja več različnih dokazovalnih sistemov, ki znotraj svojih implementacij vključujejo principe protokola ZKP. V nadaljevanju bomo najbolj znane na kratko opisali. Primerjava med izbranimi je predstavljena v tabeli 1.

zk-SNARK (Zero-Knowledge Succinct Non-Interactive Argument of Knowledge)

Najbolj razširjena implementacija protokola ZKP je različica, ki je vključena v delovanje protokola digitalne kriptovalute Zcash[1]. Z vključitvijo protokola ZKP v delovanje protokola omenjene kriptovalute je uporabnikom omogočena uporaba t. i. zaščitenih transakcij (ang. shielded transactions). Te uporabnikom omogočajo dokazovanje veljavnosti opravljenih transakcij, ne da bi jim pri tem bilo treba razkriti njihov javni naslov v omrežju verige blokov, kot tudi sam znesek transakcije. Poglavitna slabost omenjene implementacije je, da je pred samo uporabo sistema potrebno opraviti t. i. fazo namestitve, ki kot rezultat postopka generira par ključev (dokazovalni in overitveni). Ta ključa sta uporabljena v postopkih generiranja dokaza s strani dokazovalca za specifičen problem kot tudi s strani preverjevalca v postopku preverjanja tega dokaza. Namen vnaprejšnjega generiranja omenjenih ključev je izločitev večkratne potrebe po interakciji med dokazovalcem in preverjevalcem, med samim postopkom preverjanja določenega dejstva. Z generiranjem para ključev je tako vnaprej izvedena izbira naključnih točk, ki bodo v postopku preverjanja predloženega dokaza preverjene. Uporabljen algoritem za naključen izbor točk, zahteva kot vhodni podatek določen niz skritih podatkov, na podlagi katerih je naključna izbira tudi izvedena. V primeru, da ti podatki postanejo javni, je tako možno generiranje lažnih dokazov, iz tega razloga obstaja potreba, da je namestitvena faza izvedena znotraj zaupanja vrednega okolja, kjer do razkritja niza podatkov, uporabljenega znotraj algoritma naključne izbire točk preverjanja, ne bo prišlo. Implementacija protokola sloni na nestandardnih, manj znanih kriptografskih predpostavkah poznavanja eksponenta (ang. knowledge of exponent). Vpeljava slednjih omogoča hitrejše delovanje sistema, vendar uporaba novejših nestandardiziranih kriptografskih metod, čeprav do zdaj še nimajo dokazanih pomanjkljivosti, predstavlja določeno varnostno tveganje [10] [15].

Bulletproofs

Ena različica implementacije protokola ZKP je uporabljena znotraj protokola digitalne kriptovalute Monero[2]. Z vključitvijo protokola ZKP v le to, je omogočena zasebna izvedba plačilnih transakcij. Implementacija protokola sloni na uveljavljenih kriptografskih predpostavkah (diskretno logaritemski računski problemi). Izvedbena hitrost overjanja s sistemom, je zato sicer počasnejša, vendar za razliko od sistema zk-SNARK, za delovanje le tega ni potrebe po izvedbi faze namestite znotraj zaupanja vrednega okolja [16].

zk-STARK (Zero-Knowledge Succinct Transparent Non-Interactive Argument of Knowledge)

Tovrstna implementacija je trenutno deležna največ pozornosti v smislu raziskav in razvoja. Zaradi relativno zgodnje faze implementacije v kateri se zk-STARK trenutno nahaja, posledično ni uporabljen znotraj nobenega protokola v dejanski uporabi. Omenjen sistem velja za izboljšano različico sistema zk-SNARKS in se od nje razlikuje v naslednjih lastnostih. Faza namestitve je izvedena s principi javno preverljivih naključnosti, kar pomeni, da za izvedbo te faze, ni potrebe po zaupanja vrednem okolju. Izvedbena hitrost verifikacije je po trenutnih meritvah hitrejša od sistema zk-SNARK. Zasnova sistema temelji na kriptografskih enosmernih funkcijah, ki so odporne na trke, kar posledično pomeni, da se lahko tak sistem smatra kot varen tudi v t. i. dobi kvantnih računalnikov [17].

[1] https://z.cash/
[2] https://ww.getmonero.org/

Tabela 1: Primerjava trenutno najbolj aktivnih dokazovalnih sistemov temelječih na ZKP [16] [17].

	zk-SNARK	Bulltetproofs	zk-STARK
Hitrost verifikacije	~16ms	~1100ms	~10ms
Potreba po zaupanju vrednem okolju	DA	NE	NE
Kriptografske predpostavke	Poznavanje eksponenta	Diskretni logaritmi	Zgoščevalne funkcije
Kvantna odpornost	NE	NE	DA

Opisane protokole ZKP je v obstoječe projekte mogoče vključevati s pomočjo različnih programskih knjižnic, seznam katerih smo zbrali v tabeli 2.

Tabela 2: Programske knjižnice implementacij dokazovalnih sistemov ZKP [18].

Ime	Programski jezik	Repozitorij	Protokol ZKP
libsnark	C++	https://github.com/scipr-lab/libsnark	zk-SNARK
websnark	WebAssembly	https://github.com/iden3/websnark	zk-SNARK
snarkjs	JavaScript	https://github.com/iden3/snarkjs	zk-SNARK
DIZK	Java	https://github.com/scipr-lab/dizk	zk-SNARK
bellman	Rust	https://github.com/zkcrypto/bellman	zk-SNARK
Dalek bulletproofs	Rust	https://github.com/dalek-cryptography/bulletproofs	Bulletproofs
Adjoint-io bulletproofs	Haskel	https://github.com/adjoint-io/bulletproofs	Bulletproofs
LibSTARK	C++	https://github.com/elibensasson/libSTARK	zk-STARK

4 PRIMERI UPORABE

Protokol ZKP je smiselno vključiti v primere uporabe, ki zahtevajo povečano stopnjo zasebnosti uporabnikov, kot tudi v primere, kjer obstaja potreba po dokazovanju korektne izvedbe določenih operacij izvedenih znotraj rešitev IKT. Uporabniku, ki protokol uporablja, v postopku preverjanja rezultatov le tega, ne bilo potrebno razkriti kakršnekoli vhod v tak sistem. Kot že omenjeno v prejšnjih poglavjih, je protokol ZKP teoretično mogoče vpeljati v kakršenkoli sistem, ki temelji na reševanju problematike, ki spada v razred kompleksnosti NP.

V nadaljevanju bomo podali nekatera področja primerov uporabe, kjer lahko vključitev protokola ZKP občutno pripomore k povečanju trenutne ravni varnosti in zasebnosti končnih uporabnikov.

Lastništvo podatkov za dostop

Trenutni sistemi overjanja običajno od uporabnika na neki točki pričakujejo vnos gesla. Podano geslo je v zalednem sistemu običajno shranjeno ter z uporabnikom povezano. Uporabnik je tako v popolnosti primoran zaupati sistemu glede varnega načina hranjenja ter obdelave podanega gesla. Z vključitvijo protokola ZKP v proces overjanja, je za uporabnika mogoče doseči bolj varen in od sistema neodvisen ter zaupanja vreden način overjanja, saj uporabnik v nobenem trenutku ni pozvan pošiljati občutljivih podatkov (gesla) v sistem in tako v nobenem trenutku ne obstaja možnost morebitne odtujitve le tega s strani zlonamerne entitete [19]. Omenjen primer uporabe je mogoče implementirati s pomočjo hevristične sheme Fiat-Shamir, znotraj katere so uporabljeni diskretni algoritmi, ki preverjevalcu in dokazovalcu omogočata hitro dokazovanje ter overjanje. Ob tem potencialno zlonamerni entiteti teoretično ne omogoča načina, s katerim bi le ta bila sposobna sestaviti pravilen dokaz o poznavanju gesla [20] [21].

Dražbeni sistemi

Delovanje elektronskih dražbenih sistemov ima z vidika uporabnika (udeleženca) kar nekaj varnostnih pomanjkljivosti v smislu razkrivanja osebnih podatkov uporabnika, filtriranja ponudb, trgovanja določenih udeležencev na podlagi notranjih informacij in nezmožnosti transparentnega preverjanja rezultatov le teh.

Nekatere probleme je mogoče reševati s pomočjo različnih že uveljavljenih kriptografskih tehnik (šifriranje, anonimizacija), vendar problematika, kot je preverjanje pravilnosti izvedbe protokola elektronske dražbe ostaja. Vključitev protokola ZKP v tak sistem, lahko poveča robustnost protokolov elektronskih dražb, saj bi ob vključitvi protokola v sistem udeležencem bilo omogočeno preverjanje pravilnosti poteka dražbe, brez potrebe po razkrivanju kakršnih kolih informacij pridobljenih tekom dražbe, tj. dejanskih ponudb [22].

Poznavanje rešitev v igrah

Protokol ZKP je mogoče vključiti v dokazovanje rešitev določenih miselnih iger. Eden izmed primerov je igra Sudoku, kjer lahko z uporabo protokola ZKP, igralec, dokaže poznavanje rešitve, brez izdajanja le te ostalim igralcem. Kot drug primer lahko podamo igro lova na zaklad, kjer lahko iskalec zaklada z uporabo protokola ZKP, ostalim tekmovalcem uspešno dokaže, da pozna koordinate skrivnega objekta, brez, da bi pri tem obstajala potreba po izdaji dejanskih koordinat, ki lokacijo kakorkoli razkrivale ostalim tekmovalcem. Rešitve iger (vsebina za vse koordinate Sudoku) so namreč s strani dokazovalca v postopek preverjanja podane v obliki kriptografskega dokaza, s pomočjo katerega lahko preverjevalec potrdi njihovo pravilnost, brez potrebe po poznavanju podrobnosti o rešitvi sami [19].

Bančni sistemi

Z vključitvijo protokola ZKP v bančne sisteme bi se uporabniku omogočilo, da ob izvedbi določene finančne transakcije, dokaže da ima za izvedbo le te na računu dovolj sredstev, brez potrebe, da bančni sistem izvede vpogled v dejansko bilančno stanje uporabnikovega računa [19]. Primer takšnega sistema je nizozemska finančna družba ING, ki je predstavila na tehnologiji veriženja blokov temelječ sistem, s katerim je strankam omogočeno dokazovanje, da za najem posojila prejemajo zadosten mesečni prihodek, brez potrebe, da bi višino le tega morali dejansko razkriti [23] [24].

Tehnologija veriženja blokov

Kot je že bilo omenjeno v uvodu tega prispevka, se je raziskovanje, razvoj ter uporaba dokazovalnih sistemov temelječih na protokolu ZKP, aktivneje pričelo s prebojem tehnologije veriženja blokov. Prav iz tega razloga, je vključenost protokola ZKP v sisteme, ki temeljijo na tehnologiji veriženja blokov občutno večje, kot v sisteme, ki so bili zasnovani in načrtovani pred prebojem te tehnologije. Večina platform, katerih delovanje temelji na tehnologiji veriženja blokov, ob izvajanju transakcij znotraj omrežij verig blokov, ne omogoča anonimne izvedbe le teh. Omenjena lastnost zato morebiti preprečuje razvoj novih poslovnih procesov, katerih izvajanje bi potencialno temeljijo na tehnologiji veriženja blokov, vendar za to zahtevajo nekoliko več zasebnosti. Platforme, kot so Zcash [25], Zcoin [10] in Monero [16], so prav z vpeljavo protokolov ZKP uporabnikom omogočile izvajanje anonimnih in varnih transakcij ter kljub tem lastnostim ohranile transparentnost delovanja na tehnologiji veriženja blokov temelječega elektronskega plačilnega sistema.

Uporabo protokola ZKP je mogoče vključevati tudi v same protokole omrežij verig blokov. Raziskovalci in razvijalci platforme Ethereum raziskujejo možnosti, kako lahko protokol ZKP pripomore pri reševanju problematike razširljivosti omrežij verig blokov, s katerim se trenutno ukvarja večina platform na tem področju. Protokol bi lahko tako bolj konkretno uporabili za dokazovanje o veljavnosti zadnjega doseženega stanja znotraj omrežja verige blokov. Verificiranje dokaza, ki potrjuje, da je bilo stanje spremenjeno na pravilen način, je namreč veliko hitreje, kot verifikacija opravljene spremembe z dejansko ponovitvijo le te.

5 ZAKLJUČEK

Omogočanje možnosti dokazovanja poznavanja določenega dejstva znotraj problema kompleksnosti NP, brez da bi pri tem obstajala potreba po razkritju le tega, v primerjavi z ostalimi kriptografskimi protokoli predstavlja poglavitno posebnost protokola ZKP. Čeprav je protokol, predstavljen že leta 1985, se vse do trenutka preboja tehnologije veriženja blokov, kjer je uporabljen za omogočanje anonimne izvedbe transakcij znotraj javnega omrežja verige blokov, praktično znotraj rešitev IKT ni razširjeno uporabljal. Protokol se trenutno sicer še vedno nahaja v fazi prehoda iz teoretičnih okvirjev v razširjeno praktično uporabo, vendar se število raziskav, kot tudi dejanskih programskih implementacij dokazovalnih sistemov, temelječih na principih ZKP, povečuje. Razlog temu je mogoče iskati tudi v povečevanju zanimanja končnih uporabnikov rešitev IKT po novih in boljših načinih, ki omogočajo ohranjanje zasebnosti uporabnika ter njegovih podatkov. Trenutna uporaba protokola ZKP je omejena predvsem na rešitve, ki izhajajo iz področja tehnologije veriženja blokov, saj omogoča že omenjeno izvedbo anonimnih transakcij znotraj javnega omrežja verig blokov. Uporabiti ga je mogoče tudi kot sistem, s katerim je mogoče transparentno in učinkovito preverjanje pravilnosti izvedenih

sprememb stanja decentraliziranega omrežja verig blokov. Število implementacij protokola ZKP je v prihodnosti mogoče pričakovati tudi znotraj rešitev IKT, ki v osnovi ne temeljijo na tehnologiji veriženja blokov, ampak bodo v ospredje postavljale zaupanje končnih uporabnikov le teh v smislu zaupnosti in pravilnosti pri obdelavi njihovih podatkov.

V prihodnosti tudi sami načrtujemo raziskovanje možnosti vključevanja protokola ZKP. Namen katere je omogočati izvedbo večjega števila sprememb stanj kot storitev izven omrežja verige blokov Ethereum in hkrati zagotavljati enako raven transparentnosti delovanja, kljub dejstvu, da se spremembe stanj ne izvedejo neposredno znotraj omrežja verige blokov.

6 LITERATURA

[1] L. Kirichenko, T. Radivilova, in A. Carlsson, „Detecting cyber threats through social network analysis: short survey“, *arXiv Prepr. arXiv1805.06680*, 2018.

[2] European Commission, „Proposal for a European Cybersecurity Competence Network and Centre“. [Na spletu]. Dostopno: https://ec.europa.eu/digital-single-market/en/proposal-european-cybersecurity-competence-network-and-centre. [Dostopano: 05-maj-2019].

[3] European Commission, „Four EU pilot projects launched to prepare the European Cybersecurity Competence Network“, 2019. [Na spletu]. Dostopno: https://ec.europa.eu/digital-single-market/en/news/four-eu-pilot-projects-launched-prepare-european-cybersecurity-competence-network. [Dostopano: 05-maj-2019].

[4] S. Goldwasser, S. Micali, in C. Rackoff, „The knowledge complexity of interactive proof systems“, *SIAM J. Comput.*, let. 18, št. 1, str. 186–208, 1989.

[5] D. Furlonger in R. Kandaswamy, „Hype Cycle for Blockchain Technologies, 2017“, *Gartner*, št. July, 2018.

[6] B. Willemsen, „Hype Cycle for Privacy, 2017“, *Gartner*, 2017.

[7] S. Vadhan, „The complexity of zero knowledge“, v *International Conference on Foundations of Software Technology and Theoretical Computer Science*, 2007, str. 52–70.

[8] M. Green, „Zero Knowledge Proofs : An illustrated primer“. [Na spletu]. Dostopno: https://blog.cryptographyengineering.com/2014/11/27/zero-knowledge-proofs-illustrated-primer/. [Dostopano: 10-maj-2018].

[9] „Matthew D. Green“. [Na spletu]. Dostopno: https://isi.jhu.edu/~mgreen/. [Dostopano: 19-apr-2019].

[10] E. Ben Sasson *idr.*, „Zerocash: Decentralized anonymous payments from bitcoin“, v *2014 IEEE Symposium on Security and Privacy*, 2014, str. 459–474.

[11] O. Goldreich, S. Micali, in A. Wigderson, „Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems“, *J. ACM*, let. 38, št. 3, str. 690–728, 1991.

[12] A. De Santis, S. Micali, in G. Persiano, „Non-interactive zero-knowledge proof systems“, v *Conference on the Theory and Application of Cryptographic Techniques*, 1987, str. 52–72.

[13] M. Blum, P. Feldman, in S. Micali, „Non-interactive zero-knowledge and its applications“, v *Proceedings of the twentieth annual ACM symposium on Theory of computing*, 1988, str. 103–112.

[14] O. Goldreich in Y. Oren, „Definitions and properties of zero-knowledge proof systems“, *J. Cryptol.*, let. 7, št. 1, str. 1–32, 1994.

[15] M. Bellare in A. Palacio, „The knowledge-of-exponent assumptions and 3-round zero-knowledge protocols“, v *Annual International Cryptology Conference*, 2004, str. 273–289.

[16] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, in G. Maxwell, „Bulletproofs: Short proofs for confidential transactions and more“, v *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, str. 315–334.

[17] E. Ben-Sasson, I. Bentov, Y. Horesh, in M. Riabzev, „Scalable, transparent, and post-quantum secure computational integrity.“, *IACR Cryptol. ePrint Arch.*, let. 2018, str. 46, 2018.

[18] „Zero-Knowledge Proofs What are they, how do they work, and are they fast yet?“ [Na spletu]. Dostopno: https://zkp.science/. [Dostopano: 10-maj-2019].

[19] „Demonstrate how Zero-Knowledge Proofs work without using maths“. [Na spletu]. Dostopno: https://www.linkedin.com/pulse/demonstrate-how-zero-knowledge-proofs-work-without-using-chalkias/. [Dostopano: 06-maj-2019].

[20] A. Fiat in A. Shamir, „How to prove yourself: Practical solutions to identification and signature problems“, v *Conference on the Theory and Application of Cryptographic Techniques*, 1986, str. 186–194.

[21] „Fiat–Shamir with a secret password“. [Na spletu]. Dostopno: https://asecuritysite.com/encryption/fiat2. [Dostopano: 08-maj-2019].

[22] B. Palmer, K. Bubendorfer, in I. Welch, „A protocol for verification of an auction without revealing bid values“, *Procedia Comput. Sci.*, let. 1, št. 1, str. 2649–2658, 2010.

[23] J. Camenisch in R. Chaabouni, „Efficient protocols for set membership and range proofs“, v *International Conference on the Theory and Application of Cryptology and Information Security*, 2008, str. 234–252.

[24] T. Koens, C. Ramaekers, in C. Van Wijk, „Efficient Zero-Knowledge Range Proofs in Ethereum“. 2017.

[25] D. Hopwood, S. Bowe, T. Hornby, in N. Wilcox, „Zcash protocol specification“, *Tech. rep. 2016–1.10. Zerocoin Electr. Coin Company, Tech. Rep.*, 2016.

OTS 2019
SODOBNE INFORMACIJSKE TEHNOLOGIJE IN STORITVE
M. Heričko in K. Kous (ur.)

Univerzitetna založba
Univerze v Mariboru

VARNA UPORABA INFORMACIJSKO-KOMUNIKACIJSKE TEHNOLOGIJE NA POTI IN V TUJEM OKOLJU

FRANCI MULEC, FRANC MOČILAR, SAMO MAČEK

Povzetek: *Značilnost modernih delovnih procesov je uporaba informacijsko-komunikacijske tehnologije (v nadaljevanju IKT) in dostop do informacij in dokumentov od kjerkoli in kadarkoli, visoka razpoložljivost tehnologije, vsebine in tudi skoraj stalna razpoložljivost uporabnikov, informatikov in poslovnih partnerjev. V našem prispevku bomo pokazali na verjetne grožnje in postopke obvladovanja tveganj pri delu na poti oziroma v tujem okolju. To je še posebej pomembno, če smo izpostavljena oseba. Gre za osebo, ki glede na svojo vlogo v delovnem procesu, v organizaciji ali v družbi sprejema ključne odločitve ali razpolaga s ključnimi informacijami, dela za tako osebo ali pa je v njeni bližini. V tej vlogi smo zelo pogosto informatiki in vedno tudi administratorji sistemov IKT.*

Ključne besede: *zaupni podatki, varnost, informacijska oprema, službene poti, tujina*

NASLOVI AVTORJEV: mag. Franci Mulec, CISA, sekretar, Ministrstvo za zunanje zadeve, Ljubljana. mag. Franc Močilar, CISSP, vodja Oddelka za informacijsko varnost in komunikacije, Ministrstvo za zunanje zadeve, Ljubljana, Slovenija. mag. Samo Maček, vodja Sektorja za informatiko, Generalni sekretariat Vlade RS, Ljubljana, Slovenija.

https://doi.org/10.18690/978-961-286-282-4.13 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 UVOD

Poleg običajnih in precej varnih lokacij je zaradi zahtev sodobnega časa (mobilnost, stalna dosegljivosti in razpoložljivost) potrebno delo opravljati tudi na manj varnih lokacijah, kor so na primer hotelske sobe, poslovni centri, gostinski lokali, čakalnice na letališčih, počitniški apartmaji in podobno.

Oprema, ki jo imamo običajno s seboj, so papirni dokumenti, ključki USB in diski, prenosni računalniki, pametni telefoni, tablice, pametne ure in druga prenosna oprema.

Pred pomembnimi potovanji naredimo oceno stopnje tveganja, tako da identificiramo morebitne neželene dogodke. Za vsak dogodek ocenimo verjetnost dogodka in kritičnost dogodka za nas oziroma za našo organizacijo.

Če potujemo v tujino, lahko osnovne aktualne varnostne informacije najdete na straneh Ministrstva za zunanje zadeve RS na http://www.mzz.gov.si/si/informacije_za_popotnike/varnost_potovanj/.

Slika 1: Spletna stran Ministrstva za zunanje zadeve z informacijami o varnosti potovanj [1]

V primeru večjih težav se za konzularno pomoč lahko obrnete na slovenska diplomatsko konzularna predstavništva RS v tujini. Če pa tega v državi ni, se lahko za konzularno pomoč obrnete na katerokoli predstavništvo druge države, ki je članica EU.

2 MOŽNI NEŽELENI DOGODKI IN OBVLADOVANJE TVEGANJ

2.1 Dostop do dokumentov in podatkov s strani nepooblaščenih oseb

Ukrepe za obvladovanje tveganj pri obravnavi zaupnih ali tajnih dokumentov prilagodimo možnim škodljivih posledicam v primeru njihovega razkritja ali zlorabe. Pri tem upoštevamo vse tri vidike informacijske varnosti (zaupnost, celovitost in razpoložljivost). Glede obsega posledic se obravnava od vpliva na posameznika, poslovanja in premoženja organizacije do ogrožanja (vitalnih) interesov države (ustavne ureditve, neodvisnosti, ozemeljske celovitosti in obrambne sposobnosti).

2.2 Izguba ali kraja poslovnih dokumentov, opreme in posledično podatkov

S seboj na pot vzamemo le tiste dokumente in opremo, ki jo nujno potrebujemo. Dokumente in opremo moramo imeti ves čas pod nadzorom. V tem primeru je verjetnost izgube ali kraje poslovnih dokumentov, prenosnika, tablice ali pametnega telefona nizka. Če smo izpostavljena oseba, opreme in dokumentov ne smemo puščati v hotelskih ali sejnih sobah, ampak jih nosimo s seboj. To velja, tudi če gremo na kosilo, večerjo

ali na WC. Zaklenjena (tuja) hotelska soba ali (tuja) sejna soba, ali hotelski sef ni varen kraj za shranjevanje občutljivih dokumentov in opreme, na kateri takšne dokumente hranimo. Tudi začasna izguba nadzora ali izguba opreme za dostop do informacijskega sistema organizacije (gesla, PINi, žetoni) lahko pomeni nepooblaščen enkraten ali dalj časa trajajoč nepooblaščen dostop do ključnih sestavin in informacij, kar posledično lahko prinese velike škode organizaciji. Še posebej za informatike je pomembno, da na poti ne uporabljajo administrativnih dostopov. Uporaba tehnologij WiFi ali BlueTooth je lahko tvegana, manj tvegana je uporaba prenosa podatkov preko mobilnih operaterjev. V telefonskih razgovorih se izogibamo obravnave občutljivih poslovnih in osebnih informacij. Varnost »brezplačnih« sistemov, kot so Viber, WhatsUp, Signal, Skype je varljiva. V nekaterih državah (na primer ZDA, Izrael) svoje državne uradnike učijo, da uporabljajo zavese za okna, prekrivala za vnos gesel ali PINov v opremo, iz instalacij izklopijo telefonsko, TV in podobno opremo v hotelski sobi. Zavedati se moramo, da v tujih prostorih nimamo nadzora. Nekdo lahko izvaja na primer nepooblaščeno snemanje. Med občutljivimi razgovori prosimo vse udeležence, da ugasnejo pametne telefone. Oddaja pametnih telefonov gostitelju je zelo tvegana. Tak poseg mora biti vnaprej napovedan, da udeleženci lahko pridejo brez telefonov ali da zagotovijo zaupanja vredno osebo, ki bo hranila telefon med sestankom.

Še pred potovanjem moramo poskrbeti, da imamo ključke USB, diske USB, prenosnike in pametne telefone šifrirane. Nujno moramo vključi tudi splošne možnosti zaščite, kot so BitLocker, Android šifriranje in podobne mehanizme. Ti mehanizmi so primeren ukrep za večino primerov. Ocena stopnje tveganja je lahko za poslovni proces VISOKA ali ZELO VISOKA. Zavedati se moramo, da je krajo papirnih (preslikava) ali elektronskih dokumentov (kopiranje medijev) zelo težko opaziti. V državah, ki niso članice EU so zgoraj opisane zlorabe lahko »zakonite aktivnosti« državnih organov.

V primeru obravnavanja zaupnih in tajnih podatkov višjih stopenj moramo implementirati tudi druge varnostne ukrepe, kot so uporaba akreditiranih šifrirnih rešitev, zaščita pred neželenimi elektromagnetnimi in konduktivnimi emisijami (uporaba posebne opreme ali postavitev t. i. Faradayevega šotora).

2.3 Izguba ali kraja osebnih dokumentov in denarja

Verjetnost izgube ali kraje osebnih dokumentov je običajno nizka. Za pomoč pri vračanju v domovino se lahko obrnete na slovensko diplomatsko konzularno predstavništvo v državi, kjer ste ali na tistega, ki to državo pokriva, lahko se obrnete na diplomatsko konzularno predstavništvo katerekoli države, članice EU. Ocena stopnje tveganja za poslovni proces boste ocenili sami.

2.4 Razkrivanje naše zasebnosti (navade, socialni stiki, zdravstveno stanje)

Razkrivanje naših navad, socialnih stikov in zdravstvenega stanja je lahko del načrtnega zbiranja podatkov o nas, še posebej, če smo izpostavljena oseba. Verjetnost, da do tega pride, moramo oceniti sami. Pametni telefoni, pametne ure in socialna omrežja stalno in trajno zbirajo in posredujejo svojim »kupcem« naše zasebne podatke. Nekdo lahko te podatke uporabi na našo škodo ali na škodo naše organizacije tudi čez več deset let. Vse več je tudi podatkov o nepooblaščenih snemanjih v prostorih, kjer imamo pričakovanje do zasebnosti. Bolj smo izpostavljena oseba, več samoomejevanja si moramo »naložiti«. Če želi nekdo oslabiti organizacijo ali vplivati na ključne odločitve, si za »tarčo« vedno izbere predstojnika, direktorja in ga »napade« z zbranimi podatki o njegovi zasebnosti.

2.5 Morebitne provokacije, manipulacije in izsiljevanja

Podobno kot v prejšnji točki. Varujmo svojo zasebnost. Varujmo tudi opremo na potovanju in v tujini. Na našo opremo brez zaščite in nadzora lahko nekdo kopira na primer kaznive vsebine. Izogibamo se tudi provokacij neznanih oseb.

2.6 Morebitne druge nevarnosti iz okolja na primer terorizem, nemiri

V tujini in na poti se ne izpostavljamo in se izogibamo tveganih lokacij, na primer množičnih demonstracij ali nevarnih predelov. Gibanje ponoči je bolj tvegano.

2.7 Razlaga stopenj ocene tveganja

Posamezna varnostna tveganja so v tej oceni razvrščena po verjetnosti dogodka in njegovem vplivu v naslednje stopnje, od najmanjšega do največjega tveganja:

Zelo nizko (ZN),
Nizko (N),
Zmerno (Z),
Visoko (V),
Zelo visoko (ZV).

Tabela 1: Razvrstitev varnostnih tveganj glede na verjetnost in vpliv negativnega dogodka

Lestvica kritičnosti	Lestvica verjetnosti negativnega dogodka				
	Zelo visoka (ZV)	Visoka (V)	Zmerna (Z)	Nizka (N)	Zelo nizka (ZN)
Zelo visoka (ZV)	Zelo visoko	Zelo visoko	Visoko	Zmerno	Nizko
Visoka (V)	Zelo visoko	Visoko	Visoko	Zmerno	Nizko
Zmerna (Z)	Visoko	Visoko	Zmerno	Nizko	Nizko
Nizka (N)	Zmerno	Zmerno	Nizko	Nizko	Zelo nizko
Zelo nizka (ZN)	Nizko	Nizko	Nizko	Zelo nizko	Zelo nizko

Lestvica kritičnosti:

- **Zelo visoka (ZV):** pomeni, da organizacija ne more izpolnjevati osnovnega namena delovanja brez uporabe komunikacijsko-informacijskega sistema ali ob razkritju podatkov, ki jih obravnava.
- **Visoka (V):** pomeni, da organizacija ne more učinkovito izpolnjevati dela osnovnega namena delovanja brez komunikacijsko-informacijskega sistema ali ob razkritju podatkov, ki jih obravnava.
- **Zmerna (Z):** pomeni, da je delovanje organizacije resno ogroženo, vendar lahko osnovni namen delovanja izpolni drugače z zmernimi stroški.
- **Nizka (N)**: pomeni, da bi bil ob neuporabi komunikacijsko-informacijskega sistema ali ob razkritju podatkov, ki jih obravnava, učinek na izpolnjevanje osnovnega namena organizacije majhen.
- **Zelo nizka (ZN)**: pomeni, da informacijski sistem ali podatki, ki jih organizacija obravnava, niso pomembni za izpolnjevanje osnovnega namena delovanja organizacije.

Lestvica verjetnosti negativnega dogodka:

- **Zelo visoka (ZV):** pomeni, da se bo dogodek skoraj zagotovo zgodil;
- **Visoka (V):** pomeni, da se bo dogodek verjetno zgodil;
- **Zmerna (Z):** pomeni, da se bo dogodek mogoče zgodil;
- **Nizka (N):** pomeni, da se dogodek mogoče ne bo zgodil;
- **Zelo nizka (ZN):** pomeni, da se dogodek skoraj zagotovo ne bo zgodil.

Sprejemljiva tveganja so: **zelo nizka** in **nizka**.

Pogojno sprejemljiva tveganja so: **zmerna** in **visoka**.

Nesprejemljiva tveganja so: **zelo visoka**.

Glede na stopnjo tveganja v oceni navajamo ustrezne varnostne protiukrepe za kompenzacijo posledic posameznih varnostnih tveganj.

Tabela 2: Protiukrepi za kompenzacijo posledic varnostnih tveganj

Neželen dogodek	Verjetnost dogodka	Kritičnost dogodka	**Ocena stopnje tveganja**
Dostop do dokumentov in podatkov s strani nepooblaščenih oseb	Zmerna	Visoka	**Visoko**
Kraja osebnih dokumentov in denarja	Nizka	Zmerna	**Nizko**
Razkrivanje naše zasebnosti (navade, socialni stiki, zdravstveno stanje)	Nizka	Visoka	**Zmerno**
Morebitne provokacije, manipulacije in izsiljevanja	Nizka	Zmerna	**Nizko**
Morebitne druge nevarnosti iz okolja na primer terorizem, nemiri	Nizka	Visoka	**Zmerno**

3 ZAKLJUČEK

Pri delu z zaupnimi podatki smo izpostavljeni številnim grožnjam in varnostnim tveganjem. Zaščitne ukrepe prilagajamo možnim škodljivim posledicam njihovega razkritja ali zlorabe. Na službenih poteh, še posebej v tujini, je obvladovanje teh groženj še veliko bolj zahtevno. Imamo slabše tehnične pogoje (kot v domačih varovanih objektih), potencialni napadalci pa lahko s pridom izkoristijo tudi prednost »domačega terena«.

Važno je, da tveganje ocenimo že pred potovanjem, vzamemo s seboj le najnujnejše dokumente in nujno potrebno informacijsko opremo. Vse dokumente in opremo imejmo vedno pod vidnim nadzorom. Ne izpostavljajmo se po nepotrebnem. Zavedajmo se, da smo lahko subjekt fizičnega in elektronskega nadzora ali manipulacije.

4 LITERATURA

[1] Ministrstvo za zunanje zadeve »Informacije za popotnike«, http://www.mzz.gov.si/si/informacije_za_popotnike/varnost_potovanj/, obiskano, 20. 5. 2019.

[2] SI.CERT, Informacijski pooblaščenec. ABC varnosti in zasebnosti na mobilnih napravah, www.ip-rs.si/fileadmin/user_upload/Pdf/novice/ABC_varnosti_in_zasebnosti_na_mobilnih_napravah.pdf.

[3] Uredba o informacijski varnosti v državni upravi, Uradni list RS, št. 29/18.

[4] https://hollandshielding.si, Faradayjev šotor, osebna zaščita, EMI zaščita in zaščitne vrečke, obiskano, 20. 5. 2019.

OTS 2019
SODOBNE INFORMACIJSKE TEHNOLOGIJE IN STORITVE
M. Heričko in K. Kous (ur.)

Univerzitetna založba
Univerze v Mariboru

IZKUŠNJE Z VPELJAVO AGILNEGA OGRODJA NEXUS PRI RAZVOJU PROGRAMSKE OPREME Z VEČ DISLOCIRANIMI TIMI

ROBERT T. LESKOVAR, JANEZ LUKAN

Povzetek: *V prispevku so predstavljene dvoletne izkušnje z vpeljavo agilnega ogrodja Nexus pri razvoju poslovne programske rešitve. Razvoj poteka v mednarodnem okolju, s petimi timi. V začetnem delu so predstavljene značilnosti tega agilnega ogrodja in razlogi za vpeljavo. Sledijo razlike glede na agilno ogrodje Scrum, ki je namenjeno razvoju produkta z enim timom, ter temeljne razlike glede na sorodni ogrodji za podporo razvoju produkta z več timi (Scrum@Scale, LeSS). V osrednjem delu so predstavljene prednosti, slabosti in tveganja, ki smo jih zaznali od vpeljave ogrodja Nexus, ter priporočila na podlagi izkušenj.*

Ključne besede: *agilne metodologije, razvoj programske opreme, Scrum, Nexus, dislocirani razvojni timi, upravljanje odvisnosti med razvojnimi timi*

NASLOVA AVTORJEV: dr. Robert Leskovar, direktor Razvoja in Interne informatike, Saop d. o. o., Šempeter pri Gorici, Slovenija, e-pošta: robert.leskovar@saop.si. Janez Lukan, vodja programerjev v skupini C1 in Scrum Master, Saop d. o. o., Šempeter pri Gorici, Slovenija, e-pošta: janez.lukan@saop.si.

https://doi.org/10.18690/978-961-286-282-4.14 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 UVOD

V okviru skupine Solitea s.a., ki združuje podjetja, ponudnike programskih rešitev za upravljanje organizacijskih virov ("sistemov ERP"), smo leta 2016 začeli projekt razvoja nove programske rešitve, delovno imenovan C1. Namen nove rešitve je, da v dolgoročnem obdobju nadomesti stare, različne rešitve vsakega podjetja v skupini ter s tem zniža stroške tehnološke in vsebinske prenove, ki bi se ju sicer moralo lotiti vsako podjetje zase. Glavne težave starih rešitev so v zahtevnem vzdrževanju kode in počasnem razvoju novih zahtev, uporabi starejših razvojnih tehnologij, kar botruje tudi težjemu zaposlovanju novih programerjev, premalo fleksibilnih arhitekturnih zasnovah, zastarelih uporabniških vmesnikih in slabši uporabniški izkušnji.

Velika večina razvojnih podjetij uporablja agilne razvojne procese (97 %, glede na raziskavo [1]), z različno stopnjo njihove zrelosti. Agilno procesno ogrodje Scrum je najbolj razširjeno med agilnimi ogrodji (54 % delež oz. v povezavi z ogrodjem Extreme Programming (XP) 64 % [1]), vendar ne odgovarja na izzive, ki se pojavijo pri razvoju z več kot enim timom, še posebej, če so timi na različnih lokacijah. Glavni izziv je koordinacija razvoja tistih produktnih zahtev, med katerimi obstajajo odvisnosti, ob hkratnem ohranjanju čim večje avtonomije vseh timov. Večji izziv je tudi usklajeno informiranje o spremembah na produktu.

Pri iskanju ustreznega procesnega ogrodja za razvoj produkta z več timi, smo izhajali iz naslednjih zahtev:

- želimo agilen razvoj, s poudarkom na:
  - hitri odzivnosti na potrebe trga,
  - poslovni vrednosti vsake razvite uporabniške funkcionalnosti, ter
  - uporabniški izkušnji, ki je skozi iterativni razvoj preskušena pri čim več potencialnih uporabnikih;
- izhajati želimo iz ogrodja Scrum, ki ga obstoječi timi tudi najbolje poznajo (kljub različnih stopnjam razumevanja ter izkušenosti);
- razvoj bosta začela 2 tima (na različnih lokacijah), vendar bo število kmalu naraslo na 5-6;
- ker so timi del različnih, samostojnih podjetij znotraj skupine, nismo želeli poenotiti organizacijske strukture ali povečevati njene kompleksnosti v posameznih podjetjih. Želeli smo procesno ogrodje, ki je čim bolj vitko in čim bolj neodvisno od organizacijske strukture.

Po pregledu in primerjavi procesnih ogrodij, ki so bila v tistem obdobju na voljo, smo se odločili za vpeljavo takrat najnovejšega ogrodja Nexus, ker je najbolje zadoščalo našim izhodiščnim zahtevam.

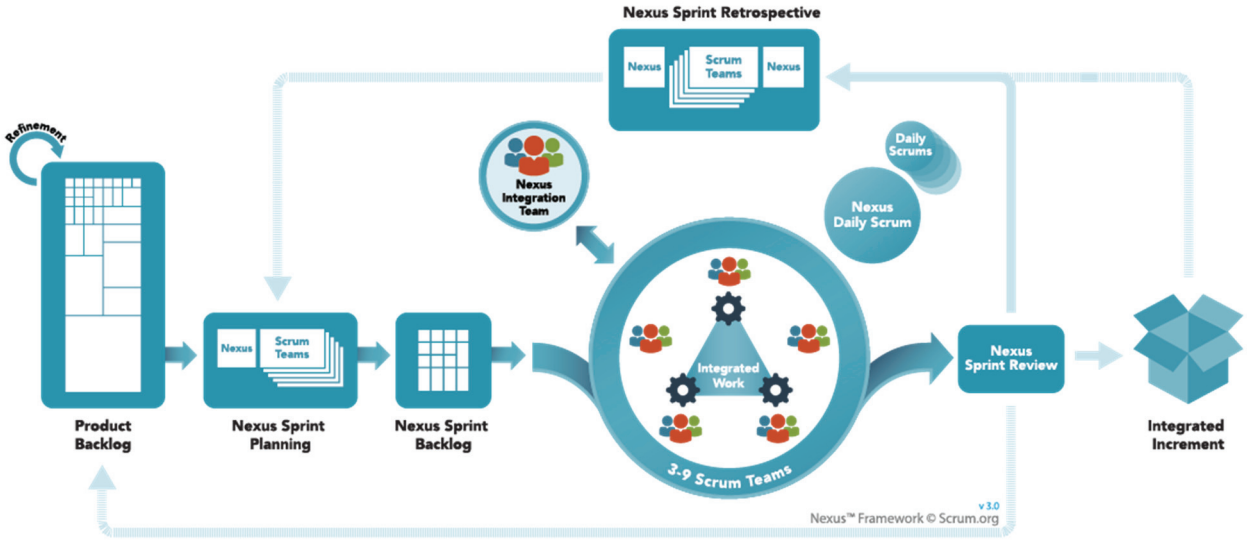
2 OGRODJE NEXUS

V prispevku predvidevamo, da je bralec seznanjen z agilnim ogrodjem Scrum, zato se osredotočamo le na razlike glede na to osnovno ogrodje. V primeru, da ni, priporočamo branje vodnika po Scrumu ("Scrum Guide") [2] ter mnogih drugih virov, ki pojasnjujejo osnove Scruma.

Ogrodje Nexus ([3]) je predstavil leta 2015 Ken Schwaber, soavtor ogrodja Scrum. Zaradi tega temelji na elementih ter filozofiji ogrodja Scrum in jih razširja le toliko, kolikor je po mnenju idejnega avtorja ter sodelavcev potrebno za učinkovit razvoj programskega produkta z več timi.

Priporočljiv predpogoj za uspešno implementacijo ogrodja Nexus je zadostna izkušenost s Scrumom v vseh timih. Ogrodje Nexus je usmerjeno v zagotavljanje učinkovitega in usklajenega dela treh do devetih Scrum timov na skupnem projektu, s poudarkom na medsebojni koordinaciji in razreševanju medsebojnih odvisnosti med timi.

Shema ogrodja Nexus je prikazana na sliki 1.



Slika 1. Ogrodje Nexus, [3]

Večja razlika ob primerjavi z ogrodjem Scrum je, da za uspešno sinhronizacijo skupnih aktivnosti v več timih ogrodje Nexus uvaja koordinacijsko skupino, imenovano Nexus Integration Team (NIT). Koordinacijska skupina skrbi za identifikacijo medsebojnih odvisnosti, razreševanje morebitnih ovir ali blokad med ekipami, mentoriranje timov ter izvedbo opravil, ki so skupna več timom v Nexusu. NIT sestavljajo produktni skrbnik ("Product Owner", v nadaljevanju PO), skrbnik Scruma ("Scrum Master", v nadaljevanju SM) in ostali člani, med katerimi so predstavniki posameznih timov. Cilj NITa je zagotoviti vsaj enkrat v razvojni iteraciji, da je uspešno integrirano vse, kar je bilo narejeno (»Done«) s strani vključenih timov. NIT svoje aktivnosti koordinira predvsem prek dnevnik sestankov (»Nexus Daily Scrum«).

Izvedba opravil, ki jih opredeli NIT, ima višjo prioriteto kot opravila, ki jih posamezniki opravljajo znotraj posameznih Scrum timov, saj opravila na nivoju NITa vplivajo na več kot en tim.

Ogrodje Nexus spreminja nekatere dogodke, ki so vezani na delo posameznega Scrum tima. Tako postane pregled iteracije ("Sprint Review") skupen dogodek vsem timom v Nexusu. Na tem dogodku timi predstavijo skupno uresničen cilj iteracije ("Sprint Goal") prek prikaza novosti na produktu. Prikazan produkt vsebuje integriran inkrement glede na prejšnjo razvojno iteracijo.

Podobno poteka načrtovanje iteracij v Nexusu ("Nexus Sprint Planning"), kjer timi določijo skupen cilj za prihajajočo iteracijo in si porazdelijo zahteve za tisto iteracijo. V nadaljevanju načrtovanja iteracije na nivoju posameznih timov, ti določijo opravila oz. naloge bolj podrobno, v okviru svojega tima.

Tudi retrospektiva poteka v treh delih. Najprej na nivoju NIT, kjer predstavniki timov in drugi člani NIT ugotavljajo, kaj bi lahko v prihodnje opravili bolje ter kako bi to dosegli. Ugotovitve takoj za tem obravnavajo posamezni Scrum timi in dodatno opredelijo tiste možne izboljšave, ki se tičejo posameznega lokalnega tima. V tretjem delu retrospektive NIT sprejme ukrepe na podlagi predlogov, ki pridejo iz posameznih timov.

V ogrodju Nexus obstaja le en PO, ki skrbi in odgovarja za produkt v celoti. Je odgovorni urednik skupnega zbira zahtev produkta ("Product Backlog") za vse time, odločevalec glede prioritet zahtev, v sodelovanju z ostalimi deležniki pripravlja vizijo in kažipot ("Roadmap") produkta. Ker je v praksi nemogoče, da bi lahko ena oseba skrbela za vse zahteve kompleksnega produkta, identifikacijo trendov in konkurenčnih točk ter ostale aktivnosti v povezavi s skrbništvom produkta, lahko PO delegira svoje aktivnosti drugim članom v posameznih timih. V našem primeru smo jih poimenovali lokalni produktni skrbniki ("Local Product Owners", LPO), tega izraza v vodniku po Nexusu ni.

Vsak posamezen Scrum tim ima svojega SM-ja, ki skrbi za pravilno in tekoče izvajanje procesa dela. Na nivoju Nexusa je dodana vloga skrbnika Nexus Scruma ("Nexus Scrum Master", v nadaljevanju NSM), ki skrbi in odgovarja za splošno razumevanje ter implementacijo ogrodja Nexus.

Pomembnejša razlika med organizacijo dela z le enim Scrum timom in več timi v okviru ogrodja Nexus je tudi pri cilju prečiščevanja zahtev ("Refinement") na nivoju Nexusa. Pri teh dogodkih se timi osredotočajo

predvsem na identifikacijo medsebojnih odvisnosti in na načrtovanje ter razgradnjo dela prek različnih Scrum timov. Namen tega dogodka je, da so lahko posamezna razvojna opravila izvedena čim bolj samostojno v posameznem Scrum timu. To pa je mogoče, če so timi opolnomočeni – imajo vse potrebne kompetence.

Podobno kot ostala ogrodja za skaliranje agilnega razvoja, tudi Nexus predvideva širitev, ko število sodelujočih timov preseže priporočeno velikost posameznega Nexusa, torej devet timov. Predvideno ogrodje za razširitev Nexusa je tako imenovan eksoskelet Nexus+ [4], za katerega ne obstaja vodnik, ampak je obravnavan le na uradni delavnici za delo z ogrodjem Nexus.

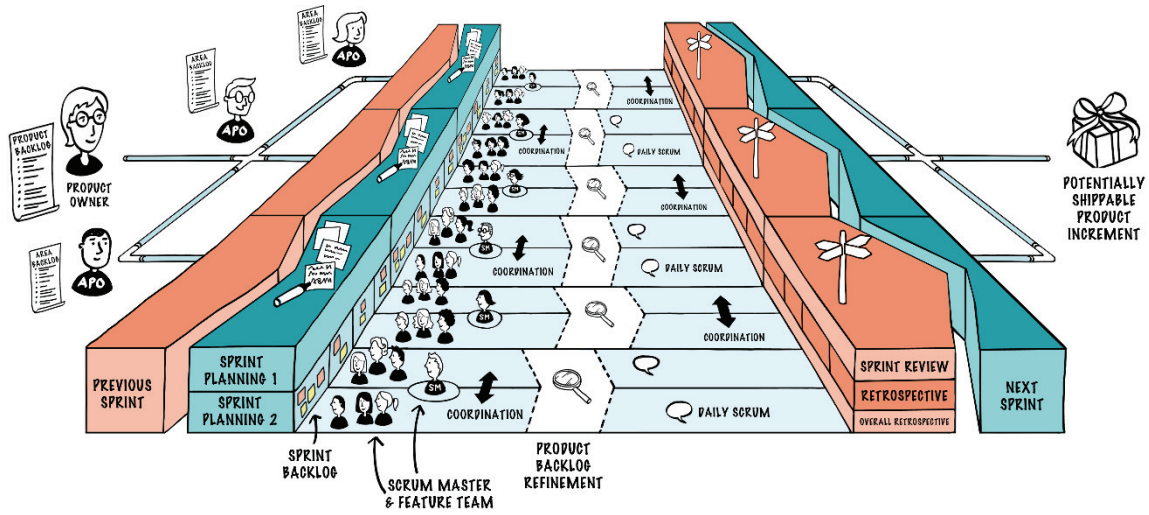
Po raziskavi [1] uporablja ogrodje Nexus 2 % sodelujočih.

3 PRIMERJAVA S SCRUM@SCALE IN LESS

3.1 Ogrodje Large-Scale Scrum (LeSS)

Ogrodje LeSS [5] prav tako razširja ogrodje Scrum in je namenjeno koordinaciji dveh do osem timov (osnovni LeSS) oziroma več kot osem timov (LeSS Huge, slika 2).

Ideja LeSS je podobna idejam ostalih tovrstnih ogrodij – ustvariti organizacijo, v kateri imajo timi vse potrebne kompetence za samostojen razvoj. Timi so zadolženi za izvedbo celovitih funkcionalnosti (t. im. "Feature Teams"). V skladu z agilno filozofijo trajnih izboljšav daje ogrodje prednost eksperimentiranju pred sledenjem predpisanih procesov.



Slika 2. Ogrodje LeSS Huge, [5]

PO je en, prav tako zbir zahtev produkta. PO je edini odgovoren za prioritiziranje zahtev, pri čemer razjasnjevanje zahtev ("Refinement") poteka neposredno med timi in tudi z uporabniki ter drugimi deležniki.

SM-ji ne skrbijo le za proces v posameznem timu, ampak v 1-3 timih in morajo biti pozorni tudi na delovanje organizacije kot celote.

Vodstvo se osredotoča na izboljšave sposobnosti, s katerimi timi ustvarjajo vrednost v celotnem sistemu razvoja produktov. Pri tem LeSS izhaja iz filozofije "Go See", kjer neposredni vodje sedijo med sodelavci. Cilj tega pristopa je, da bolje razumejo težave in da uvidijo priložnosti za izboljšave, za vzpostavljanje zaupanja ter pomoč, kar je manj verjetno, če se odločajo zgolj na podlagi poročil in ožjih sestankov.

Visokonivojsko planiranje poteka s predstavniki vseh skupin, s čimer to ogrodje zagotavlja koordinacijo med njimi.

Planiranje iteracij se deli na dva dela. Prvi del je namenjen skupnemu planiranju vseh timov, kjer udeleženci izbirajo naloge za posamezne time. Timi identificirajo točke, kjer se njihovo delo povezuje. Drugi del planiranja je namenjen planiranju znotraj posameznih timov.

Vsak tim ima svoj seznam zahtev za razvojno iteracijo ("Sprint Backlog"). Za koordinacijo skrbijo timi sami. Prednost ima decentralizirana in neformalna koordinacija.

Pregled iteracije ("Sprint Review") je eden, za celoten produkt.

Po individualnih retrospektivah se izvede celotna retrospektiva, ki se dogaja teden dni po koncu iteracije in na kateri sodelujejo predstavniki skupin, da prediskutirajo večje razvojne težave, ki obstajajo na nivoju več timov, ter definirajo izboljšave, vključno z morebitnimi eksperimenti za dosego teh.

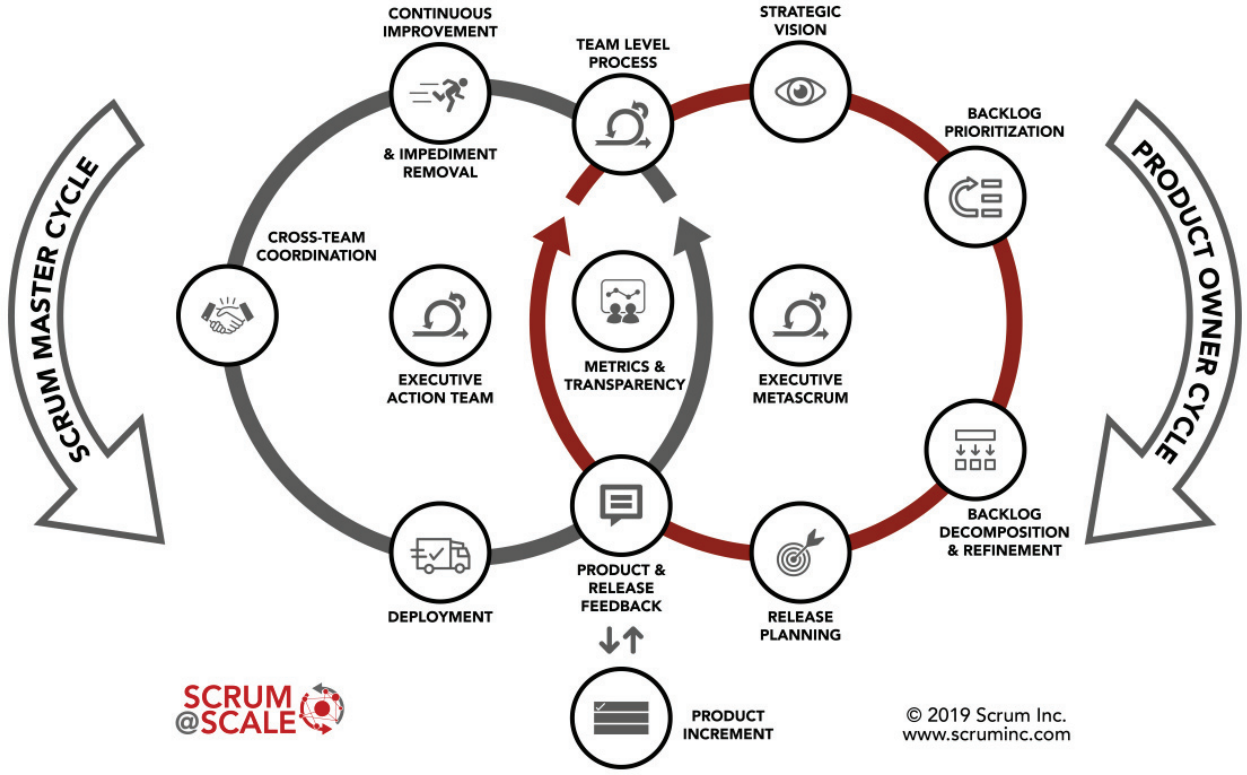
Pri koordinaciji več kot 8 timov so dodane vloge področnih PO-jev ("Area Product Owner", APO), področnih seznamov zahtev produkta in vloge ter aktivnosti za koordinacijo na tem nivoju.

Po raziskavi [1] uporablja ogrodje LeSS 3 % sodelujočih.

3.2 Ogrodje Scrum@Scale

Najstarejše ogrodje za povečevanje oz. skaliranje agilne organizacije je Scrum of Scrums. Nastalo je skozi sodelovanje Sutherlanda in Schwaberja, soavtorjev ogrodja Scrum. To ogrodje uporablja po raziskavi [1] 16 % podjetij.

Na osnovi ogrodja Scrum of Scrums je Sutherland nato razvil in predstavil v začetku leta 2018 ogrodje Scrum@Scale [6], katerega osnovne komponente so prikazane na sliki 3.



Slika 3. Osnovne komponente ogrodja Scrum@Scale, [6]

Sutherland izhaja iz izsledkov harvardske študije [7], ki pravijo, da je optimalno število ljudi v učinkovitem timu 4-5 in da je takšno tudi optimalno število timov v okviru samostojne koordinacijske enote. Avtor ogrodja se zgleduje po bioloških organizmih in smatra, da se s povezovanjem temeljnih timov v večje strukture in z ustrezno koordinacijo lahko zagotavlja linearna rast učinkovitosti, brez omejitve števila timov.

Ogrodje uvaja dva temeljna procesna cikla – cikel SM-jev ("Scrum Master Cycle") in cikel PO-jev ("Product Owner Cycle").

Skupina razvojnih timov, ki se morajo koordinirati, se imenuje Scrum of Scrums (SoS). Ta skupina ima dnevne sestanke ("Scaled Daily Scrums") za koordinacijo timov in odpravljanje ovir. Vsak SoS ima svojega SM-ja, ki skrbi za proces, in seznam zahtev za odpravo ovir ter koordinacijo. V večjih okoljih se skupine SoS povezujejo v skupine Scrum of Scrum of Scrums (SoSoS).

Poleg te skupine, uvaja Scrum@Scale akcijsko skupino vodij ("Executive Action Team", EAT) – ta je v vlogi SM-ja za celotno organizacijo. EAT ima organizacijske ter finančne vzvode, zato na najvišji ravni odpravlja

ovire, ki se pojavijo v posameznih SoS in jih ti ne morejo odpraviti sami. Naloga tega telesa je tudi, da implementira skupna pravila Scruma znotraj celotne organizacije. EAT ima svojega PO-ja, SM-ja in seznam nalog.

PO-ji in drugi odločevalci se organizirajo v skupine MetaScrum, ki združujejo do 5 timov. V tej skupini definirajo, kaj bo razvito v okviru določenega produkta. Tudi ta skupina ima svojega PO-ja, ki se imenuje glavni PO ("Chief Product Owner"), SM-ja in seznam nalog. Skupine MetaScrum se lahko naprej povezujejo v višje enote MetaScrum. MetaScrum za celotno organizacijo se imenuje izvršni MetaScrum ("Executive MetaScrum", EMS).

3.3 Ogrodje Scaled Agile Framework (SAFe)

Kot zadnje na kratko omenimo še metodološko ogrodje za velike organizacije, ki je med temi precej razširjeno. To je SAFe (Scaled Agile Framework) [8]; po raziskavi [1] ga uporablja 30 % sodelujočih. Za razliko od ostalih navedenih ogrodij, ki zagovarjajo minimalizem, je SAFe ogrodje, ki natančno predpisuje širšo strukturo in procese – od večnivojske strukture organizacije (upravljanje produktov ("product management"), sistemska arhitektura in inženiring, izdaje verzij, produktni in razvojni timi ...), definira specializirane vloge, večnivojske hierarhije zahtev ("solutions", "capabilities", "business epics", "enabler epics", "features", "stories" ...), ocenjevanje njihove poslovne vrednosti, kriterije sprejemljivosti zanje, metrike na različnih nivojih in drugo.

Zaradi svoje obširnosti ter zaradi večnivojske strukturiranosti lahko prinese veliko kompleksnost v organizacijo, če ga ne vpeljujejo ljudje, ki dobro razumejo namen posameznih elementov ogrodja – in prek tega presodijo, ali posamezen element vpeljati ali ne. Sicer je lahko dosežen nasprotni učinek od namenov agilnega, vitkega pristopa [9]. Običajno se za to ogrodje odločajo tiste velike organizacije, v katerih je zahtevana višja stopnja formalnega delovanja in standardizacije.

3.4 Primerjava drugih ogrodij z ogrodjem Nexus

V primerjavi z ostalimi navedenimi ogrodji je ogrodje Nexus:

- bolj osredotočeno na razvojni proces, ne na celotno organizacijo,
- velik poudarek daje planiranju, komuniciranju in razreševanju razvojnih odvisnosti med timi,
- najvitkejše, najbližje osnovnemu Scrumu in ga je zato možno najhitreje vpeljati, če imajo razvojni timi izkušnje s Scrumom.

Organizacije, ki želijo vpeljati skalabilno agilno ogrodje širše, torej ne le v razvoju, se lahko odločajo med LeSS in Scrum@Scale. Med tema ogrodjema je LeSS nekoliko fleksibilnejši in zagovarja eksperimentiranje pri iskanju trajnih izboljšav znotraj organizacije, Scrum@Scale pa prinaša večjo definiranost in strukturiranost. Za velike organizacije z višjo stopnjo standardizacije je primerno ogrodje SAFe, ki je obenem najkompleksnejše za vpeljavo ter najbolj podrobno predpisuje strukturo, vloge in aktivnosti.

4 NAŠ PRISTOP K VPELJAVI OGRODJA NEXUS V RAZVOJNI PROCES

4.1 Začetek projekta – prototipno delo po Scrumu

Pred začetkom razvoja projekta C1 smo identificirali razvojne kapacitete v posameznih podjetjih znotraj skupine, ki bi jih lahko namenili razvoju novega produkta. Pred razvojem smo spisali tudi strateški dokument projekta, v katerem smo zastavili cilje in temeljne usmeritve glede tehnologije, arhitekture, kakovosti, skladno z naravo produkta, kompetencami razvojnih timov in sodobnimi razvojnimi smernicami.

Razvoj projekta C1 se je začel s fazo izgradnje prototipa, s katerim smo želeli potrditi kompetence začetnih skupin in predpostavke v strateških ciljih, obenem pa pripraviti osnovo za morebiten nadaljnji razvoj.

Začetne vloge na projektu je določila operativna skupina projekta, ki je bila formirana za ta namen, sestavljali so jo člani vodstva posameznih podjetij znotraj skupine Solitea in vnaprej določen vodja arhitekture, sicer zunanji strokovnjak.

Kot smo navedli v uvodu, smo začeli z dvema timoma, v Sloveniji in na Češkem. Tim v Sloveniji je bil sistematično vpeljan v delo po ogrodju Scrum, z delavnicami in testnimi iteracijami. Češki tim je imel pomanjkljivo vpeljavo v Scrum, kar se je kasneje pokazalo kot ovira. Podobno glavni PO in glavni SM nista

imela ustreznih znanj ter izkušenj, prej sta bila v vlogah poslovodij oz. projektnih vodij v okviru čeških podjetij. Vendar v fazi prototipiranja zavestno še nismo dajali večjega poudarka sami organizaciji.

Timi se pred začetkom projekta niso poznali in tudi jezik je bil (oziroma do neke mere še vedno je) ovira, saj znanje angleščine na Češkem v splošnem ni tako dobro kot v Sloveniji.

Organizacijo dela in celotnega razvojnega procesa smo vzpostavili v spletnem okolju Microsoft Azure DevOps Services (prej imenovan Visual Studio Team Services oz. VSTS). Organizirali smo seznama zahtev ("Product Backlog" in "Sprint Backlog") ter elektronske table za spremljanje stanj vseh zahtev oziroma njihove realizacije znotraj iteracije ("Sprint Board"). Vzpostavili smo upravljanje napak in samodejno gradnjo izvršnih verzij produkta s predhodnim izvajanjem avtomatskih testov.

Faza začetnega prototipiranja je trajala pol leta.

4.2 Vpeljava ogrodja Nexus

Po uspešno zaključeni fazi prototipa, je vodstvo skupine odobrilo nadaljevanje, torej fazo produkcijskega razvoja, kjer smo več pozornosti namenili tudi organizaciji dela skupin.

V podjetju Saop smo naš Scrum tim okrepili z dodatnimi razvojniki, na skupno 7 članov. Tim so sestavljali: PO, ki je opravljal tudi naloge poslovnega analitika, dodatni analitik, tester za avtomatske teste na nivoju uporabniškega vmesnika ("UI"), trije programerji z različnimi nivoji izkušenosti, od katerih je bil en usmerjen tudi v delo na arhitekturi, ter SM, ki je hkrati sodeloval pri razvoju kot izkušen programer. Testiranje je bilo v domeni vseh vlog, funkcionalno testiranje zlasti v domeni analitikov, testi enot in integracijski testi v domeni programerjev, avtomatski testi na nivoju UI v domeni testerja ter testi sprejemljivosti v domeni PO-ja.

Na Češkem so po prehodu v drugo fazo sprva sodelovali le razvojniki iz podjetja v Pribramu, razdeljeni v dva Scrum tima, nato sta se pridružila še dva tima iz Brna.

Kmalu po prehodu v drugo fazo razvoja smo se trije predstavniki dveh podjetij udeležili delavnice za vpeljavo in delo s procesnim ogrodjem Nexus, »Scaled Professional Scrum with Nexus«, poleg SM-ja in direktorja razvoja iz Saopa še glavni SM iz Češke. Glavni SM je imel nalogo, da uvede ostale češke time v ogrodje Nexus, kar pa se kasneje ni pokazalo kot dovolj uspešno, predvsem zaradi pomanjkanja izkušenj in slabšega znanja angleščine.

V tistem času smo imeli razvojno delo organizirano po horizontalnih nivojih arhitekture; en tim je skrbel za razvoj dinamičnega uporabniškega vmesnika, drugi za razvoj t. im. sistemske platforme, ki je podpirala ostale logične poslovne module oz. storitve, dodatna aplikacijska skupina je bila zadolžena za implementacijo poslovne logike.

Po nekaj mesecih produkcijske faze smo torej imeli na projektu pet Scrum timov. Aktivnosti med timi smo organizirali na podlagi priporočil ogrodja Nexus, ki smo jih povzeli v poglavju 2. Oblikovali smo skupino NIT. NIT so sestavljali predstavniki vsakega od sodelujočih Scrum timov, vanj pa so bili vključeni tudi specialist za "DevOps" ter vodji arhitekture in analitike. Po priporočilih vodnika po Nexusu, so dogodki v okviru NIT vedno vključevali tudi glavnega PO-ja in NSM-ja. Kasneje se je struktura NIT nekoliko spreminjala: dodani so bili predstavniki posameznih vlog v timih (predstavniki t.i. "guildov"), na primer predstavnik načrtovalcev uporabniške izkušnje ter uporabniških vmesnikov in predstavnik testerjev.

Ker projekt pokriva mnogo različnih vsebinskih domen v okviru sistema ERP in ker se je pet timov nahajalo na treh različnih lokacijah, smo se odločili, da bo imel v posameznem timu glavni PO svojega predstavnika – lokalnega PO-ja (LPO), ki bo predstavljal glavnega PO-ja znotraj tima, skrbel za zahteve z domenskih področij tistega tima in urejal seznam zahtev za tim. Glavni PO je moral skrbeti za vsebinsko usklajenost lokalnih PO-jev. Za ustrezno izvajanje procesa v vseh timih je moral skrbeti NSM z lokalnimi SM-ji.

4.3 Trenutno stanje in organiziranost

V skladu z dobro prakso smo v Sloveniji težili k vertikalni organizaciji Scrum timov, kjer bi bil vsak tim sposoben implementirati posamezno funkcionalnost v celoti, torej prek celotnega sklada nivojev: od uporabniškega vmesnika do poslovne logike v storitvah poslovnega nivoja, pa vse do funkcionalnosti na sistemski platformi. Takšna organizacija timov omogoča manj odvisnosti in odpravlja potrebo po čakanju timov na specializiran tim (npr. čakanju aplikativnega tima na platformski tim). Izkušnje iz zgodovinske

delitve na osprednje ("front-end") in zaledne ("back-end") time govorijo v prid vertikalni organizaciji timov, torej timov z vsemi potrebnimi kompetencami za razvoj zahtev prek celotnega sklada nivojev.

Po nekaj mesecih razvoja smo vzpostavili tako organiziranost timov. Toda kasneje so se pokazale pomanjkljivosti pri kompetencah in izkušenosti znotraj nekaterih timov, zaradi česar tovrstna organiziranost ni učinkovito zaživela. Največja težava se je pokazala pri implementaciji zahtevnejših funkcionalnosti na nivoju platforme.

Zato smo se kasneje odločili za začasno vrnitev na horizontalne time. V obdobju enega do dveh let (sedaj smo sredi tega obdobja) želimo manj izkušene člane različnih timov usposobiti, da bodo lahko zastopali svoje področje znotraj vertikalnega tima.

Pomanjkanje transparentnosti s strani posameznih timov, slabše znanje angleščine ter posebej pomanjkanje izkušenj vodilnih vlog na projektu je pogosto vodilo v neusklajeno delo ter preseganje rokov, ki so bili resda ambiciozno zastavljeni. Vodilo je tudi v eskalacije in predloge izboljšav s strani bolj izkušenega, slovenskega tima.

4.4 Priporočila ob vpeljavi ogrodja Nexus

Lastne izkušnje na podlagi dveh let dela s procesnim ogrodjem Nexus lahko strnemo v nekaj priporočil.

Zelo pomembno je, da vodilne vloge na projektu – vloge NSM-ja, glavnega PO-ja, lokalnih PO-jev (kot smo omenili v poglavju 2, teh ogrodje Nexus ne pozna oz. zgolj omogoča PO-ju delegiranje določenih aktivnosti drugim vlogam v timih; vendar so dobra praksa in jih uvajajo tudi druga orodja za skaliranje Scruma) ter SM-jev v posameznih timih – zasedejo ljudje, ki imajo nekajletne praktične izkušnje s Scrumom na uspešno izpeljanih projektih. Tem ljudem priporočamo tudi izvedbo delavnice »Scaled Professional Scrum with Nexus« – ter izvedbo certificiranja, ki sledi delavnici.

Poleg SM-jev v posameznih timih je koristno imeti v vseh timih tudi dovolj razvojnikov z izkušnjami s Scrumom. Vodnik po Nexusu priporoča uvedbo ogrodja le v time, ki so izkušeni s Scrumom. V nasprotnem primeru lahko posamezniki težijo k izvajanju aktivnosti na star, poznan in manj transparenten način.

Scrum je potrebno razumeti kot razvojno procesno ogrodje, ki odpravlja najpogostejše vzroke za težave pri razvojnih projektih. Ker ni vezano na določeno področje razvoja, se ne posveča ostalim potrebnim inženirskim procesom ali tehnikam. Te mora organizacija oz. posamezni tim vzpostaviti okrog Scruma.

Zaradi napačnih interpretacij Scruma in ogrodij za njegovo skaliranje prihaja pri začetnikih pogosto do zmotnih mnenj, da Scrum odpravlja potrebo po arhitekturnem načrtovanju (oziroma da se lahko arhitektura razvije sama od sebe skozi iteracije), da odpravlja potrebo po razvojni dokumentaciji, da je potrebno vse funkcionalnosti načrtovati in razvijati le s pogledom na naslednjo iteracijo in podobno. Zaradi njune vitkosti je mogoče vodnika po Scrumu ter Nexusu razumeti preveč lahkotno in ju napačno interpretirati. Čeprav se vse od začetka povsod omenja izjava, da je Scrum enostavno razumeti, a težko usvojiti, se marsikdo tega zave šele ob težavah na projektu. Zato sta pomembna trajno izobraževanje in izmenjava izkušenj.

Dislociranost timov in še posebej različno znanje skupnega jezika sta lahko dodatni oviri v komunikaciji. Potrebno je poskrbeti za čim boljšo komunikacijsko infrastrukturo (interaktivni zasloni, kamere in zmogljivi mikrofoni) ter tudi za konverzacijske tečaje za skupni jezik.

Pomembni sta tudi jasna opredelitev poslovnih ciljev in merljivih rezultatov. Izvajanje korakov za dosego ciljev mora biti transparentno (pri tem lahko zelo pomagajo sodobna razvojna okolja).

Sestanki, ki jim pravimo retrospektive, so namenjeni ugotavljanju ovir pri delu tima oz. več timov ter skupnemu iskanju rešitev. Seznam aktivnosti za odpravo ovir je koristno voditi kot seznam zahtev na produktu, in jih izvajati osredotočeno ter postopoma – po pomembnosti, ne vseh hkrati.

Deležniki na projektu (vodstvo, sponzorji, domenski strokovnjaki, predstavniki uporabnikov in po možnosti tudi uporabniki) naj z razvojnimi timi aktivno sodelujejo ter jim dajejo povratne informacije.

5 ZAKLJUČEK

Ogrodje Nexus izhaja iz ogrodja Scrum in ga minimalno razširja. Namen Nexusa je zagotoviti učinkovitost pri razvoju produkta z več timi, ob ohranjanju vitkosti ter agilnosti razvojnega procesa. Ogrodje Nexus se ne posveča strukturi ter delovanju širše organizacije, prav tako ne dobrim inženirskim praksam. Oboje morajo organizacije oz. njihovi razvojni timi vpeljati sami.

Zaradi lahkotnosti ogrodja ga je relativno enostavno vpeljati v prakso. Vendar je pri tem potrebno upoštevati priporočila za uspešno vpeljavo. Podobno kot velja pri Scrumu, je tudi Nexus enostavno razumeti in težko usvojiti. Pri vpeljavi tega procesnega ogrodja za usklajeno delo več timov je pomembno, da imajo ljudje v vodilnih vlogah na projektu dovolj izkušenj. Pri dislociranih timih iz različnih držav je koristno, da vsaj predstavniki timov dobro govorijo skupni jezik, sicer se pojavlja ovira tudi v komunikaciji, kar vodi v netransparentno stanje projekta ali v prepozno ugotavljanje in odpravljanje ovir.

Ob ustrezni vpeljavi ogrodja Nexus, pravilni organiziranosti ter dovoljšni izkušenosti lahko timi oz. organizacije pričakujejo učinkovito koordinacijo pri skupnem razvoju, pravočasno zaznavo in odpravo odvisnosti ter trajne izboljšave procesa razvoja.

Ogrodij za skaliranje agilnih procesov v slovenskem prostoru ne zasledimo pogosto v praksi. Zato ob koncu prispevka vabimo tudi druge organizacije, ki so jih vpeljale, da podelijo svoje izkušnje na strokovnih konferencah in interesnih srečanjih ("meetup-ih").

6 LITERATURA

[1] explore.versionone.com/state-of-agile/13th-annual-state-of-agile-report, 13th Annual State of Agile Report, obiskano 8. 5. 2019.

[2] www.scrum.org/resources/scrum-guide, The Scrum Guide, obiskano 8. 5. 2019.

[3] www.scrum.org/resources/nexus-guide, The Nexus Guide, obiskano 8. 5. 2019.

[4] www.thescrummaster.co.uk/scrum/scaled-professional-scrum-and-nexus-nexus-plus, Scaled Professional Scrum and Nexus - Nexus+, obiskano 8. 5. 2019.

[5] less.works/less/framework/index.html, LeSS Framework, obiskano 8. 5. 2019.

[6] www.scrumatscale.com/scrum-at-scale-guide, Scrum at Scale Guide, obiskano 8. 5. 2019.

[7] HACKMAN J. Richard, Leading teams: Setting the stage for great performances, Harvard Business School Press, 2002.

[8] www.scaledagileframework.com, SAFe Framework, obiskano 8. 5. 2019.

[9] www.cio.com/article/2974436/comparing-scaling-agile-frameworks.html, Comparing scaling agile frameworks, obiskano 8. 5. 2019.

OTS 2019
SODOBNE INFORMACIJSKE TEHNOLOGIJE IN STORITVE
M. Heričko in K. Kous (ur.)

Univerzitetna založba
Univerze v Mariboru

PRAKTIČNE IZKUŠNJE PRI AVTOMATIZACIJI RAZVOJA MOBILNIH APLIKACIJ

DAMIJAN RAČEL, ŽAN SKAMLJIČ

Povzetek: *V procesu razvoja mobilnih aplikacij se v določenih korakih od zasnove do razpoložljivosti aplikacije v mobilnih trgovinah izgubi veliko časa z čakanjem, da računalnik zaključi s procesiranjem. Prispevek opisuje optimizacijo korakov, ki so bili prepoznani kot ozka grla. Osredotočili smo se na avtomatizacije testiranja in distribucije testnih ter produkcijskih različic. Predstavili bomo naše izkušnje z različnimi orodji in končni pristop, ki temelji na sistemu git za verzioniranje kode, okolju Gitlab, ki ima vgrajeno podporo za neprekinjeno integracijo in nalaganje (CI/CD), platformi Fabric, preko katere se izvrši distribucija testnih in beta verzij ter skupku orodij fastlane, ki nam omogočajo podpisovanje aplikacij in nalaganje v mobilne trgovine.*

Ključne besede: *git, continuous integration, fabric, avtomatizacija, distribucija, mobilna aplikacija*

NASLOVA AVTORJEV: Damijan Račel, tehnični vodja iOS ekipe, Drugi vid d.o.o., Maribor, Slovenija, e-pošta: damijan.racel@equaleyes.com. Žan Skamljič, mobilni razvijalec, Drugi vid d.o.o., Maribor, Slovenija, e-pošta: zan.skamljic@equaleyes.com.

https://doi.org/10.18690/978-961-286-282-4.15 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 UVOD

V podjetju Equaleyes vedno znova prepoznavamo in se trudimo zmanjšati vpliv ponavljajočih se procesov, s katerimi se srečujemo, ko razvijamo in vzdržujemo aplikacije. Če odkrijemo procese, ki jih moramo pogosto ponavljati, jih poskusimo avtomatizirati, četudi posamezno morda niso časovno zahtevni. Ugotovili smo, da v primerih, ko več razvijalcev dan za dnem opravlja rutinske naloge, ki bi jih lahko avtomatizirali, takšne operacije posegajo v zbranost razvijalcev, ki bi se lahko med tem časom nemoteno osredotočili na zahtevnejše aktivnosti. Ena izmed takšnih nalog je distribucija aplikacije testerjem, stranki in končnim uporabnikom. Izboljšave na tem področju so vodile do definicije poenotenega načina razvoja v podjetju, večinoma za projekte, v katerih so naši razvijalci zadolženi za postavitev in vodenje tega vidika projekta[1].

2 VERZIONIRANJE

2.1 Uporaba orodja Git in GitLab

Za učinkovito delo v ekipi več programerjev in boljši pregled nad zgodovino sprememb se pri vsakem projektu zanašamo na orodje git. S pomočjo git-a lahko tudi prožimo različne akcije, ko npr. razvijalec dodaja ali ureja programsko kodo projekta. Brez git-a bi bilo naše delo zelo oteženo. Gostovanje projektov imamo urejeno na spletni strani GitLab. GitLab smo izbrali zato, ker ponuja hkrati s hrambo kode tudi orodje za neprekinjeno integracijo, GitLab-CI, ki smo ga uporabili tudi za avtomatizirano distribucijo.

2.2 Zahteve za združevanje kode

Ko se razvijalec loti razvoja nove funkcionalnosti, mora narediti novo git vejo iz razvojne veje projekta. Na svoji veji doda vse, kar je zahtevano v specifikacijah, zraven pa napiše še teste za funkcionalnost, ki jo je razvil. Ko zaključi z delom, odpre zahtevo za vključitev svoje programske kode v razvojno vejo projekta. GitLab podpira avtomatično obveščanje ob raznolikih dogodkih, ki se zgodijo v sklopu razvoja projekta. Ustvarjanje zahtevka za združitev kode smo povezali s programom Slack, ki ga uporabljamo za komunikacijo med zaposlenimi v podjetju. V programu Slack smo ustvarili skupine glede na področje specializacije, na primer razvoj iOS aplikacij ali razvoj Android aplikacij. Zahtevki za združitev kode se razvrstijo v primerno skupino, tako da jih vidijo vsi, ki so zadolženi za določeno področje.

Preden se delo na veji zaključi, mora vsaj eden izmed preostalih razvijalcev pregledati programsko kodo in podati svoj komentarje in opozoriti na pomanjkljivosti ali nejasnosti. Te mora razvijalec, ki je zahtevek odprl, odpraviti, preden se lahko koda združi v razvojno vejo.

3 AVTOMATIZACIJA

GitLab in git uporabljamo tudi z namenom, da se ob določenih dogodkih sprožijo avtomatizirani postopki, ki nadomestijo ročna rutinska opravila, ki so jih prej izvajali razvijalci sami. Ker se lahko zgodi, da razvijalec pozabi pognati teste na svojem računalniku, preden želi združiti svoje delo z razvojno vejo, smo morali dodati omejitev, da se lahko koda združi samo takrat, ko so testi uspešno zaključeni. Teste poženemo na strežniku za izgradnjo, običajno ko razvijalec sproži zahtevo po združitvi kode. Ob posodobitvi razvojne veje se sproži distribucija testne verzije preko orodja Fabric. To omogoči takojšnje testiranje med našimi QA inženirji.

3.1 GitLab CI

GitLab služi kot osrednja točka celotnega projekta, kamor vsi razvijalci pošiljajo svojo kodo, in kjer se zgodi kontrola kvalitete in vključi tudi ne-razvijalce. Prednost tega je, da lahko določamo pravila dostopa in distribucije končnega produkta preko Gitlab-a s pomočjo vgrajenega orodja Gitlab CI. Omejimo lahko tudi dostop do glavnih vej projekta in razdelimo razvijalce po vlogah.

Gitlab CI je orodje, ki nudi veliko dodane vrednosti in je v omejeni različici na voljo zastonj. Gitlab uporabljamo tudi zato, da se ni potrebno ukvarjati z dodatnim delom gostovanja na lastnih strežnikih, odločimo pa se lahko tudi za kak plačljiv paket in tako po potrebi dodajamo orodja ki jih rabimo.

Gitlab CI je zelo dobro dokumentiran in je relativno preprost za uporabo. Omogoča poganjanje skript ob spremembah programske kode ali kakšnih drugih dogodkih v našem projektu, ki se nahaja v Gitlabu. Da pričnemo uporabljati Gitlab CI, moramo v projekt dodati datoteko s končnico "yaml". Ta datoteka opisuje vse naloge, ki se bodo pognale, in dogodke, ki bodo delovali kot sprožilci.

```
stages:
  - build
  - test

job 1:
  stage: build
  script: make build dependencies
  only:
    - development

job 2:
  stage: test
  script: make test
```

Primer yaml datoteke, ki definira dve nalogi. Naloga 1 se bo sprožila samo na veji z imenom "development". Seveda Gitlab CI ponuja še mnogo več ukazov za specifične potrebe projektov. [2]

3.2 GitLab runner

Ko želimo pognati teste ali poslati testno verzijo aplikacije, GitLab CI prenese izvorno kodo projekta na oddaljen računalnik, kjer je nameščeno orodje GitLab runner. Runner mora biti ustrezno nastavljen glede na potrebe projekta in teči na pravilni platformi. Strežnik za izgradnjo za iOS zahteva uporabo operacijskega sistema macOS in nameščeno razvojno orodje Xcode, Android projekt pa ni omejen na nobeno specifično platformo. Postopek registracije runnerja je enak tako za iOS kot za Android, le da pri izgradnji iOS aplikacij ni možno uporabiti sistema Docker.

Kot smo že omenili, ta pristop zahteva strežnik za izgradnjo. Odločili smo se za uporabo ločenih strežnikov, torej strežnik za Android in strežnik za iOS. Na strežniku za Android je nameščen operacijski sistem Linux. Dodatno smo namestili tudi git, Android SDK in program gitlab-runner. Gitlab-runner smo registrirali na razvijalno skupino na GitLabu. Dodali smo značke, ki strežnik dodajo v seznam, da ga lahko projekti kasneje uporabijo. Za izvajanje postopkov izgradnje uporabljamo t.i. shell mode, kar pomeni da se ukazi neprekinjene integracije poganjajo neposredno na napravi. Dodatno lahko uporabljamo tudi Docker, kar pomeni da se vsaka naloga izvede v svoji virtualni napravi.

Vsak runner označimo z značkami, ki jih lahko nato uporabimo, da v spletnem vmesniku na GitLab-u izberemo, kateri runner se naj uporabi za določen projekt.

3.3 Samodejna distribucija

Pri razvoju mobilnih aplikacij se hitro pojavi potreba po uporabniškem testiranju, bodisi pri internih inženirjih zagotavljanja kvalitete (Quality Assurance – QA) ali pa za interno testiranje pri stranki. Najbolj osnoven pristop takšne distribucije od razvijalcev zahteva, da zgradijo namestitveno datoteko za mobilno napravo. Namestitveno datoteko je nato potrebno poslati testerju, ki jo mora sam namestiti. Izgradnja namestitvene datoteke lahko traja več minut, in ta čas narašča s kompleksnostjo projekta. Med postopkom izgradnje razvijalec ne more nadaljevati svojega dela. Odvisno od internih zahtev in zahtev naročnika je to potrebno izvajati pogosto, običajno po vsaki spremenjeni, dodani ali popravljeni funkcionalnosti. Takšen pristop od razvijalcev lahko skupno od razvijalcev zahteva tudi več ur na dan.

3.3.1 *Uporaba orodja Fabric*

Prvi korak avtomatizacije je bila distribucija preko orodja Fabric. To razvijalcem omogoči nalaganje namestitvenih datotek na strežnik in tako olajša distribucijo ciljnim osebam. Vsako preneseno datoteko lahko orodje nato samodejno distribuira med eno ali več skupin. V skupine lahko dodamo uporabnike preko e-naslovov. Preko e-naslova nato uporabnik dobi sporočilo, preko katerega si namesti aplikacijo in jo lahko začne uporabljati. Ta pristop pospeši in olajša distribucijo med več skupin testerjev.

3.3.2 *Faze izgradnje*

Distribucija preko orodja Fabric pa od razvijalcev še vedno zahteva ročno izgradnjo. Ta postopek smo avtomatizirali z uporabo orodja za neprekinjeno integracijo GitLab CI. Za vzpostavitev smo potrebovali strežnik, na katerem potekajo zahtevane operacije. Za naše projekte smo definirali več stopenj: izgradnja, testiranje, interna distribucija, distribucija testerjem stranke in objava na trgovinah Google Play in App Store. Za razvoj uporabljamo pristop vejitev git-flow[3].

Takšen pristop nam omogoča hitrejši pregled pred združevanjem vej. Fazi izgradnje in testiranja se poženeta na strežniku. Tako pregledovalcem ni potrebno lokalno kopirati veje in pognati testov, saj že strežnik poroča o uspešnosti le-teh. Pregledovalec mora tako pregledati le ustreznost kode, kar prihrani dodaten čas in pospeši razvoj.

Faza interne distribucije se požene ob združevanju v razvojno vejo. Pri tem koraku se izgradi namestitvena datoteka in se samodejno naloži na Fabric. Med korakom podamo tudi podatek, kateri skupini naj se aplikacija distribuira. Ta korak dodatno razbremeni razvijalce, saj ni več potrebe po ročni izgradnji in distribuciji testerjem. Distribucija testerjem stranke deluje na podoben način, le da se samodejno požene ob združevanju v drugo vejo.

Zadnja izmed faz, distribucija preko Google Play in App Store se požene ob združevanju v glavno (master) vejo. Pri tem koraku se izgradi produkcijska različica aplikacije in se tudi ustrezno podpiše. Po izgradnji se nato prenese na ustrezni strežnik podjetij Google in Apple in se objavi v želenih kanalih (Alpha, Beta, Production). Po potrebi lahko tudi povišamo aplikacijo iz posameznega kanala v višjo stopnjo. Takšna distribucija za razvijalce pomeni, da ni več potrebe po izmenjavi certifikatov, ključev in gesel, saj za to skrbi strežnik.

3.4 Prednosti samodejne izgradnje

Zaradi samodejne izgradnje se tudi zmanjša število napak, ko delamo z aplikacijami, ki si delijo kodo, ampak se razlikujejo po izgledu za več podjetij (t.i. whitelabel), saj se z avtomatizacijo izognemo človeški napaki, kjer bi lahko razvijalec napačno različico poslal napačni končni stranki. To se je izkazalo za posebno koristen pristop pri dveh aplikacijah (Android in iOS), kjer sta aplikaciji imeli skupaj kar 24 različic. Vse te različice se zdaj izgradijo, podpišejo in distribuirajo samodejno, kar razvijalcem prihrani tudi do več ur časa pri vsaki distribuciji.

3.5 Podpisovanje aplikacij

Za podpisovanje aplikacij so zahtevani skrivni ključi in certifikati. GitLab nam omogoča, da več vej označimo kot zaščitene. To nam omogoči, da dodamo na strežniku za izgradnjo več spremenljivk. Na zaščitenih vejah lahko dodamo spremenljivke, ki so običajnim uporabnikom skrite in so dostopne le med poganjanjem na zaščitenih vejah. Med te spremenljivke smo za Android projekt dodali vsebino datoteke z shrambo ključev, geslo datoteke, ime ključa in geslo ključa. Za nalaganje na trgovino Google Play je potrebna tudi datoteka s ključem za dostop do konzole Google Play.

Pri podpisovanju aplikacij za iOS sta za podpisovanje potrebna račun in certifikat. Te podatke hranimo na zasebnem repozitoriju. Ko na projektu začne z delom nov razvijalec, mora le pognati ukaz fastlane match. Pri tem se iz repozitorija z certifikati prenesejo šifrirani podatki in se dodajo v verigo ključev na napravi. Ta postopek traja okoli 30 sekund.

Dodajanje računa in certifikata brez tega postopka zahteva od obstoječih razvijalcev dodajanje novega uporabniškega računa, kreiranje certifikata s pomočjo developer.apple.com in kreiranje razvijalskega profila na napravi. S postopkom avtomatizacije smo tako zmanjšali potrebo po dodatnih razvijalcih pri vzpostavljanju okolja, saj si lahko vsak razvijalec sam uredi vse potrebno za razvoj, brez sodelovanja ostalih ali dodatnih pravic na Applovem računu.

Pri obeh platformah zato več ni potrebe, da bi katerikoli razvijalec imel dostop do podatkov, ki so zahtevani za objavo v produkciji.

Z uporabo zaščitenih spremenljivk tako razvijalcem ni potrebno dodeliti dostopa do ključev in gesel za izgradnjo, podpisovanje in objavo nove različice. Skupaj z zahtevo po pregledovanju zahtev za združevanje tako zmanjšamo verjetnost po zlorabi gesel.

4 PRAKTIČNE IZKUŠNJE

Pri aktivnem razvoju projekta ob istem času na projektu dela več ljudi. To seveda pomeni, da so zahteve po združevanju kode pogoste, ker se večkrat dodajo funkcionalnosti ali popravijo napake. Te spremembe je običajno potrebno testirati. V primerih, ko avtomatizacija ni vzpostavljena, si mora tester sam zgraditi aplikacijo, ali pa jo mora zgraditi eden izmed razvijalcev. Izgradnja aplikacije traja nekaj minut, kar je dovolj da razvijalca in testerja zamoti dovolj dolgo, da izgubita sled o prejšnjem delu.

Pri vzpostavljenem sistemu neprekinjene integracije to od razvijalcev in testerjev ni več zahtevano. Tako razvijalcem prihranimo več minut z vsako izgradnjo, hkrati pa testerjem omogočimo enostavnejšo namestitev. Pri aktivnem razvoju lahko tako skupina prihrani več ur na dan, saj se razvoj lahko nadaljuje, medtem ko strežnik izvaja izgradnjo različice.

5 ZAKLJUČEK

Avtomatizacija procesa za distribucijo aplikacij nam je zelo olajšala delo. V kratkem času smo z željo po izboljšavah spremenili potek dela na vseh projektih v podjetju in naš način dela se še vedno spreminja in prilagaja glede na pridobljene izkušnje in glede na probleme, ki se pojavijo ob vsakodnevnem delu na projektih. Trenutno še nismo dokončno razrešili problema podpisovanja in distribucije iOS aplikacij, saj lahko zdaj vsak razvijalec pošlje novo verzijo aplikacije v testiranje in na Apple TestFlight kar iz svojega računalnika in mu ni potrebno iti skozi celoten postopek preverjanja kode na GitLab-u. To bi lahko rešili z ločenimi certifikati za posameznega razvijalca in omejitvijo dostopa.

Za aplikacije Android je v nekaterih primerih zahtevan podpis namestitvene datoteke s specifičnim ključem. Kadar se knjižnice na to zanašajo (npr. Facebook login) je potrebno v konzolo dodati SHA1 vrednosti ključev. To lahko povzroča težave pri razvoju, kadar razvijalec testira funkcionalnost in njegovega ključa ni v konzoli. Enaka težava se pojavi kadar namestitveno datoteko izdela strežnik, saj se za razvijalne načine ne uporabljajo enaki ključi kot za izdajo aplikacije. To je mogoče rešiti tako, da v repozitorij dodamo datoteko za podpisovanje, ki se nato uporablja za podpisovanje razvojnih različic. Prepričani smo, da bomo uspeli pristop v prihodnje še izboljšati in pohitriti, avtomatizirati še druge rutinske stvari in tako pridobiti čas za razvoj.

6 LITERATURA

[1] The Pragmatic Programmer: From Journeyman to Master, 1999, Chapter 8: Pragmatic Projects

[2] https://docs.gitlab.com/ee/ci/README.html, dokumentacija za Gitlab CI

[3] Introduction to Git-flow: A Git Workflow and Development Model, 2014

OTS 2019
SODOBNE INFORMACIJSKE TEHNOLOGIJE IN STORITVE
M. Heričko in K. Kous (ur.)

Univerzitetna založba
Univerze v Mariboru

IMPLEMENTACIJA PROGRAMSKIH REŠITEV S POMOČJO INTELIGENTNIH ASISTENTOV

MITJA GRADIŠNIK, TINA BERANIČ, SAŠO KARAKATIČ

Povzetek: *Razvijalci neprestano iščejo nove pristope in prakse, s katerimi bi optimizirali in pohitrili razvoj programske opreme. V zadnjih letih postaja vse bolj zanimiva vpeljava pristopov podpore procesu razvoja programske opreme z metodami umetne inteligence. Z umetno inteligenco podprta orodja nadgradijo preproste avtomatizacije ponavljajočih se trivialnih opravil, z avtomatizacijo intelektualno zahtevnejših delov, ki razbremenijo razvijalce. Trenutno je na trgu mogoče najti različna s strojnim učenjem podprta orodja, ki razvijalcem nudijo pomoč v vseh fazah razvojnega cikla programske opreme. V prispevku smo se osredotočili na fazo implementacije in rešitve podprte s strojnim učenjem, ki razvijalcem v realnem času svetujejo glede optimalne implementacije programske opreme ter uporabe dobrih praks. Trenutno razpoložljivi inteligentni asistenti so nadgradnja obstoječih orodij za samodokončanje programske kode, ki predlagajo bolj celostno in kontekstu primerno dopolnitev programske kode. Pregledali in analizirali smo izbrane inteligentne asistente razvijalca za različne programske jezike in razvojna okolja. Poleg praktičnega preizkusa rešitev prispevek predstavlja tudi podroben opis delovanja algoritmov strojnega učenja, ki so ključni del tovrstnih inteligentih sistemov.*

Ključne besede: *inteligentni asistenti programiranja, umetna inteligenca, programsko inženirstvo, nevronske mreže, code2vec*

NASLOVI AVTORJEV: Mitja Gradišnik, strokovni sodelavec, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Maribor, Slovenija, e-pošta: mitja.gradisnik@um.si. dr. Tina Beranič, asistentka, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Maribor, Slovenija, e-pošta: tina.beranic@um.si. dr. Sašo Karakatič, docent, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Maribor, Slovenija, e-pošta: saso.karakatic@um.si.

https://doi.org/10.18690/978-961-286-282-4.16 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 UVOD

Tudi na področju procesov razvoja programske opreme je mogoče zaznati vse bolj razširjeno uporabo metod in pristopov umetne inteligence. Te je mogoče zaslediti v okviru različnih orodij, ki ponudijo podporo različnim aktivnostim v okviru standardnih faz procesa razvoja programske opreme. Proces razvoja programske opreme zajema več korakov, katerih skupen cilj je uspešna implementacija delujoče programske rešitve, ki temelji na podanih zahtevah naročnika. Razvojni cikel programske opreme vključuje več zaporednih faz, od zajema poslovnih zahtev, ki mu sledi načrtovanje ter implementacija in testiranje rešitve, zadnji fazi pa predstavljata nameščanje ter vzdrževanje razvite programske rešitve [20]. Na umetni inteligenci temelječa orodja je mogoče zasledili v okviru vseh faz razvojnega cikla, pri čemer pa podprtost nekaterih faz in aktivnosti znotraj njih še posebej prednjači. Takšen primer je faza implementacije programskih rešitev, kjer omenjena orodja ponudijo podporo razvijalcem na najrazličnejše načine, od nasvetov uporabe dobrih praks pa vse do priporočil uporabe različnih vzorcev. Orodja razvijalcem svetujejo v realnem času, pri tem pa gradijo v smeri optimizacije implementacije programskih rešitev.

Umetna inteligenca je krovni in posplošen izraz za več družin pristopov, ki se med seboj pomembno razlikujejo. Področje je v grobem mogoče razdeliti na dve veji, šibke in močne pristope umetne inteligence [20]. Šibke pristope umetne inteligence predstavljajo tehnike predstavitve znanja, sklepanje na podlagi poslovnih pravil, metode razpoznave slik in govora ter tehnike avtomatizirane obdelave naravnega jezika. Na drugi strani, avtonomno učenje na podlagi podanih podatkov temelji predvsem na močnih pristopih, in sicer strojnem učenju. Pri vpeljavi v proces razvoja programske opreme je ključna kombinacija obeh vej, saj je mogoče z uporabo močnih pristopov doseči zmožnost avtomatizacije intelektualno zahtevnih opravil preko strojnega učenja, medtem ko pa šibki pristopi avtomatizirajo strojno ekstrakcijo podatkov iz dokumentov s pomočjo procesiranja naravnega jezika ali razpoznave elementov iz podanih slik.

Uporaba z umetno inteligenco podprtih orodij pa od sodelujočih zahteva pomembne spremembe. Z preseganjem omejitev, ki jih prinašajo vnaprej kodirana pravila reševanja problemov zahtevajo miselni premik od do sedaj uveljavljenih pristopov. Izdelava pravil in strategij reševanja problemov je v rokah orodij, kar omogoči pomembno zmožnost prilagajanja različnim kontekstom uporabe. Dostopna so številna z umetno inteligenco podprta orodja [27], ki naslavljajo zelo širok nabor aktivnosti v okviru razvojnih faz programske opreme. Bodisi gre za področja identifikacije natančnejših zahtev projekta, avtomatiziranega generiranje programske kode ali identifikacijo hroščev in napak, so na trgu na voljo številna [20], zaenkrat še v večini prototipna, orodja.

Analiza in zajem poslovnih zahtev velja za pomembno področje, ki v veliki meri vpliva na uspešnost projekta v razvoju. Orodja, temelječa na metodah umetne inteligence v okviru te faze, skušajo z metodami obdelave naravnega jezika procese skrajšati, poenostaviti in jih naredili natančnejše. Primer takšnega pristopa iz opisov sistema v naravnem jeziku izlušči zahteve z uporabo konvolucijske nevronske mreže in sisteme klasificirali glede na zahteve [28]. Še en primer inteligentnih orodij je spletna platforma Bookmark [14], namenjena gradnji spletni strani, ki ponuja rešitev Artificial Intelligence Design Assistant. Ta z umetno inteligenco razume uporabnikove želje in potrebe ter na osnovi tega ustvari prototipno verzijo spletne strani.

Načrtovanje in implementacija, kot osrednji del procesa razvoja, združuje skupek aktivnosti, ki se končajo z izvršljivo programsko kodo. V okviru faze načrtovanja je mogoče uporabiti pristop [25], v katerem se orodje, ki uporablja strojno učenje, nauči skupek specifičnih gest, ki se lahko uporabijo za načrtovanje aplikacij namenjenih za naprave na dotik [30]. Prav tako je mogoče pri pretvorbi slike uporabniškega vmesnika v kodo uporabniškega vmesnika uporabiti nevronske mreže [26], kar podpira tudi sistem SketchCode, ki iz skice zaslonske maske avtonomno generira HTML in CSS programsko kodo [18]. Prav tako je raziskava pokazala, da je mogoče z analizo programske kode s pomočjo nevronski mrež ugotoviti delovanje kode in jo še boljše opisati [29]. Pri tem se uporabi arhitektura nevronske mreže Code-NN, ki ustvari vektorski zapis kode, ki se na to primerja z znanimi opisi kod in pri opisi generira opise iz dobljenih vektorjev. Na drugi strani pa sistem Semantic Search [15] podpira iskanje po kodi s pomočjo naravnega jezika. Na podlagi predhodnih raziskav, tudi v tej raziskavi uporabijo dvojno kodiranje v vektorsko obliko - tako besedila (iskalnega niza) kakor tudi programske kode. Nenazadnje pa pomembno podporo fazi implementacije predstavljajo inteligentni asistenti razvijalcev, ki le tem svetujejo na osnovi dobrih praks in vzorcev. Inteligentni asistenti neprestano spremljajo in opazujejo delo razvijalca ter se ob tem tudi učijo, razvijalcem pa lahko tako ob podobnih situacijah ponudijo optimalno rešitev.

Tudi testiranje programskih rešitev ni izjema iz vidika podpore različnih orodij temelječih na metodah umetne inteligence. Orodje Functionize [16] omogoča testiranje v oblaku podprto z umetno inteligenco. Uporabnik v besedilni obliki vnese plan testiranja, ki se avtomatsko pretvori v testni primer [19]. Applitools [17] omogoča testiranje grafičnih uporabniških vmesnikov z emulacijo človeških oči in tehnologijo primerjave slik s pomočjo umetne inteligence. Omogočeno je pregled in zaznavanje nepravilnosti kot na primer postavitve spletnih strani, omogočeno pa je tudi preverjanje obnašanja spletne strani [19]. Prav tako velja izpostaviti sistema SapFix in Sapienz [21], pri čemer slednji uporablja umetno inteligenco za avtomatsko testiranje in razhroščevanje kode. Nadgradnja tega sistema, sistem SapFix, pa iz najdenih pomanjkljivosti predlaga popravke na osnovi preteklih sprememb v programski kodi organizacije.

Prispevek se v nadaljevanju osredotoča na fazo implementacije, natančneje na inteligentne asistente razvijalcev. Tako poglavje 2 predstavi omenjeno področje in praktične primere dostopnih orodij, poglavje 3 pa podrobneje naslovi delovanje inteligentnih asistentov.

2 INTELIGENTNI ASISTENTI RAZVIJALCEV

2.1 Potreba po inteligentnih asistentih

Sodobni pristopi v programskem inženirstvu, razvoj naprednih razvojnih orodij in programskih jezikov ter nenazadnje kontinuirane izboljšave v strojni opremi nam omogočajo, da gradimo čedalje kompleksnejše informacijske rešitve, s katerimi uspemo uspešno naslavljati izzive v sodobnih poslovnih okoljih. Neprestano povečevanje kompleksnosti programskih rešitev so nenazadnje zaznale mnoge raziskave [1]. S ciljem po obvladovanju nastale kompleksnosti informacijske rešitve razbijamo programske rešitve v neodvisne podsisteme oz. module. Če čas postanejo takšni sistemi prepleteni z odvisnostmi med moduli, prav tako postane kompleksna integracija posameznih nivojev med zalednimi sistemi in uporabniškim vmesnikom. Ker je upravljanje odvisnosti zaenkrat povsem v domeni razvijalec, lahko takšno stanje hitro pripelje do nekonsistentnosti sistema ali nerešljivih programskih napak [6].

Razvijalci programskih rešitev se torej pogosto znajdejo v okoljih, v katerih se morajo za nadgradnjo funkcionalnosti ali popravek programske napake prebiti skozi vrstice programske kode, ki po možnosti niti niso napisane v programskem jeziku, v katerem bi se počutil domače. Zaradi hitrega tempa razvoja, kot smo ga vajeni v IT sektorju, ne gre pričakovati, da bodo razvijalci lahko ves čas delali na izključno na njim lastni programski kodi, saj je okolje izredno dinamično in potrebuje visoko stopnjo prilagodljivosti razvijalcev.

Da bi zmanjšali obremenitve razvijalcev, ki se pojavljajo kot posledica kompleksnosti programskih rešitvami in hitrega tempa razvoja, iščemo nove pristope in orodja, ki bi lahko razvijalcem pri njihovem delu učinkovito pomagale. V iskanju rešitev problema je potrebno natančno razumeti naravo dela razvijalcev programskih rešitev. Ključna naloga razvijalcev je pretvarjanje zahtev naročnikov v delujočo programsko kodo. Pri tem namenjajo velik del svojega časa razhroščevanju programske kode ter prebiranju najrazličnejših dokumentov in virov [3]. Proces pisanja programske zahteva veliko analitičnega razumevanja problemskega področja in iskanje optimalne rešitve za opazovan problem. Razvijalec torej ne le da piše programsko kodo, ampak tudi sprejema načrtovalske odločitve ter poskuša ugotoviti, kakšna programska koda bo za dano situacijo najbolj optimalna. Poleg lastnih izkušenj je so primarni viri informacij, na katere se razvijalci upirajo med razvojem, dokumentacija tehnoloških rešitev, s katerimi delajo, ter primeri programske kode, ki jo je mogoče najti v javnih repozitorijih kode ali spletnih skupnostih, kot sta na primer GitHub ali StackOverflow. Programerji se pogosto zgledujejo po programski kodi, ki so jo pri reševanju podobnih programskih problemov ustvarili drugi razvijalci. V osnovi gre za souporabo kolektivnega znanja, ki pomaga razvijalcem hitreje pisati boljšo kodo. Kolektivni repozitoriji znanja razvijalcem prihranijo čas, ki bi ga porabili, če bi se do rešitve problema dokopali samostojno s študijo razpoložljive dokumentacije in virov [4].

Da bi bila razbremenitev razvijalcev s pomočjo namenskih orodij učinkovita, morajo biti orodja zmožna učinkovito prevzeti del nalog razvijalca. Ker je delo razvijalcev primerno povezano z intelektualno zahtevnimi opravili, se velika priložnost za preboj na tem področju vidi ravno vpeljavi pristopov umetne inteligence v proces razvoja programske opreme. V praksi bi vpeljava inteligentih asistentov razvijalca zmanjšala potrebo razvijalca, da brska za informacijami po spletu ali spletnih skupnostih [4].

2.2 Kaj ponujajo inteligentni asistenti razvijalcev?

Področje strojnega učenja je v zadnji letih doživelo bliskovit razvoj, kar daje možnost razvoja različnim področjem, ki uspejo sodobne pristope strojnega učenja vključiti v inovativne rešitve. Po ocenah analitikov bodo z umetno inteligenco podprta orodja do leta 2021 ustvarila 2,9 milijarde USD [9]. Vendar je potrebno napredek, ki ga prinaša vpeljava pristopov strojnega učenja v proces razvoja programske opreme, bolj razumeti v evolucijo obstoječih pristopov in ne revolucije, ki bi čez noč v svojih temeljih spremenila pristope razvoja informacijskih rešitev. Imenu navkljub trenutni pristopi in zmožnosti stojnega učenja ne bodo povsem zamenjali razvijalcev ali kar povsem avtomatizirali celoten proces razvoja programske opreme. Možnosti vpeljave stojnega učenja se tako kažejo predvsem v vpeljavi inteligentnih asistentih programerja, naloga katerih je spremljati delo programerja in mu sproti pripravljati predloge, ki bi jih ta pri svojem delu lahko potreboval. Z uporabo tovrstnih inteligentnih asistentov razvijalca se bi naj občutno zmanjšala potreba po prekinjanju dela, da bi razvijalec na primer v dokumentaciji poiskal pravilen način uporabe nekega razreda, na katerega je naletel pri programiranju.

Pristop vpeljave inteligentnih asistentov programerjev bi najlažje primerjali s programiranjem v paru, ki velja za uveljavljeno prakso pri nekaterih agilnih metod razvoja programske opreme. V splošnem gre pri programiranju v paru za pristop, pri kateri za isti monitor posedemo dva razvijalca, ki se s skupnimi močmi lotevata razvijalskih izzivov. Programiranje v paru je sicer preizkušena metoda, ki učinkovito spodbuja izmenjavo veščin med razvijalcema, sprotnemu preverjanju kakovosti napisane kode in spodbujanju sodelovanja med razvijalci [4]. Kljub prepoznanim in v praksi potrjenim številnim prednostim, ki jih opisana praksa razvoja programske opreme prenaša, so mnenja glede smotrnosti njene vpeljave deljena. Pogosto se namreč pojavljajo pomisleki o enakomernosti in enakopravnosti vključenosti obeh razvijalcev v razvoj programske kode, še večjo težavo prestavlja produktivnost takšne ekipe, ki se praktično prepolovi, saj dva razvijalca istočasno delata na enem opravilu. Razvojne skupine se posledično odločajo za vpeljavo programiranja v paru izključno, če povečana kakovost nastale programske kode resnično odtehta stroške povezane z zmanjšano produktivnostjo razvijalcev.

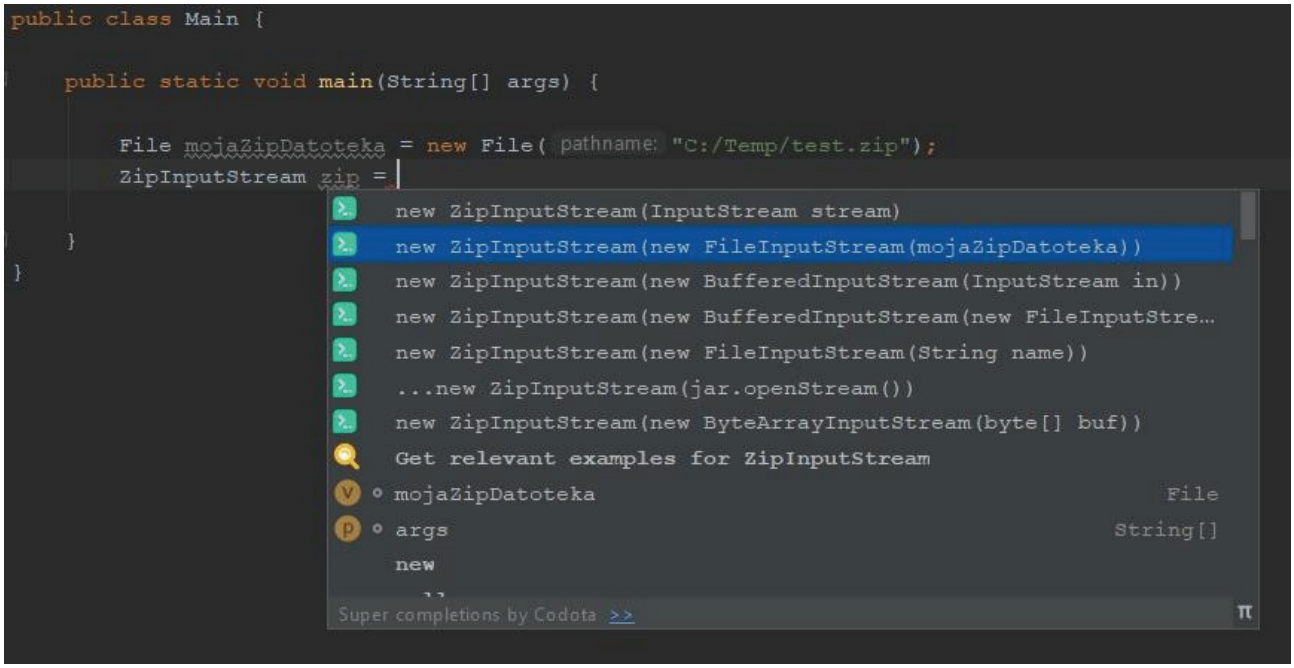
Poskusi razvoja in vpeljava inteligentnih asistentov razvijalca v proces razvoja programske opreme skušajo ohraniti prednosti pristopa programiranja v paru, hkrati pa zmanjšati stroške povezane s produktivnostjo razvijalcev ter pomisleke o neenakomernem prispevku obeh članov para. Enega izmed razvijalcev v paru zamenja inteligentni asistent, katerega naloga je konstanto spremljati razvijalca, poskušati razumeti njegovo delo v kontekstu in mu na podlagi podatkov iz različnih zunanjih spletnih virov ponujati predloge s čim bolj optimalnimi rešitvami problemov, na katere razvijalec naleti med razvojem.

Vpeljava strojnega učenja nam torej daje priložnost, da povečamo zmožnost posameznih razvijalcev s praktično neomejenim znanjem, ki ga lahko črpamo iz interneta [4]. Seveda je uporaba pristopov strojnega učenja na tej točki nujna, saj je količina programskih problemov in njihovih rešitev na drugi strani enostavno preveč, da bi jih zapisali v orodja v obliki pravil, kot to počnemo v do zdaj uveljavljenih razvojnih orodjih pri funkcionalnostih kot sta preoblikovanje programske kode in samodopolnjevanje programske kode.

2.3 Inteligentni asistent Codota

Zmožnosti rešitve Codota presegajo zmožnosti vtičnikov za samodopolnjevanje programske kode na podlagi vnaprej programiranih pravil, ki ga po večini ponujajo sodobna integrirana razvojna orodja, hkrati pa od inteligentnega asistenta v tej fazi ne moremo pričakovati samodejnega in povsem avtomatiziranega generiranja programske opreme na podlagi v naravnem jeziku podanih zahtev, kot bi lahko morda zaradi omembe umetne inteligence v povezavi z orodjem napačno sklepali. Orodje Codota se torej umešča nekje vmes med obema opisanima poloma. Naloga inteligentnega asistenta Codota je spremljati delo programerja ter mu ponuditi povsem delujoče bloke programske kode, ki bi jih lahko razvijalec lahko glede na kontekst uporabil v naslednjem koraku razvoja programske kode. Običajno predstavljajo ponujeni bloki kode sestavljene klice API-ja. Ker rešitve programskih problemov, ki jih ponuja Codota, niso vnaprej programirane v orodje in ker orodje spremlja in skuša razumeti kontekst, v katerem je bila neka programska kode napisana, so lahko ponujeni bloki programske kode bistveno bolj specifični in skladni kontekstom problema, ki ga rešujejo. Prilagajanje že napisani programski kodi razvijalcu preprečuje, da bi postal ujet v slog pisanje programske kode, ki ga narekuje orodje [2, 4, 5].

Inteligentni asistent Codota je za zadaj celostno podpira le programski jezik Java, sicer pa je v beta fazi testiranja podpora programskemu jeziku JavaScript. Dolgoročno se obeta tudi podpora programskemu jeziku C#. Orodje je na voljo v obliki vtičnika za priljubljena integrirana razvojna orodja, in sicer IntelijJ, Android Studio in Eclipse. Orodje se z nameščanjem v IDE enostavno integrira v razvojni proces in je razvijalcem pri roki ob vsakem času, ko bi jih ta lahko potreboval. Inteligentni asistent Codota črpa znanje, ki ga uporabi za generiranje predlogov blokov programske kode, iz treh velikih spletnih baz znanja, in sicer javnih repozitorijev projektov, ki domujejo na storitvah GitHub in BitBucket, in spletnega skupnosti razvijalcev Stack Overflow [10]. Primer uporabe inteligentnega asistenta Codota v orodju IntelliJ Idea prikazuje slika 1.



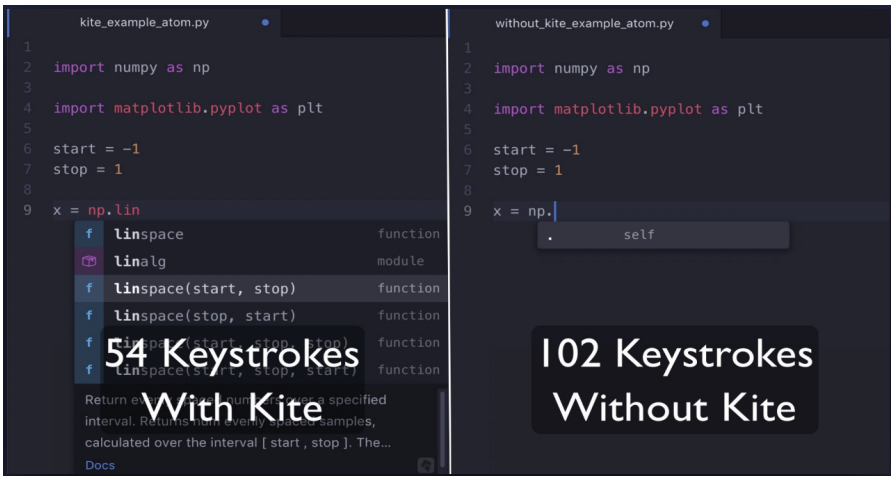
Slika 1: Codota inteligentni asistent v orodju IntelliJ Idea.

Seveda pa ima orodje v trenutni fazi razvoja tudi številne omejitve [5]. Čeprav Codota neprestano spremlja delo razvijalca in skuša razumeti kontekst napisane programske kode, končnega cilja programske kode ne pozna. Ker ne razume funkcionalnosti, na kateri dela razvijalec, in s tem nima širše slike programskega problema, lahko predlaga zgolj naslednji statistično najverjetnejši korak. Pogosto se zgodi, da je v naslednjem koraku mogoče uporabiti več blokov programske kode, zaradi česar ponudi Codota razvijalcu več alternativnih blokov programske kode, med katerimi razvijalec izbere tistega, ki mu najbolj ustreza. Seveda je povsem mogoče, da razvijalcu ne bo ustrezal nobeden izmed ponujenih blokov programske kode. Ker je proces učenja orodja Codota povsem avtomatiziran in nenadzorovan, je povsem možno, da se bo med predlaganimi izseki kode znašel tudi kakšen, ki ni povsem skladen z dobrimi praksami ali pravilu pisanja kode, ki ga predpisuje podjetje. Vsa odgovornost za izbiro ustrezne rešitve je še vedno v celoti na plečih programiranja veščega in dobro usposobljenega razvijalca. Trenutna zrelost samega orodja predstavlja naslednjo omejitev, zaradi katere bi lahko bil malokateri razvijalec, ki glede na poimenovanje inteligentni asistent pričakuje preveč, nad orodjem razočaran. Orodje je v tej fazi odlično pomagalo pri pravilnem sestavljanju klicev API-ja v ustrezen vrstni red, manjka mu pa zmožnost predlaganja kompleksnih blokov kode, kot so na primer regularni izraz ali implementacije določenih pogosto uporabljenih algoritmov. Je pa zagotovo razreševanje tovrstni predlogov v agendi razvoja orodij, saj so tovrstni problemi z uporabo pristopov strojnega učenja zagotovo rešljivi.

Zasebnost in intelektualna lastnine uporabnikov rešitve Codota je v celoti zaščiteno, saj se inteligentni asistent uči zgolj na primerih programske kode iz javno dostopnih repozitorijev in storitev, programske kode odjemalcev pa ne prenaša nikamor [2]. Proti plačilu je sicer mogoče v proces učenja asistenta vključiti tudi obstoječo lastniško kodo, vendar so predlogi kode iz lastniške kode dostopni zgolj dotičnemu odjemalcu in ne tudi ostalim uporabnikom orodja.

2.4 Inteligentni asistent Kite

Če se inteligentni asistent Codota v tem trenutku primarno posveča razvijalcem programskega jezika Java, je inteligentni asistent Kite namenjen razvijalcem, ki razvijajo programske rešitve s programskim jezikom Python. Programski jezik Python zaseda pomembno mesto med programskimi jeziki, sej velja za programski jezik, ki v zadnjih letih občutno pridobiva na priljubljenosti [8] in tako postaja pomemben igralec med programskimi jeziki. Vzporedno s rastjo razširjenostjo in pomembnostjo se bo čedalje več razvijalcev odločajo, da jezik uporabijo pri svojih projektih. Inteligentni asistenti, kot je na primer Kite, odigrajo odločilno vlogo pri mehčanju prehoda razvijalca na nov programski jezik, saj programskega jezika in njemu pripadajočega API-ja neveščemu razvijalcu s predlogi relevantnih blokov programske kode bistveno olajšajo in pohitrijo programiranje. Na sliki 2 je mogoče videti primer z izsekom programske kode napisane v programskem jeziku Python, katera terja od razvijalca 102 pritiska na tipkovnica. Če razvijalcev uporabi inteligentni asistent Kite, je za isto programsko kodo potrebnih le 54 pritiskov na tipkovnici [11]. Podobno kot v taboru inteligentnega asistenta Codota se tudi v taboru orodja Kite spogledujejo s širitvijo na preostale trenutno aktualne programske jezike. Tudi rešitev Kite je na voljo v obliki vtičnika, ki ga enostavno integriramo v integrirano razvojno orodje, kje je razvijalcem vedno na dosegu roke. Trenutno podprta orodja so Atom, PyCharm/IntelliJ, Sublime Text, Microsoft Visual Studio Code in Vim [11].



Slika 2: Primerjava števila pritiskov brez in z inteligentnim asistentom Kite.

Orodje Kite za učenje uporablja programsko kodo, ki je objavljena v javno objavljenih repozitorijih storitve GitHub. Učenje inteligentnega sistema sicer ne poteka neposredno na analizirani programski kodi projektov, temveč se programska kode pred fazo analize in učenja najprej pretvori v abstraktno sintaktično drevo, kar omogoča inteligentnemu asistentu boljši in natančnejši model učenja rešitev brez nepotrebnega šuma, ki ga vnaša sintaksa programskega jezika v tekstovni obliki [7].

2.5 Inteligentni asistent IntelliCode

Razvijalcem, ki so bližje tehnologije Microsoftovega tabora, bo zagotovo v pomoč Microsoftov inteligentni asistent IntelliCode, ki ga je mogoče kot vtičnik namestiti v integrirano razvojni orodji Visual Studio [13] ter Visual Studio Code. IntelliCode ponuja razvijalcem pomoč pri programiranju s predlaganjem blokov programske kode, ki bodo glede na kontekst programske kode najverjetneje relevantni. Pomoč, ki jo ponuja je za zdaj osredotočena na priporočila glede nadaljnjih klicev API-ja, izbire tipov, samodolopnjevanja listov parametrov ter razbiranje pravil in konvencij stila programiranja na podlagi zapisane programske kode [12].

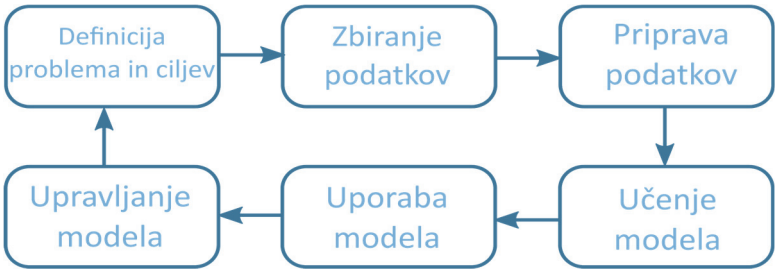
Inteligentni asistent IntelliCode sicer primarno cilja na razvijalce, ki razvijajo programske rešitve na Microsoftovih tehnologijah v programskih jezikih C#, C++ in JavaScript/TypeScript. Od primarno podprtih jezikov je v produkcijski fazi podpora programskemu jeziku C#, programska jezika C++ ter JavaScript/TypeScript pa se trenutno nahajata v "preview" fazi. Sicer pa so razvijalci inteligentnega asistenta so trenutno korak pred konkurenčnimi produktoma Codota in Kite. IntelliCode v tem trenutku ponuja razširitev s podporo programskima jezikoma Python in Java, ki sta sicer prav tako na seznamu podprtih programskih jezikov integriranega razvojnega okolja Visual Studio Code [13].

Učenje inteligentnega asistenta je temeljilo na več kot tisoč javnih projektih iz GitHuba, pri čemer so bili v učno množico asistenta vzeti le dobro ocenjeni projekti z oceno več kot 100 zvezdic [14]. S tem korakom so se razvijalci inteligentnega asistenta skušali izogniti učenju iz slabih praks. Učenje v splošnem poteka na javno dostopnih repozitorijih GitHuba, lastniška koda uporabnikov pa ni vključena v proces učenja. Sicer pa platforma inteligentnega asistenta IntelliCode omogoča tudi učenje asistenta na podlagi lastne programske kode. Na podlagi lastne programske kode si lahko inteligentni asistent izdela personalizirane modele specifične za posameznega uporabnika sistema, zaradi so njegova priporočila razvijalcem še bolj relevantna. Izdelava personaliziranih modelov je smiselna predvsem v primerih, ko so v kodi uporabljeni lastniški pomožni in temeljni razredi knjižnic ali druge domensko specifične knjižnice, ki jih sicer v odprtokodnih projektih ne zasledimo prav pogosto [12]. Platforma inteligentnega asistenta sicer omogoča deljenje ustvarjenih personaliziranih modelov med uporabniki inteligentnega asistenta.

3 INTELIGENTNI ASISTENTI POD POKROVOM

Ključna prednost vpeljave pristopov strojnega učenja v orodja za podporo razvoju programske opreme je zmožnost obvladovanja in avtomatizacije kompleksnejših opravil, kot to omogočajo do zdaj uveljavljena razvojna orodja. Trenutno uporabljeni pristopi gradnje orodij terjajo ekspliciten način zapisa obnašanja orodja v obliki pravil. Takšna orodja torej zmorejo le izvesti opravila, ki so vnaprej kodirana v orodju. Za primer vzemimo recimo nadgradnjo programskega jezika. Razvijalci lahko začnejo uporabljati novo vpeljane koncepte programskega jezika komaj, ko si posodobijo razvoja orodja, ki so prilagojena spremembam sintakse nove različice jezika. Pristop gradnje orodij z vnaprej kodiranimi pravili obnašanja hitro trči ob omejitve povezane z zmožnostjo doseganja kompleksnosti takšnih pravil. Pri programiranju namreč naletimo na cel kup opravil in odločitev, ki so preveč kompleksna, da bi jih kodirali v obliki eksplicitnih pravil [6].

Vpeljava strojnega učenja v orodja razvoja programske opreme pomeni odmik od ustaljenega kodiranja obnašanja orodij s pravili obnašanja. Razvijalci orodij se več ne ukvarjajo s pravili, temveč iskanje poti do rešitve nekega problema prepustijo algoritmom strojnega učenja. Algoritmom strojnega učenja je potrebno zagotoviti dovolj veliko učno množico, iz katere samostojno izpeljejo pravila za reševanje problemov oz. opravil s tem, da si izgradijo model reševanja problemov. Osnova razvojnih orodij, ki temeljijo na strojnem učenju, so jasno definirani cilji, ki bi jih takšno orodje naj reševalo. Na podlagi ciljev se nato zberejo kakovostni učni podatki. Ker so ti pogosto pridobljeni iz različnih virov in so v različnih formatih, jih je pred uporabo potrebno pogosto prečistiti in transformirati v ustrezno obliko, ki je skladna z izbranimi algoritmi strojnega učenja. Kakovostno pripravljeni podatki predstavljajo osnovo za učenje model, ki so uporabljeni kot jedro namenskih orodij. V želji po neprestanem izboljševanju in prilagajanju novim primerom seveda model neprestano izboljšujemo tako, da celotno fazo razvoja modela iterativno ponavljamo nad izboljšano množico učnih podatkov. Celoten življenjski cikel razvoja modela prikazuje slika 3.



Slika 3: Življenjski cikel razvoja modela umetne inteligence.

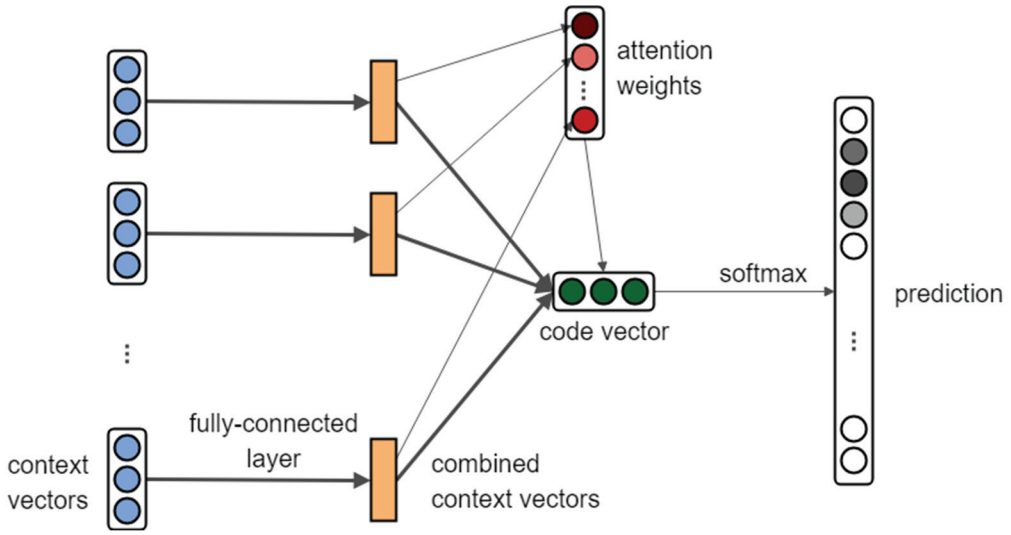
3.1 Umetna inteligenca v inteligentnih asistentih

Inteligentni asistenti za pomoč pri programiranju torej delujejo na podlagi pregledane ogromne količine programske kode in iz tega iščejo vzorce za nadaljevanje ali prilagoditev pisanja kode. Da lahko stroj (računalnik) preuči programsko kodo, jo je potrebno najprej zapisati v taki obliki, ki ima računalniku smisel. Če računalnik obravnava programsko kodo le kot niz znakov, bi ga lahko zmedlo že drugačno zaporedje ukazov, drugačno poimenovanje spremenljivk, stil pisanja, komentiranja in dokumentiranja kode ter sama struktura razredov, metod in zank. Kakor ljudje lahko poiščemo podobne razrede ali metode, pa tudi, če so te zapisane v drugačnem stilu, želimo to sposobnost omogočiti tudi računalniku.

S podobnim problemom so se srečali že pri prosto zapisanem besedilu, kjer lahko na nešteto različnih načinov zapišemo odstavek, ki opisuje en pojem. V ta namen se že na področju obdelava prostega besedila uporabljajo kodirana besed, povedi, odstavkov in drugih vrst besedila v vektorje števil - temu pravimo vložitev besed (angl. word embeddings). S postopkom vektorizacije besede ustvarimo vektor števil med 0 in 1, kjer vrednosti števil v vektorju povedo semantično vsebino besede. Vsako število v vektorju nosi nek pomen o besedi, še več pa lahko o besedi izvemo, ko gledamo kombinacije več števil iz vektorja. Posebnost takega pristopa je, da človek ne sodeluje pri procesu odločanja, kakšen pomen ima katero število v vektorju - tega izbere računalnik sam in človeku ni neposredno jasen. V zadnjem času je najpogostejša tehnika vektorizacije besed mehanizem, ki se imenuje word2vec, ki so ga razvili pri Google-u [23], ki deluje na sledeč način. Uporabi se plitka nevronska mreža z enim nivojem, ki ima toliko nevronov na skritem nivoju, kolikor želimo, da bo vektor besed imel števil. Nato se pregleda celotno besedilo v izbranem jeziku, večja količina besedila prinaša boljše rezultate, in skozi iteracije se prilagodijo uteži nevronov v nevronski mreži na tak način, da nevronska mreža uspe ugotoviti kontekst za vsako besedo - najde najpogostejše besede, ki se pojavljajo pred in po izbrani besedi. Že iz te definicije vidimo, da po naučeni nevronski mreži lahko računalnik sam najde podobne besede, saj imajo te najverjetneje podobne besede okoli njih - se pojavljajo v podobnem kontekstu.

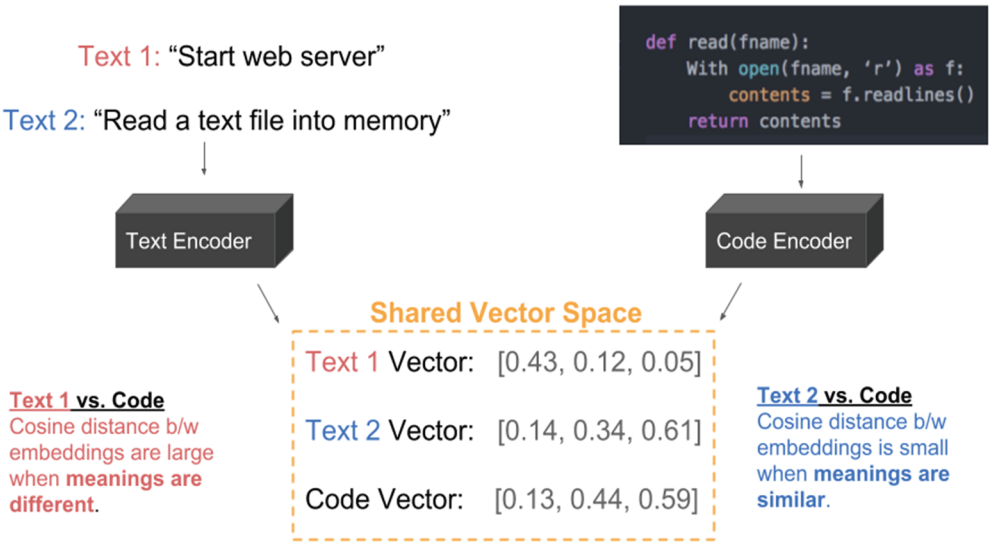
Podoben postopek se uporablja tudi pri pretvorbi programske kode v računalniku obvladljivo obliko. Postopek z imenom code2vec so razvili raziskovalci Facebook-a v sodelovanju z univerzami, ki pregleda ogromne količine kode in za vsak del kode naredi vektor, ki opisuje kontekst tistega dela kode [24]. Raziskovalci code2vec-a so šli s kopiranjem word2vec-ca še en korak naprej. Vektorje word2vec besed ene povedi lahko agregiramo skupaj, da dobimo en vektor za poved, vektorje povedi lahko agregiramo v vektorje stavkov (sentence2vec) ali še dlje kombiniramo kar v vektorje celotnih dokumentov (doc2vec). To lastnost agregacije delov kode pozna tudi pristop code2vec, kjer se lahko deli kode agregirajo v vektorje metod.

Uporaba postopka agregacije v code2vec kodiranju prinaša s seboj nekatere pozitivne lastnosti word2vec sistema. Vektorje word2vec kodiranja lahko tudi primerjamo med seboj - manjša kot je kosinusna razdalja med vektorjema, bolj semantično podobni sta besedi. Podobno je tudi pri code2vec sistemi, kar so praktično prikazali pri GitHub [22].



Slika 4: Uporaba nevronske mreže za kreacijo vektorske reprezentacije objektov [22].

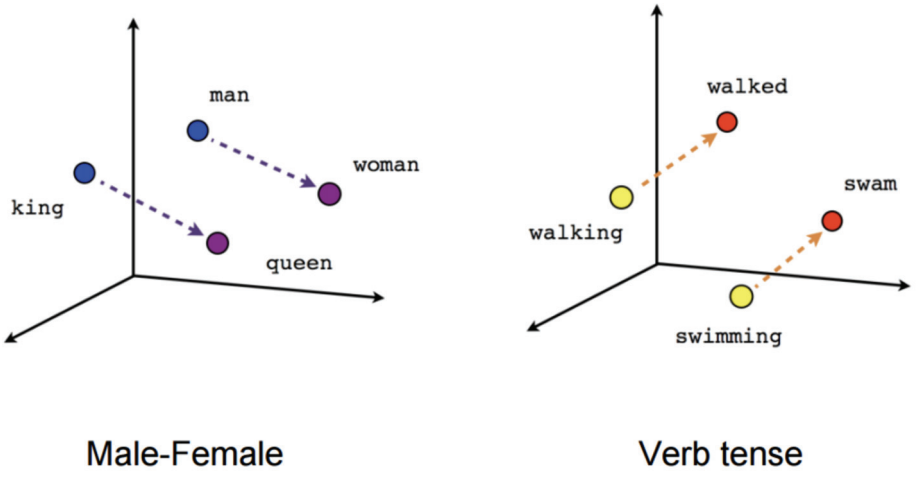
Tako delujejo tudi inteligentni asistenti, saj iz vsega nabora ogromne količine pregledane kode, poiščejo tako kodo (npr. tak vektor metode ali vrstice), ki najbolj ustreza že napisani kodi v uporabljenem razvijalnem okolju in na podlagi tega priporoča kaj popraviti ali dopolniti. Popravke in dopolnitve pa prav tako vzame iz najbolj podobnih pregledanih delov kode že obstoječih repozitorijev.



Slika 5: Primer vektorske oblike besedila in programske kode [22].

Pri GitHub-u so demonstrirali tudi, da kopiranje code2vec sistema iz idej word2vec sistema prinese še eno pozitivno lastnost - vektorji zapisane dokumentacije kode so zgradili tako, da so ti čim bolj podobni vektorjem kode, ki jo dokumentacija opisuje (na sliki 5). Na podlagi tega so zgradili semantični iskalnik, ki iz iskalnega niza zgradi vektor niza in tega primerja z vektorji kode ter najde najbolj podobno kodo.

Še marsikaj ni raziskanega na tem področju. Na primer, ena lastnost word2vec kodiranja je, da lahko nad vektorji uporabljamo tudi aritmetične operacije, kjer razlika vektorjev "Nemčija" in "Berlin" vrne vektor "prestolnica", ali pa seštevek "kralj" + "ženska" vrne vektor za besedo "kraljica" (na sliki 6). Zanimivo bi bilo uporabiti aritmetične lastnosti tudi nad code2vec vektorji, kjer bi lahko ugotovili manjkajoče dele metod ali razredov ter jih avtomatsko dopolnili. Računalniški prevajalniki, kot so Google Translate ali Bing Translate, delujejo prav na lastnosti word2vec vektorjev, da števila v vektorju nosijo semantični pomen besede ali stavka - ta je skoraj enak v vseh jezikih. Prevajanje deluje tako, da iz stavka izvornega jezika izračuna vektor, ter iz vsega nabora znanj najde in sestavi tak vektor v končnem jeziku, ki je čim bolj podoben vektorju izvornega stavka. Ta aplikacija v programski kodi bi bila zanimiva, saj bi iz preprostega avtomatskega prepisovanja kode iz enega programskega jezika v drugega upoštevala stil pisanja kode v vsakem izmed jezikov in uporabila knjižnice primerne za vsako nalogo posebej glede na programski jezik.



Slika 6: Aditivna lastnost vektorske reprezentacije objektov.

4 ZAKLJUČEK

Bliskovit razvoj široke palete raznovrstnih pristopov k strojnemu učenju ter vzporedni napredek v zmogljivosti strojne opreme, ki omogoča učinkovito poganjanje kompleksnih modelov strojnega učenja, odpirajo številne nove možnosti reševanja problemov na najrazličnejših področjih. Eno takšnih področji je zagotovo področje razvoja programske opreme, ki velja za izredno dinamično okolje, v katerem se pričakuje hitre odzive in rešitve za nastale tekoče izzive. Tesni roki dostave in kompleksnost sodobnih programskih rešitev predstavljajo za razvijalce dodaten pritisk. Inteligentni asistenti razvijalca predstavljajo rešitve, ki ponujajo odgovore na tovrstne sodobne izzive razvoja programskih rešitev. Njihova naloga je spremljati delo razvijalca ter mu sproti predlagati bloke programske kode, ki bi jih glede na kontekst lahko uporabil v naslednjem koraku programiranja. Cilj inteligentnih asistentov je poenostaviti in pohitriti delo razvijalca tako, da temu ob tem, ko naleti na nepoznani razred ali kakorkoli drugače ne more nadaljevati s pisanjem kode, ne bo več potrebno prekinjati dela in po spletu brskati za rešitvijo. Naloga inteligentnega asistenta, da to postori zanj ter mu predlaga rešitev takoj, ko jo ta potrebuje.

Trenutno so se na trgu pojavile številne rešitve, ki sledijo zastavljenemu konceptu inteligentih asistentov in jih je že mogoče preizkusiti tudi v produkcijskih okoljih. Najvidnejši predstavniki tovrstnih orodij so rešitev Codota, ki primarno cilja na razvijalsko skupno Java razvijalec, rešitev Kite, ki je primarno osredotočena na razvijalce programskega jezika Python ter Microsoftov IntelliCode, zasnovan primarno v podporo razvijalcem, ki razvijajo svoje rešitve na Microsoftovi tehnološki platformi. Čeprav imajo vsa tri navedena orodja svoj primarni krog razvijalcev na katerega ciljajo, je pri vseh treh orodjih zaznana močna tendenca po širitvi na druge programske platforme. Potrebno je poudariti, da se vsa navedena orodja trenutno nahajajo v relativno zgodnji fazi zrelosti, saj gre za relativno mlade programske produkte. V skladu s tem je potrebno prilagoditi naša pričakovanja. Kljub svojemu imenu, inteligentni asistenti razvijalca v tej fazi zrelosti ne bodo povsem izničila potrebo razvijalca po prekinjanju dela in brskanju za informacijami povezani z uporabo konstruktov programskega jezika ali API-ja, ki ga želijo vključiti v programsko kodo. Njihova pomoč je trenutno fokusirana v predvidevanje naslednjih blokov programske kode iz vidika pravilne uporabe API-ja. Tovrstni predlogi so seveda v veliko pomoč mlajšim in manj platforme programskega jezika manj veščim razvijalcev, izkušenejšim razvijalcem pa bi najverjetneje le zmanjšali količino zapisane kode, kar pa je z vidika produktivnosti solidna izboljšava. Kompleksnejših izzivov, zaradi katerih bi morali napredni programerji prekinjati delo in rešitev iskati po spletnih virih, primerjana orodja zaenkrat ne naslavljajo celostno.

Inteligentni asistenti razvijalca se, pogledano pod pokrov, učijo iz programske kode, ki je javno dostopna v spletnih skupnostih, kot je na primer StackOverflow, in v repozitorijih kode, kot sta na primer GitHub in BitBucket. Tovrstne spletne storitve predstavljajo skrivajo veliko zbirko znanja s področja razvoja programskih rešitev, katere je mogoče učinkovito uporabiti v navezavi z metodami strojnega učenja.

Čeprav bi kateri izmed naprednih razvijalcev znal biti nad trenutno ponujenimi zmožnostmi inteligentih asistentov morda razočaran, imajo sam koncept zagotovo v prihodnosti velik potencial. S tem, ko izkorišča pomembne repozitorije znanja, ki se je v njih nabira več desetletij, in jih s pomočjo pristopov strojnega učenja pretvori v modele, ki so sposobni razvijalcem ponuditi rešitve v trenutku, ko jih ta potrebuje, predstavljajo inteligentni asistenti močna razvojna orodja, ki daleč presegajo zmožnosti orodij z vnaprej kodiranimi pravili obnašanja.

5 LITERATURA

[1] https://savi.avsi.aero/about-savi/savi-motivation/exponential-system-complexity/, Exponential Growth of System Complexity, obiskano 5. 5. 2019.

[2] https://medium.com/lingvo-masino/codota-an-ai-coding-partner-6fb198381a61, Codota — An AI coding partner, obiskano 5. 5. 2019.

[3] https://www.interventure.info/blog/how-artificial-intelligence-will-change-software-development/, How Artificial Intelligence Will Change Software Development, , obiskano 5. 5. 2019.

[4] https://thenewstack.io/codota-offers-ai-pair-programming/, Codota Offers Pair Programming with Artificial Intelligence, obiskano 5. 5. 2019.

[5] https://jaxenter.com/codota-using-ai-make-code-better-135461.html, Codota: Using AI to make our code better, obiskano 5. 5. 2019.

[6] https://www.forbes.com/sites/mariyayao/2018/04/18/6-ways-ai-transforms-how-we-develop-software/#22e8111826cf, 6 Ways AI Transforms How We Develop Software, obiskano 5. 5. 2019.
[7] https://timesofdatascience.com/2019/01/29/497/ai-powered-programming-assistant-for-python/, AI-powered programming assistant for Python, obiskano 5. 5. 2019.
[8] https://www.tiobe.com/tiobe-index/, Tiobe Index, obiskano 5. 5. 2019.
[9] https://blogs.wsj.com/cio/2017/12/15/ai-to-drive-job-growth-by-2020-gartner/, AI to Drive Job Growth by 2020: Gartner, obiskano 5. 5. 2019.
[10] https://www.codota.com/, Codota, obiskano 5. 5. 2019.
[11] https://kite.com/blog/product, Kite, obiskano 5. 5. 2019.
[12] https://docs.microsoft.com/en-us/visualstudio/intellicode/, Microsoft, obiskano 13. 5. 2019.
[13] https://marketplace.visualstudio.com/items?itemName=VisualStudioExptTeam.VSIntelliCode, Microsoft Marketplace, obiskano 13. 5. 2019.
[14] https://www.bookmark.com, Bookmark, obiskano 10. 5. 2019.
[15] https://githubengineering.com/towards-natural-language-semantic-code-search/, Towards Natural Language Semantic Code Search, obiskano 10. 5. 2019.
[16] https://www.functionize.com/, Functionize, obiskano 10. 5. 2019.
[17] https://applitools.com/features/frontend-development, Applitools, obiskano 10. 5. 2019.
[18] https://blog.insightdatascience.com/automated-front-end-development-using-deep-learning-3169dd086e82, Automated front-end development using deep learning, obiskano 10. 5. 2019.
[19] https://hub.packtpub.com/5-ways-artificial-intelligence-is-upgrading-software-engineering/ , 5 Ways Artificial Intelligence Is Upgrading Software Engineering, obiskano 10. 5. 2019.
[20] https://www.forrester.com/report/How+AI+Will+Change+Software+Development+And+Applications/-/E-RES121339, How AI Will Change Software Development And Applications, obiskano 10. 5. 2019.
[21] https://code.fb.com/developer-tools/finding-and-fixing-software-bugs-automatically-with-sapfix-and-sapienz/, Finding and Fixing Software Bugs Automatically with SapFix and Sapienz, obiskano 10. 5. 2019.
[22] https://github.blog/2018-09-18-towards-natural-language-semantic-code-search/, Towards Natural Language Semantic Code Search, obiskano 10. 5. 2019.
[23] MIKOLOV Tomas, SUTSKEVER Ilya, CHEN Kai, CORRADO Greg, DEAN Jeffrey: "Distributed representations of words and phrases and their compositionality", International Conference on Neural Information Processing Systems 2013, str. 3111-3119.
[24] ALON Uri, ZILBERSTEIN Meital, LEVY Omer, YAHAV Eran "code2vec: Learning distributed representations of code", ACM on Programming Languages, letnik 3, 2019.
[25] DE SOUZA ALCANTARA Tulio, DENZINGER Jörg, FERREIRA Jennifer, MAURER Frank: "Learning Gestures for Interacting with Low-Fidelity Prototypes." International Workshop on Realizing AI Synergies in Software Engineering 2012, str. 32–36.
[26] BELTRAMELLI Tony: "Pix2Code: Generating Code from a Graphical User Interface Screenshot", Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems, 2018.
[27] GRADIŠNIK Mitja, KARAKATIČ Sašo, MAUŠA Goran, BERANIČ Tina, HERIČKO Marjan: »Možnosti vpeljave umetne inteligence v proces razvoja programske opreme«. Dnevi Slovenske Informatike 2019.
[28] JU Reyes, LICEA Guillermo: "Towards Supporting Software Engineering Using Deep Learning: A Case of Software Requirements Classification," International Conference v Software Engineering Research and Innovation 2017, str. 116-120.
[29] IYER Srinivasan., KONSTAS Ioannis., CHEUNG Alvin, ZETTLEMOYER Luke: »Summarizing source code using a neural attention model«, Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, str. 2073-2083.
[30] FELDT Robert, DE OLIVEIRA NETO Francisco, TORKAR Richard: "Ways of Applying Artificial Intelligence in Software Engineering". Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, 2018, str. 35–41.

OTS 2019
ADVANCED INFORMATION TECHNOLOGY AND SERVICES
M. Heričko in K. Kous (eds.)

University of Maribor Press

ON-DEMAND DATA MIGRATION AND GO-LIVE STRATEGIES FOR TRANSITIONING TO A NEW SYSTEM

TOMAŽ KORELIČ, MITJA BOMBAČ

Abstract: *Transitioning from a legacy system to a new one presents many challenges, including how customer data is migrated. A "big bang" cutover presents many risks regarding system online time, readiness of the new system, fallback strategies and others. In an agile development environment features are delivered continuously, so a go-live point in time where not all features are available in the new system demands different migration approaches. This paper presents a migration strategy where both systems run in parallel, but the data of a customer is present only in one and it is migrated on-demand. It was introduced as part of a business project where customer data was migrated on-demand, based on order processing and feature availability in the new system. Additionally, this paper also covers the topic of back migrations when requested features are not yet available in the new system and therefore have to be done in the legacy world. Finally, advantages, disadvantages, challenges, and lessons learned are presented for the chosen approach.*

Keywords: *data migration, migration strategies, migration architecture, go-live strategies, mass migration*

CORRESPONDENCE ADDRESS: Tomaž Korelič, Technology Architect, BearingPoint Technology GmbH, Premstätten, Austria, e-mail: tomaz.korelic@bearingpoint.com. Mitja Bombač, Senior System Engineer, BearingPoint Technology GmbH, Premstätten, Austria, e-mail: mitja.bombac@bearingpoint.com.

https://doi.org/10.18690/978-961-286-282-4.17 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 INTRODUCTION

Transitioning IT systems is clearly important in any business. For taking the big leap to switch to a newly developed system, it takes confidence that it is fully tested. There is no room for critical errors in any business environment. How to migrate customer data to the new system is an essential part of this process. There are many best practices on how to execute such transitions. All have advantages and disadvantages and are applicable to different business scenarios. This paper presents a customer data migration approach implemented in a transitional business project. It presents the approach of on-demand data migration and its architecture and concludes with the lessons learned.

2 CUSTOMER DATA MIGRATION

There are many best practices and guidelines on how to approach customer data migration [3]. The following chapters summarize common strategies before our use-case is presented in detail.

2.1 Common strategies

When switching business users and customers from the old world to the new world there are essentially two major options for the cutover strategy – big bang or phased [1][2][8].

2.1.1 *Big Bang cutover*

Big-Bang data migration is where an entire dataset is moved (extracted, transformed and loaded [3]) from source to target systems in one operation. This is typically carried out over a weekend or planned downtime period [2]. The main advantages of this approach are lower costs, shorter implementation time, no need for changeover operations, and no synchronization issues. On the other hand, there is a high risk involved during the rushed migration phase, failures in the new system's functionality may have a business impact and fallback strategies can be challenging.

2.1.2 *Phased cutover*

This approach is also called iterative data migration, trickle-feed data migration or synchronized data migration [2]. It reduces the risk of failure by letting both systems run in parallel for a longer duration, and data is migrated in smaller increments. There is no stressful and failure-prone short timeframe, where all data is migrated. Found errors can be fixed more easily. Effective execution of such cutover may be challenging due to the complexity of the underlying applications, data, and business processes. Additionally, operators may have to be able to work with two systems in that phase. Additional costs for running two systems also apply.

Both approaches have advantages and disadvantages. However, the implemented strategy does not need to strictly follow one or another [6]. It needs to be selected and implemented based on business needs and compromises. The following chapters present our approach implemented in a business project which is a variant of a phased cutover.

2.2 Use-case and motivation

The scope of the transformation customer project was to replace multiple legacy system landscapes by a new one. The goal was to decommission the affected systems and to set up the new one based on a modern technology stack. The legacy functionality was analysed, optimized and integrated. Both system landscapes were integrated with the same interfaces, where the internal business processes differ slightly. The systems processed orders containing services of a customer that needed to be provisioned and be inventoried.

The customer requested to go-live with the new system as soon as possible. As soon as the first order could be processed, and the new system supported the first service, the system went live with the minimum possible functionalities delivered. Incremental support for other services was delivered until all data was migrated and all functionality integrated, and the legacy landscape could be shut down.

2.3 Migration, distribution and orchestration proxy microservice

Based on the requirements and the need to go-live as soon as possible, a big bang migration was not an option for us. Additionally, we wanted to decrease the risk of failure when performing migrations. Because not all services were supported at the time of go-live an on-demand migration strategy where both systems run in parallel was introduced.

A "migration proxy" microservice was introduced which handles the order distribution and orchestration between the various systems. Based on the customer inventory location, order content and feature support, on-demand migrations are triggered and when finished, the original order is routed accordingly. The feature support is configurable in the database of the migration service and is definable based on services that were processed in the system.

The following figure 1 presents the process flow with the migration proxy microservice.

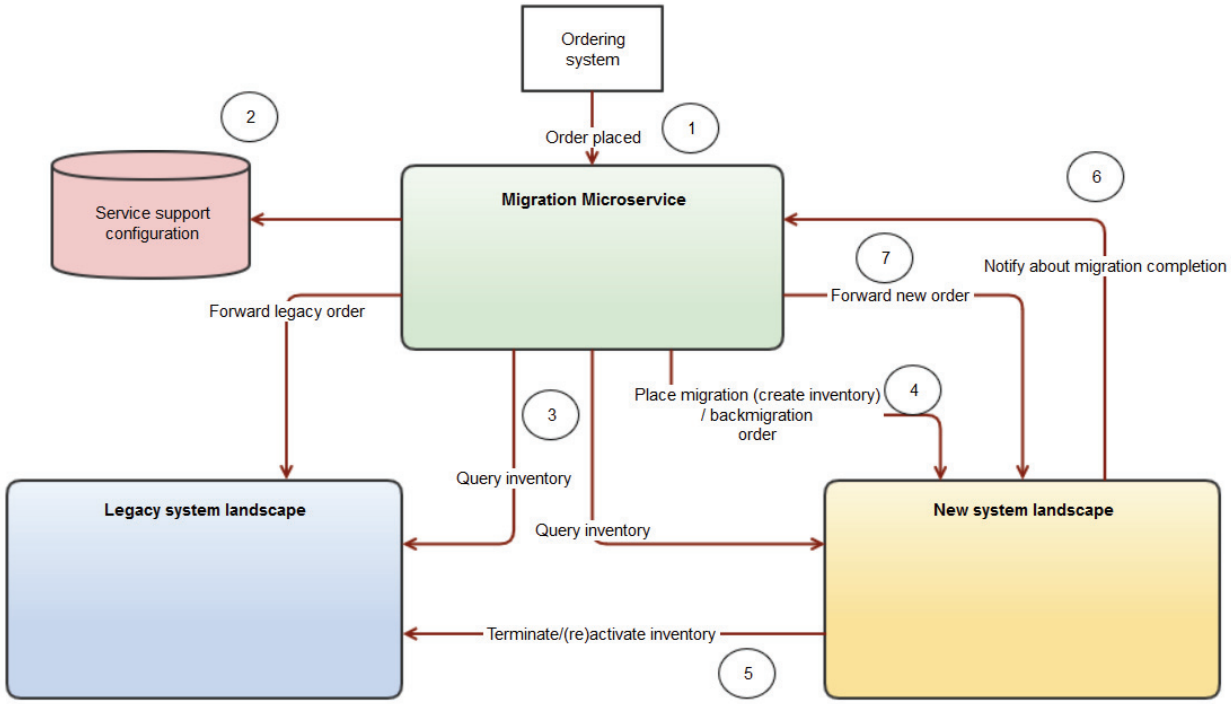


Figure 1: System architecture

The process flow in the migration proxy is composed of the following steps:

1. The order is placed.
2. Determine support of services in the new system.
3. Determine customer inventory location.
4. If determined that the inventory is in the legacy system and the service in the order is supported, then an on-demand migration order is created and placed in the new system.
5. The inventory in the legacy system is terminated.
6. A migration order completion notification is sent.
7. Place initial order in the new system.

The current inventory state is queried based on APIs provided by both systems. Next, the order content is evaluated with a configuration of supported features, which is contained in the new proxy microservice. When detected that a customer inventory is persisted in the legacy system, but the accepted order is supported by the new one, a new on-demand migration order is placed in the new system. It migrates the inventory by using provided APIs and terminates the inventory in the legacy system. Finally, the initial order is placed in the new system. Using this approach, the inventory is migrated only when needed and supported. At any point in time, the inventory of a specific customer is only available in one system.

2.3.1 *On-demand back migration*

When an order requests a deactivated (or not yet implemented) feature, but the inventory of the affected customer is contained in the new system, then a fallback on-demand-back-migration order is triggered which migrates the inventory back to the legacy system where finally the order is also routed to.

The process flow in the migration proxy is composed of the following steps, which is also presented in Figure 1:

1. The order is placed.
2. Determine support of services in the new system.
3. Determine customer inventory location.
4. If determined that the inventory is in the new system and the service in the order is not supported, then an on-demand back-migration order is created and placed in the new system.
5. The inventory in the legacy system is created or reactivated when already existing.
6. A back-migration order completion notification is sent.
7. Place initial order in the legacy system.

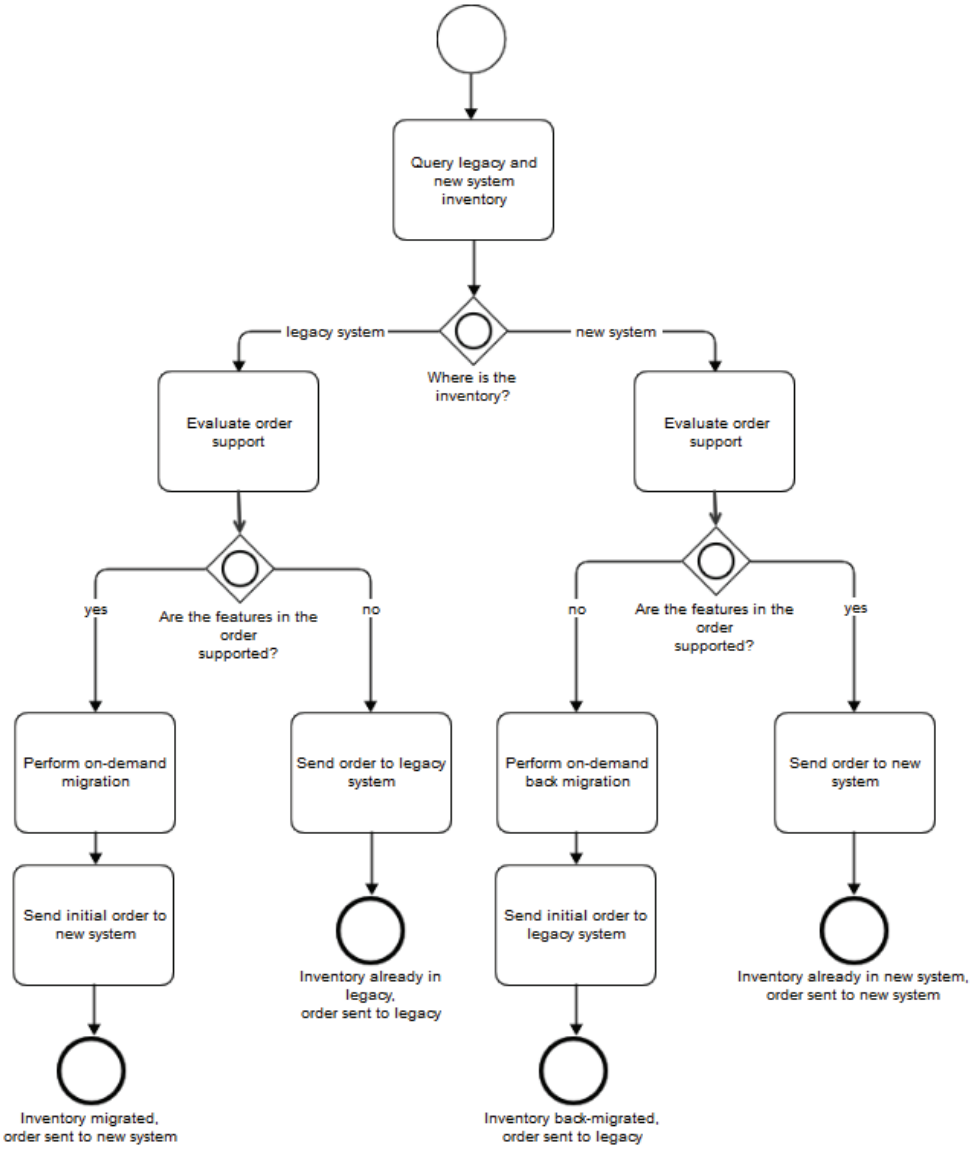


Figure 2: Migration determination process

Figure 2 presents a decision tree of the functionality in our migration microservice.

2.3.2 *On-demand (back) migration order behaviour*

Special handling was introduced in the new system for the on-demand migration and back migration orders as presented in figure 3. The goal of both was to move the customer inventory from one to another system without provisioning any platform. After that, the initial order can make changes on the migrated inventory.

The on-demand migration order is by design a service creation order which additionally terminates the inventory in the legacy system.

On the other hand, an on-demand back-migration order is by design a service termination order which additionally creates or reactivates the customer inventory in the legacy system. The initial order can then make changes on the active inventory in the legacy system.

A disadvantage of this process is the introduction of new interfaces in the legacy system that supports the activation and termination of the inventory. Additionally, special order behaviour needs to be introduced in the new system to trigger such actions.

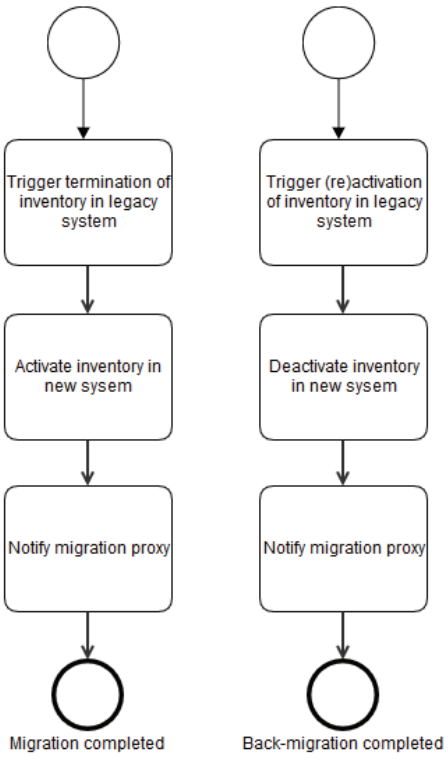


Figure 3: Migration and Back Migration order process

2.4 Go-live approach

An advantage of the described approach is to have the possibility to choose which feature will be provided in the new system at a specific point in time. The support of services is configurable in the database of the migration proxy. This configuration can be edited at any time.

So-called “go-live soft-launches” by software features can be scheduled where only specific service support can be activated and if necessary deactivated immediately after a controlling timeframe. They significantly minimize the risk of the transformation for the customer, development partner, management and others.

When it is decided to deactivate the support for a service, some inventory is may be already migrated to the new system. This inventory remains there until a subsequent order triggers a back migration.

2.5 Mass migration

The final feature of the migration microservice is the support for mass migrations. As only the inventory where changes are triggered by orders is migrated, most of the inventory remains in the legacy system. For that purpose, the on-demand migration logic is used to trigger mass migrations in the following way:

1. A list of customer identifiers is given as an input.
2. A scheduler processes this list.
3. For every customer identifier, the location of the inventory is determined.
4. If the inventory is in the legacy system an on-demand migration order is placed.
5. When finished the inventory is migrated and terminated in the legacy.

3 CONCLUSION

The paper presents the strategy of an on-demand inventory (back) migration that was introduced in scope of a customer project. It presents the motivations on why such a strategy was applied and implemented, and how the migration microservice orchestrates the migrations. The chosen approach gave us several advantages such as:

- Low risk in migration execution.
- Quick recovery possibility in case of failures.
- Flexibility in activating and deactivating supported services/software features.
- Possibility of back-migrate into the legacy system.
- Very early go-live date in the project time span.
- No long term go-live planning necessary.
- Simple mass migration integration.

When reflecting on the experience and lessons learned we see the following drawbacks of our approach:

- Two systems needed to run in parallel for a long duration.
- High complexity in migration logic and determination.
- The introduction of new interfaces was necessary for both systems. Additional features for inventory manipulation needed to be implemented.
- The inventory is split between two systems.

Such a migration strategy was chosen in order to be able to go into production with the new system as soon as possible. Continuously new features were supported by an agile development project and more orders were routed to it. With the chosen approach additional complexity was introduced and new APIs had to be provided. In conclusion, the migration approach fulfilled our customer's needs and requirements, it gave us the needed flexibility to schedule soft-launches and minimized the risk of failures.

4 REFERENCES

[1] JONES Dylan "The data migration go-live strategy – what is it and why does it matter?" https://datamigrationpro.com/data-migration-go-live-strategy/ visited 28.4.2019.

[2] MISTRY Dhiren "Big Bang vs Phased Migration", https://xceedgroup.worldsecuresystems.com/insights/big-bang-vs-phased-migration visited 28.4.2019.

[3] MORRIS John, Practical Data Migration, British Informatics Society Limited, April 2006.

[4] Oracle Corporation, "Successful Data Migration", An Oracle whitepaper November 2011, https://www.oracle.com/technetwork/middleware/oedq/successful-data-migration-wp-1555708.pdf visited 28.4.2019.

[5] SPACEY John, 4 Types of Data Migration, https://simplicable.com/new/data-migration visited 28.4.2019.

[6] DONAGHER John, Big Bang Versus Phased ERP Implementations, https://lumeniaconsulting.com/en/insights/blogs/big-bang-versus-phased-erp-implementations visited 28.4.2019.

[7] RENWICK Catherine, Migrating From Paper to HER, https://www.chiropractic.on.ca/event/migrating-paper-ehr/#.XMWgRaRS9EY visited 28.4.2019.

[8] HOWARD Philip, Data Migration, Bloor Research, https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=9&ved=2ahUKEwiCxPrl-vLhAhXlsosKHYszBp4QFjAIegQIBhAC&url=https%3A%2F%2Fwww.existbi.com%2Fwp-content%2Fuploads%2F2013%2F09%2F9744c08a90461e58daf0fcf26e2ddd3e_11921334155093_770351314340475_Data_MigrationWhitePaper.pdf&usg=AOvVaw0h7Ww9-c5Uy7J0MqVG5mKG visited 28.4.2019.

OTS 2019
SODOBNE INFORMACIJSKE TEHNOLOGIJE IN STORITVE
M. Heričko in K. Kous (ur.)

Univerzitetna založba
Univerze v Mariboru

POSODOBITEV REŠITVE ZA MOBILNI MARKETING Z MODULARNO PLATFORMO LIFERAY IN TEHNOLOGIJO OSGI

DIMITAR IVANOVSKI, MIROSLAV BERANIČ

Povzetek: *Mobilno trženje postaja vse bolj priljubljeno med podjetji po vsem svetu z naraščajočo priljubljenostjo pametnih telefonov. Za razliko od standardnega trženja, ki učinkovito spodbuja zanimanje občinstva s pomočjo radijskih spotov, časopisov in tiskanih medijev, mobilni marketing podjetjem omogoča, da negujejo bolj osebne odnose s svojimi strankami. Z mobilnim trženjem lahko podjetja svojim strankam posredujejo vsebine z osebno in stroškovno učinkovito komunikacijo. S tem namenom smo za A1 Slovenija razvili in prenovili spletno rešitev za mobilni marketing, ki je namenjena vsem podjetjem, ne glede na velikost, obseg ali poslovne dejavnosti. Rešitev omogoča samostojno oblikovanje mobilnih oglaševalskih akcij, nadzor baze strank in pregled sedanjih in preteklih akcij ter pomaga partnerjem pri iskanju novih strank, graditvi zvestobe obstoječih kupcev in povečanju prodajne dejavnosti. V prispevku bomo predstavili potek razvoja in posodobitve spletne aplikacije za mobilni marketing, ključne tehnologije, s katerimi smo delali (njihove prednosti in slabosti), kaj nam OSGi kot modularna tehnologija omogoča ter praktične izkušnje, ki smo jih pridobili pri izgradnji aplikacije na odprtokodni platformi za digitalno izkušnjo Liferay Portal 7.1.*

Ključne besede: *Mobilni marketing, telekomunikacijski sektor, digitalna izkušnja, Liferay Portal 7.1, OSGi tehnologija*

NASLOVA AVTORJEV: Dimitar Ivanovski, Bintegra d.o.o., Maribor, Slovenija, e-pošta: dimitar.ivanovski@bintegra.com. Miroslav Beranič, Bintegra d.o.o., Maribor, Slovenija, e-pošta: miroslav.beranic@bintegra.com.

https://doi.org/10.18690/978-961-286-282-4.18 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 UVOD

S popularizacijo pametnih telefonov in drugih mobilnih naprav, kot so tablice, je zelo pomembno, da se mobilni marketing (v nadaljevanju - MM) vključi kot bistveni del digitalnih tržnih strategij. Mobilno trženje je nabor tehnik in formatov za promocijo izdelkov in ponudbo storitve, ki kot komunikacijsko orodje uporabljajo mobilne naprave. Velika prednost je, da odpira osebni in neposredni kanal med oglaševalcem in njegovimi ciljnimi skupinami in ponuja različne prilagoditve. Druge prednosti so, da dopolnjuje obstoječe tržne dejavnosti; je stroškovno učinkovito; je merljivo; ter je zelo dobro orodje za pridobivanje novih kupcev/strank [1], [2].

Podatki IAB o mobilnem oglaševanju nam dajo idejo o pomembnosti MM [3]:

- Povprečni čas dnevno porabljen na spletu je 2 uri in 34 minut za pametne telefone in 1h19m za tablice.
- V povprečju uporabniki mobilnih telefonov mesečno prenesejo 2 aplikaciji, imajo 17,8 nameščenih in redno uporabljajo 9,1.
- Eden od treh mobilnih uporabnikov pogosto uporablja svojo napravo kot »drugi zaslon« v kombinaciji s televizijo.
- 9 od vsakih 10 uporabnikov pametnih telefonov ga je nekoč uporabilo v procesu nakupa (zlasti tistih med 16. in 45. letom starosti).
- 4 od 10 uporabnikov je uporabljalo mobilno poslovanje.

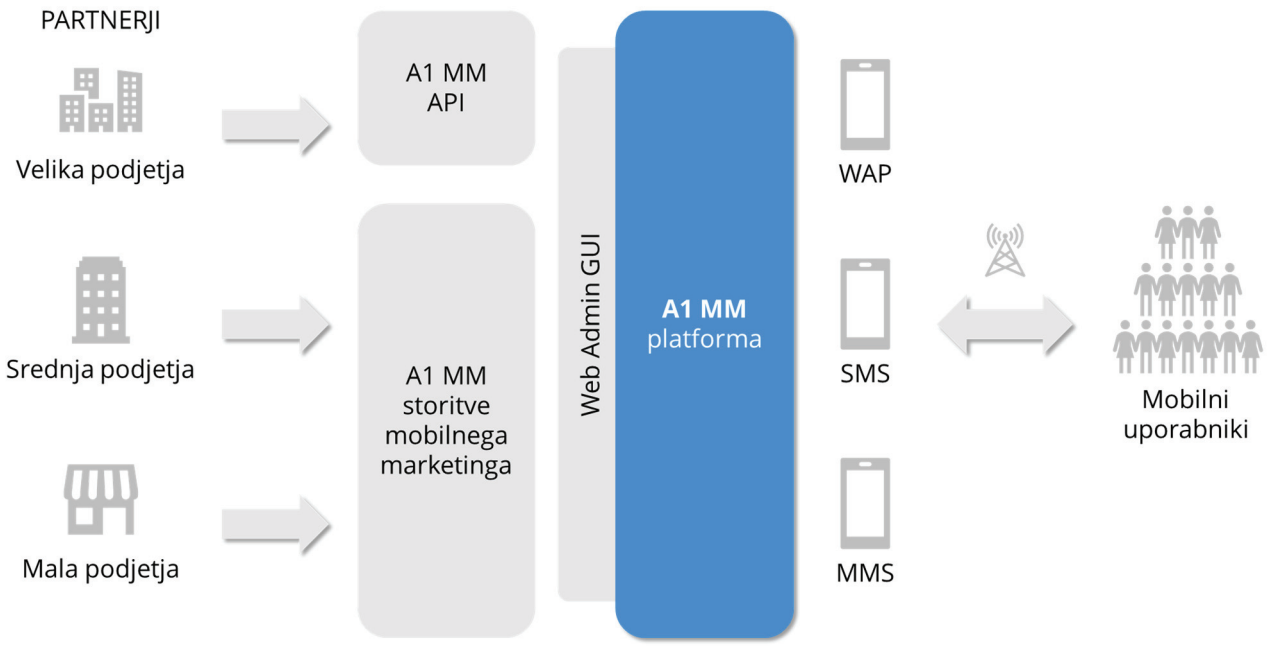
MM je večkanalni pristop in eden najmočnejših kanalov, ki ga je treba upoštevati, je SMS [4]. Po besedah strokovnjakov za digitalni marketing v Teradati [5] se 94% SMS sporočil odpre v treh minutah. Poleg tega pravijo, da mobilni uporabniki berejo 98% vseh svojih sporočil v primerjavi z le 29% tvitov, 20% e-poštnih sporočil in le 12% objav v Facebooku.

S poslovnega vidika je pomembno, da se zavedamo, da je MM pomemben in lahko ustvari dodano vrednost podjetju. S tem v mislih, smo za drugi največji mobilni operater v Sloveniji - A1 Slovenija (prejšnji Si.mobil) načrtovali, razvili in posodobili rešitev MM, ki jo že več let uporabljajo in jo bomo predstavili v naslednjih poglavjih. Dodatno bomo predstavili potek razvoja in posodobitev spletne aplikacije, ključne tehnologije s katerimi smo delali, dobre prakse in potenciale ter možnosti za uporabo tehnologije v prihodnosti.

2 PREDSTAVITEV REŠITVE A1 MOBILNI MARKETING

A1 MM Portal je mobilna marketinška rešitev, ki podjetjem omogoča ustvarjanje in izvajanje akcij neposrednega trženja ter interaktivnih storitev (Slika 1). A1 MM zajema sporočila SMS / MMS, USSD in mobilne internetne strani. Ključne prednosti rešitve so:

- Prilagodljiva rešitev, ki podjetjem omogoča ustvarjanje mobilnih oglaševalskih akcij (enosmerne in dvosmerne interakcije);
- Večkanalna rešitev, ki pokriva sporočila (SMS / MMS), mobilni internet, USSD;
- Zmogljiva administracija, ki omogoča prilagodljivost za ustvarjanje in upravljanje partnerjev, cenovnih in storitvenih paketov;
- Arhitektura rešitev omogoča hitro uvajanje novih storitev.
- Združljiv s TAG (Telekom Austria Group) infrastrukturo in tehničnimi platformami, ki omogoča zelo kratek čas do trženja.



Slika 1: Platforma A1 mobilni marketing

2.1 Platforma Liferay

A1 MM je portalska rešitev, ki je bila zgrajena na odprtokodni platformi Liferay. Po podatkih podjetja Gartner [6] je Liferay vodilna svetovna platforma za odprtokodne rešitve, ki širokemu krogu občinstva zagotavlja dosledni, varni in osebni dostop do informacij in aplikacij na številnih digitalnih točkah. Organizacije običajno uporabljajo podobne okvire za izgradnjo, uvajanje in stalno izboljševanje spletnih strani, portalov, mobilnih in drugih digitalnih izkušenj. Liferay Platform upravlja predstavitveni sloj (angl. layer) na podlagi vloge, varnostnih privilegijev in preferenc posameznika. Združuje in usklajuje aplikacije, vključno z upravljanjem vsebin, iskanjem in navigacijo, personalizacijo, integracijo in združevanjem, sodelovanjem, potekom dela, analitiko, mobilno in večkanalno podporo [7].

A1 MM je bil prvič zgrajen na platformi Liferay 6.0 in smo ga sedaj nadgradili na 7.1. Spodaj so glavne razlike med različicami:

- Verzija 6.0. Glavni mehanizem za doseganje modularnosti so Liferay vtičniki (angl. plugins). Vtičniki Liferay se vedno distribuirajo kot spletni arhivi (.war datoteke) in so razporejeni preko mehanizmov za uvajanje portala Liferay.
- Verzija 6.2. Uvod v OSGi runtime. OSGi runtime je eksperimentalen in nepodprt. So ga obravnavali kot tehnološki predogled.
- Verzija 7.0. Liferay Portal je modularen. Vsebuje module, ki so izdelani in testirani neodvisno in vzporedno. Liferay postane platforma, na katero nameščamo module in modularne aplikacije, ki jih lahko namestimo, zaženemo, zaustavimo in odstranimo. Komponente Liferay Portala uporabljajo standard modularnosti OSGi.
- Verzija 7.1. Standard OSGi je v celoti vključen.

Končna rešitev (Slika 2) je B2B portal Liferay 7.1, kjer lahko podjetja vseh velikosti ustvarijo in popolnoma neodvisno upravljajo svoje storitve in stike, dostopajo do poročanja in plačevanja.

Slika 2: Primer ustvarjanje in aktivacijo storitev

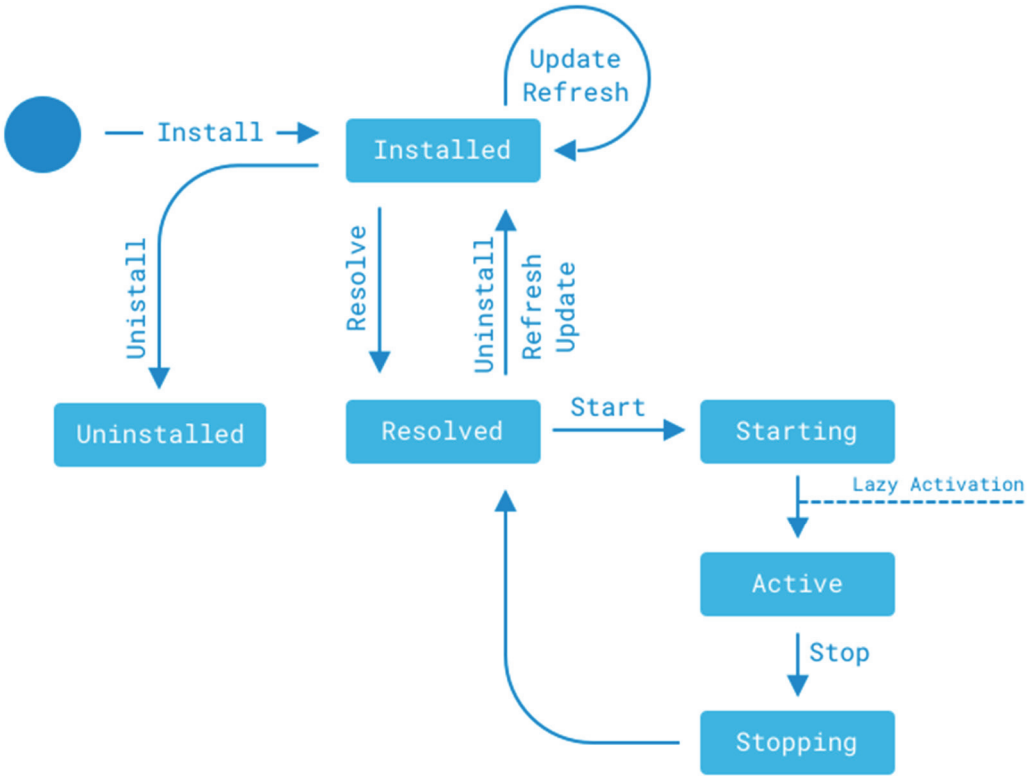
Pomembno je omeniti, da A1 MM je bil zgrajen na Liferay Portal Community Edition - CE. Liferay Portal CE je osnova za manjša podjetja, z omejenimi vgrajenimi funkcionalnostmi. Odvisno od kompleksnosti, veliko funkcionalnosti se "enostavno" implementira ali prilagodi. Podpora je s strani forumov, mailing list, itd. Po drugi strani, plačljivo različico Liferay Enterprise oz. Liferay DXP ima več razširitev oz. razširitve imajo večji nabor funkcionalnosti (za katere potrebno je vedeti zakaj se bodo uporabljaje) in podpora s strani Liferay-a. Liferay Portal CE podpira "odprtokodne" projekte, Liferay Portal EE oz. DXP pa je naravnan za velika podjetja in podpora EE rešitve. Obstaja še Liferay Commerce – majhni koraki Liferay-a v vode B2B in B2C nakupovalnih okolij ter Liferay Sync – sinhronizacija datotek z lokalnim računalnikom.

2.2 Tehnologije in orodja

Cilj pri izbiri tehnologij je bil dati našim razvijalcem in uporabnikom najnovejšo, preprosto in stabilno platformo za gradnjo storitev. Liferay Portal 7.1 temelji na priljubljenih, dobro znanih in dobro podprtih tehnologijah. V svoji osnovi je Liferay JavaEE aplikacija, ki vključuje standard OSGi. S tem so združili najboljše iz obeh svetov: dostop do najbolj robustne in popolne poslovne platforme na svetu ter prednosti najbolj celovitega in stabilnega modularnega vsebnika na svetu. Z Java 8 in OSGi smo v dinamičnem okolju, ki temelji na modulih, uspeli uvesti poslovno, prilagodljivo, spletno in mobilno aplikacijo.

Posodobljen portal je bil zgrajen modularno. Modul je enotna enota distribucije in uvajanja v modularni arhitekturi s standardnim OSGi (Slika 3). OSGi med drugim opredeljuje, kako so lahko moduli odvisni drug od drugega ter kako komunicirajo. Določa tudi obliko pakiranja za module: svežnji (angl. bundles) OSGi. Modul OSGi je navadna datoteka JAR, ki jo razvijalci Java poznajo kot datoteko ZIP, ki vsebuje prevedeno kodo, predloge, vire in nekaj meta podatkov [8].

Moduli opišejo svoje potrebe in sposobnosti v meta-podatkih, okolje poskrbi, da je potrebam zadoščeno in da so sposobnosti propagirane. Uvoz in izvoz se dela na nivoju Java paketov, zato še toliko bolj pomembno primerno poimenovanje Java paketov. OSGi razvoj nam omogoča enostavnejše objavljanje in pridobivanje funkcionalnosti. Kodira/prevaja se napram API-ju, pri izvajanju se ožiči izbrana implementacija. V primerjavi z Liferay 6.2, kjer imamo WAR datoteko, funkcionalnosti pri Liferay 7.1+ so razdrobljene v posamezne Java arhive – JAR datoteke, ki imajo dodatne meta datotečne opise. Dodatno, določen in znan je življenjski cikel modula. V Liferay 7.1 je prvič OSGi "first class citizen". Trenutno Liferay "še vedno" dodaja lastne anotacije, ki niso del splošne specifikacije, tako da pogosto moduli niso prenosljivi, a se bo to v prihodnjih verzijah izboljšalo/spremenilo.



Slika 3: OSGi standard [7]

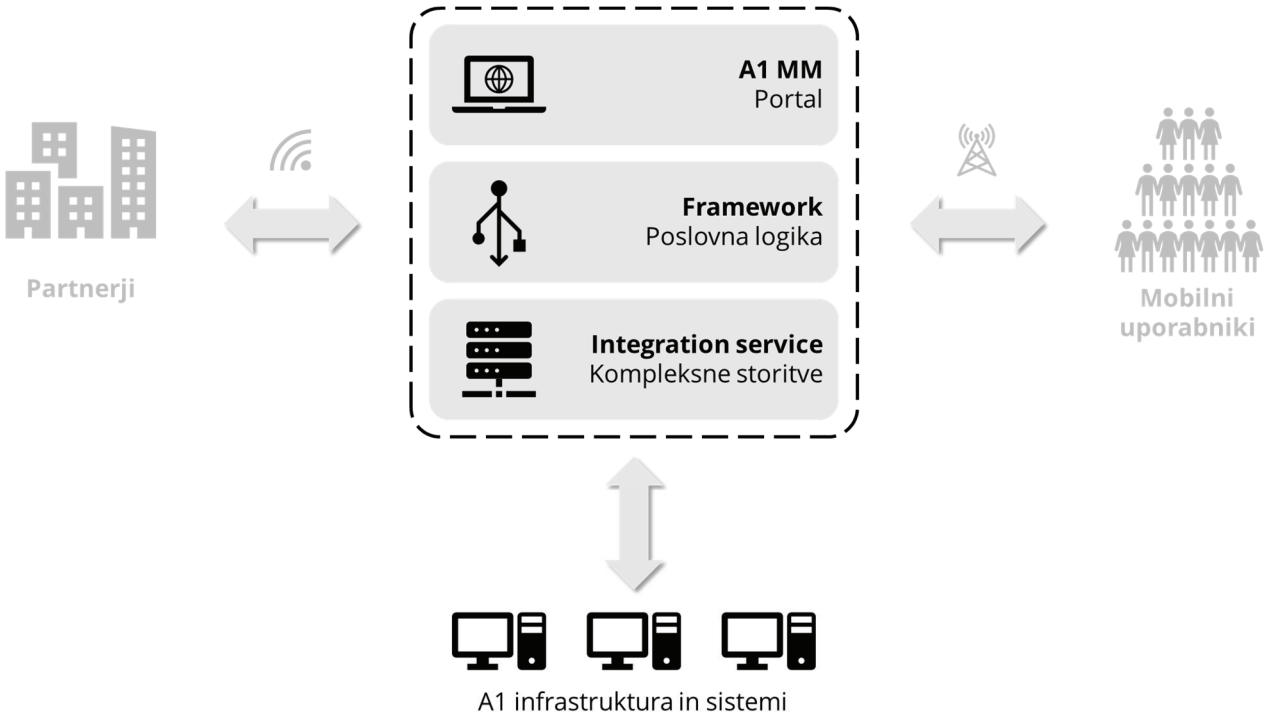
Za komunikacijo s SOAP spletno storitvijo se praviloma uporablja orodje/ogrodje/knjižnica Apache CXF v povezavi z Apache Maven "build" orodjem. Predvsem s stališča, ker se ta nabor orodja uporablja skozi vse druge komponente v arhitekturi (primarno v storitvenem vodilu in/ali povezanih aplikacijah). Prednost je predvsem v poenotenju nabora knjižnic/orodij, s katerimi se srečujejo razvijalci. Praviloma se najprej napiše/definira spletni vmesnik z WSDL, nato se naredi implementacija strežnika in odjemalca. Pri implementaciji SOAP odjemalca je potrebno poskrbeti za preverjanje dostopnih pravic.

Na čelni del rešitve smo spremenili Liferayjev Alloy UI z Bootstrap in SASS. Vključili smo različne JavaScript knjižnice, vključno z jQuery in Highcharts. Skupaj z oblikovalskim jezikom in stilskim vodnikom A1 smo sledili oblikovalskemu jeziku, ki so ga oblikovali oblikovalci v Liferayu, imenovanem Lexicon Experience Language. Za razvoj in namestitev teme smo uporabili naslednja orodja:

- NodeJS. Node.js je sodobna tehnologija za izdelavo spletnih aplikacij, zlasti v realnem času. Uporabili smo ga kot front-end tehnologijo, ki bi razumela našo javascript, sass kodo in jo izvedel (angl. execute), da bi ustvarili WAR datoteko za temo.
- NPM. Znan tudi kot upravljalec paketov Node (angl. Node package manager) je orodje, ki nam je omogočilo namestitev knjižnic tretjih oseb. Uporabili smo ga za namestitev Gulp-a.
- GULP. Gulp je majhen komplet orodij, ki avtomatizira ponavljajoče se naloge. Te ponavljajoče se naloge so ponavadi sestavljanje datotek CSS, JavaScript, Apache FreeMarker ali takrat, ko uporabljate ogrodje, ki se ukvarja z nestandardnimi datotekami JavaScript / CSS, boste želeli uporabiti orodje za avtomatizacijo, ki zgrabi te datoteke, jih pakira skupaj in prevede vse, da lahko vaš brskalnik to razpozna.

2.3 Arhitektura rešitve

A1 MM arhitektura je večplastna (Slika 4). Vsaka plast je ločena aplikacija, ki jo je mogoče namestiti ločeno na več strežnikih. To nam omogoča uravnoteženje obremenitve in varno delovanje. Komunikacija med različnimi plastmi poteka preko protokola JMS. Komunikacija z zunanjimi storitvami poteka prek različnih protokolov (SMPP, SOAP…). Podporo za dodatne protokole lahko dodamo hitro in enostavno.



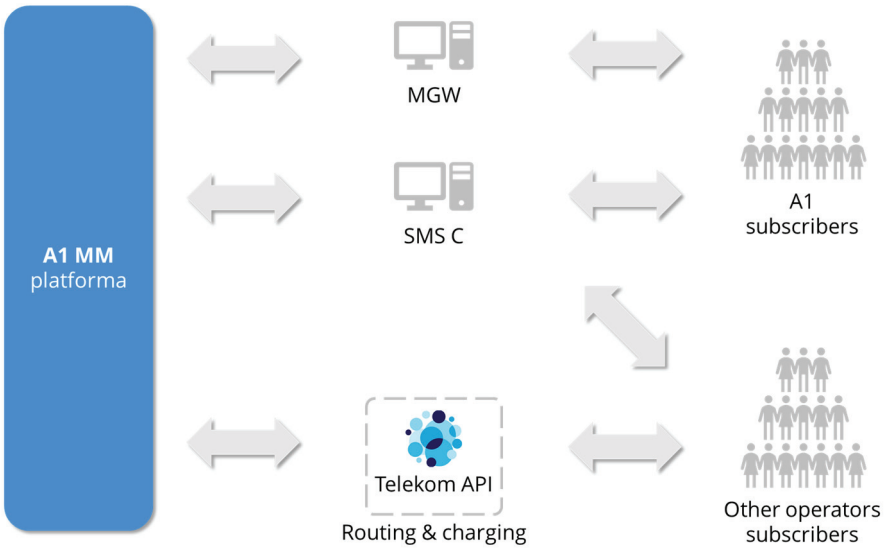
	Stara	**Nova**
Portal	Liferay Portal 6.0	Liferay Portal 7.1
Framework	Java 6, JBoss AS 5.1.0 GA	Java 8, Apache ServiceMix 7
Integration Services	Java 6	Java 8

Slika 4: Arhitektura A1 MM

Trenutno uporabljen aplikacijski strežnik je JBoss AS 5.1.0 GA. Zadnja izdaja JBossa je bila različica JBoss AS 7.1.1. Od takrat so ustavili nadaljnji razvoj. Eden od dveh zaporednih projektov je Apache ServiceMix z najnovejšo različico 7. V tej nadgradnji je bil uporabljen Apache ServiceMix različice 7.x. Apache ServiceMix je fleksibilen, odprtokodni integracijski vsebnik, ki združuje funkcije in funkcionalnost Apache ActiveMQ, Camel, CXF in Karaf v zmogljivo izvajalno platformo, ki smo jo lahko uporabili za izgradnjo lastnih integracijskih rešitev. Zagotavlja popolno storitveno vodilo (ESB), ki je pripravljeno za podjetja in temelji na OSGi.

2.4 Usmeritev sporočil

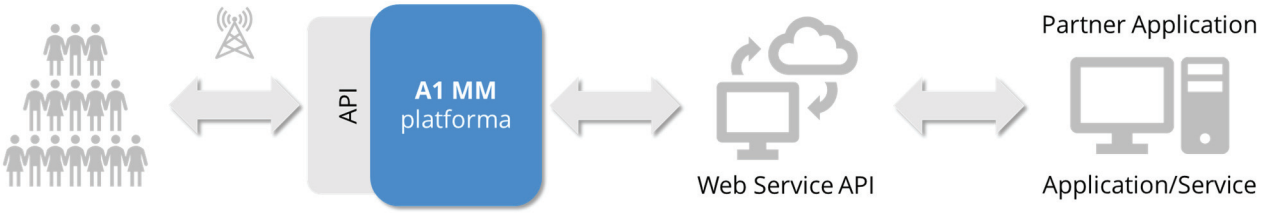
Usmerjanje sporočil poteka na SMS Gateway-u ali katerikoli drugi podobni storitvi (Slika 5). Uporabimo lahko več storitev SMS in MMS usmerjanja in jih povežemo z A1 Slovenija. Konfiguracijo usmerjanja sporočil opravijo administratorji SMS centra in je celoti odvisna od A1 Slovenija.



Slika 5: Primer usmerjanje sporočil

2.5 Uporaba zunanjega vmesnika

Partner lahko uporablja tudi zunanji API in izvaja svoje storitve. Dostop do poročanja ima prek A1 portala (Slika 6).



Slika 6: Primer uporaba zunajnega vmesnika

2.6 Administracija

A1 MM zagotavlja zmogljivo upravljanje z zagotavljanjem fleksibilnosti pri ustvarjanju in upravljanju partnerjev, oblikovanju cenovnih paketov in paketih storitev, ki jih lahko zahtevajo zaposleni na področju trženja ali IT:

- Cenovni načrti
  - Predplačilo - vključuje kvoto sporočil
  - Naročnina - vključuje kvoto sporočil / mesečno
  - Kalkulator - uporaba
- Partner
  - Uporabniki in vloge
    - Vloge: Admin, Urednik, Pošiljatelj, Bralec
  - Cenovni načrt
- Storitve
  - Konfiguracija pretoka
  - zasebnost sporočil - preverjanje blokade
  - VAS integracija zaračunavanja
  - Partner je lahko omejen na čas za pošiljanje sporočil
  - Zamenjava besedilnega imena originatorja
  - poročanje v realnem času

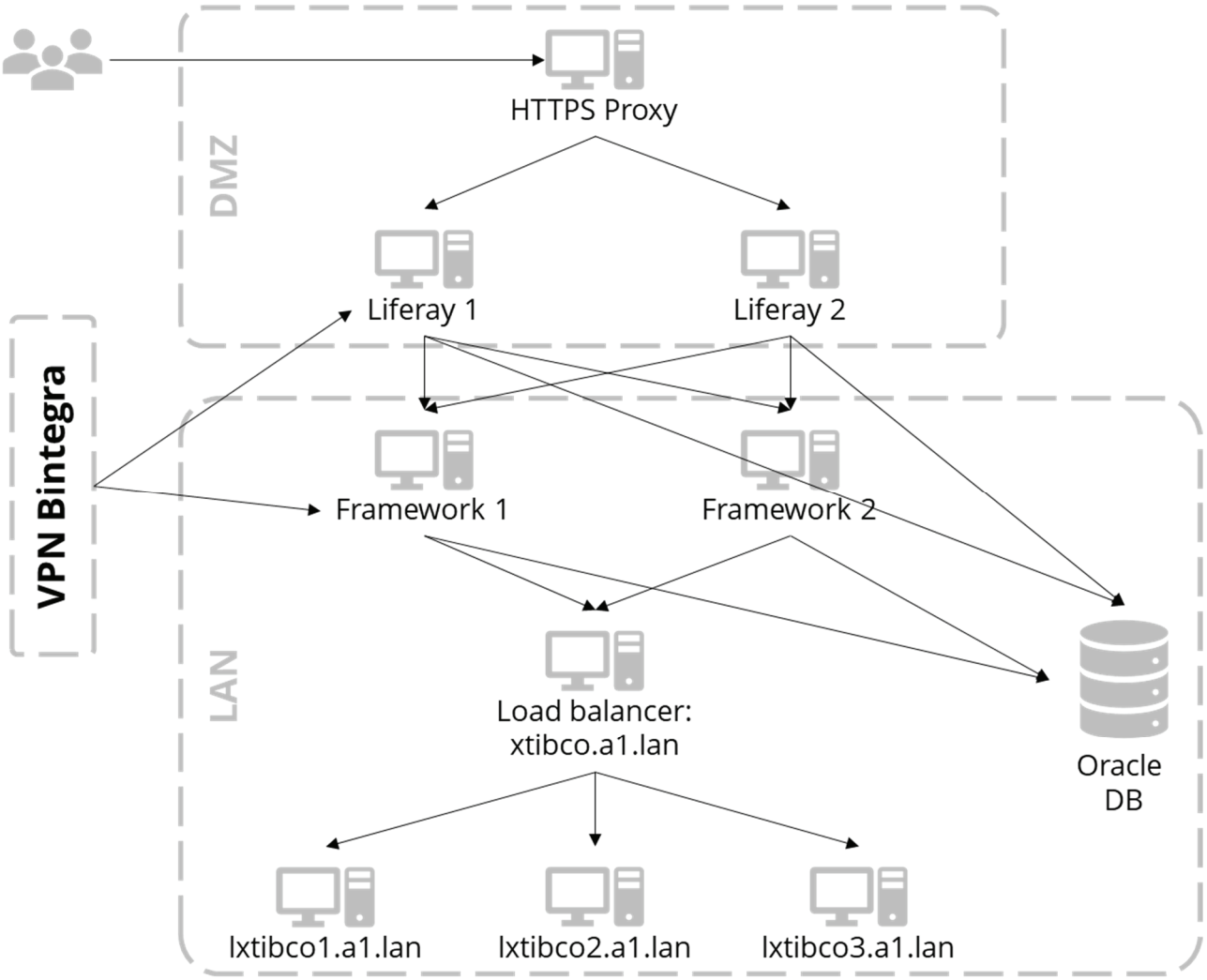
2.7 BSS aplikacija: Ocena partnerjev in zaračunavanje storitev

BSS je ločena aplikacija, ki omogoča konsolidiran izračun izkoriščenosti partnerjev, zaračunavanja in delitve prihodkov glede na njihove cenovne načrte. Aplikacija ima naslednje funkcionalnosti:

- Avtomatizirano zbiranje podatkov iz sistemov, vključenih v sporočanje (A1, zaračunavanje VAS, MGW)
- Centralizirano upravljanje cen v A1 MM (tudi za MGW in VAS)
- Priprava in izvedba obračunskega cikla
- Dostop do vseh potrebnih poročil, ki jih je potrebno poslati partnerjem v odobritev (vključno z izračunom deleža prihodkov)
- Avtomatizacijo vnosa podatkov o zaračunavanju v ERP za račune
- Zgodovina informacij za zaračunavanje in promet sporočil

2.8 Okolje rešitev

A1 MM je nameščen v dveh okoljih: testnem in produkcijskem okolju (Slika 7). Testno okolje je potrebno za namene razvoja in testiranja. Vsaka sprememba mora biti najprej preizkušena v testnem okolju. Testno okolje se uporablja tudi za osnovno stresno testiranje.



Slika 7: Okolje A1 MM

Novo vzporedno testno okolje je sestavljeno iz dveh strojev. Eno za front-end in eno za ogrodje (angl. framework). Novo vzporedno produkcijsko okolje je sestavljeno iz štirih strojev. Dva za front-end in dva za okvir. Operacijski sistem na novih strojih je zadnja različica Redhat Enterprise Linux.

3 ZAKLJUČEK

Za mobilni marketing podjetja vseh velikosti potrebujejo skalabilne in robustne rešitve. Liferay Portal kot rešitev ponuja točno to – zanesljivo platformo na katero lahko zgradimo in posodobimo enostavno za uporabo portalno rešitev za stranke, zaposlene, namizne in mobilne uporabnike. Velika prednost je, da zagotavlja vse standardne aplikacije, ki jih potrebujemo za delovanje spletnega mesta ter zagotavlja enostavno razvojno okolje za nove aplikacije ali prilagoditve. Posodobitev A1 MM Portal je pokazala, da razvoj v Liferay Portal 7.1+ je veliko enostavnejši, kot kadarkoli prej, po zaslugi OSGi. Zadnja leta so pokazala, da Liferay Portal raste v smislu nabora funkcionalnosti in uporabnosti le teh in da Liferay nenehno dodaja nove projekte/produkte in ostreje ločuje med CE in EE. Liferay Portal je primeren za intranet in extranet spletne strani. Zaključek je, da Liferay Portal je eden od mnogih, a zagotovo eden bolj atraktivnih za naše razvojno okolje in podjetje. Mogoče so številne integracije, pri tem je potrebno uporabiti smiselno rešitev, glede na zastavljeno arhitekturo.

4 LITERATURA

[1] M. Marrs, “What Is Mobile Marketing & Why Does it Matter So Much?,” 2018.

[2] D. Tomas, “How to optimize your mobile marketing strategy,” 2019.

[3] “IAB,” 2019. [Online]. Available: https://www.iab.com/. [Accessed: 20-May-2019].

[4] N. Smith, “A Little SOS for SMS Marketing,” 2015.

[5] “Teradata | Data and Analytics, Cloud Analytics & Consulting.” [Online]. Available: https://www.teradata.com/. [Accessed: 20-May-2019].

[6] I. Guseva, G. Phifer, G. Tay, and M. Lowndes, “Magic Quadrant for Digital Experience Platforms,” *Gartner Inc. Report*, Feb-2019.

[7] “Liferay | Digital Experience Software.” [Online]. Available: https://www.liferay.com/. [Accessed: 20-May-2019].

[8] “OSGi™ – The Dynamic Module System for Java.” [Online]. Available: https://www.osgi.org/. [Accessed: 20-May-2019].

OTS 2019
SODOBNE INFORMACIJSKE TEHNOLOGIJE IN STORITVE
M. Heričko in K. Kous (ur.)

Univerzitetna založba
Univerze v Mariboru

RAZVOJ UNIVERZALNEGA PODATKOVNEGA NIVOJA – GONILO DIGITALIZACIJE GENERIČNIH FARMACEVTSKIH PODJETIJ

ALEŠ TEMELJOTOV, SAŠA SOKOLIĆ, DEJAN DOVŽAN, MARJAN KALIGARO

Povzetek: *Industrijo generičnih farmacevtskih izdelkov samo v EU sestavlja preko 700 podjetij, ki zaposlujejo preko 130.000 ljudi. V zadnjem času so se omenjena podjetja znašla pod vse večjimi pritiski: od intenzivnejše konkurence, zmanjševanja cen zdravil, večanja regulatornih zahtev in naraščajočih stroškov. Rešitve je možno iskati preko digitalizacije proizvodnje ter popolno uvajanje tehnologij industrije 4.0. Podjetje Metronik se je tega lotil na izvoru, in sicer pri podatkih, ki nastajajo v procesu proizvodnje. Preko enovite podatkovne platforme omogočamo dostop do produkcijskih podatkov neglede na njihov vir. Podatke pripravljamo (tj. ustrezno kontekstualiziramo) za nadaljno uporabo v poslovnih procesih, oz. aplikacijah.*

Ključne besede: *Industrija 4.0, kontekstualizacija podatkov, enovit dostop do procesnih podatkov, legacy sistemi in podatki, digitalizacija proizvodnje*

NASLOVI AVTORJEV: Aleš Temeljotov, vodja projektne pisarne, Metronik d.o.o., Ljubljana, e-pošta: ales.temeljotov@metronik.si. dr. Saša Sokolić, direktor marketinga in prodaje, Metronik d.o.o., Ljubljana, e-pošta: sasa.sokolic@metronik.si. dr. Dejan Dovžan, vodja projektov, Metronik d.o.o., Ljubljana, e-pošta: dejan.dovzan@metronik.si. mag. Marjan Kaligaro, pomočnik direktorja oddelka digitalizacija, Metronik d.o.o., Ljubljana, e-pošta: marjan.kaligaro@metronik.si.

https://doi.org/10.18690/978-961-286-282-4.19 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 UVOD

Industrijo generičnih farmacevtskih izdelkov (GP) v EU sestavlja 700 podjetij, katera zaposlujejo 130.000 ljudi in ustvarjajo približno 50 milijard EUR prihodkov letno. Omenjena podjetja se ukvarjajo s proizvodnjo in prodajo stroškovno učinkovitih ter življenjsko pomembnih zdravil. Poleg tega ima omenjena industrija ključno vlogo pri zmanjševanju finančnega bremena za javne sisteme zdravstvenega varstva v EU, kateri porabijo povprečno 1,5% državnega BDP za zdravila. Vendar je GP pod vse večjimi pritiski: intenzivnejša konkurenca (tudi iz držav izven EU), zmanjševanje cen zdravil, večanje regulatornih zahtev za večjo varnost zdravil in naraščajoči stroški. Zaradi omenjenega se marže teh podjetij zmanjšujejo, kar se posledično kaže v zmanjševanju (potrebnih) naložb in z zapiranjem proizvodnih obratov (npr. Novartis v Sloveniji, Teva na Madžarskem). Slednje povzroča veliko izgubo delovnih mest in nepotrebne zamude.

Na srečo je rešitev za te izzive na obzorju, in sicer digitalizacija proizvodnje in popolno uvajanje tehnologij industrije 4.0. Podjetja GP lahko z njimi pridobijo globlji vpogled v svoje proizvodne procese, jih izboljšajo z odkrivanjem optimizacijskega potenciala, zagotavljanjem večje stabilnosti procesa, izboljšanjem kakovosti, povečanjem učinkovitosti, odkrivanjem medsebojnih povezav parametrov in pospeševanjem tržnih zagonov. Poleg omenjenega lahko drastično zmanjšajo proizvodne stroške (za 1,3 - 2,5 milijarde EUR/leto) in povečajo svojo konkurenčnost. Omenjena podjetja se zavedajo potenciala digitalizacije, vendar so trenutno omejena, saj noben udeleženec na trgu ni uvedel izvedljive rešitve za celovito digitalizacijo. Zato ne morejo premagati sedanjih digitalizacijskih ovir, katere so nekorektirani, nekontekstualizirani podatki v različnih oblikah, razpršenih med različnimi proizvodnimi stroji, procesnimi sistemi in bazami podatkov, ki niso uporabni za celovito analizo. Zaradi omenjenega intenzivno iščejo rešitev, ki omogoča hiter, širok in stroškovno učinkovit dostop do kontekstualiziranih produkcijskih podatkov za neomejeno uporabo.

V tem prispevku želimo podati ugotovitve, do katerih smo, v skoraj treh desetletij dela v GP industriji v EU, prišli v podjetju Metronik. Prikazali bomo rešitev, ki naslavlja dve ključni oviri, ki v GP (in ostalih proizvodnih) podjetjih onemogočata kakovostno uporabo velike količine podatkov:

- obstoječi podatki so zaklenjeni v lokaliziranih skladiščih (z velikokrat ne najbolj intuitivnim načinom programskega dostopa do njih),
- pridobivanje novih podatkov je drago.

2 DIGITALIZACIJA RIGIDNIH PROCESNIH SISTEMOV?!

V GP industriji je proizvodnja pretežno strojna, kjer je praktično vsak stroj (npr. granulator, mlin, sušilnik, tabletirka, itd.) sposoben generirati podatke o aktivnostih, ki jih izvaja ali jih tudi uporabiti v smislu navodil za upravljanje svojega delovanja. Posledično pridemo v situacijo, ko je teh podatkov ogromno in jih je izziv pravočasno generirati, oz. uporabiti.

Osnovni problem pri uporabi je običajno v zaklenjeni in razpršeni podatkovni infrastrukturi, kar pomeni, da se podatki hranijo v samih strojih, proizvodnih sistemih, relacijskih podatkovnih bazah, Historian podatkovnih strežnikih, itd.. Poleg tega so praviloma hranjeni v različnih oblikah, nepovezani med seboj in nekontekstualizirani, kar seveda pomeni, da tudi če uspemo priti do podatkov, ti niso neposredno primerni za uporabo (npr. analize, ...).

Glede na izkušnje lahko ocenimo, da imajo v tipični generični farmacevtski družbi, ki je običajno podjetje srednje velikosti, približno 200 strojev v proizvodnji. Omenjeni stroji uporabljajo vsaj 10 različnih proizvodnjih procesov, vsak s svojo bazo podatkov in posledično tudi mehanizmi za zbiranje in shranjevanje podatkov. Tudi če se osredotočimo izključno na beleženje časovnih procesnih podatkov in vzpostavimo t.i. centraliziran historian strežnik (npr. GE Historian ali OSI PI, kar je običajna praksa omenjenih podjetij za reševanje težave dostopa do procesnih podatkov), vidimo, da je to samo del rešitve, saj takšna postavitev omogoča le časovno beleženje časovnih vrst podatkov brez učinkovitega mehanizma za kontekstualizacijo.

Uporaba obstoječih praks tako pogosto vodi v situacije, ko se isti podatki po nepotrebnem hranijo v večih podatkovnih bazah, kar bi lahko označili kot nerazumno višanje stroškov, ali pa se podatki pripravljajo, oz. hranijo samo za določeno aplikacijo, brez prave (in/ali enostavne) možnosti, da bi jo ponovno uporabile druge aplikacije (slika 1).

Slika 1: Business intelligence (BI) in Operational intelligence (OI) aplikacije

To seveda pomeni, da je vsaka nova, oz. dodatna aplikacija obsojena na izvedbo analize podatkovnih virov in implementacijo ustreznih konektorjev, s katerimi aplikacija dostopa do potrebnih podatkov, ki jih mora pred uporabo praviloma tudi ustrezno obdelati (normalizacija, povezava podatkov iz večih različnih virov, itd.). Prav tako takšne aktivnosti neizogibno vključujejo tudi posege v (praviloma) kritično IT infrastrukturo, nesorazmerno povečujejo obremenitev IT omrežij, s čimer se povzroča stroške uvajanja in vzdrževanja.

Možno je zatrditi, da želijo podjetja GP izkoristiti svoje proizvodne podatke, da bi izboljšali svoje proizvodne procese in odkrili optimizacijski potencial, zagotovili višjo stabilnost procesa, povečali učinkovitost in odkrili medsebojne korelacije parametrov. Poleg tega bodo izboljšave njihovih procesov pomenile varnejša zdravila (višja kakovost), hitrejše doseganje novih zdravil na trgu in nižje cene, kar so pomembne koristi za bolnike v EU (in seveda tudi izven EU).

Študija skupine The Boston Consulting Group (BCG) je ugotovila, da lahko podjetja GP (kot procesna podjetja) v naslednjih 10. letih zmanjšajo svoje skupne proizvodne stroške za 5 - 10%, če lahko popolnoma izkoristijo potencial digitalizacije s tehnologijami Industrije 4.0. Glede na povprečne proizvodne stroške industrije GP, ki znašajo približno 50% prihodkov, in skupni prihodki industrije v višini 220 milijard EUR, znaša potencial za letno izboljšanje med 5,5 do 11,0 milijarde EUR. Hkrati bo digitalizacija pomagala podjetjem GP, da prihranijo čas in denar, ki je potreben za zagotavljanje skladnosti z vedno strožjimi regulativnimi zahtevami (npr. za preprečevanje ponarejanja - večja varnost zdravil, poročila o proizvodnji in sledljivost (CFR 210) - višja kakovost zdravil).

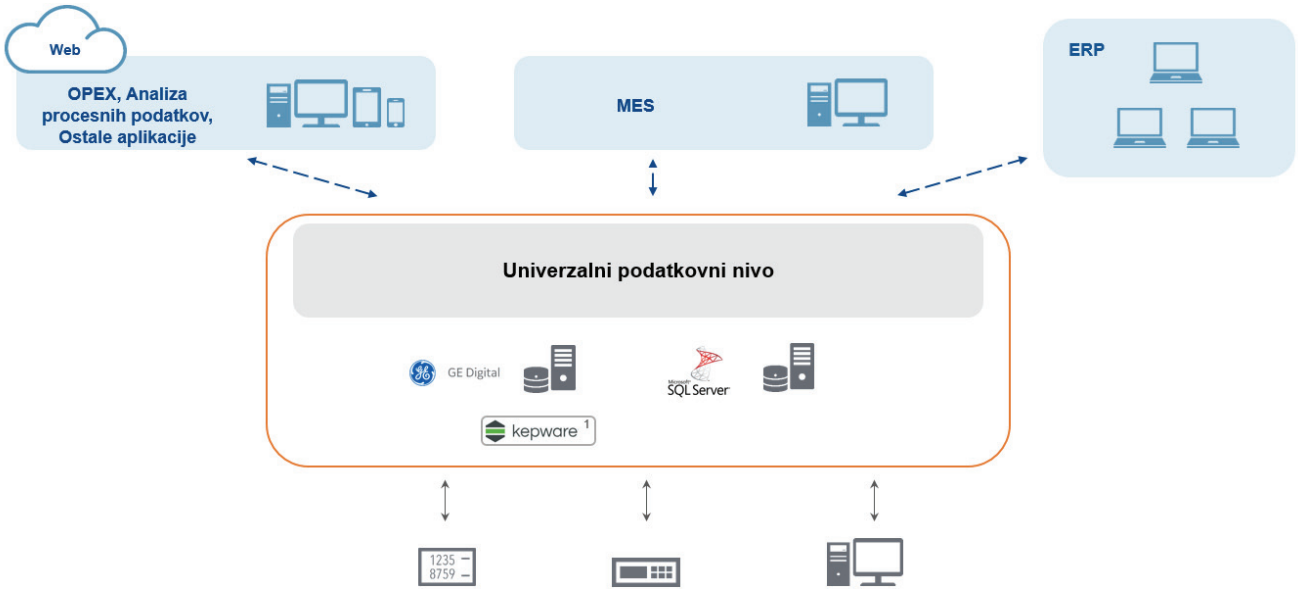
Na žalost, kot je bilo že omenjeno, obstoječe razmere ne dovoljujejo njihovim poslovnim analitikom uporabe specializiranih aplikacij za kakršno koli bolj sofisticirano analizo med proizvodnjo, da bi pridobili globlji vpogled v proizvodnjo in jo (posledično) optimizirali. To za podjetja GP pomeni, da imajo ogromne možnosti za izboljšave na dosegu roke, vendar jih zaradi trenutnih digitalizacijskih ovir ne morejo izkoristiti, saj imajo težave z kompleksnimi, izoliranimi in razpršenimi produkcijskimi podatkovnimi viri, ki izhajajo iz ogromnega števila različnih proizvodnih strojev in sistemov, ki ustvarjajo različne vrste podatkov in jih hranijo na več lokacijah in podatkovnih bazah. V praksi se težave prezentirajo tako, da so posamezna podjetja že prenehala dodajati nove aplikacije za analizo proizvodnih procesov, saj želijo počakati do točke, ko bo omogočen poenostavljen dostop do podatkov iz strojev in sistemov, stroški povezljivosti pa se bodo primerno zmanjšali.

Danes večina naprav v proizvodnih linijah v podjetjih GP, iz že navedenih razlogov, še vedno ni sistematično integrirana v informacijske sisteme in kot takšne dejansko ne zagotavljajo podatkov za večnamensko uporabo (tudi za BI in OI aplikacije z dodano vrednostjo). Zaradi tega je še vedno veliko dokumentacije v fizični obliki, ki že po definiciji zelo otežuje tako operativno delo kot tudi izvajanje morebitnih analiz na podlagi zgodovinskih podatkov. Ker regulatorji predpisujejo vedno strožja pravila za elektronsko sledenje proizvedenih farmacevtskih izdelkov in želijo razširiti isto načelo tudi na druge proizvodne procese, podjetja GP čutijo vedno večjo potrebo po digitalizaciji. Ta digitalizacija je prav tisto, kar jim lahko pomaga povečati

konkurenčnost proti proizvajalcem iz tretjih držav, ki nastopajo na trgu z agresivno cenovno politiko. Vedno bolj se zavedajo možnega vpliva (obstoječih in novih) podatkov o proizvodnji, če se obdelujejo s pravimi orodji, oz. programi. Potrebujejo samo sredstvo, ki omogoča učinkovito povezovanje obeh in izkoriščanje ugodnosti.

3 UČINKOVITA DIGITALIZACIJA PROIZVODNJE

Rešitev vsakega problema je potrebno iskati in reševati pri samem viru. V dosedanjem tekstu smo jasno identificirali izvorni problem, ki je težava z dostopom do podatkov. V prispevku tako analiziramo možnost načrtovanja, razvoja ter implementacije celovitega in stroškovno učinkovitega univerzalnega podatkovnega nivoja (slika 2).



Slika 2: Umestitev univerzalnega podatkovnega nivoja

Ideja takšnega univerzalnega podatkovnega nivoja je v omogočanju široke in hitre digitalizacije proizvodnje GP, z vzpostavitvijo neomejenega dostopa do kontekstualiziranih produkcijskih podatkov, s čimer se preoblikuje digitalizacijska vrednostna veriga. Preoblikovanje verige je možno preko zmožnosti modularnega združevanja optimalnih BI/OI aplikacij in s skladiščenjem podatkov (npr. nižji operativni stroški, višja kakovost, višja stabilnost delovanja preko manj posegov v procesne enote, zaradi uvajanja novih aplikacij, odpiranje trga tudi za ponudnike aplikacij, ki niso neposredno povezani s procesnimi elementi proizvodnje).

V splošnem takšen univerzalni podatkovni nivo prinaša naslednje prednosti:

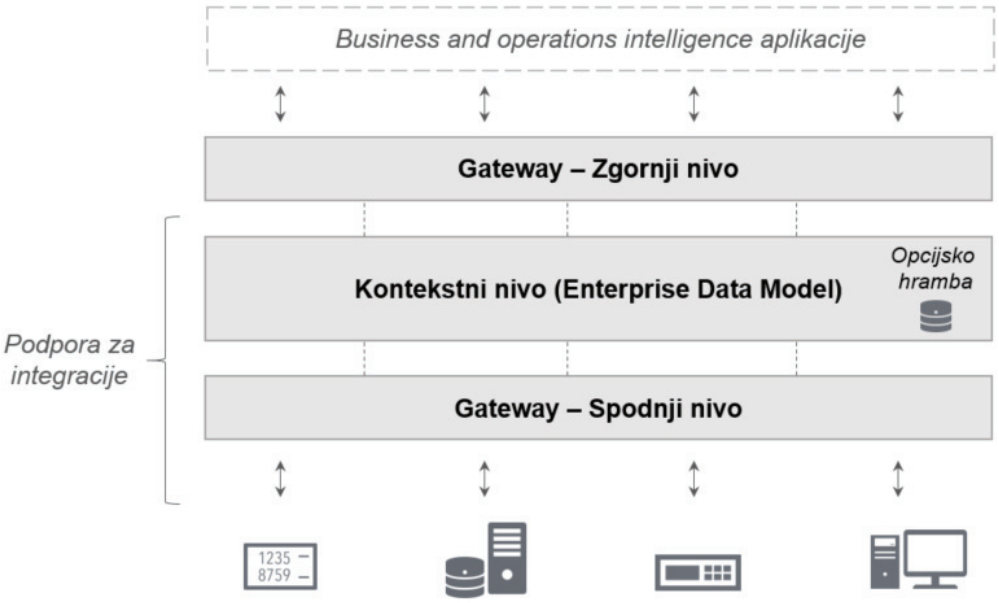
- omogoča podjetjem GP popolno prilagodljivost za aplikacijski sloj: prosto izbiro optimalnih aplikacij BI in OI, ki niso le vnaprej določeni in morajo poleg samih vsebinskih funkcionalnosti vedno znova implementirati tudi mehanizme za dostop do virov podatkov,
- omogoča podjetjem GP popolno prilagodljivost plasti za shranjevanje: prosto izbiro programske opreme za shranjevanje,
- najhitrejši (in najcenejši) za uvajanje in povezovanje z viri podatkov: namesto 4-5 milijonov EUR za srednje veliko GP podjetje, bi takšna implementacija stala približno 1 milijon EUR; prav tako bo pospešilo izvajanje novih aplikacij za 3-4 krat,
- zagotovitev najhitrejše podatkovne poizvedbe (s povprečja 12 sekund na manj kot 0,5 sekund),
- omogoča kontekstualizacijo podatkov in varnost na najvišji ravni (v skladu s ključnimi standardi proizvodnje GP, kot sta CFR, ISA),
- ni v nasprotju z obstoječimi rešitvami in ne zahteva sprememb obstoječih aplikacij (npr. baze podatkov, aplikacije), saj deluje kot čisti prehodni sloj, ki lahko z njimi deluje skladno.

Univerzalni podatkovni nivo lahko razumemo kot informacijsko rešitev, ki igra vlogo infrastrukturne komponente v podjetjih GP. Posledično lahko rečemo, da bo imela transformacijski učinek na podjetje - tako v smislu tehnologije kot poslovanja. Tehnološko je revolucionarna v smislu, da bo prelomila meje med podatkovnimi bazami in aplikacijami tradicionalnih vertikalnih rešitev, ki smo jih navajeni sedaj. Poleg tega temelji na najsodobnejših tehnologijah industrije 4.0, kot so umetna inteligenca, povezljivost platform in napredna analitika velike količine podatkov.

3.1 Arhitektura rešitve

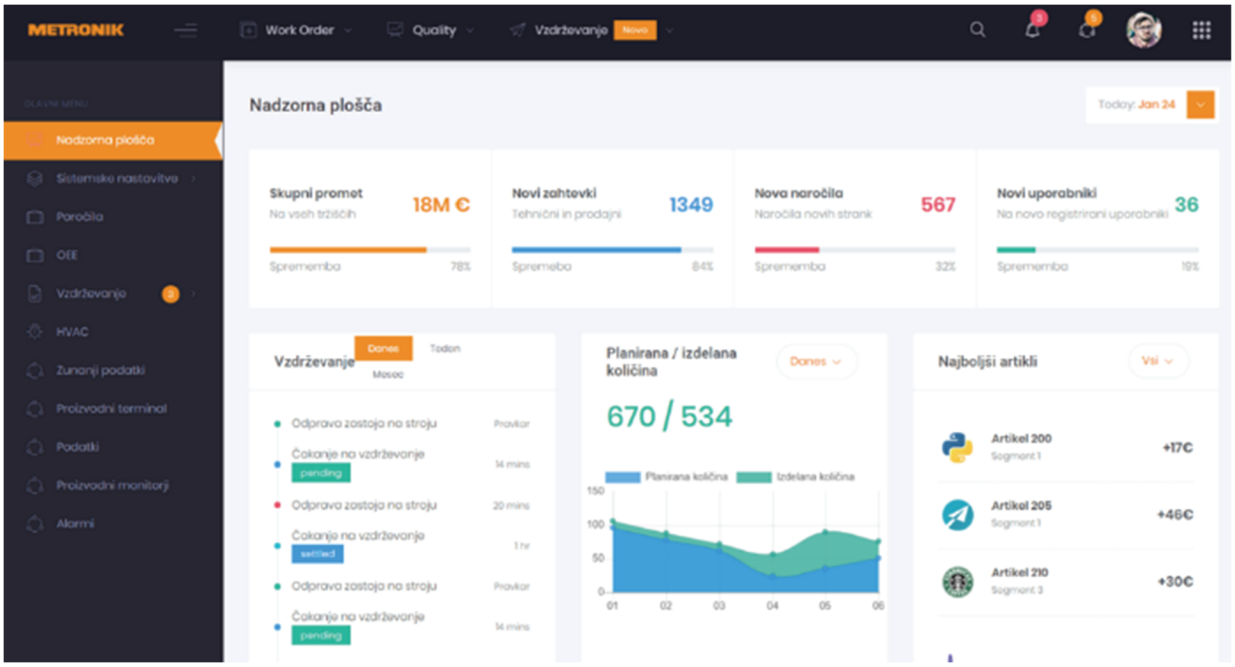
Rečemo lahko tudi, da z uporabo univerzalnega podatkovnega nivoja nastaja nova vrednostna veriga. Z aplikacijskim prehodom (slika 3) kot neodvisnim elementom, ki deluje kot platforma za povezavo produkcijskih podatkovnih virov z aplikacijami, se lahko predhodno izolirane vertikalne aplikacije razdelijo na elementarne komponente z možnostjo ponovno uporabe. Te komponente se nato lahko fleksibilno izberejo in združijo z uporabo aplikacijskega prehoda. Z brezšivnim dostopom do osnovnih podatkov bodo različni ponudniki aplikacij pridobili sposobnost prilagajanja svojih BI /OI aplikacij za določene vire podatkov na stroškovno učinkovit in hiter način ter dodali svojo poslovno vrednost nad univerzalni podatkovni nivo. Pri tem je (v kontekstu GP podjetij) potrebno upoštevati tudi dejstvo, da ponudniki (drugače kakovostnih) aplikacij velikokrat nimajo lahkega vstopa v farmacijo, predvsem zaradi izredno strogih predpisov.

Sam univerzalni podatkovni nivo je torej mogoče razumeti kot premik paradigme na relativno zaprtem trgu kompetentnih podjetij za avtomatizacijo. Poleg tega je interoperabilen z obstoječimi rešitvami in podatkovno infrastrukturo, kar pomeni, da bo zagotovil svoje prednosti brez (skritih) dodatnih potrebnih naložb (npr. v podatkovnih bazah).



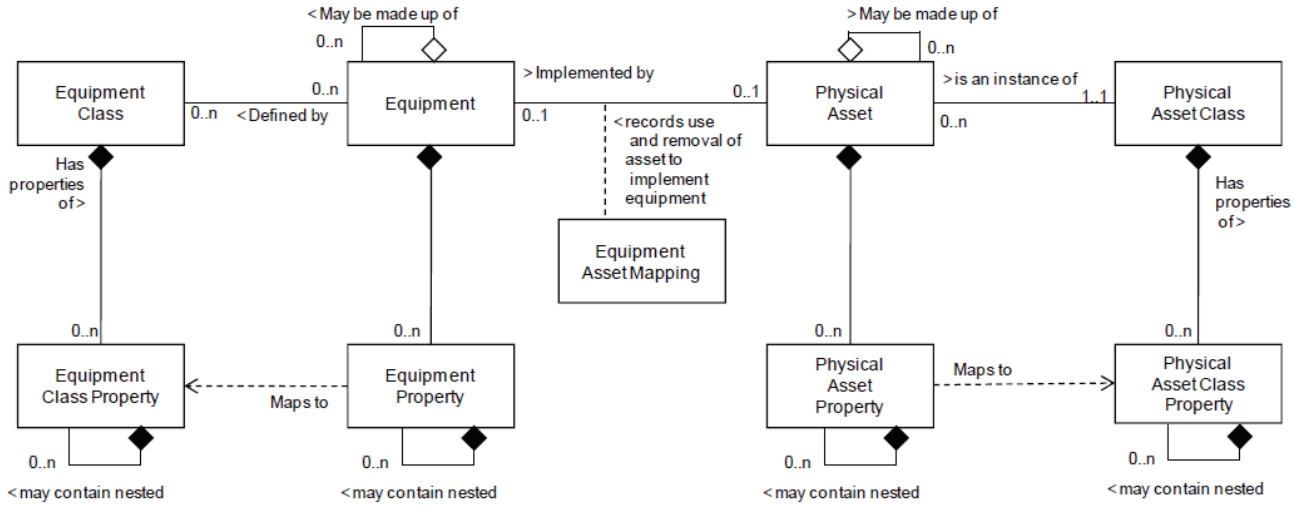
Slika 3: Vrhnji pogled na arhitekturo

Zgornji sloj aplikacijskega prehoda zagotavlja enoten dostop CRUD (ustvarjanje, branje, posodabljanje, brisanje) do proizvodnih podatkov (in po izbiri za shranjevanje podatkov) za BI in OI. Zgornja aplikacijska plast je tako zadolžena za izpostavitev podatkov na standardiziran način v skladu s strukturo standardov ISA / ANSI 88/95. Aplikacije se morajo zavedati samo standardizirane podatkovne strukture, brez skrbi po načinu dostopa do dejanskih podatkov v zalednih sistemih/napravah. Iz tega izhaja že zapisana trditev glede hitrejšega in lažjega razvoja odjemalskih aplikacij; primer POC uporabniške aplikacije na sliki 4.



Slika 4: Primer uporabniške aplikacije, ki do podatkov dostopa preko univerzalnega podatkovnega nivoja

Spodnja plast aplikacijskega prehoda je platforma za produkcijske podatkovne vire - zagotavlja mehanizme za komunikacijo z napravami in drugimi viri podatkov v proizvodnji prek standardnih komunikacijskih storitev. Omogoča preslikavo strukture podatkov proizvodnje v strukturo ISA/ANSI. S preprogramirano logiko konektorjev naprave je sistemski integrator odgovoren le za določanje preslikave, ki pospeši proces povezave do 7-krat v primerjavi s klasičnimi rešitvami, kjer je integrator odgovoren za razvoj celotnega priključka.



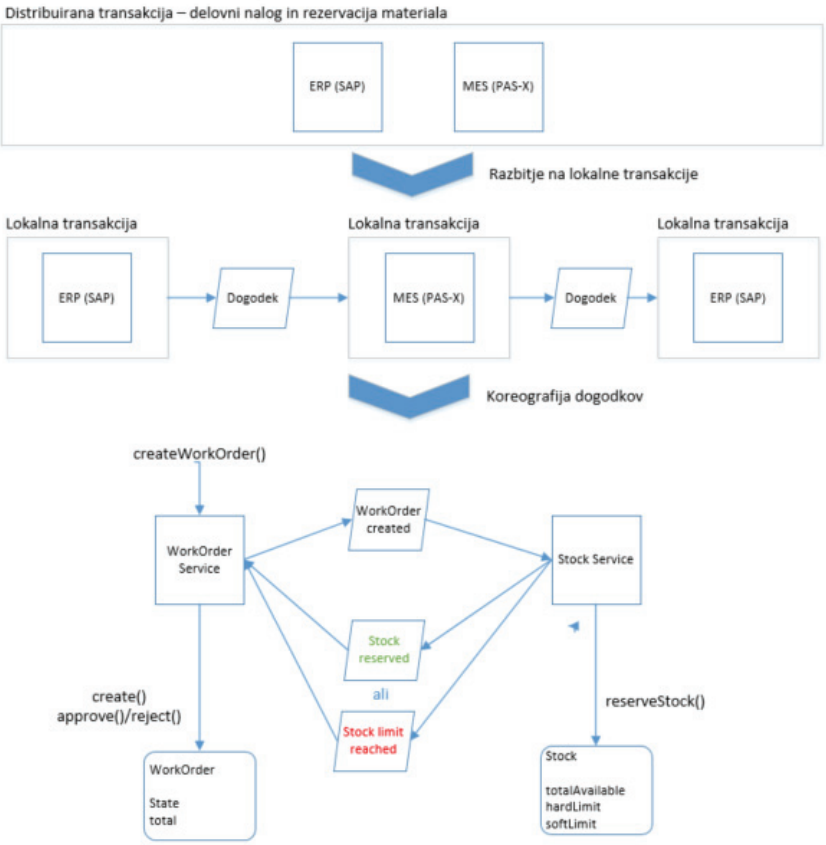
Slika 5: Standardni podatkovni model (izsek) [7]

Mehanizem za kontekstualizacijo je odgovoren za ustrezne klice API-jev in kontekstualizacijo podatkov (tj. standardni podatkovni model podjetja, prikazan na sliki 5), ki (med drugim) omogoča generiranje poročil in skrbi za celoten varnostni model rešitve. Vključuje osrednje upravljanje pravic in revizijsko sled o izvajanju CRUD operacij, ki omogočajo skladnost CFR in GMP v aplikacijskem prehodu (ključnega pomena za regulatorje - FDA in EudraLex). V primerjavi s trenutnimi pristopi, kjer je varnost rešena na aplikacijskem sloju, to omogoča veliko hitrejše varnostne nastavitve, boljši nadzor in uporabo aplikacij, ki nimajo vgrajenega varnostnega modela. Poleg tega omogoča upravljanje uporabniških privilegijev prek nastavitvenih datotek (in ne s klikom), kar sistemskim integratorjem omogoča, da to dejavnost opravljajo do 10-krat hitreje.

Integracijski modul je niz podpornih programskih orodij, ki pospešujejo projekte sistemske integracije in nastavitev kontekstualizacije podatkov za 3-4 krat (tj. posledično tudi cenejše) z usmerjanjem in pomočjo integratorju skozi proces. Ponuja vnaprej pripravljene modele povezav in avtomatizira določene transakcijske procese.

3.1.1 *Zagotavljanje podatkovne konsistence*

Dodaten izziv pri implementaciji univerzalnega podatkovnega nivoja so operacije, ki spreminjajo podatke na samih podatkovnih virih. V primeru standardnih podatkovnih skladišč bi se v tej situaciji lahko zanesli na že obstoječe, oz. implementirane mehanizme izvajanja distribuiranih transakcij. Žal v opisanem okolju to ni (vedno) možnost. Posledično se kot rešitev pojavlja implementacija koreografiranega SAGA vzorca, ki vsako transakcijo, ki vpliva na več kot en podatkovni vir razbije na več lokaliziranih, oz. ločenih transkacij in nadzoruje njihovo izvajanje preko uporabe ustreznih dogodkov (slika 6).



Slika 6: Implementacija SAGA vzorca

Na zgornji sliki je prikazan primer izvajanja distribuiranih transakcij SAGA s koreografskim pristopom dveh ločenih lokalnih transakcij: ustvarjanje delovnega naloga v MES in zagotavljanje zalog v ERP. Na podlagi dogodkov sta ti dve lokalni transakciji povezani in omogočata skladno potrjevanje sprememb ali povrnitev v primeru napake.

4 ZAKLJUČEK

Na podlagi analize ter pripravljene arhitekture je bil izdelan POC za omenjen univerzalni podatkovni nivo. Preko razvoja so bile potrjene vse predpostavke, na katerih smo utemeljili samo rešitev. Z dokončno razvito rešitvijo tako pričakujemo, da bomo lahko našim odjemalcem ponudili stabilno in skalabilno rešitev, ki bo omogočala:

- razvijalcem aplikacij: 2x hitrejšo in cenejšo implementacijo poslovnih vsebin, omogočila vstop na trg tudi tistim, ki nimajo znanja/izkušenj pri povezovanju s procesnimi podatkovnimi viri,
- GP podjetjem: do 70 % znižanje TCO iz naslova skladiščenja procesnih podatkov, razširjen prostor za implementacijo različnih BI in OI aplikacij, preprost in hiter odziv na regulatorne spremembe (čista in jasna infrastruktura), hitro izvajanje raznih analiz, ki se opirajo na podatke iz različnih podatkovnih virov,
- sistemskim integratorjem: nova, oz. dodatna ponudba v portfelju za digitalizacijo proizvodnje in hitro izvajanje integracijskih projektov.

5 LITERATURA

[1] Metronik d.o.o. (2017 – 2019). MxWeb 4.0 razvoj (interno gradivo). Ljubljana: Metronik d.o.o.

[2] Metronik d.o.o. (2018 – 2019). Unified Data Gateway Architecture (interno gradivo). Ljubljana: Metronik d.o.o.

[3] Haile, N., & Altmann, J. (2016). Value creation in software service platforms. *Future Generation Computer Systems, 55*, 495–509.

[4] Hustad, E., & De Lange, C. (2014). Service-oriented architecture projects in practice: A study of a shared document service implementation. *Procedia Technology, 16*(2014), 684–693.

[5] Antikainen, J., & Pekkola, S. (2009). Factors influencing the alignment of SOA development with business objectives. *Proceedings of the 7th European Conference on Information Systems (ECIS 2009)* (str. 2579–2590). Verona: ECIS Standing Committee.

[6] Bell, M. (2008). Service-Oriented Modeling (SOA): Service Analysis, Design, and Architecture. Hoboken: Wiley & Sons.

[7] ANSI/ISA-95.00.02-2010 (IEC 62264-2 Mod), Enterprise-Control System Integration – Part 2: Object Model Attributes

OTS 2019
SODOBNE INFORMACIJSKE TEHNOLOGIJE IN STORITVE
M. Heričko in K. Kous (ur.)

Univerzitetna založba
Univerze v Mariboru

ORODJA ZA PODPORO CELOVITEMU RAZVOJU DECENTRALIZIRANIH APLIKACIJ

PATRIK REK, MUHAMED TURKANOVIĆ

Povzetek: *Razvoj decentraliziranih aplikacij v omrežju Ethereum je zaradi vključitve številnih komponent (npr. pametne pogodbe, IPFS, RPC, Truffle) in knjižnic (npr. Web3.js, TruffleContract, IPFS) kompleksen in od razvijalca zahteva številne spretnosti in znanja. Hkrati ima razvijalec za razvoj decentraliziranih aplikacij na voljo številna ogrodja kot so React.js, Vue.js ali Angular, ki pa ne ponujajo načina za enostavno vključevanje in združevanje omenjenih komponent ter konsistentnega razvoja z uporabo teh. Trenutno so v nastajanju orodja, ki naslavljajo zgoraj omenjeno problematiko in so namenjena podpori celovitemu razvoju decentraliziranih aplikacij (npr. Drizzle, Vortex, Web3-React). Namen orodij je samodejno združevati omenjene komponente decentraliziranih aplikacij in omogočiti konsistenten in poenostavljen način razvoja. V prispevku bomo raziskali uporabnost različnih podpornih orodij in njihove specifične funkcionalnosti pri razvoju decentraliziranih aplikacij. Prav tako bomo orodja med seboj primerjali in izpostavili prednosti posameznih. Zaključili bomo s predstavitvijo uporabe teh na praktičnem primeru.*

Ključne besede: *ethereum, veriga blokov, decentralizirane aplikacije, web3, podpora razvoju*

NASLOVA AVTORJEV: Patrik Rek, podiplomski študent, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Blockchain Lab:UM, Maribor, Slovenija, e-pošta: patrik.rek@um.si. dr. Muhamed Turkanović, docent, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Blockchain Lab:UM, Maribor, Slovenija, e-pošta: muhamed.turkanovic@um.si.

https://doi.org/10.18690/978-961-286-282-4.20 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 UVOD

Decentralizirane aplikacije so aplikacije, ki svoje podatke in zapise operacij hranijo v porazdeljeni glavni knjigi (ang. distributed ledger). Imenujemo jih decentralizirane, saj so specifične iz stališča, da niso povezane s poslovno logiko in podatki odloženimi na centralni entiteti ali shrambi (tj. strežniku), temveč s poslovno logiko, ki temelji na pametnih pogodbah verig blokov ali s podatki, ki so shranjeni na verigi blokov. Decentralizacija ne pomeni nujno popolne decentralizacije, temveč zgolj dejstvo, da je njihovo delovanje odvisno od decentraliziranega sistema, ki je v večini primerov veriga blokov v podpori s pametnimi pogodbami [1].

Ne obstaja uradna definicija decentraliziranih aplikacij, temveč neuradno uveljavljeno pravilo, da se aplikacija smatra kot decentralizirana, če zagotavlja določene lastnosti, kot so: (1) odvisnost od decentraliziranih in porazdeljenih sistemov, kot so verig blokov, (2) ne nujno tudi odprtokodnost, (3) praviloma spletna oblika izvajanja in (4) kriptografsko podpisani podatki.

V primeru klasičnih spletnih aplikacij so le te sestavljene iz HTML, CSS in Javascript izvorne kode za prikazovanje vsebine ter izvorne kode, ki se bo s pomočjo programskega vmesnika (ang. application programming interface - API) povezala na centralizirano ali porazdeljeno podatkovno bazo. Ključna razlika decentraliziranih aplikacij je ravno v programskem vmesniku, katerega v tem primeru predstavljajo pametne pogodbe na verigi blokov in je dostop do podatkov prav tako praviloma iz verige blokov. Primerjava izvajalnega okolja klasičnih in decentraliziranih aplikacij je predstavljena v tabeli 1, iz katere je razvidno tudi poimenovanje le teh na osnovi Web3 knjižnice, ki je osnova za delovanje decentraliziranih aplikacij.

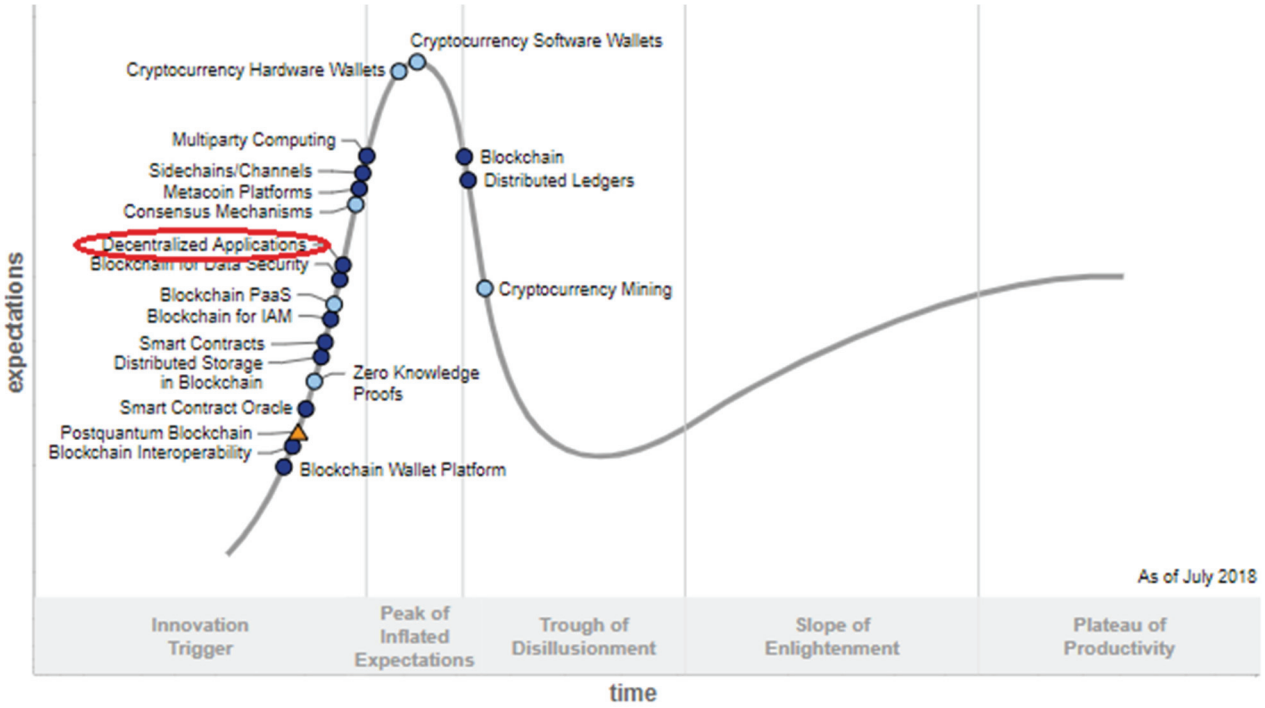
Tabela 1: Primerjava izvajalnega okolja klasičnih in decentraliziranih aplikacij.

	Klasične spletne aplikacije – Web2	Decentralizirane aplikacije – Web3
Izvedba poslovne logike	Strežnik	Omrežje P2P
Gostovanje	Spletni strežnik	Porazdeljeni CDNS IPFS ali SWARM
Storitveni sloj	HTTP API	Pametne pogodbe
Shranjevanje	Podatkovna baza	Porazdeljeni datotečni sistemi Porazdeljene glavne knjige (verige blokov)

Kot že v uvodu izpostavljeno, decentralizirane aplikacije praviloma niso popolnoma decentralizirane, saj temelji njihov oblični del (ang. frontend) praviloma na spletnih tehnologijah, katere so gostovane na centraliziranih spletnih strežnikih. Kljub temu je možno doseči tudi popolno decentralizacijo na način, da se vse izvorne datoteke obličnega dela aplikacije shranijo na porazdeljene in decentralizirane datotečne sisteme, kot sta IPFS ali Swarm. Prav tako se lahko vse potencialne druge komponente končne aplikacije, kot je izmenjava sporočil, zamenjajo s svojimi decentraliziranimi ekvivalenti, kot npr. uporaba protokola Whisper (Slika 2).

S svojo specifično arhitekturo, so decentralizirane aplikacije vzpostavile način, ki zmanjša vpliv potencialnih posrednikov, saj so lahko uporabniki le teh neposredno povezani s poslovno logiko ponudnika. Takšen zaključek je veljaven, saj lahko ponudnik določene poslovne storitve, ki temelji na pametni pogodbi, le to naloži na javno verigo blokov, katere del je tudi sam. S tem doseže stanje v katerem lahko uporabniki neposredno izkoriščajo definirano poslovno logike, pri čemer se povežejo na katero koli vozlišče izbrane verige blokov. S tem decentralizirane aplikacije dosežejo tudi večjo varnost, saj se odstrani centralna točka odpovedi oz. je ponujena storitev teoretično zagotovljena s pomočjo porazdeljenega okolja. Hkrati si ponudniki storitev na takšen način zagotovijo necenzurirano okolje v razliko od plačljivih centraliziranih ali oblačnih ponudnikov gostovanja.

Decentralizirane aplikacije se glede na Gartnerjev graf navdušenja za tehnologije veriženja blokov iz leta 2018, prikazanem na Sliki 1, nahajajo v fazi zasnove koncepta in posledično zaradi tega prožijo večje zanimanje javnosti [1].



Slika 1: Gartnerjev graf navdušenja za tehnologije veriženja blokov [1].

Ker so decentralizirane aplikacije in celotno področje decentraliziranih rešitev še v fazi razvoja in ker temeljijo na številnih specifičnih lastnosti kot tudi izvajalnem okolju (Tabela 1), je razumevanje teh ter načrtovanje in razvoj decentraliziranih aplikacij za razvijalce precej zahtevno in predstavlja precejšnjo oviro.

Z namenom reševanja omenjene problematike obstajajo različna orodja, ki bi naj razvijalcem olajšala celovit razvoj decentraliziranih aplikacij. V nadaljevanju prispevka bomo predstavili določena orodja, s katerimi je mogoče doseči preusmerjanje pozornosti razvijalcev na razvoj samih funkcionalnosti, ki jih lahko decentralizirane aplikacije ponudijo, brez potrebe po večjem osredotočanju na koncepte, na katerih le te temeljijo.

Prispevek se nadaljuje s kratkim opisom platforme Ethereum in s tem povezanimi decentraliziranimi aplikacijami, saj so le te najbolj razširjene in uveljavljene oblike decentraliziranih aplikacij [2] in se bomo na te tudi osredotočili. V poglavju 3 predstavimo in opišemo določene komponente za razvoj decentralizirani aplikacij. Osrednje poglavje je poglavje 4, kjer bomo predstavili in primerjali orodja za celovit razvoj decentraliziranih aplikacij. Zaključke predstavimo v poglavju 5.

2 ETHEREUM IN DECENTRALIZIRANE APLIKACIJE

V nadaljevanju se bomo osredotočili in predstavili orodja in ogrodja s katerimi je mogoče olajšati razvoj decentraliziranih aplikacij, temelječih na platformi verig blokov Ethereum.

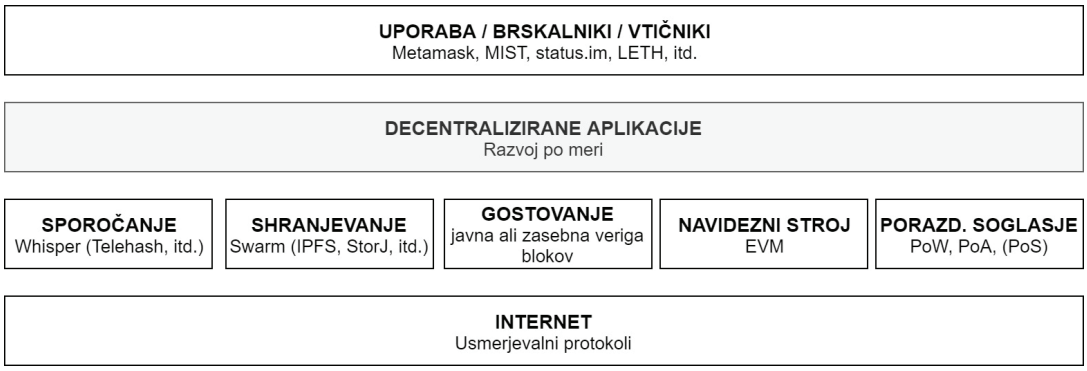
Platforma verig blokov Ethereum je razvita leta 2015 in je trenutno najbolj pogosto uporabljena platforma za razvoj decentraliziranih aplikacij [3]. Je prva platforma verig blokov, ki je omogočila razvoj in podporo pametnim pogodbam, podprtih s Turingovo polnim jezikom, kar je doprineslo do možnosti razvoja novih tipov aplikacij in poslovnih procesov povezanih s tehnologijo veriženja blokov.

Snovalci platforme Ethereum so decentralizirane aplikacije razdelili na tri osnovne tipe in sicer: (1) tiste, ki upravljajo s kriptovaluto Ether, (2) tiste, ki vključuje kriptovaluto Ether, vendar so osredotočene na drug primer uporabe in (3) tiste, ki vključujejo poljubne primere uporabe, kot so decentralizirani sistemi upravljanja (ang. decentralized autonomous organisation) itd. [4].

Kot že v uvodu omenjeno, je razvoj decentraliziranih aplikacij specifičen zaradi zasnove platform verig blokov in s tem arhitekture izvajalnega okolja, kateri je osrednja lastnost ravno decentraliziranost. Razvijalci so tako primorani upoštevati specifične načine upravljanja decentraliziranih aplikacij, saj se jih uporablja in upravlja s pomočjo namenskih brskalnikom (npr. MIST, status.im) oz. vtičnikov (npr. Metamask), kot prikazuje slika

2. Omenjeni namenski brskalniki, vključujejo tudi t. i. digitalno denarnico in so po neuradni definiciji sami po sebi decentralizirane aplikacije. Predstavljen način uporabe in upravljanja decentraliziranih aplikacij je nujen, saj se morajo v sklopu tehnologije veriženja blokov, vse spremembe stanj digitalno podpisovati z zasebnimi kriptografskimi ključi, pri čemer temelji podpisovanje na algoritmih ECDSA (ang. Elliptic Curve Digital Signature Algorithm) [5].

V odvisnosti od tega, kolikšno stopnjo decentraliziranosti končnih aplikacij želimo, temu primerno lahko načrtujemo arhitekturo le te. Kot primer smo že v uvodu navedli uporabo sporočilnega protokola Whisper, ki temelji na decentralizirani osnovi platforme Ethereum. Podobno je s porazdeljenim datotečnim sistemom Swarm, kateri prav tako temelji na platformi Ethereum. Obe komponenti lahko nadomestimo tudi s alternativnimi rešitvami, ki niso neposredno povezane s platformo Ethereum, kot npr. uporaba sporočilnega protokola Telehash ali porazdeljenih datotečnih sistemov IPFS oz. StorJ (Slika 2).



Slika 2: Abstrahiran Web3 sklad, temelječ na platformi Ethereum [6].

Potrebno je tudi omeniti, da lahko decentralizirane aplikacije temeljijo na javnih ali zasebnih oz. konzorcijskih omrežjih, pri čemer izbira le tega vpliva tudi na načrtovanje, razvoj in delovanje končnih decentraliziranih aplikacij. Razlike so lahko v sami učinkovitosti izvedbe končnih aplikacij, kot tudi konceptualnem delovanju le teh, saj se v primeru zasebnih omrežjih lahko izognemo plačevanju goriva za izvedbo transakcij, ki spreminjajo stanje. Na delovanje decentraliziranih aplikacij, lahko v primeru zasebnih omrežij, vpliva tudi izbira porazdeljenega soglasja, na katerem temelji izbrana veriga blokov, saj je s tem povezana tudi končnost transakcij, tj. čas potreben, da smo lahko prepričani, da bo izvedena transakcija dokončno sprejeta (ang. persistent). Zaradi samega razumevanja delovanja decentraliziranih aplikacij in učinkovitega načrtovanja le teh, morajo razvijalci poznati in razumeti tudi delovanje navideznega stroja Ethereum, ki predstavlja dejansko izvajalno okolje pametnih pogodb. V povezavi s prej omenjenima dejstvoma, je izvajanje decentraliziranih aplikacij nujno asinhrono, kar predstavlja dodatno breme razvijalcem, ki želijo načrtovati in razviti uporabniško prijazno končno aplikacijo.

V nadaljevanju se bomo osredotočili na določene tehnične značilnosti razvoja decentraliziranih aplikacij. Izpostavili bomo orodja, ogrodja in knjižnice, ki so pomembne pri razvoju decentraliziranih aplikacij. Večina jih temelji na programskem jeziku JavaScript [7] ter izvajalnem okolju NodeJS [8].

2.1 JavaScript ogrodja

Z namenom lažjega povezovanja logičnega s predstavitvenim nivojem spletnih aplikacij, obstajajo številna ogrodja, med katera spadajo React, Angular in Vue.js [9]–[11].

React je JavaScript deklarativno ogrodje za oblikovanje in gradnjo uporabniških vmesnikov, ki je zasnovan na različnih komponentah. Komponente so enkapsulirane in imajo svoje stanje, kar z združevanjem več različnih komponent omogoča razvoj kompleksnejših uporabniških vmesnikov. Logika komponent je zapisana s programskim jezikom JavaScript, predstavnostni nivo knjižnice je zapisan v XML podobni sintaksi, imenovani JSX. Aplikacijo, temelječo na ogrodju React je mogoče poganjati tako na strežniški, kot tudi na strani odjemalca. Zaradi na programskem jeziku JavaScript zasnovanih komponentah, so le te prilagodljive in vodijo stanje, ki se v ozadju pošlje na strežnik ali, v našem primeru, v omrežje verig blokov ter z njim tudi komunicira. Naprednejši razvijalci, ki uporabljajo ogrodje React, hkrati uporabljajo tudi Redux, ki je napredno orodje za hranjenje in upravljanje stanja aplikacij na enem mestu. Brez uporabe Reduxa je potrebno hraniti stanje

aplikacije za vsako komponento aplikacije posebej, kar v primeru verig blokov pomeni vnovična vzpostavitev povezave z omrežjem ob vsaki interakciji uporabnika z aplikacijo [9], [12].

Angular je ogrodje podobno React, le da je zasnovano na tipiziranem programskem jeziku TypeScriptu in omogoča olajšan razvoj spletnih aplikacij. Omogoča deklarativne predloge in sprotno razhroščevanje. Sama platforma prav tako omogoča gradnjo aplikacij, ki se bodo uporabljale na različnih napravah in platformah. Za ustvarjanje predlog, je mogoče uporabiti obogaten jezik HTML. Podobno, kot pri ogrodju React, tudi Angular vzpodbuja razvoj modulov oz. komponent, razdeljenih med manjše dele uporabniškega vmesnika, saj je s takim načinom razvoja mogoče doseči lažje sodelovanje med razvijalci [10].

Vue je progresivno ogrodje za gradnjo uporabniških vmesnikov. Vue je tako kot React, zasnovan na programskem jeziku JavaScript. Za razliko od ostalih prej navedenih ogrodij, je Vue zasnovan tako, da je lahko uporabljen inkrementalno, kar pomeni, da ga ni potrebno uporabljati ekskluzivno, ampak se lahko uporablja v kombinaciji z ostalimi ogrodji in se ga uporablja le delno. Ogrodje je usmerjeno zgolj na predstavitveni nivo, kar omogoča preprosto integracijo z ostalimi knjižnicami in projekti. Kljub omenjenim lastnostim je s pomočjo ogrodja Vue mogoče poganjati sofisticirane spletne aplikacije v kombinaciji z modernimi orodji in podpornimi knjižnicami. Tudi pri Vue je tako kot pri Angular, predloga zapisana v obogateni obliki HTML. Pri gradnji aplikacij s pomočjo ogrodja Vue.js je prav tako priporočeno ločevanju celotnega projekta na posamične komponente [11].

2.2 Web3.js

Za povezavo decentralizirane aplikacije z omrežjem verige blokov je potrebna posebna JavaScript knjižnica. Ena takšnih, ki to omogočajo in je tudi v praksi največkrat uporabljana, je web3.js. To je zbirka knjižnic, ki omogočajo interakcijo z lokalnim ali oddaljenim vozliščem Ethereum, z uporabo povezave HTTP, WebSocket ali IPC (inter-process communication). Zbirka knjižnic se deli na tri pomembnejše dele, in sicer (1) web3-eth, ki predstavlja dostopno točko do verige blokov Ethereum in pametnih pogodb, (2) web3-shh, ki predstavlja način za povezavo s protokolom Whisper za komunikacijo 'vsak z vsakim', in (3) web3-utils, ki vsebuje uporabne funkcije za razvijalce decentraliziranih aplikacij (pretvorba med različnimi tipi kodiranj, preverjanje veljavnosti zapisov …) [13].

S pomočjo knjižnice web3.js je mogoča tudi implementacija naročanja na dogodke (ang. events), ki so ob uspešno opravljeni spremembi stanja, s pomočjo pametne pogodbe, proženi znotraj te. Pomemben podatek je, da se web3.js in njene funkcije izvajajo sinhrono, kar prinaša dodaten izziv pri razvoju decentraliziranih aplikacijah, saj v prejšnjem poglavju opisana ogrodja delujejo asinhrono, tako da je potrebno implementirati asinhronost tudi v sodelovanju s knjižnico web3.js [13].

2.3 Orodja za razvoj

Truffle je razvojno in testno orodje za Ethereum verige blokov. Orodje Truffle omogoča prevajanje, povezovanje in nalaganje pametnih pogodb na omrežje, samodejno testiranje pametnih pogodb, upravljanje omrežij in nalaganje pametnih pogodb na poljubno število javnih in zasebnih omrežij, interaktivno konzolo za neposredno komunikacijo s pametnimi pogodbami in orodje za enostavno neprekinjeno integracijo pametnih pogodb. Ob uporabi orodja Truffle za nalaganje pametnih pogodb, lahko za delo z njimi znotraj decentraliziranih aplikacij, uporabimo JavaScript knjižnico *truffle-contract*, ki pri inicializaciji in uporabi pametnih pogodb razvijalcem olajša delo. Tako več ni potrebno ročno pridobivanje aplikacijskega binarnega vmesnika, saj za to poskrbi Truffle. S tem se razvojni cikel nekoliko skrajša in olajša, ker je uporaba funkcij bolj podobna standardom, ki jih uporablja JavaScript [14].

Embark je, podobno kot Truffle, razvojno okolje za gradnjo decentraliziranih aplikacij in nalaganje na porazdeljena omrežja. Poveže se z omrežjem verig blokov Ethereum in porazdeljeno shrambo IPFS. Ponuja samodejno nalaganje pametnih pogodb na omrežje verig blokov, izdelavo obličja, testiranje, porazdelitev decentraliziranih aplikacij na porazdeljen datotečni sistem IPFS in sporočanje 'vsak z vsakim'. Vsebuje tudi aplikacijo Cockpit, ki predstavlja vmesnik za upravljanje s pametnimi pogodbami in verigo blokov. Ob vsem pa prinaša še JavaScript knjižnico EmbarkJS, ki prinaša podobne funkcionalnosti kot knjižnica truffle-contract v primeru uporabe okolja Truffle. Omogoča povezavo decentralizirane aplikacije z omrežjem verig blokov, proženje transakcij in klicev, naročanje na dogodke, uporabo porazdeljenega datotečnega sistema IPFS in delo z uporabniškimi računi [15].

2.4 JS-IPFS

IPFS je porazdeljen datotečni sistem, ki teži k povezovanju vseh računalnikov z enakim sistemom datotek. Je datotečni sistem z različicami, ki upravlja datoteke na različnih mestih in tudi sledi njihovim različicam. Sistem je podoben delovanju protokola Bittorrent [16].

IPFS podpira razvoj decentraliziranih aplikacij, ki so popolnoma porazdeljene in nimajo centralnega strežnika ali osrednje točke izpada. Namesto naslavljanja strežnikov, v omrežju IPFS naslavljamo datoteke po njihovi zgoščeni vrednosti. Odjemalec tako po omrežju povpraša, v katerih vozliščih se iskana datoteka nahaja in od enega teh vozlišč dobi tudi odgovor – datoteko. Odjemalec jo tako prenese na svoj datotečni sistem in jo ponudi na voljo ostalim. Zato so datoteke, ki imajo večje povpraševanje, bolj porazdeljene po omrežju in imajo večjo verjetnost, da ne bodo nikoli izgubljene. Ko je datoteka spremenjena, ima novo zgoščeno vrednost, tako da lahko na tak način s pomočjo verige blokov vodimo različice datotek [17].

Na omrežje IPFS se lahko povežemo s pomočjo HTTP klicev na eno od vozlišč ali, v primeru decentraliziranih aplikacij, s pomočjo JavaScript knjižnice js-ipfs, ki omogoča povezovanje na IPFS vozlišče in dodajanje ter odpiranje datotek. Knjižnico js-ipfs je potrebno po namestitvi inicializirati in povezati z lastnim ali javnim vozliščem. Ko se povezava vzpostavi, razvijalec dobi dostop do aplikacijskega programskega vmesnika za datotečno shrambo IPFS. Posluša lahko različne dogodke, ki se sprožijo ob napakah ali prekinitvah. Ob tem pa lahko dodaja datoteke ali druge podatke na omrežje IPFS, kjer kot odgovor prejme njihovo zgoščeno vrednost, datoteke pa lahko iz omrežja ob vnosu te zgoščene vrednosti tudi prebere. Vse funkcije knjižnice js-ipfs so asinhrone [18].

3 ORODJA ZA PODPORO CELOVITEMU RAZVOJU

Glede na specifičnost razvoja decentraliziranih aplikacij in glede na številna ogrodja in knjižnice, ki jih pri tem moramo ali lahko uporabimo, so z namenom olajšanja razvoja, v nastajanju različna orodja, ki poenostavijo in poenotijo razvoj celovitih decentraliziranih aplikacij. Primeri takšnih orodij so Drizzle, Vortex in Web3-React. V nadaljevanju bomo omenjena orodja opisali in jih med seboj primerjali glede na funkcionalnosti, ki jih omogočajo. Povzetek primerjave je predstavljen v tabeli 2.

Drizzle je zbirka knjižnic namenjena razvoju obličja decentralizirane aplikacije. Cilj orodja je narediti razvoj le teh lažji in bolj predvidljiv. Jedro orodja Drizzle je zasnovano na shrambi za Redux. Drizzle poskrbi za sinhronizacijo podatkov pametne pogodbe in transakcijskih podatkov. Prinaša odzivne podatke, pridobljene iz pametnih pogodb, vključujoč stanja, dogodke in transakcije. Te lahko razvijalec uporablja na podoben način kot tradicionalne funkcije in metode v JavaScript aplikacijah. Drizzle se lahko pohvali tudi z deklarativnostjo, kjer razvijalec določi, katere podatke potrebuje in tako ne porablja računskih sredstev za podatke, ki so pri razvoju decentralizirane aplikacije nepotrebni [19].

Ker je Drizzle razvit s strani organizacije ConsenSys v sklopu zbirke Truffle Suite, je za pravilno delovanje zaželena uporaba orodja Truffle. Drizzle se namesti s pomočjo upravitelja paketov *npm* in se lahko uporablja v kombinaciji s katerim koli ogrodjem, ki omogoča Redux (npr. React). Zato je najbolj primerno ogrodje React, za katerega so razvijalci Drizzle pripravili tudi komponente, ki olajšajo razvoj decentraliziranih aplikacij. Drizzle je kompatibilen z različico 1.0 knjižnice web3 in protokolom WebSocket [19].

React podporne komponente za Drizzle vključujejo komponento, ki skrbi za preverjanje stanja povezave (LoadingContainer) z verigo blokov in pametnimi pogodbami in med nalaganjem ter ob napaki prikazuje uporabniško izbran pogled, komponento za izvajanje klicev metod pametne pogodbe in prikaz vrnjenih podatkov (ContractData) in komponento, ki iz metode pametne pogodbe pripravi vnosni obrazec (ContractForm). S temi komponentami in uporabo celotnega ogrodja React se pohitri razvoj decentraliziranih aplikacij [19].

Vortex je shramba za Redux, ki poskrbi za transakcije, pametne pogodbe, dogodke, račune, klice metod, stanje web3, pridobivanje datotek iz porazdeljene shrambe IPFS, itd. Uporaben je predvsem z ogrodjem React, saj decentralizirani aplikaciji omogoča boljšo odzivnost brez osveževanja in naredi manj zahtevkov web3 za še boljše rezultate. Uporabi se lahko tudi za prebiranje datotek iz porazdeljenega datotečnega sistema IPFS in za predpomnenje teh [20].

Omogoča hitro nalaganje vseh pametnih pogodb znotraj shrambe Redux. V predpomnilniku hrani informacije o vseh uporabnikovih dejanjih in natančno sledi transakcijam uporabnika in stanje Ethereum računa. Vsi

podatki iz pametnih pogodb so konstantno pridobljeni iz verige blokov in zapisani v predpomnilnik. Ob sproženem dogodku ob izvajanju pametne pogodbe je ta samodejno ujet. Razvoj decentraliziranih aplikacij je tako mogoč na bolj poenoten način [20].

Vortex lahko deluje v sožitju z orodjema Embark ali Truffle. Namestitev je možna z uporabo upravitelja paketov *npm*, uporabijo pa se lahko funkcije ali pripravljene komponente za React. Trenutna slabost orodja Vortex je v tem, da je trenutna različica že za časom, ker se ne sklada z nekaterimi novejšimi knjižnicami, na katere se sklicuje. V razvoju je že nova različica, imenovana *ethvtx*, ki pa še ni razvita do takšne mere kot je bila prva različica orodja Vortex [20].

Web3-react je preprosto ogrodje za gradnjo decentraliziranih aplikacij platforme Ethereum s pomočjo ogrodja React. Podpira večino najpogosteje uporabljanih orodij, ki vzpostavljajo povezavo web3 z omrežjem verige blokov (MetaMask, Infura, Portis). Je razvijalcem prijazen, saj sam vzpostavi instanco web3.js in upravlja z nastavitvami. Ne ponuja tolikšne količine pripravljenih funkcionalnosti kot prej opisana Drizzle in Vortex, a ponuja osnovo za razvoj naprednih funkcionalnosti, ki upravljajo vsak vidik decentralizirane aplikacije z verigo blokov Ethereum in uporabniškimi računi. Pri razvoju reši težavo asinhronosti, ki bi jo sicer moral razvijalec implementirati sam z izdelavo asinhronih funkcij (ang. promises) za vse obstoječe Web3.js funkcije [21].

Natančna primerjava funkcionalnosti opisanih ogrodij je prikazana v tabeli 2.

Tabela 2: Primerjava ogrodij za podporo celovitemu razvoju decentraliziranih aplikacij

	Drizzle	Vortex	Web3-React
Pomoč pri vzpostavitvi povezave	Da	Da	Da
Delo s pametnimi pogodbami	Da	Da	Ne
Izvajanje transakcij	Da	Da	Ne
Poslušanje dogodkov	Da	Da	Ne
Povezovanje z IPFS	Ne	Da	Ne
Asinhrono delovanje	Da	Da	Da
Uporaba Redux	Da	Da	Ne
Omejenost na ogrodje	Truffle in ogrodja s podporo Redux	React	React
Razvojno stanje	Pripravljeno za produkcijo, skladno z zadnjimi različicami.	Pripravljeno za produkcijo, a ni skladno z zadnjimi različicami. Nova različica je še v razvoju.	Pripravljeno za produkcijo.

4 ZAKLJUČEK

Pri celovitem razvoju decentraliziranih aplikacij obstajajo številna orodja, ki ta razvoj pohitrijo in olajšajo. Že pri razvoju pametnih pogodbah na platformi Ethereum sta v veliko podporo orodji Truffle in Embark, ki olajšata upravljanje z le temi in povezavo z decentralizirano aplikacijo. Oba ponujata podobne funkcionalnosti, tako da je odločitev za izbiro odvisna od želja vsakega posameznika.

Pri razvoju obličja decentralizirane aplikacije je smiselna uporaba orodja Drizzle, Vortex ali Web3-React, ki odpravijo težavo asinhronosti, opisano v prejšnjih poglavjih. Pri teh glede uporabnosti nekoliko zaostaja Web3-React, a je vseeno primeren za uporabo v produkciji. Funkcionalno je najbolj napreden Vortex, ki pa ni skladen z zadnjimi različicami pripadajočih knjižnic, kot so Web3 in React. Zato je trenutno najbolj primeren Drizzle, ki je sicer omejen na Truffle in ogrodja, ki podpirajo Redux. Glede na kompatibilnost vseh treh orodij je za razvoj decentralizirane aplikacije najbolj priporočena uporaba ogrodja React v kombinaciji z Redux. Posameznik pa mora oceniti svoje potrebe in se odločiti, ali takšno orodje sploh potrebuje, saj se nekatere decentralizirane aplikacije lahko brez težav razvijejo tudi brez takšnih namenskih orodij.

5 LITERATURA

[1] D. Furlonger in R. Kandaswamy, „Hype Cycle for Blockchain Technologies, 2018“, *Gartner*, 2018. [Na spletu]. Dostopno: https://www.gartner.com/document/3883991?ref=solrAll&refval=218071076&qid=4a3d0d5adbc19525818689d85. [Dostopano: 11-mar-2019].

[2] DappRadar, „Dapps Rankings | DappRadar“, 2019. [Na spletu]. Dostopno: https://dappradar.com/rankings. [Dostopano: 28-maj-2019].

[3] Ethereum, „Beginners | ethereum.org“, 2019. [Na spletu]. Dostopno: https://www.ethereum.org/beginners/. [Dostopano: 15-maj-2019].

[4] Ethereum, „Ethereum White Paper“, 2019. .

[5] D. Ter Heide, „A Closer Look At Ethereum Signatures“, *Hacker Noon*, 2018. [Na spletu]. Dostopno: https://hackernoon.com/a-closer-look-at-ethereum-signatures-5784c14abecc. [Dostopano: 28-maj-2019].

[6] Easy Ethereum, „Components of Ethereum (Part 1)“, 2019. [Na spletu]. Dostopno: https://www.easyeth.com/components-of-ethereum-part-1.html. [Dostopano: 28-maj-2019].

[7] Pluralsight, „JavaScript.com“, 2019. [Na spletu]. Dostopno: https://www.javascript.com/. [Dostopano: 24-maj-2019].

[8] Node.js Foundation, „Node.js“, 2018. [Na spletu]. Dostopno: https://nodejs.org/en/. [Dostopano: 16-maj-2018].

[9] Facebook Inc., „React – A JavaScript library for building user interfaces“, 2019. [Na spletu]. Dostopno: https://reactjs.org/. [Dostopano: 15-mar-2019].

[10] Google, „Angular“, 2018. [Na spletu]. Dostopno: https://angular.io/. [Dostopano: 19-mar-2019].

[11] Evan You, „Vue.js“, 2018. [Na spletu]. Dostopno: https://vuejs.org/. [Dostopano: 15-mar-2019].

[12] D. Abramov in Redux, „Redux · A Predictable State Container for JS Apps“, 2019. [Na spletu]. Dostopno: https://redux.js.org/. [Dostopano: 28-maj-2019].

[13] Ethereum, „web3.js - Ethereum JavaScript API — web3.js 1.0.0 documentation“, 2016. [Na spletu]. Dostopno: https://web3js.readthedocs.io/en/1.0/. [Dostopano: 16-maj-2018].

[14] Consensys, „Truffle Suite | Sweet Tools for Smart Contracts“, 2018. [Na spletu]. Dostopno: https://truffleframework.com/. [Dostopano: 17-okt-2018].

[15] Embark, „Getting Started | Embark“, 2019. [Na spletu]. Dostopno: https://embark.status.im/docs/overview.html. [Dostopano: 17-maj-2019].

[16] Protocol Labs, „IPFS is the Distributed Web“, 2018. [Na spletu]. Dostopno: https://ipfs.io/. [Dostopano: 21-maj-2018].

[17] ConsenSys, „An Introduction to IPFS“, *Medium*, 2016. [Na spletu]. Dostopno: https://medium.com/@ConsenSys/an-introduction-to-ipfs-9bba4860abd0. [Dostopano: 16-maj-2019].

[18] Protocol Labs, „IPFS Documentation“. [Na spletu]. Dostopno: https://docs.ipfs.io/. [Dostopano: 15-maj-2019].

[19] ConsenSys, „Truffle Suite | Documentation | Drizzle | Drizzle Quickstart“, 2018. [Na spletu]. Dostopno: https://truffleframework.com/docs/drizzle/quickstart. [Dostopano: 17-maj-2019].

[20] „Introduction - Vortex“. [Na spletu]. Dostopno: https://vort-x.readthedocs.io/en/develop/. [Dostopano: 10-mar-2019].

[21] N. Zinsmeister, „web3-react“, 2019. [Na spletu]. Dostopno: https://noahzinsmeister.gitbook.io/web3-react/v/latest/. [Dostopano: 17-maj-2019].

OTS 2019
SODOBNE INFORMACIJSKE TEHNOLOGIJE IN STORITVE
M. Heričko in K. Kous (ur.)

Univerzitetna založba
Univerze v Mariboru

RAZVOJ APLIKACIJE ZA OŽIVLJANJE KULTURNE DEDIŠČINE: ZASNOVA INTERAKTIVNEGA PROTOTIPA

KLEMEN BABUDER, VID STROPNIK, EMILIJA STOJMENOVA DUH

Povzetek: *Ljudje pri obiskovanju njim atraktivnih lokacij uporabljajo mobilne in spletne aplikacije, ki jim olajšajo obisk in pomagajo pri beleženju in deljenju novih doživetij. Pri razvoju aplikacije za oživljanje kulturne dediščine se je torej potrebno vprašati, na kakšen način omenjenim ljudem približati in izboljšati izkušnjo odkrivanja ter doživljanja kulturne dediščine. Uporabnikom in uporabnicam je najlažja uporaba tiste tehnologije, ki je intuitivna in enostavna za uporabo, ter so jo navajeni oziroma jo že dobro poznajo. Da bi aplikacijo naredili uporabnikom čimbolj poznano, je potrebno poznati uporabnikove karakteristike, potrebe, navade, itn. Z drugimi besedami je uporabnike potrebno postaviti v ospredje. To nam omogoča uporabniško naravnano načrtovanje (angl. User-centered design – UCD), kjer snovalci aplikacij aktivno sodelujejo z uporabniki tekom celotnega procesa načrtovanja in razvoja aplikacije. Pri vsaki fazi razvoja, snovalci in razvijalci preverjajo uporabnikov odziv ter prosijo uporabnike naj jim zaupajo kaj se jim zdi znano, prijazno ter intuitivno za uporabo, in kaj ne. Ker je proces iterativen, nosi veliko težo prva oblika ali prvi prototip naše aplikacije, ko so stroški razvoja relativno nizki. V članku opisujemo uporabljene metode in postopek ustvarjanja interaktivnega prototipa za mobilno aplikacijo za oživljanje kulturne dediščine. Pri načrtovanju prototipa smo deloma izhajali iz analize stanja priljubljenih aplikacij, ki rešujejo našim podobne probleme, deloma pa iz rezultatov izvedene uporabniške študije. Opišemo nadgradnjo skic v prvi srednje funkcionalni ter interaktivni prototip aplikacije. V zaključku članka dodajamo še kratek razmislek o prednostih in slabostih uporabljenih orodij in vplivu le teh na prototip*

Ključne besede: *kulturna dediščina, uporabniško naravnano načrtovanje, uporabniško usmerjen razvoj, interaktivni prototip, persone*

NASLOVI AVTORJEV: Klemen Babuder, Fakulteta za elektrotehniko Univerza v Ljubljani, Ljubljana, Slovenija, e-pošta: klemen.babuder@ltfe.org. Vid Stropnik, Fakulteta za elektrotehniko Univerza v Ljubljani, Ljubljana, Slovenija, e-pošta: vid.stropnik@ltfe.org. Emilija Stojmenova Duh, Fakulteta za elektrotehniko Univerza v Ljubljani, Ljubljana, Slovenija, e-pošta: emilija.stojmenova@fe.uni-lj.si.

https://doi.org/10.18690/978-961-286-282-4.21 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 UVOD

UNESCO kulturno dediščino opisuje kot zapuščino fizičnih predmetov in nepredmetnih atributov skupine ali družbe, ki so podedovani od preteklih generacij, ohranjeni v sedanjosti in namenjeni predstavljanju in izročanju novim rodovom za voljo njihovega učenja in koristi. [1] Način omenjenega izročila formalno ni opredeljen, vendar avtorji članka razumemo, da morajo prejemniki za kar se da veliko mero uporabnosti in koristi od sprejetega, dojeti vrednost kulturne dediščine in njeno vsebino posledično ponotranjiti.
Življenja današnje generacije so zelo odvisna od rabe mobilnih naprav. V letu 2018 smo 52,2% vsega spletnega prometa generirali prek mobilnih telefonov. [2] Druga raziskava kaže, da 79% vseh Evropejcev svoje mobilne telefone uporablja za brskanje po internetu. [3] V letu 2017 je uprava največjega svetovnega portala za pretočne video vsebine – YouTuba, naznanila, da na dnevni ravni uporabniki predvajajo več kot milijardo ur spletnega videa [4]; 70% tega pa je storjenega z rabo mobilnih platform. [5]
Današnjo generacijo se več kot očitno da doseči z rabo mobilnih platform. Še več: poročanja, da v Združenih državah Amerike 75 % vsega denarja, porabljenega za digitalno trženje, podjetja namenijo za oglase v mobilnih aplikacijah [6] zakoličijo dejstvo, da so te mesto, kjer je smiselno iskati razumevanje vsebin, kot je kulturna dediščina in jih tam tudi distribuirati.
Aplikacijo s tovrstnim ciljem avtorji članka tudi ustvarjamo. Pod okriljem projekta kulTura, ki financiranje prejema iz programa Interreg Slovenija - Hrvaška in Javne agencije za raziskovalno dejavnost RS, bomo z izdelavo intuitivne rešitve uporabnikom mobilnih telefonov predstavili kulturno dediščino čezmejnega para mest Črnomelj v Sloveniji in Jastrebarsko na Hrvaškem. V želji, da uporabniki vsebine, ki jim jih predajamo v aplikaciji ne samo razumejo, temveč jih ponotranjijo in se k njim ponovno vračajo, smo med določanjem funkcionalnosti naše rešitve uporabnika postavili v ospredje. To smo storili z uporabo Uporabniško naravnanega načrtovanja; (angl. *User-centered design* – UCD). V prihajajočih poglavjih predstavljamo razloge za odločitev znanj in dileme, s katerimi smo se pri njegovi implementaciji spoprijeli. Seveda v nadaljevanju predstavljamo tudi rešitve slednjih, te razlage pa v koncu članka pripeljejo do nadrobnejše predstavitve interaktivnega prototipa - enega izmed ključnih prijemov UCD.

2 ZGODNJI PROCES ODLOČANJA O FUNKCIONALNOSTIH

2.1 Uporabniško naravnano načrtovanje

Uporabniško naravnano načrtovanje (angl. *User-Centered Design*) je iterativen proces načrtovanja in razvoja produktov, kjer produktna ekipa analizira navade uporabnikov in na osnovi rezultatov analize snuje uporabniško izkušnjo. Ustreznost vizualizacije testira na uporabnikih in jo odzivom primerno prilagodi. Proces nadaljuje z iterativnim ponavljanjem testiranja in prilagajanja.

Tekom načrtovanja lahko ekipa v posameznih korakih uporablja različne metode za pripravo konceptov in vrednotenje uporabniške izkušnje. V nadaljevanju so na kratko opisane metode uporabljene pri razvoju naše rešitve.

2.2 Persone

Persona je osebnostno gnan element, ki ekipo opominja, komu je aplikacija namenjena. Je namišljen lik, ki personificira ciljnega uporabnika [7]. Pri ustvarjanju posamezne persone je pomembna vključitev dovoljšnega števila podrobnosti, ki razvijalcu omogoča pristno poistovetenje z likom. Med pripravo produktov z raznoliko ciljno skupino ali večjim številom ciljnih skupin, je smiselna uporaba primernega števila person [8].

Tekom snovanja smernic za razvoj aplikacije smo želeli oblikovati zalogo person, ki bi nas med razvojem spremljala in nas opominjala na naše predvidene končne uporabnike, njihove potrebe in predstave o izdelku. Persone smo namreč prepoznali kot uporabno orodje ne le pri internih diskusijah o nadaljnjem razvoju, temveč tudi pri rabi drugih uveljavljenih prijemov UCD, kot so "kognitivni sprehodi" (angl. *Cognitive Walk-throughs),* [9] priprava zgodborisov *(angl. Storyboard)* [10] in "empatijskih zemljevidov" (angl. *Empathy map)* [11] ter igranju vlog. V praksi smo po tem, ko smo jih uspešno pripravili, potrdili tudi smiselnost njihove rabe kot povzetek dosedanjega dela pri prepoznavanju končnih uporabnikov, saj sta novo pridružena člana projektne ekipe z njihovo rabo hitro razumela uporabnike, za katere produkt razvijamo. [12]

Nemudoma po začetku priprave prvih person smo naleteli na mnoga vprašanja. Zagotovo je največja težava, ki smo jo med razvojem našli, bil primanjkljaj informacij za korektno pripravo person na osnovi določenih ciljnih skupin.

Ker rešitev, ki jo pripravljamo, temelji na kulturni dediščini v Črnomlju in Jastrebarskem, je zaključek, da sta vsaj dve izmed ciljnih skupin končnih uporabnikov aplikacij prebivalci omenjenih mest, smiseln. Dvomili smo, da bi se z vrednotami, ki bi omenjeni skupini (oz. Personi, osnovanih na njih) tipološko najbolje opisali, kot zunanji izvajalci znali popolnoma poistovetiti oziroma jih prepoznati. Zato smo se za pomoč odločili obrniti na projektne partnerje, živeče in delujoče v Črnomlju in Jastrebarskem. Od njih študije teorije o personah nismo pričakovali. V želji po tem, da bi persone, ki jih izdelajo, čim natančneje ustrezale našim željam, smo zanje z rabo brezplačnega spletnega orodja Google Forms [13] pripravili vprašalnik [14], na osnovi odgovorov na katerega smo nato persone pripravili mi. Kot končni izdelki procesa izdelave person so se v ekipi uveljavili povzetki rešenih vprašalnikov v obliki posameznih diapozitivov powerpointove predstavitve. [15]

Slika 1. Povzetek persone v obliki powerpointovega diapozitiva

Iz postopka je moč izvzeti več lekcij. Sprva - števila person, ki bodo spremljale razvoj produkta ni smiselno določati vnaprej. Izkazalo se je, da jih v samem začetku ni bilo treba pripraviti veliko, saj smo k njim lahko iterativno dodajali brez večjega problema. Pripraviti je smiselno persone primarnega, sekundarnega in terciarnega tipa, kot smiselno pa smo ocenili tudi pripravo *antipersone*; tipa uporabnika, za katerega naše aplikacije izrecno *ne* razvijamo. Kot zadnjo lekcijo izpostavljamo dobro izkušnjo priprave enostranskih povzetkov person, nameščenih v delovni prostor: Na ta način smo tekom celotnega razvoja rešitve opomnjeni na karakteristike naših različnih ciljnih skupin oziroma likov, ki jih poosebljajo.

2.3 Uporabniške ankete

Uporabniške ankete so uporaben in časovno učinkovit način za doseganje velikega števila uporabnikov. Uspešnost pridobivanja koristnih in zanesljivih informacij je predvsem odvisna od načina postavljanja vprašanj in ustrezne sestave izprašane skupine [16].

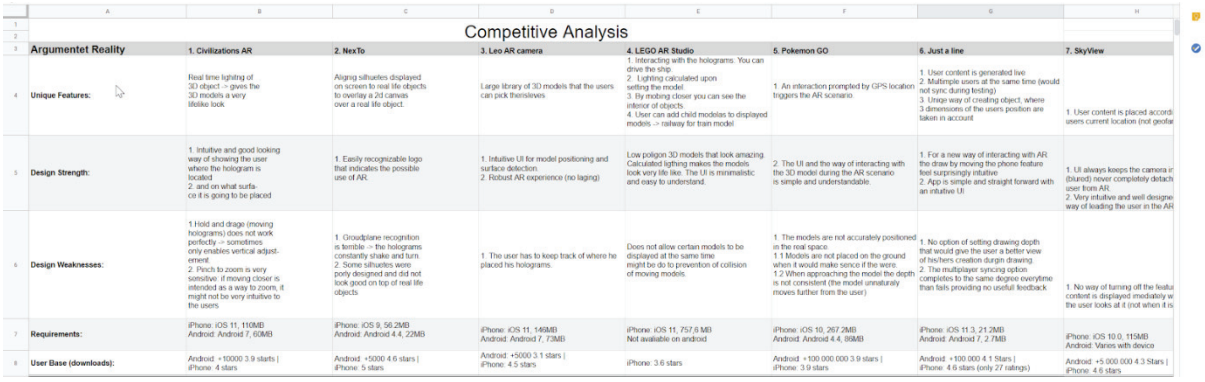
Kot velijo priporočila uporabniško naravnanega načrtovanja, smo ankete snovali z več ponovitvami in pregledi. Sodelovanje med pripravo nam je omogočil modularen uporabniški vmesnik spletnega orodja Google Forms. Ob vrnitvah na vprašalnik smo se pogosto trudili izločiti podatke, ki nam eksplicitno ne bi koristili, prav tako pa ne bi informacijskega prispevka nosili v korelaciji z odgovori na katerokoli drugo vprašanje. Med drugim smo revidirali tudi vprašanja, na katere odgovore smo zlahka predvideli. Po tem ključu smo izmed kategorij v vprašanju 'Kako pogosto na svojem pametnem telefonu uporabljate te vrste aplikacij?' izločili družbena omrežja oziroma jih od ostalih aplikacij ločili. Prikaz podatkov v končni analizi je bil zato čistejši in ni zahteval dodatnega transformiranja podatkov. Prav tako smo pripravili več končnih razvrstitev izbranih vprašanj, končno podobo pa smo med drugim izdelali z rabo osnovnih prijemov sortiranja s kartami (angl. *Card sort*) [17]. Dovršena anketa se je izvajala mesec dni in je vsebovala 49 vprašanj razdeljenih v 6 tematskih

sklopov. Nanjo je odgovarjalo 272 posameznic in posameznikov z zadovoljivo raznolikim demografskim profilom. Vzorčenje je bilo ne verjetnostno in je bilo izvedeno v obliki določitve namenskega vzorca, sicer pa je bila anketa posredovana specifičnim ciljnim skupinam - potencialnim uporabnikom aplikacije.

2.4 Analiza konkurence

Analiza konkurence zajema pregled produktov, ki rešujejo podoben problem, kot rešitev, ki jo razvijalec razvija. Z razdelitvijo problema na manjše a še smiselne elemente, lahko sorazmerno povečamo število produktov, ki ustrezajo analizi. Koristnost analize lahko povečamo z obtežitvijo ugotovitev glede na poznavanje in priljubljenost produktov med uporabniki.

Tudi analizo smo z ekipo opravili. Lotili smo se je z rabo Googlovih oblačnih urejevalnikov dokumentov. Sprva smo v aplikaciji prepoznali področja, ki se jih skuša vsaj v omembe vrednem obsegu dotakniti oz. jih uporabiti. Nato smo na vsakem izmed določenih področij poiskali uspešne aplikacije, vsaki izmed katerih smo subjektivno določili vsaj 1 unikatno lastnost, moči in pomanjkljivosti v dizajnu: prav tako pa smo za potrebe nadaljnjega dela zabeležili njihove programske zahteve in število prenosov.



Competitive Analysis

	Argumentet Reality	1. Civilizations AR	2. NexTo	3. Leo AR camera	4. LEGO AR Studio	5. Pokemon GO	6. Just a line	7. SkyView
4	Unique Features:	Real time lighting of 3D object -> gives the 3D models a very lifelike look	Aligning silhuettes displayed on screen to real life objects to overlay a 2d canvas over a real life object.	Large library of 3D models that the users can pick themselves	1. Interacting with the holograms. You can drive the ship 2. Lighting calculated upon setting the model. 3. By mobing closer you can see the interior of objects. 4. User can add child modelas to displayed models -> railway for train model	1. An interaction prompted by GPS location triggers the AR scenario	1. User content is generated live 2. Multimple users at the same time (would not sync during testing) 3. Unige way of creating object, where 3 dimensions of the users position are taken in account	1. User content is placed accordi users current location (not geofe
5	Design Strength:	1. Intuitive and good looking way of showing the user where the hologram is located 2. and on what surfa- ce it is going to be placed	1. Easily recognizable logo that indicates the possible use of AR.	1. Intuitive UI for model positioning and surface detection. 2. Robust AR experience (no laging)	Low poligon 3D models that look amazing. Calculated lighing makes the models look very life like. The UI is minimalistic and easy to understand.	2. The UI and the way of interacting with the 3D model during the AR scenario is simple and understandable.	1. For a new way of interacting with AR the draw by moving the phone feature feel surprisingly intuitive 2. App is simple and straight forward with an intuitive UI	1. UI always keeps the camera in (blured) never completely detach user from AR. 2. Very intuitive and well designe way of leading the user in the AR
6	Design Weaknesses:	1 Hold and drage (moving holograms) does not work perfectly -> sometimes only enables vertical adjust- ement 2. Pinch to zoom is very sensitive. if moving closer is intended as a way to zoom, it might not be very intuitive to the users	1. Groudplane recognition is terrible -> the holograms constantly shake and turn. 2. Some silhuettes were poorly designed and did not look good on top of real life objects	1. The user has to keep track of where he placed his holograms.	Does not allow certain models to be displayed at the same time might be do to prevention of collision of moving models.	1. The models are not accurately positioned in the real space. 1.1 Models are not placed on the ground when it would make sence if the were. 1.2 When approaching the model the depth is not consistent (the model unnaturaly moves further from the user)	1. No option of setting drawing depth that would give the user a better view of his/hers creation durgin drawing. 2. The multiplayer syncing option completes to the same degree everytime than fails providing no usefull feedback	1. No way of turning off the featu content is displayed imediately w the user looks at it (not when it is
7	Requirements:	iPhone: iOS 11, 110MB Android: Android 7, 60MB	iPhone: iOS 9, 56.2MB Android: Android 4.4, 22MB	iPhone: iOS 11, 149MB Android: Android 7, 73MB	iPhone: iOS 11, 757,8 MB Not available on android	iPhone: iOS 10, 267.2MB Android: Android 4.4, 86MB	iPhone: iOS 11.3, 21.2MB Android: Android 7, 2.7MB	iPhone: iOS 10.0, 115MB Android: Varies with device
8	User Base (downloads):	Android: +10000 3.9 starts | iPhone: 4 stars	Android: +5000 4.6 stars | iPhone: 5 stars	Android: +5000 3.1 stars | iPhone: 4.5 stars	iPhone: 3.6 stars	Android: +100 000 000 3.9 stars | iPhone: 3.9 stars	Android: +100 000 4.1 Stars | iPhone: 4.6 stars (only 27 ratings)	Android: +5 000 000 4.3 Stars | iPhone: 4.6 stars

Slika 2. Naš pristop k analizi konkurence

2.5 Prototipiranje

Prototipiranje je proces izdelave preprostih ali kompleksnih prototipov. Služi hitrejšemu in učinkovitejšemu preizkušanju uporabniške izkušnje produkta z uporabniki. Ponujajo možnost evalvacije izgleda in oblike produkta pred znatnim časovnim vložkom v njegovo izdelavo [18]. O našem pristopu h prototipiranju več pišemo v naslednjih poglavjih.

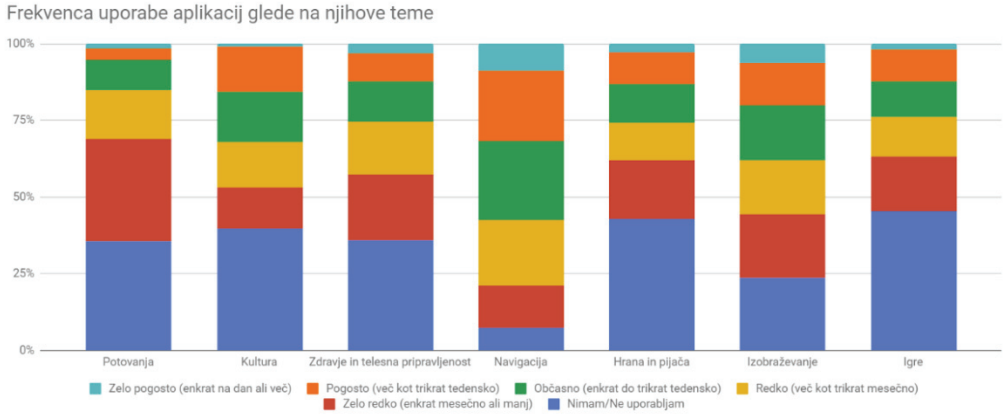
3 NAŠA REŠITEV

Določeni prijemi naše rešitve so bili opredeljeni že v razpisni dokumentaciji projekta kulTura. Mednje spada tudi raba najnovejših mobilnih tehnologij, med katere med drugimi uvrščamo obogateno resničnost; interaktivno izkušnjo resničnega sveta, v kateri dejanske objekte aplikacija “obogati” z računalniško-generiranimi zaznavnimi informacijami preko različnih oblik modalnosti, med drugim vidne, slušne, haptične idr. [19]

Spet druge prijeme smo določili na osnovi raziskave načinov za čim efektivnejšo predajo kulturno-dediščinskih vsebin aplikacije njenim uporabnikom. Za uresničevanje tega cilja smo kot ustrezen atribut prepoznali *igrifikacijo (angl. Gamification)*, rabo gradnikov iger s ciljem motiviranja specifičnih vzorcev obnašanja v igrificirani situaciji. [20] Praksa se je za uspešno izkazala pri težnji po doseganju različnih ciljev - za naš primer pa je najbolj smiselno omeniti uporabniško angažiranost, [21-22] učenje [23] in nadaljevanje ciljno-usmerjenega obnašanja. [24] To spoznanje se v končni rešitvi odraža tako, da se dediščino trudimo uporabnikom predstaviti z rabo igrificiranega raziskovanja vsakega izmed mest - podane informacije pa se trudimo popestriti z *mini igrami* na vsaki izmed točk interesa.

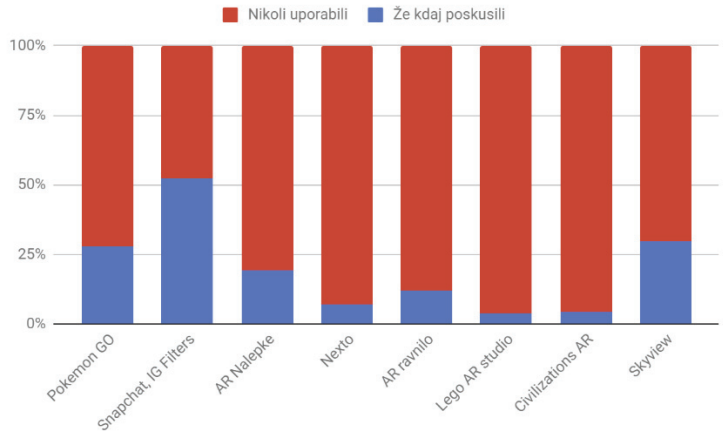
Ostale funkcionalnosti aplikacije smo določili s pomočjo rezultatov uporabniške ankete, opisane v prejšnjem poglavju.

Uporabniška anketa je pokazala, da imajo aplikacije nišnih kategorij med seboj podoben vzorec rabe, večina uporabnikov pa jih ne uporablja niti enkrat tedensko. Kot izjema temu pravilu so se izkazale navigacijske aplikacije, ki so med navedenimi kategorijami prejele najpozitivnejši vzorec glasov. Na osnovi teh odgovorov smo sprejeli odločitev, da bo velik del aplikacije tvoril vmesnik za navigacijo po mestih, ki jih uporabnik raziskuje. Prav tako smo navdih za pripravo uporabniškega vmesnika zato črpali pri uveljavljenih navigacijskih aplikacijah.



Slika 3. Izsek iz analize uporabniške ankete: Prikaz frekvence uporabe aplikacij glede na nišno kategorijo, ki ji pripadajo.

Z zavedanjem novosti tehnologije obogatene resničnosti in poznavanjem dejstva, da na uporabniško izkušnjo močno vplivajo uporabnikovo predhodno stanje in njegove izkušnje z rabo dotične tehnologije [25], smo pri izbiri načina predstavitve funkcionalnosti, vezanih na obogateno resničnost, morali biti previdni. Želeli smo minimizirati možnost, da bi se uporabniki v vmesniku izgubili oz. ga uporabljali s težavo. Uporabniška anketa je pokazala, da se je največje število uporabnikov s tehnologijo obogatene resničnosti srečalo pri rabi aplikacij Instagram [26] in Snapchat. [27] Vmesnik za bogatenje resničnosti smo posledično zasnovali na osnovi teh.

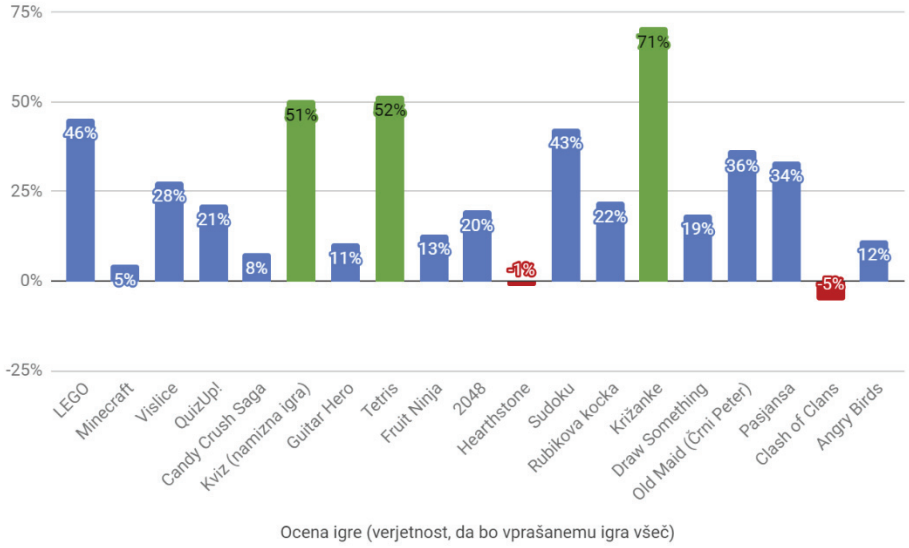


Slika 4. Izsek iz analize uporabniške ankete: Prikaz poznanosti AR funkcionalnosti.

Rdeča nit aplikacije je z zgoraj navedenimi odločitvami začenjala dobivati obliko: Uporabniki bodo aplikacijo uporabljali med raziskovanjem kulturne dediščine mest s sprehodom skoznje. Aplikacija jim bo z navigacijskim vmesnikom pomagala najti točke interesa, na teh pa jim bo nudila geografsko zajezene (angl. *Geo-fenced*) vsebine, ki bodo bodisi igrificirane, bodisi popestrene z rabo obogatene resničnosti. V slednjem primeru jim bo omogočila tudi zajem fotografij. Raziskovanje bo realizirano z uporabo *zgodb*; uro do dve dolgim sprehodom do določenih točk interesa, na katerih bo uporabnik opravljal izzive. Uporabniku bo v aplikaciji omogočeno tudi branje dodatnih informacij in upravljanje z elementi aplikacije skozi naprednejše menijske elemente, kot sta *inventar* in *knjižnica*.

Zadnja odločitev, ki smo jo sprejeli na osnovi podatkov uporabniške študije, je bila vsebina iger na točkah interesa. Izkazalo se je, da med najbolje ocenjene igre spadajo tiste tradicionalne in dobro prepoznane, saj med

najbolje ocenjene štejemo tetris, križanke, LEGO in Sudoku. Visoki oceni križank in Kviza pričata o priljubljenosti kvizov splošne razgledanosti (angl. Trivia). Dejstvo, da sta ti igri ocenjeni višje kot npr. QuizUp. [28] (in sorodne pripisane aplikacije v anketnem vprašalniku - Trivia Crack [29] in SongPop [30]), nam je dalo vedeti, da uporabniških vmesnikov teh mobilnih ekvivalent ne rabimo prekomerno upoštevati pri lastnem prototipiranju; vsekakor pa nam je študija dala usmeritev za nadaljnji razvoj.

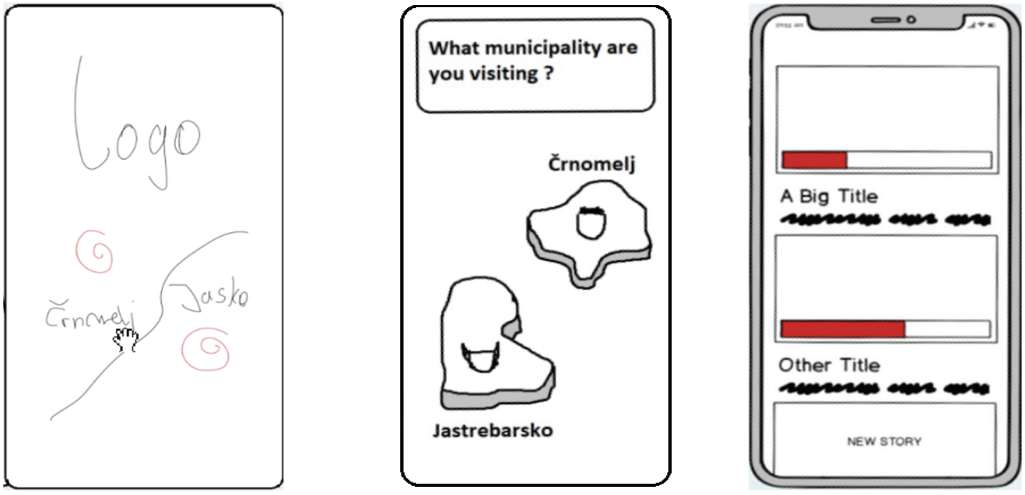


Slika 5. Izsek iz analize uporabniške ankete: prikaz utežene verjetnosti, da bo uporabnik dano igro hkrati poznal in maral.

4 ZASNOVA INTERAKTIVNEGA PROTOTIPA

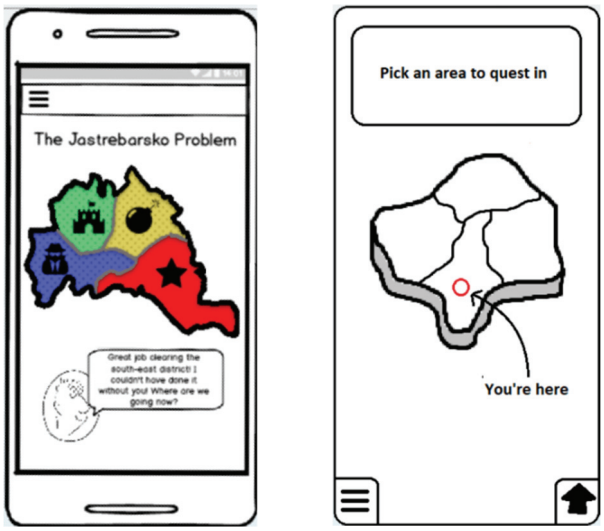
4.1 Priprava konceptov zaslonov

Proces prototipiranja smo začeli z izdelavo preprostih modelov (angl. Mock-ups) uporabniškega vmesnika. Izdelave so se lotili trije člani ekipe. Po eksplicitnem dogovoru si člani ekipe pred in med izdelavo modelov med seboj niso delili svojih ugotovitev in napredka. Namen dogovora je bil preprečiti podobnosti med modeli, ki bi lahko nastale kot posledica medsebojne komunikacije. Posledično, smo kakršnekoli podobnosti med končanimi modeli interpretirali kot posledico podobnih predhodnih mentalnih modelov razvijalcev. Podobnosti smo ocenili za koristne, pri predpostavki, da so mentalni modeli razvijalcev podobni mentalnemu modelu povprečnih uporabnikov pametnih telefonov.



Slika 6. Primer modelov različnih članov ekipe.

Eden izmed članov ekipe je za izdelavo modela uporabil namensko programsko opremo Balsamiq Mockups 3 [31], v katero je uvozil skice, pripravljene z rabo grafične tablice in programa Adobe Photoshop CC 2018 [32], druga dva člana pa se pri izdelavi nista posluževala posebne namenske programske opreme. Svoje prototipe sta ustvarila s pomočjo kombinacije orodja Slikar (*Microsoft Paint*) [33] in skic, pripravljenih s svinčnikom in papirjem.



Slika 7. Primer podobnih zamisli dveh članov ekipe - posledice predhodnega pogovora o podobi modelov.

Izdelane modele smo strnili v predstavitev [34] (za pripravo katere smo uporabili spletno orodje Prezi [35]), kjer smo zaslone, ki so reševali podoben problem združili v skupine.

Celotna ekipa je s pregledom nastalih skupin, prepoznala in primerjala različnie pristope reševanja mikro problemov, ki so si jih razvijalci zamislili skozi ustvarjanje modelov. Prepoznali smo pristope za katere smo ocenili, da jih je smiselno realizirati v prvem interaktivnem prototipu. Pri tem se nismo omejili na eno rešitev na mikro problem. Ocenili smo namreč, da je odločitev glede ustrezne rešitve za izbrani problem smiselno prepustiti fokusnim skupinam.

4.2 Pregled dobrih praks uporabniških vmesnikov

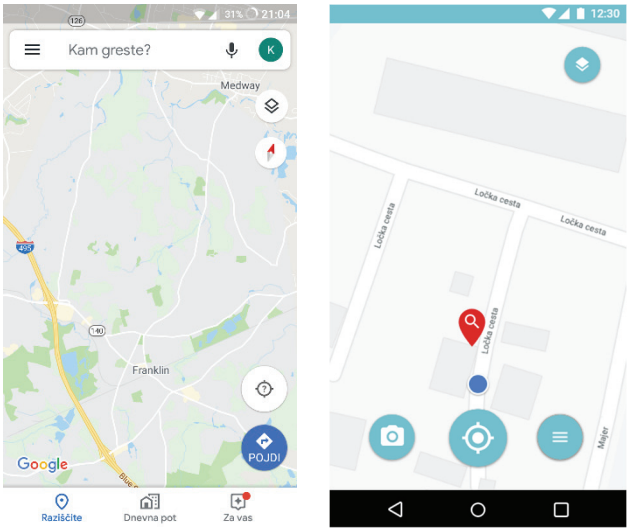
Pred začetkom izdelave prototipa smo pregledali dobre prakse uporabniških vmesnikov izbranih aplikacij. Izbrane aplikacije in utemeljitve za njihovo izbirno navajamo na koncu podpoglavja. Izbrali smo tiste, ki so vsebovale pristope podobne ali enaki tistim, ki smo jih označili kot primerne za realizacijo in pri tem upoštevali tudi število prenosov posamezne aplikacije.

Namen pregleda je bil izpostaviti ustrezno funkcionalne in lepo oblikovane uporabniške vmesnike. Dodatna teža je bila dodana vmesnikom tistih aplikacij, ki so bile v uporabniški anketi uvrščene med najbolj uporabljene. Cilj tovrstne obtežitve je bil ustvariti prototip, ki se bo čim bolj prilagajal že obstoječim mentalnim modelom uporabnikov.

V nadaljevanju so navedene izbrane aplikacije in obrazložitev njihove izbire.

4.2.1 *Google maps*

Na podlagi ugotovitev uporabniške ankete, kjer je 57.48% izprašanih (n = 247) odgovorilo, da aplikacijo uporabljajo več kot enkrat tedensko, ocenjujemo, da so uporabniki dobro seznanjeni z načinom uporabe aplikacije. Predpostavljamo, da jim je način orientacije, pomikanja in spreminjanja izgleda mape uporabnikom jasen in menimo, da je podobna realizacija funkcionalnosti v naši aplikaciji smiselna.



Slika 8. Primer 2D navigacijskih ekranov aplikacij Google Maps [36] (levo) in kulTura (desno)

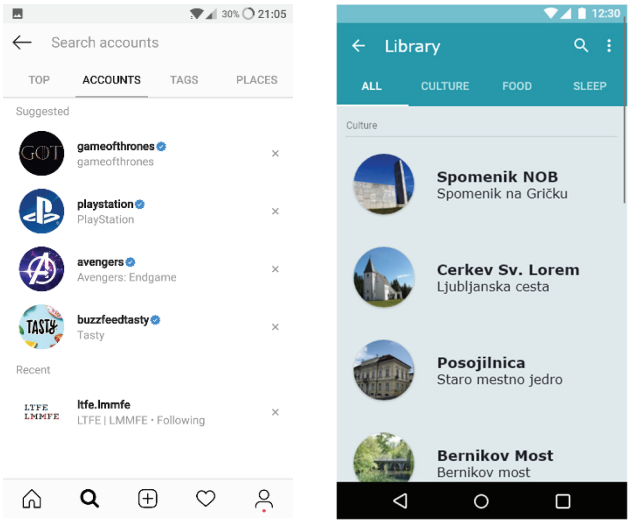
4.2.2 *PokemonGO*

Aplikacijo je do maja 2019 na terminalih z Android operacijskim sistemom preneslo več kot 11.700.000 uporabnikov. Med anketiranci je 28% izprašanih odgovorilo, da je aplikacijo že uporabljalo. Zaradi relativno male prepoznavnosti aplikacije med izprašanci, smo se osredotočili predvsem na elemente, ki jih ni med aplikacijami, ki jih uporabniki pogosteje uporabljajo. Menimo, da je podobna realizacija tri dimenzionalne mape in podobna tehnika postavitve relevantnih objektov na mapo, smiselna.

Slika 9. Primerjava 2.5D navigacijskih ekranov aplikacij Pokemon GO [37] (levo) in kulTura (desno)

4.2.3 *Instagram*

70.70% izprašanih je dejalo, da aplikacijo uporabljamo za spremljanje družbenih medijev, 48.37% izprašanih pa je dejalo da aplikacijo uporablja zelo pogosto. Ocenjujemo, da so uporabniki seznanjeni z načinom prikazovanja daljših vsebinskih seznamov in prepoznavamo smiselnost podobne realizacije funkcionalnosti v naši aplikaciji.



Slika 10. Primerjava prikazov seznamov aplikacij Instagram (levo) in kulTura (desno)

1.1. Oblikovanje vmesnika na osnovi konceptov in dobrih praks

Za izdelavo interaktivnega prototipa smo izbrali namensko programsko opremo AdobeXD [38].

Barve uporabljene v prototipu so bile izbrane iz celostne grafične podobe projekta, v okviru katerega nastaja aplikacija. Za ilustracijo gumbov in drugih interaktivnih elementov smo izbrali znake oblikovni jezik Material Design. [39]

Uvodni zaslon (angl. splash screen) prototipa predstavlja logo projekta kulTura znotraj katerega rešitev nastaja. Sledita zaslona za izbiro lokacije in števila uporabnikov. Oba zaslona uporabnika pozivata k akciji prek namenskega območja na vrhu zaslona. Uporabniku sta na obeh zaslonih na voljo dva gumba, ki ob pritisku peljeta na naslednji pogled. Večino preostalih gumbov v aplikaciji si deli njun izgled, kjer je oblika gumbov okrogla, znak na njih je bele barve, preostanek pa je obarvan v primarni modro-zeleni barvi aplikacije.

Sledi zaslon, ki ponuja izbiro med zgodbami, ki bodo uporabnika popeljale po predhodno izbranem kraju. Osrednji del zaslona vsebuje naslov, sliko in opis zgodbe. Na zaslonu je naenkrat prikazana le ena zgodba, saj smo mnenja, da na tak način uporabniku podamo dovolj opisnih in vizualnih informacij in ga pri izbiri ne obremenjujemo.

Izbiri zgodbe sledi prehod na osrednji zaslon aplikacije. Uporabnika z rabo besedila v pojavnem oknu pozdravi fiktivni lik zgodbe. Vsi nagovori fiktivnih likov in sistemska opozorila so prikazana z rabo modalnih oken, nagovore prej omenjenih likov pa pospremi tudi slikovna podoba.

Navigacijski zaslon lahko uporabnik prilagaja s pomočjo okroglega gumba, ki se nahaja v desnem zgornjem kotu ekrana. Inspiracijo za postavitev gumba in za način interakcije z menijem za prilagajanje navigacijski zaslona smo dobili iz uporabniškega vmesnika aplikacije Google Maps.

Preostanek grafičnega vmesnika na navigacijskem zaslonu sestavljajo trije okrogli gumbi na dnu ekrana. Srednji iz med treh gumbov je nekoliko večji od ostalih dveh. Z razliko v velikosti smo želeli poudariti njegovo pomembnost, saj je edini izmed treh gumbov, čigar akcija se navezuje na trenutni zaslon. Inspiracij za tovrstno razporeditev in število gumbov smo našli v aplikacijah Snapchat in Pokemon GO. Podobno prakso Levi gumb uporabniku omogoča prehod med navigacijskim zaslonom in pogledom skozi kamero, desni pa vodi v uporabniški meni.

Pogled skozi kamero ohranja gumbe na dnu ekrana, kjer desni gumb ohranja svojo funkcionalnost iz navigacijskega zaslona, desni pa nanj vodi. Osrednji gumb podobno, kot standardne aplikacije za fotografiranje, Instagram in Snapchat, uporabniku omogočata zajem fotografije. Slednji dve aplikaciji isto postavitev uporabljata tudi pri prikazu AR vsebin, kar ocenjujemo kot dobro popotnico za bodoče delo na njih.

Uporabniški meni sestavljajo okrogli gumbi za galerijo, knjižnico, inventar in nastavitve. Preostanek zaslona sestavlja zamegljen zaslon iz katerega je uporabnik odprl meni.

Za izgled lista elementov v zavihku knjižnica smo inspiracijo jemali v aplikaciji Instagram, kjer na levo stran posameznega elementa postavimo okroglo izrezano sliko in s tem podatkom o točki interesa dodamo še vizualno ponazoritev. Podobno razporeditev elementov si z knjižnico deli tudi inventar.

Galerijo sestavlja niz kvadratno izrezanih slik, prikazan v dveh stolpcih, kjer klik na posamezno sliko, le to razširi čez cel ekran in ozadje zatemni. Tako razporeditev smo si izbrali v želji, da bi ob preletu zajetih slik, vsaka slika ujela uporabnikovo oko in mu s tem ustvarila učinek potopisa.

Inspiracijo za zaslone nastavitev smo v želji, da bi imeli uporabniki z prilagajanjem aplikacije kar se da malo težav, iskali v izgledu standardne android aplikacije Nastavitve (angl. Settings).

5 UPORABLJENO ORODJE: PREDNOSTI IN SLABOSTI ADOBE XD

V nadaljevanju navajamo naše mnenje in ugotovitve glede rabe namenske programske opreme Adobe XD za izdelavo interaktivnih prototipov.

Kot jasno prednost rešitve smo prepoznali dejstvo, da je na voljo brezplačno. Ob hitri primerjavi s konkurenčnimi orodji *InVision* [40] in *JustInMind* [41] v tem cenovnem razredu nismo našli tekmeca, ki bi nam nudil primerljivo široko paleto funkcionalnosti.

Ocenjujemo tudi, da so brezplačne predloge standardnih zaslonov in znakov za različne sisteme [42] - med drugimi predloge za snovanje aplikacij za operacijski sistem iOS in v oblikovnem jeziku Material Design, znatno pospešile čas izdelave prototipa. K efektivnosti našega dela so pripomogle tudi druge funkcionalnosti programa. Dobro realiziran sistem za uporabo in spreminjanje barvnih shem nam je omogočil shranjevanje in poimenovanje uporabljenih barv. Kasneje je bilo na enostaven način omogočeno spreminjanje odtenka vseh elementov, pobarvanih z njo. Prav tako smo bili zadovoljni z delovno površino, ki razvijalcem ponuja zadovoljivo mero svobode pri razporejanju in navigaciji med zasloni.

Program ne omogoča prilagoditve zaslona glede na njegovega predhodnika. Pomanjkanje te funkcionalnosti se občuti pri urejanju prehoda na zaslon, ki je na voljo skozi več drugih zaslonov v aplikaciji (npr. nastavitve). Zavoljo ohranjanja kontinuitete, je potrebno omenjen zaslon in vse globlje ležeče zaslone, do katerih lahko preko njega dostopamo, podvojiti tolikokrat, kolikor je možnih prehodov vanj. Na vsako novo instanco je nato potrebno vpeti eno izmed možnih poti prihoda na zaslon. Čas izdelave prototipa se posledično znatno podaljša z vsako podvojeno instanco zaslona. Pri tem se občutno poveča tudi velikost izvožene datoteke. Pojavi se tudi težava pri izvozu prototipa v .pdf formatu, saj se podvojeni zaslon v dokumentu ponavlja, funkcija za izvoz pa ne omogoča preproste izločitve dvojnikov.

Program ponuja omejeno število animacij. Med ustvarjanjem prototipa smo, kot pomanjkljivost prepoznali predvsem pomanjkanje horizontalnega premikanja elementov. Pomanjkanje tovrstne animacije smo nadomestili s podvajanjem zaslonov ali pa smo število horizontalnih elementov primerno zmanjšali. Prvi kompromis predstavlja težave pri izvozu prototipa v .pdf formatu, saj se podvojeni zaslon v dokumentu ponavlja, funkcija za izvoz pa ne omogoča preproste izločitve dvojnikov. Potrebo po uporabi predvsem drugega od omenjenih kompromisov pa prepoznavamo kot eno večjih omejitev aplikacije pri zagotavljanju pristne izkušnje uporabe interaktivnega prototipa.

V namene deljenja in bodočega testiranja prototipov z uporabniki v veliko korist prepoznavamo raznolike možnosti deljenja in izvažanja prototipov. Med njimi bi izpostavili možnost snemanja interakcije s prototipom, saj bi za pridobivanje tovrstnega videa sicer potrebovali namensko programsko opremo. Nazadnje bi izpostavili še mobilno aplikacijo, ki skupaj s povezljivostjo programa v oblačni sistem Adobe Creative Cloud [43] razvijalcem omogoča prikaz in simulacijo vseh prototipov shranjenih v oblaku.

6 SKLEPI IN NADALJNJE DELO

Z rabo metod UCD smo se v zgodnjih fazah razvoja aplikacije kulTura seznanili z njihovo koristnostjo in prepoznali, do katere mere njihova raba ustreza naši skupini sodelavcev.

Glavna lekcija, ki smo jo dognali, pa je dejstvo, da tudi implementacija rabe UCD lahko poteka po njegovem glavnem načelu - iterativnosti. Ob začetku projekta smo k implementaciji person, zgodborisov in večih oblik prototipiranja pristopali z zadržki. Večina slednjih je bila povezana z uporabnostjo elementov uporabniško naravnanega načrtovanja in smotrnostjo rabe časa za njihovo pripravo. Naše izkušnje so pokazale, da proces uporabe omenjenih elementov ne zahteva prekomerne priprave in sistemizacije dela v samem začetku.

Uporabniško naravnano načrtovanje ni pogojeno z rabo prav vseh orodij v njegovem repertoarju. Nasprotno - med našim delom so opisane tehnike najbolje delovale takrat, ko smo naleteli na problem, na neenotno dojemanje obravnavane rešitve ali na prelomno točko v njenem snovanju. Menimo, da lahko skupine v nam podobnem položaju po opisanih in pohvaljenih orodjih in konceptih poprimejo na katerikoli točki priprave svojih rešitev, ne da bi se obremenjevali z njihovo širšo umestitvijo v okvir metodologije UCD.

Aplikacija kulTura prehaja v prvo fazo programske izdelave. Po prejetih mnenjih in kritikah projektnih partnerjev, ki smo jih interno obravnavali tudi kot prvo zunanjo fokusno skupino, že načrtujemo izdelavo posodobljenega prototipa, v katerem bomo ob spremembah evalvirali tudi bolj točno podobo aplikacije. Slednjo bodo med drugim diktirali tudi uporabljena programska oprema (npr. Unity [43] za postavljanje hologramov v prostor obogatene resničnosti in MapBox SDK [44] za realizacijo navigacijskega zaslona) in izdelani 3D modeli. Za namene prototipiranja nameravamo nadaljevati z rabo programa AdobeXD, ki se je, kljub kritikam podanim v članku, izkazal za fleksibilno, enostavno in hitro orodje, ki ga bomo z lahkoto uporabljali v mnogih iteracijah ne samo tega projekta, temveč tudi prihajajočih.

7 LITERATURA

[1] http://www.unesco.org/new/en/cairo/culture/tangible-cultural-heritage/, Tangible Cultural Heritage (n.d.), obiskano 10. decembra 2018.

[2] https://www.statista.com/statistics/241462/global-mobile-phone-website-traffic-share/, Delež obiskanih spletnih strani iz mobilnih naprav 2018, obiskano 16. maja 2019.

[3] EUROSTAT, 2016, EUROSTAT, 2016, Almost 8 out of 10 internet users in the EU surfed via a mobile or smart phone in 2016… Dostop: https://ec.europa.eu/eurostat/documents/2995521/7771139/9-20122016-BP-EN.pdf/f023d81a-dce2-4959-93e3-8cc7082b6edd, obiskano 15. maja 2019.

[4] You know what's cool? A billion hours, 2017. Youtube Official Blog. Dostop: https://youtube.googleblog.com/2017/02/you-know-whats-cool-billion-hours.html., obiskano 15. maja 2019.

[5] https://www.comscore.com/Insights/Infographics/Unlocking-Mobile-Measurement-for-YouTube-in-the-US, Odklepanje mobilnega merjenja v ZDA., obiskano 15. Maja 2019.

[6] https://www.forbes.com/sites/johnkoetsier/2018/02/23/mobile-advertising-will-drive-75-of-all-digital-ad-spend-in-2018-heres-whats-changing/#5e516c39758b, V mobilni market se izteka 75% denarja vloženega v internetno oglaševanje, obiskano 22. avgusta 2018.

[7] LOWDERMILK, Travis. User-centered design: a developer's guide to building user-friendly applications. " O'Reilly Media, Inc.", 2013 str. 43.

[8] UNGER, Russ; CHANDLER, Carolyn. A Project Guide to UX Design: For user experience designers in the field or in the making. New Riders, 2012 str. 114.

[9] POLSON, Peter G., LEWIS, Clayton, RIEMAN, John in WHARTON, Cathleen, 1992, Cognitive walkthroughs: a method for theory-based evaluation of user interfaces. International Journal of Man-Machine Studies. 1992. Vol. 36, no. 5, str. 741-773.

[10] LONG, Frank. Real or Imaginary: The effectiveness of using personas in product design. Irish Ergonomics Review, Proceedings of the IES Conference 2009, Dublin. str. 5–7. Dostop: https://s3.amazonaws.com/media.loft.io/attachments/Long, obiskano 11. maja 2019

[11] FERREIRA, Bruna, SILVA, Williamson, OLIVEIRA, Edson in CONTE, Tatyana. Designing Personas with Empathy Map. SEKE. Maj 2015. Vol. 152.

[12] COURAGE, CATHERINE and BAXTER, KATHY, 2005, Understanding your users. 2. San Francisco, CA : Morgan Kaufmann Publishers. str 41

[13] Google.com. (2012). Google Forms: Free Online Surveys for Personal Use. Dostop: https://www.google.com/forms/about/, obiskano 16. maja 2019.

[14] Stropnik, V. (2018). Kreacija Person. Google Forms. Dostop: https://forms.gle/s2FQ2DfSw5moHiXUA , obiskano 16. maja 2019.

[15] Microsoft Powerpoint. Microsoft (1990).

[16] LOWDERMILK, Travis. User-centered design: a developer's guide to building user-friendly applications. " O'Reilly Media, Inc.", 2013 str. 81.

[17] SPENCER, Donna, 2011, Card Sorting. Sebastopol : Rosenfeld Media.

[18] LOWDERMILK, Travis. User-centered design: a developer's guide to building user-friendly applications. " O'Reilly Media, Inc.", 2013 str. 89.

[19] SCHUEFFEL, Patrick. The Concise Fintech Compendium. Fribourg/Switzerland : School of Management, 2017.

[20] SAILER, Michael, HENSE, Jan Ulrich, MAYR, Sarah Katharina and MANDL, Heinz. How gamification motivates: An experimental study of the effects of specific game design elements on psychological need satisfaction. Computers in Human Behavior. 2017. Vol. 69, p. 371–380.

[21] HAMARI, Juho. Do badges increase user activity? A field experiment on the effects of gamification. Computers in Human Behavior. 2017. Vol. 71, p. 469–478.

[22] RUHI, Umar. Level Up Your Strategy: Towards a Descriptive Framework for Meaningful Enterprise Gamification. Technology Innovation Management Review. 2015. Vol. 5, no. 8p. 5–16.

[23] HAMARI, Juho, SHERNOFF, David J., ROWE, Elizabeth, COLLER, Brianno, ASBELL-CLARKE, Jodi and EDWARDS, Teon. Challenging games help students learn: An empirical study on engagement, flow and immersion in game-based learning. Computers in Human Behavior. 2016. Vol. 54, p. 170–179.

[24] SCHUNK, Dale H., MEECE, Judith L. and PINTRICH, Paul R. Motivation in education: theory, research, and applications. Boston : Pearson, 2014.

[25] HASSENZAHL, Marc and TRACTINSKY, Noam. User experience - a research agenda. Behaviour & Information Technology. 2006. Vol. 25, no. 2p. 91–97.

[26] Instagram. Instagram (2012). Dostop: https://instagram-press.com/our-story/, obiskano 16. maja 2019

[27] Snapchat. Snap Inc. (2012). Dostop: https://whatis.snapchat.com/, obiskano 16. maja 2019

[28] QuizUp. Glu (2014).

[29] Trivia Crack. Eternax (2013).

[30] SongPop. FreshPlanet (2012).

[31] Balsamiq Mockups 3. Balsamiq Studios, LLC (2008).

[32] Adobe Photoshop CC 2018. Adobe Inc. (2017).

[33] Microsoft Paint. Microsoft (2015).

[34] Stropnik, V. How will our kulTura app look? (2018). Prezi.com. Dostop: https://prezi.com/daykipjp8s0y/?utm_campaign=share&utm_medium=copy, obiskano 14. maja 2019.

[35] Prezi. Prezi (2009). Dostop: https://prezi.com/product/ , obiskano 16. maja 2019

[36] Maps - Navigate & Explore. Google LLC (2005). Dostop: https://www.google.com/maps/about/, obiskano 16. maja 2019.

[37] Pokemon GO. Niantic (2016).

[38] Adobe XD. Adobe Inc. (2016). Dostop: https://www.adobe.com/products/xd.html, obiskano 16. maja 2019

[39] Material Design. Introduction (2014). Dostop: https://material.io/design/introduction/#, obiskano 12. maja 2019.

[40] InVision. (n.d.). InVisionApp Inc. Dostop: https://www.invisionapp.com/about/, obiskano 16. maja 2019.

[41] JustInMind. (n.d.). justinmind. Dostop: https://www.justinmind.com/, obiskano 16. maja 2019.

[42] Xdresources.co. (n.d.). Material UI Kit. Dostop: https://xdresources.co/resources/material-ui-kit , obiskano 16. maja 2019.

[43] Adobe Creative Cloud. Adobe Systems (2011).

[44] Unity. Unity Technologies (2005). Dostop: https://store.unity.com/products/unity-personal?_ga=2.174284002.1791839523.1558557152-1818264421.1554517046, obiskano 17. maja 2019.

[45] MapBox SDK. MapBox (2010). Dostop: https://www.mapbox.com/navigation/, obiskano 17. maja 2019.

OTS 2019
SODOBNE INFORMACIJSKE TEHNOLOGIJE IN STORITVE
M. Heričko in K. Kous (ur.)

Univerzitetna založba
Univerze v Mariboru

KO SOČASNOST IN FUNKCIJA PRESEŽETA OBJEKTNOST

BOJAN ŠTOK, TEODOR VEINGERL

Povzetek: *Večina modernih programskih jezikov vedno dobiva nove funkcionalnosti tako, da kopirajo novosti in koncepte iz drugih jezikov. Tako postajajo jeziki vedno bolj zapleteni in med sabo tekmujejo (C, C++, C#, Java, Python, Javascript, Kotlin…). Pri jeziku Go pa so se ob nastanku odločili, da bodo razvili enostaven in čist jezik, kjer bosta sočasnost (ang. concurrency) in gola funkcija (brez objekta) primarno vodilo. Pri tem želijo, da je sintaksa čim preprostejša in nima nepotrebnih konstruktov - manj je več. Za razliko od večine drugih modernih jezikov, ki se prevajajo v neko interpretirano vmesno kodo, se koda jezika Go prevede v strojno kodo za določeno arhitekturo procesorja in operacijski sistem. Tudi sam prevajalnik za Go je napisan v jeziku Go. Druga posebnost je, da jezik ni objektno orientiran, čeprav ima koncept vmesnikov (interface) in polimorfizma. Za razliko od jezikov C, C++ in Rust, ki se prav tako prevajajo v strojno kodo, ima Go vgrajen upravljalec pomnilnika (ang. garbage collector). Koncept vmesnikov je zelo močan konstrukt, ki omogoča ponovno uporabo programske kode. Kanal (ang. channel) je vgrajeni podatkovni tip. Jezik Go ne pozna niti, pač pa so implementirali kooperativni časovni razporejevalnik (ang. cooperating scheduler), ki upravlja z Go rutinami. Go rutine so lahke, saj jih lahko znotraj enega procesa hkrati teče tudi 500.000 in več. Go je sicer splošno-namenski jezik za strežniške programe, vendar je namenjen predvsem obdelavam v visoko zahtevnih paralelnih okoljih (npr. strežniške farme), kjer izpodriva klasične večnitne in večprocesne pristope.*

Ključne besede: *programski jezik Go, sočasnost, porazdeljeno procesiranje, CSP, goroutine*

NASLOVA AVTORJEV: mag. Bojan Štok, Institut informacijskih znanosti, Maribor, Slovenija, e-pošta: bojan.stok@izum.si. Teodor Veingerl, Institut informacijskih znanosti, Maribor, Slovenija, e-pošta: teodor.veingerl@izum.si.

https://doi.org/10.18690/978-961-286-282-4.22 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 UVOD

Večina modernih programskih jezikov dobiva nove in nove funkcionalnosti tako, da kopirajo novosti in koncepte iz drugih jezikov. Tako postajajo jeziki vedno bolj zapleteni in med sabo tekmujejo (C, C++, C#, Java, Python, Javascript, Kotlin…). Pri jeziku Go pa so se ob nastanku odločili, da bodo razvili enostaven in čist jezik [1]. Jezik Go so leta 2007 v podjetju Google zasnovali Rob Pike, Ken Thompson in Robert Griesemer. Pike in Thompson sta sodelovala tudi pri razvoju operacijskega sistema Plan 9 (predhodnik sistema Unix) in programskega jezika B (predhodnik jezika C) v podjetju Bell Labs [2,3]. Griesemer pa je sodeloval pri prevajalniku za Java HotSpot [4]. Vsi trije so imeli veliko različnih izkušenj in različen pogled na svet. Strinjali so se, da želijo razviti jezik, ki bo imel čim preprostejšo sintakso brez nepotrebnih konstruktov, da bo omogočal enostavno programiranje kot ga omogočajo dinamično tipizirani jeziki (npr. Python, Ruby), vendar z učinkovitostjo in robustnostjo statično tipiziranih jezikov, ki se prevajajo v strojno kodo. Jezik naj bi bil moderen, podpirati bi moral omrežne funkcionalnosti in imeti vgrajeno podporo za izvajanje na več jedrnih sistemih (ang. multicore computing) [5].

Za razliko od večine modernih jezikov, ki se direktno interpretirajo oz. se prevajajo v neko interpretirano vmesno kodo, se koda jezika Go prevede v strojno kodo za ciljni operacijski sistem in arhitekturo procesorja. Tudi sam prevajalnik za Go je napisan v jeziku Go. Druga posebnost je, da jezik ni objektno orientiran, čeprav ima koncept vmesnikov (ang. interface) in polimorfizma. Za razliko od jezikov C, C++ in Rust, ki se prav tako prevajajo v strojno kodo, ima Go vgrajen avtomatski upravljalnik pomnilnika (ang. garbage collector). Avtorji jezika so bili mnenja, da mora programer pri jeziku C in C++ preveč napora vložiti v zaseganje in sproščanje pomnilnika. Zavedali so se, da ima upravljalnik pomnilnika svojo ceno, vendar sintaksa in semantika jezika omogočata, da je to upravljanje učinkovitejše kot pri jeziku Java [5].

2 SINTAKSA JEZIKA

Čeprav je osnovna sintaksa jezika podobna jeziku C, je nanjo vplival tudi jezik Pascal oz. Modula – zlasti kar se tiče deklaracije in paketov [5]. Ime navajamo pred tipom.

Primer:

- var x,y int
- var f1 func([]int) int
- type T struct {a, b, c int}

Nov stavek napišemo v novi vrstici. Znaka podpičje ne uporabljamo. Komentar označuje // ali /* ... */ .

Osnovni vgrajeni tipi:

- int8, int16, int32, int64, uint8, uint16, uint32, uint64 (byte je sinonim za uint8 ipd.)
- float32, float64
- bool
- complex64, complex128
- string (nespremenljiv, kodiran po standardu UTF-8)

Kontrolne strukture:

- **if** pogoj {…} **else** {…} – pogoja ne dajemo v oklepaj, zaviti oklepaji so obvezni.
- **switch** c { **case** x: … } – ni ukaza "break". Če je pogoj izpolnjen se avtomatsko izvede "break".
- **for** {…} – lahko je brez pogoja. Znotraj zanke lahko uporabimo ukaza "continue" in "break".

Funkcija ima lahko nič ali več vhodnih parametrov. Vrača lahko več kot eno vrednost. (Pri večini jezikov lahko vrača le eno vrednost.)

Primeri oblike funkcij:

- **func** fname()
- **func** fname(par1 type1, par2 type2) returnType
- **func** fname(par1 type1, par2 type2) (retType1, retType2) // return 2, 5

- **func** fname(par1 type1, par2 type2) (y1 retType1, y2 retType2)
  {
  // named return values
    y1=2
    y2=5
    return
  }

Posebnost so "metode", kjer pred imenom funkcije v oklepaju navedemo tip (struct) na katerega se nanaša implementacija določene metode:

- **func** (rArg ReceiverType) fname(par1 type1, par2 type2) (retType1, retType2)
- **func** (rArg *ReceiverType) fname(par1 type1, par2 type2) (retType1, retType2)

Domiselno so določili kako definiramo ali so funkcije, spremenljivke ali tipi, javne ali privatne. Mala začetnica pomeni, da je metoda/spremenljivka/tip vidna le znotraj paketa (ang. private), velika pa pomeni, da je vidna tudi navzven (ang. public, exported).

Go obravnava funkcije kot "prvorazredne državljane" (ang. first class citizen). Kazalec na funkcijo lahko pošljemo kot argument v drugo funkcijo. Funkcija lahko vrne funkcijo. Funkcija lahko nastopa kot tip. Lambda funkcije so anonimne funkcije, ki lahko uporabijo tudi spremenljivke iz "okolice" (tj. iz klicajoče funkcije).

Veliko truda so namenili omogočanju ponovne uporabe kode in možnosti modularizacije. V enem paketu imamo lahko eno ali več datotek Go. Paketu, ki ga uvozimo, določimo ime oz. oznako (ang. identifier), preko katere se sklicujemo na imena funkcij in tipov znotraj paketa. Čeprav je uporaba paketov zelo preprosta, sta koncept in izvedba razreševanja referenc v ozadju, kompleksna in domišljena. Nepotrebni uvozi paketov niso dovoljeni. Če uvozimo paket, ki ga nikjer v kodi ne uporabimo, prevajalnik javi usodno napako (ang. fatal error).

Podobno velja, če deklariramo spremenljivko, ki je v kodi ne uporabimo – takšne kode ne moremo prevesti. Na ta način so avtorji jezika želeli doseči, da že prevajalnik prepreči čim več potencialnih programerskih napak.

2.1 Obravnava napak pri izvajanju

V nasprotju z drugimi programskimi jeziki, napake niso obravnavane kot izjeme, zato tudi ni ukazov try/catch. Napake so le poseben tip. Avtorji so bili mnenja, da obravnava izjem terja preveč dodatnega procesiranja in zmanjša preglednost kode. Zato jezik Go definira vmesnik **error**. Seveda lahko definiramo lasten tip (struct), ki poleg kode napake vsebuje še dodatne informacije. Dovolj je, da definira vmesnik **error**. Tipično tako funkcije vrnejo dvoje: rezultat in status. Primer klica funkcije za odpiranje datoteke:

```
f, err := os.OpenFile(fnm, os.O_RDONLY, 0644)
if err != nil {
  return err
}
```

2.2 Testiranje kode

Paket "testing" vsebuje podporo za avtomatsko testiranje Go paketov. Testi se zaženejo z ukazom **go test**. Testna funkcija mora biti oblike: **func** TestXxx(*testing.T)"

Testne funkcije morajo biti v datoteki s pripono "_test.go" (npr.: my_test.go). Datoteke *_test.go se prevajajo le v fazi testiranja in se ne vključijo v končno verzijo paketa.

Na podoben način lahko vključimo tudi zmogljivostne teste. Njim namenjene funkcije morajo biti oblike: **func** BencmarkXxx(*testing.B). Zmogljivostne teste vključimo s kretnico **bench**. Primer ukaza:

```
go test -v -bench=. -benchtime=100ms
```

3 VMESNIKI

Čeprav jezik Go ni objektno orientiran, pozna koncept vmesnikov. Nimamo razredov (ang. class), imamo pa tipe (ang. type). Tipe določimo na tri načine:

- **type** T **struct** – struktura (vrednost)
- **type** T **interface** – obnašanje (določimo seznam funkcij, ki definirajo vmesnik)
- **type** T **func**(x,w int) int – funkcija kot tip

Vmesnik določa množico funkcij z določenimi podpisi. Tako vsak tip implicitno definira prazen vmesnik (interface{}). V Javi ima podobno vlogo razred Object.

V jeziku Go pri vmesniku tipično definiramo eno samo metodo. Tako na primer vmesnik "fmt.Stringer" definira metodo "String()" tipa "string":

```
type Stringer interface {
        String() string
}
```

Vsako spremenljivko lahko torej obravnavamo kot par (value, interface). Ko implementiramo nek vmesnik tega eksplicitno ne navedemo. Prevajalnik sam preveri, katere vmesnike posamezni tip implementira. Koncept vmesnikov nam omogoča ponovno uporabo iste kode v drugih modulih. Primer:

```
type Shape interface {
    Surface() float
}

type Rectangle struct {
    A, B float
}

type Circle struct {
    R float
}

func (r Rectangle) Surface() float {
    return r.A * r.B
}

func (c Circle) Surface() float {
    return math.Pi * c.R * c.R
}
```

Definirali smo tip "Shape" kot vmesnik. Tipa "Rectangle" in "Circle" implementirata vmesnik "Shape", če obstaja implementacija metode "Surface() float", kjer je prejemnik (ang. receiver) tip "Circle" oz. "Rectangle".

Primer uporabe vmesnika in konkretnega vrednostnega tipa (struct):

```
func main() {
      var shapes []Shape
      rec1 := Rectangle{A: 2.1, B: 3.1}
      rec2 := Rectangle{A: 5.2, B: 9.33}
      cir1 := Circle{R: 15}
      shapes = append(shapes, rec1, rec2, cir1)
      for _, shape := range shapes {
            surf := shape.Surface()
            fmt.Printf("Shape:%T Surface:%12.3f\n", shape, , surf)
      }
}
```

4 SOČASNOST

Večina programskih jezikov (Java, C++, Python…) rešuje sočasno izvajanje z nitmi (ang. threads). Niti med seboj tipično komunicirajo z uporabo skupnega pomnilnika. Da ohranimo konsistentnost deljenih struktur, moramo dostop do njih sinhronizirati oz. uporabiti zaklepanje.

4.1 Gorutine

Go elegantno rešuje problem sočasnosti in sinhronizacije s t.i. "Gorutinami" (ang. Goroutine), ki med seboj komunicirajo s pošiljanjem podatkov po kanalih (ang. channels) [6]. To zagotavlja, da lahko le ena hkrati dostopa do posameznega podatka. Sicer pa ta pristop temelji na konceptu "Komunikacije sekvenčnih procesov" (ang. Communicating Sequential Processes – CSP), ki ga je opisal Tony Hoare leta 1978.

Gorutine so precej lažje od niti. Operacijski sistem jih ne pozna, zato tečejo znotraj internega razporejevalnika (ang. scheduler) "Go runtime". Gola Gorutina zasede pribl. 2 kB pomnilnika, torej jih lahko znotraj procesa, ki ima 2 GB pomnilnika, zaženemo skoraj milijon. (Pri večnitnih procesih praviloma ne moremo pognati več kot nekaj tisoč niti na proces.)

4.2 Kanali

Gorutine komunicirajo preko kanalov. Pri definiciji kanala vnaprej določimo tip sporočil, ki jih bo sprejemal. Torej skozenj ne moremo pošiljati sporočil z različnimi podatkovnimi tipi. Kanal ima privzeto dodeljen prostor le za eno sporočilo. Gorutina, ki skuša pisati v poln kanal, bo blokirana, dokler se kanal ne sprosti. Seveda lahko pri definiciji kanala določimo tudi drugačno velikost pomnilnika za sporočila (ang. buffer). Definirajmo na primer kanal, ki lahko hrani 20 sporočil tipa **int**:

```
c := make(chan int, 20)
```

Podobno je Gorutina, ki želi brati iz praznega kanala, tako dolgo blokirana, dokler sporočilo v kanalu ni na voljo.

Z drugimi besedami: Gorutine lahko pišejo v nek kanal, dokler ni dosežena velikost njegovega predpomnilnika, zatem bodo blokirane. Po drugi strani Gorutine, ki berejo iz kanala, ne bodo blokirane, dokler je v njem še vsaj eno sporočilo.

Sporočilo vpišemo v kanal z operatorjem "<−":

```
ch <- data
```

Z enakim operatorjem sporočilo preberemo:

```
data <- ch
```

4.3 Select

Go pozna poseben konstrukt **select**, s katerim lahko beremo iz večih kanalov hkrati. V tem primeru Gorutina čaka, da se v kateremkoli kanalu, navedenem v **select case**, pojavi sporočilo. Posebna vrsta kanala je časovnik (ang. timer), ki v kanal vpiše sporočilo, ko poteče naveden čas. Primer:

```
timeout := time.After(100 * time.Millisecond)
      select {
      case result := <-ch1:
      case result := <-ch2:
      case <-timeout:
      }
```

Izvajanje konstrukta **select** bo blokirano, dokler se v "ch1", *ali* v "ch2", *ali* v "timeout" ne pojavi neko sporočilo. Pomeni, da se bo čez 100 ms izvajanje nadaljevalo, tudi če bosta ch1 in ch2 še vedno prazna.

Koncept kanalov je zelo fleksibilen. Ni nujno, da posamezna Gorutina zgolj bere iz nekega kanala. Vanj in v druge kanale lahko tudi piše ali bere iz njih.

Mnogokrat koncept CSP primerjajo z t.i. modelom **akter** (ang. actor), ki ga uporabljatata jezik Erlang ter knjižnica Akka v jeziku Java. Ta model lahko implementiramo v jeziku Go tako, da vsakemu akterju dodelimo eno Gorutino z enim vhodnim kanalom. Posamezni akter svoje izhodne podatke pošilja drugim akterjem v njihove kanale kot sporočila. V teoriji tak model potrebuje neomejene vhodne kanale, da posamezna nit oz.

Gorutina, ki pošilja sporočilo akterju ne bo nikdar blokirana. V nasprotnem primeru si lahko akterji medsebojno zaklenejo kanale (ang. deadlock).

Go ne omogoča neomejenih kanalov. Če zazna, da so medsebojno zaklenjeni, avtomatsko prekine izvajanje z izjemo "deadlock".

4.4 Primer programa z uporabo kanalov in funkcijskih spremenljivk

V EU, RU in US imamo nameščene strežnike. Želimo implementirati program, ki zahtevo pošlje na vse lokacije in zbere njihove rezultate.

Na vsaki lokaciji imamo vsaj dva strežnika v gruči. Na vsakem od njih se izvede ista zahteva, uporabili pa bomo le rezultat tistega strežnika, ki bo najhitreje izračunan. Primer programa je na sliki 1 in tudi na naslovu https://bit.ly/DistributedServer.

```
1  package main
2
3  import (
4          "fmt"
5          "math/rand"
6          "time"
7  )
8
9  var (
10         EU1 = DbServer("EU1")
11         EU2 = DbServer("EU2")
12         EU3 = DbServer("EU3")
13         RU1 = DbServer("RU1")
14         RU2 = DbServer("RU2")
15         US1 = DbServer("US1")
16         US2 = DbServer("US2")
17         US3 = DbServer("US3")
18         US4 = DbServer("US4")
19 )
20
21 type Result string
22 type Request func(request string) Result
23
24 func DbServer(location string) Request {
25         return func(request string) Result {
26                 time.Sleep(time.Duration(rand.Intn(130)) * time.Millisecond)
27                 return Result(fmt.Sprintf("%s result for %q\n", location,
28                         request))
29         }
30 }
31
```

```
32 func FirstResult(request string, replicas ...Request) Result {
33         replChan := make(chan Result)
34         replicaRequest := func(i int) { replChan <- replicas[i](request) }
35         for i := range replicas {
36                 go replicaRequest(i)
37         }
38         return <-replChan
39 }
40
41 func DistributedRequest(query string) (results []Result) {
42         c := make(chan Result)
43         go func() { c <- FirstResult(query, EU1, EU2, EU3) }()
44         go func() { c <- FirstResult(query, RU1, RU2) }()
45         go func() { c <- FirstResult(query, US1, US2, US3, US4) }()
46         timeout := time.After(90 * time.Millisecond)
47         for i := 0; i < 3; i++ {
48                 select {
49                 case result := <-c:
50                         results = append(results, result)
51                 case <-timeout:
52                         fmt.Println("timed out")
53                         return
54                 }
55         }
56         return
57 }
58
59 func main() {
60         rand.Seed(time.Now().UnixNano())
61         start := time.Now()
62         results := DistributedRequest("find go")
63         elapsed := time.Since(start)
64         fmt.Println(results)
65         fmt.Println(elapsed)
66 }
```

Slika 1: Primer programa z uporabo kanalov

V vrsticah 10 do 18 instanciramo spremenljivke, ki vsebujejo kazalce na funkcije za posamezne strežnike. Funkcija "DbServer" namreč vrne funkcijo "**func**(request string) Result".

Funkcija FirstResult (vrstica 32) vrne rezultat najhitrejšega strežnika v gruči. V prvem argumentu ("request") prejme zahtevo, drugi argument ("replicas …Request") pa je polje funkcij, ki izvajajo zahtevo na posameznih strežnikih v gruči. V vrstici 33 kreiramo kanal, kamor strežniki v gruči pišejo rezultate. V vrstici 34 definiramo kazalec na lambda funkcijo, ki posreduje zahtevo posameznemu (i-temu) strežniku. Rezultat, ki ga ta strežnik vrne, vpiše v kanal "replChan". (Lambda funkcija lahko dostopa do "zunanjih" spremenljivk, zato "vidi" kanal "replChan", čeprav je definiran izven nje.) V vrsticah 35 do 37 v zanki kreiramo Gorutine, ki kličejo to lambda funkcijo. Nato v vrstici 38 počakamo, da najhitrejši strežnik vpiše rezultat v "replChan" in ga vrnemo kot najhitrejši rezultat celotne gruče.

V vrstici 42 kreiramo kanal "c", kamor bodo Gorutine z vsake lokacije vpisale rezultat. V vrsticah 43 do 49 za vsako lokacijo kreiramo Gorutino z lambda funkcijo, ki v kanal "c" vpiše prvi rezultat posamezne gruče (klic funkcije "FirstResult"). Takoj zatem pokličemo to lambda funkcijo (oklepaj-zaklepaj na koncu vrstice).

V vrstici 46 zaženemo časovnik, ki se zaključi po 90ms. Časovnik ustvari nov kanal "timeout", kamor se po izteku časa vpiše sporočilo. V zanki beremo tri rezultate iz kanala "c" (vsak je odgovor posamezne gruče na poslano zahtevo). Stavek "select" prebere, ali rezultat iz kanala "c", ali pa sporočilo o pretečenem času iz kanala "timeout". Če uspe prebrati rezultat, ga doda v seznam (ang. slice) "results". Odgôvori v kanal "c" prihajajo v takem vrstnem redu kot jih gruče izračunajo.

Če od katerekoli lokacije ne dobimo odgovora v 90ms, iskanje zaključimo in izpišemo "timed out". Že prejeti rezultati se (v seznamu "results") v vsakem primeru vrnejo klicatelju.

V glavnem programu v vrstici 62 pokličemo funkcijo, ki pošlje zahtevo na vse lokacije. Funkcija vrne seznam rezultatov.

V kakšnem drugem programskem jeziku bi bila koda precej bolj kompleksna, verjetno manj robustna in najbrž bi bil program vsaj dvakrat daljši.

5 ROBUSTNOST

Z jezikom Go lahko napišemo zelo robustne programe, saj ima vgrajenih ogromno mehanizmov, ki povečajo robustnost:

- varno upravljanje pomnilnika (ang. memory safety),
  - omogoča delo s kazalci (ang. pointers), ne omogoča pa aritmetike z njimi,
  - avtomatsko določanje alokacije pomnilnika na skladu (ang. stack) ali kopici (ang. heap) – to povečuje zmogljivost programa,
  - upravljalnik pomnilnika s samodejnim čiščenjem (ang. garbage collector),
  - preverjanje mej rezin (ang. slices) in polj (ang. arrays),
- varno delo s tipi – ni direktnega prirejanja tipov (ang. casting),
  - statični tipi,
  - eksplicitna konverzija številčnih tipov (int64 + int32: napaka),
  - ni mogoča nevarna implicitna konverzija med tipi (13 + "go"),
  - vmesniki (ang. interfaces) se implicitno preverjajo že med prevajanjem (vmesnik vedno hrani tudi tip vrednosti),
- neuporabljene spremenljivke javijo napako pri prevajanju,
- napake niso izjeme, temveč tipi (obravnavanje izjem pogosto pripelje do napak v programu),
- kanali in Gorutine elegantno rešujejo probleme sočasnosti.

6 RAZVOJNO OKOLJE

Razvojno okolje za Go je na voljo za Linux, Windows in Apple macOS. A ne glede na to na kateri platformi razvijamo, imamo v razvojno okolje že vključen navzkrižni prevajalnik (ang. cross-compiler), ki omogoča, da kodo prevedemo v binarno kodo za naslednje operacijske sisteme in arhitekture oz. procesorje:

- Linux: x86, amd64, arm, arm64, ppc64, ppc64le, mipsle, mips64, mips64le;
- Windows: x86, amd64;
- Darwin: x86, amd64, arm, arm64;
- FreeBSD: x86, amd64, arm;
- OpenBSD: x86, amd64, arm;
- NetBSD: x86, amd64, arm;
- Android: arm;
- Solaris: amd64;
- Plan9: x86, amd64.

Če na primer želimo na razvojnem računalniku Windows prevesti kodo za RaspberryPi, to storimo z naslednjim ukazom:

```
set "GOOS=linux" && set "GOARCH=arm" && go build myprogram.go
```

Obstaja več IDE okolij za razvoj v jeziku Go. Najbolj se uporabljata odprtokodna IDE: Visual Studio Code in Eclipse z vtičnikom (ang. plug-in) Go. Zelo dober IDE pa je tudi komercialni GoLand podjetja JetBrains.

7 STANDARDNE KNJIŽNICE

Sam jezik že vsebuje precej standardnih knjižnic, ki nam omogočajo, da program razvijemo skoraj brez uporabe zunanjih. V nadaljevanju je naštetih nekaj knjižnic.

Paket "os" implementira vmesnik do operacijskega sistema, neodvisen od platforme (os.OpenFile, os.Mkdir, os.Chmod, os.Rename, os.Remove).

Paket "net" omogoča delo z mrežnimi protokoli (funkcije: Dial, Listen, Accept) in sicer podpira tcp, tcp4, tcp6, udp, udp4, udp6, ip, ip4, ip6, unix, unixgram in unixpacket. Podpira ipV4 in ipV6.

Net/http je paket, ki implementira HTTP odjemalca in strežnik. Z eno programsko vrstico lahko ustvarimo HTTP strežnik. Primer:

```
      http.HandleFunc("/hello",      func(w      http.ResponseWriter,      r
*http.Request) {
            w.Header().Set("Content-Type", "text/plain")
            w.Write([]byte("Hello Go\n"))
      })
      http.ListenAndServeTLS(":8080", "cert.pem", "key.pem", nil)
```

Paket "sql" omogoča generični vmesnik za dostop do baz podatkov:

- db, err := sql.**Open**(driverName, dataSourceName)
- rows, err := db.**Query**(queryString, args)
- err := row.Scan(dest…) – kopira vrednosti stolpcev trenutne vrstice v polje "dest"
- stmt, err := db.**Prepare**("DELETE FROM item WHERE code=$1 AND location='$2' ") – pripravi parametrični SQL stavek s parametroma $1 in $2
- res, err := stmt.**Exec**(15, "loc2") – vpiše parametre in izvede predpripravljen SQL stavek

8 SLABOSTI JEZIKA

Jezik Go ne podpira generičnih tipov. Zato programska koda včasih postane kompleksnejša oz. jo je treba podvajati.

Čeprav razvojno okolje jezika Go že vsebuje mnogo knjižnic (http, json, socket, database), je njihov seznam precej krajši kot pri starejših in bolj razširjenih programskih jezikih (npr. Java ali Python).

9 ZAKLJUČEK

Jezik Go so razvili v podjetju Google leta 2007, ker nobeden izmed obstoječih jezikov (Java, C++, Python…) ni imel dobrih rešitev za razvoj programske infrastrukture v Googlu [5]. Zasnova jezika naj bi zagotavljala hitrejši oz. produktivnejši razvoj programov. Ima vgrajene mehanizme za sočasnost, upravljalnik pomnilnika, dober mehanizem odvisnosti, možnost prilagajanja programske arhitekture, ko sistem raste in zagotavlja robustne programe tudi pri kompleksnih rešitvah, sestavljenih iz mnogih komponent.

V jeziku Go so napisani naslednji najbolj znani odprto-kodni projekti: Docker, Kubernetes, etcd, InfluxDB, Prometheus itd. To dokazuje, da je jezik zrel in primeren za produkcijo.

Ker se koda v jeziku Go prevede v strojno kodo, je primeren za razvoj programov, ki tečejo tudi na napravah IoT z malo pomnilnika in slabšimi procesorji.

Največja moč jezika Go je v tem, da elegantno rešuje probleme sočasnosti, kar nam omogoča, da sorazmerno enostavno razvijamo zelo učinkovite paralelne strežniške programe. Za ta namen moramo na primer v programskem jeziku Java uporabiti kakšno knjižnico (npr. Akka), ali pa precej dodatnega truda vložiti v logiko zaklepanja.

10 LITERATURA

[1] https://golang.org/doc, The Go Programming Language, Documentation, obiskano 6. 5. 2019

[2] https://en.wikipedia.org/wiki/Rob_Pike, Rob Pike, obiskano 6. 5. 2019

[3] https://en.wikipedia.org/wiki/Ken_Thompson, Ken Thompson, obiskano 6. 5. 2019

[4] https://www.computerhope.com/people/robert_griesemer.htm, Robert Griesemer, obiskano 6. 5. 2019

[5] https://talks.golang.org/2012/splash.article, Go at Google: Language Design in the Service of Software Engineering, obiskano 6. 5. 2019

[6] https://blog.golang.org/share-memory-by-communicating, Share Memory By Communicating, obiskano 6. 5. 2019

OTS 2019
SODOBNE INFORMACIJSKE TEHNOLOGIJE IN STORITVE
M. Heričko in K. Kous (ur.)

Univerzitetna založba
Univerze v Mariboru

PRAKTIČNI PRIMERI ELEGANTNIH REŠITEV ZA ORKESTRACIJO ASINHRONIH OPERACIJ V PROGRAMSKEM JEZIKU GO

MARKO GAŠPARIČ

Povzetek: *Osnova programskega jezika Go je podpora sočasnosti, ki temelji na modelu imenovanem Communicating Sequential Processes. Go rutina in kanal sta primitivna tipa, namenjena enostavni in učinkoviti implementaciji sočasnega izvajanja. Pri razvoju tretje generacije Nicehash Miner aplikacije smo uporabili programski jezik Go in tri enostavne vzorce, ki so se tekom razvoja izkazali za uspešne. Prvi vzorec je namenjen prioritizaciji nalog znotraj modula, drugi hkratnemu izvajanju operacij in tretji testiranju enot. Opazili smo, da je nov pristop pohitril razvoj, poenostavil razumevanje in vzdrževanje programske kode, ter zmanjšal število napak med izvajanjem.*

Ključne besede: *asinhronost, orkestracija, programiranje, testiranje enot, Go*

NASLOV AVTORJA: Marko Gašparič, MaG IT d.o.o., Ptuj, Slovenija, e-pošta: m.gasparic@gmail.com.

https://doi.org/10.18690/978-961-286-282-4.23 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 UVOD

Začetki programskega jezika Go segajo v leto 2007, ko so Robert Griesemer, Rob Pike in Ken Thompson, pri podjetju Google, zasnovali cilje novega jezika, ki je konec leta 2009 postal javen in prosto-dostopen [1]. Čeprav Go vsebuje veliko konceptov, ki jih najdemo tudi v drugih programskih jezikih, se zdi, da ponuja boljše rešitve za razvoj aplikacij, ki so namenjene izvajanju na več-jedrnih in omrežnih napravah. Programi, ki so napisani v programskem jeziko Go, na način, kot so si ga zamislili njegovi avtorji, se občutno razlikujejo od programov spisanih v drugih priljubljenih jezikih, kot so na primer C++, Java ali JavaScript [2]; kljub temu, da je sintaksa jezika Go podobna sintaksi jezika C [3].

Go zagotavlja čiščenje pomnilnika in zahteva statično definiranje tipov. Pred izvajanjem je potrebno izvorno kodo prevesti v strojni jezik. Osnova jezika je podpora sočasnosti, ki temelji na modelu imenovanem Communicating Sequential Processes (CSP): komunkacija med procesi se ne izvaja preko deljenega pomnilnika, temveč se podatki delijo preko komunikacije [4]. Go rutina in kanal sta primitivna tipa namenjena enostavni in učinkoviti implementaciji sočasnega izvajanja.

V podjetju H-bit, pri razvoju tretje generacije Nicehash Miner aplikacije, smo uporabili programski jezik Go in nekaj enostavnih vzorcev, ki temeljijo na prej omenjenih konceptih. Opazili smo, da je nov pristop pohitril razvoj, poenostavil razumevanje in vzdrževanje programske kode ter zmanjšal število napak med izvajanjem. V tem prispevku je najprej predstavljen CSP model, skupaj z osnovami jezika Go, ki so potrebne za razumevanje nadaljnje vsebine. V tretjem poglavju so opisani trije vzorci, ki so namenjeni prioritizaciji operacij znotraj modulov, hkratnemu izvajanju operacij in testiranju enot. Ob vsakem vzorcu je predstavljen tudi primer njegove implementacije. V zaključku prispevka so povzete glavne prednosti in slabosti teh vzorcev kot celote, glede na njihovo uporabnost v praksi.

2 CSP MODEL

Že leta 1978 je Charles A. R. Hoare [5] predstavil predlog, po katerem bi "vhod" in "izhod" postala osnovna programska primitivna tipa, paralelna kompozicija - kot je zasnovana v CSP modelu - pa bi predstavljala temeljno strukturo vsakega programa. V tistem času so bili splošno sprejeti le trije programski konstrukti: klasična sekvenčna kompozicija oz. vrstica kode, ki se običajno konča s podpičjem, pogojni stavek in zanka. Vseeno so tudi leta 1978 obstajale potrebe po pohitritvi izvajanja programov na več-procesorskih računalnikih, zato je Hoare izdelal model, ki bi celovito rešil težave s komunikacijo med procesi, ki se izvajajo sočasno.

CSP model temelji na "vhodni" in "izhodni" komandi, ki se uporabljata za komunikacijo med procesi. Proces v tem kontekstu predstavlja sekvenco ukazov oz. komand. Komunikacija med dvema procesoma se izvede, ko en proces imenuje drug proces kot destinacijo "izhoda", ta drugi proces pa imenuje prvi proces kot vir "vhoda"; v takšnih primerih se vrednost "izhoda" kopira v drugi proces. Ker se v CSP modelu sporočila ne hranijo avtomatsko, je potrebno en proces zaustaviti,č dokler drug proces ni pripravljen na komunikacijo. V tem kontekstu ni pomembno ali je čakajoči proces blokiran na "vhodnem" ali "izhodnem" ukazu.

V programskem jeziku Go so za prenos sporočil na voljo kanali, ki lahko hranijo v naprej določeno število sporočil. Privzeta vrednost je sicer nič, kar je skladno z zahtevami CSP modela, lahko pa je začasna shramba tudi večja. V takšnem primeru, dokler shramba ni polna, se Go kanal obnaša kot klasična čakalna vrsta, ki po principu "prvi-noter-prvi-ven" posreduje sporočila naprej; ob tem pa proces ni blokiran na "vhodni" komandi, tudi, če noben drug proces ne čaka na sporočilo v danem trenutku.

Kadar proces pričakuje sporočilo, ki še ni bilo vstavljeno v kanal, "izhodna" komanda vedno blokira proces. V primeru, da več procesov čaka na "vhod" iz istega vira, je naključno izbran en proces, ki prejme kopijo "vhoda" (in lahko nadaljuje z izvajanjem), na ostale procese pa ta komunikacija nima vpliva. Zahteva CSP modela je, da sočasni procesi komunicirajo med seboj izključno na ta način in nikoli preko globalnih spremenljivk. Čeprav nam Go omogoča oba načina komunikacije, smo se pri naših vzorcih držali tega načela.

CSP model predvideva uporabo funkcije za preverjanje strukture "vhodov" in "izhodov", kar zagotavlja varno uporabo kopiranih sporočil. Ta zahteva je povsem skladna z zasnovo jezika Go, ki vsebuje izključno statično definirane tipe spremenljivk. Posledično je potrebno kanal za prenos sporočil definirati z rezervirano besedo "chan" in tipom sporočil. Npr. "var mojKanal chan string" definira spremenljivko z imenom "mojKanal", ki jo lahko uporabljamo za prenos tekstovnih sporočil med procesi. Instanciacija spremenljivke se izvede z ukazom

"make". Pri tem ukazu se tudi definira ali bo kanal vseboval začasno shrambo ali ne. Privzet ukaz, brez shrambe, izgleda tako: "mojKanal = make(chan string)". Če želimo definirati velikost shrambe, ki je večja od nič, npr. sto, moramo dodati ukazu še en parameter: "mojKanal = make(chan string, 100)".

Ukaz za vstavljanje sporočila v kanal oz. "vhodna" komanda ima naslednjo obliko: "kanal <- spremenljivka"; npr. "mojKanal <- "Pozdravljen OTS!"". Ukaz za branje sporočila iz kanala oz. "izhodna" komanda ima naslednjo obliko: "<-kanal". Izhod iz kanala je vrednost, ki jo lahko uporabljamo na enak način kot vse ostale vrednosti; npr. lahko je to vrednost nove spremenljivke: "pozdravnaVrstica := <-mojKanal", lahko jo izpišemo: "fmt.Println(<-mojKanal)", lahko jo zavržemo ipd.

Go rutine so funkcije, ki se znotraj istega programa izvajajo sočasno z drugimi Go rutinami [6]. Za izvajanje potrebujejo zelo malo virov. Zagon Go rutine se izvede na podlagi ukaza, ki je sestavljen iz rezervirane besede "go" in funkcije, ki jo želimo zagnati asinhrono. Npr. "go list.Sort()" v ločeni Go rutini pokliče funkcijo "Sort", v imenskem prostoru "list".

Poleg Go rutine, kanala in splošnih konceptov, kot so zanke in pogojni stavki, je za učinkovito implementacijo sočasnosti v jeziku Go pomembna še ena odločitvena struktura, to je "select". V drugih programskih jezikih, ki so nam poznani, ta koncept ne obstaja. Uporaba "select" strukture omogoča čakanje na več "izhodov" hkrati [7]. Z uporabo rezervirane besede "case" in različnih kanalov lahko določimo možne načine nadaljnjega izvajanja Go rutine. Stavek "select" zagotavlja izvedbo enega primera in je blokiran, dokler vsaj en kanal ne vsebuje sporočila, ki je namenjeno tej Go rutini. V primeru, da več kanalov čaka na branje "izhoda", je odločitev naključna. Z uporabo rezervirane besede "default" lahko določimo način nadaljnjega izvajanja, ki nikoli ne bo blokiral Go rutine pri "select" stavku: koda, ki spada pod "default", se namreč izvede takrat, ko noben "case" ni izvedljiv.

3 VZORCI

Poleg priljubljenih Go vzorcev, kot so npr. cevovod (angl. pipeline) in izrecna prekinitev (angl. explicit cancellation) [8], so v Nicehash Miner aplikaciji implementirani tudi nekateri drugi vzorci, ki so se tekom razvoja izkazali za uspešne. V nadaljevanju bomo predstavili tri.

3.1 Prioritizator

Prvi vzorec je namenjen prioritizaciji nalog znotraj modula in se uporablja v vsakem modulu, ki je implementiran v Nicehash Miner aplikaciji. Tudi v primeru asinhronega določanja nalog, s strani večih enot, naš vzorec zagotavlja zaporedno izvajanje nalog istega tipa.

Predloga vzorca je predstavljena v Tabeli 1. Bistvo vzorca je, da se naloge v modulu dodeljujejo izključno preko kanalov, ki so z uporabo "select" odločitvenih struktur, definiranih znotraj "default" primera, razvrščeni kaskadno. Takšna struktura omogoča enostavno določanje vrstnega reda sporočil, ki jih prenaša določen kanal. V primeru, da modul med aktivnim izvajanjem naloge prejme več novih nalog, bo po opravljeni nalogi vedno nadaljeval z nalogo, ki ima najvišjo prioriteto, glede na kaskadno strukturo. Npr., če modul prejme po eno nalogo po kanalih "ch_1_1", "ch_2_1" in "ch_2_2" (glej Tabelo 1), bo zagotovo najprej prebrano sporočilo iz kanala "ch_1_1", kar pomeni, da bo izvedena funkcija "handleMsg11(msg)". V kolikor med izvajanjem druge naloge modul ne prejme novih sporočil, sledi izvedba "handleMsg21(msg)" ali "handleMsg22(msg)" - kot smo že omenili, je izbira funkcije v takšnem primeru naključna.

Kadar so po opravljeni nalogi, ali ob inicializaciji Go rutine, vsi kanali prazni, se Go rutina zaustavi na zadnjem "select" stavku. To je koristno, saj na ta način modul ne odžira procesorske moči ostalim modulom, kljub neskončni zanki[1]. Potrebna je le previdnost, da ne pozabimo definirati vseh možnih primerov na zadnjem nivoju, saj se v nasportnem primeru lahko zgodi, da bo rutina blokirana na najnižjem nivoju, vse dokler ne prispe sporočilo, ki je tam podprto; in to tudi, če bi že prej prejeli sporočilo z višjo prioriteto, ki bi ga pozabili dodati v zadnji "select". Torej, v primeru pravilne implementacije, bo proces na najnižjem nivoju čakal na prvo sporočilo, ki bo prispelo, in bo ob prejemu le-tega takoj nadaljeval z izvajanjem dodeljene funkcije.

Vzorec Prioritizator temelji na treh predpostavkah: opisana Go rutina je edini vmesnik po katerem modul prejema sporočila iz okolja; sporočila se berejo iz kanalov uporabljenih v Go rutini le na enem mestu, se pravi

[1] Neskončna zanka se v Go jeziku definira z rezervirano besedo "for" in brez parametrov (glej Tabelo 1).

znotraj te rutine; in na nivoju izvajanja nalog je zagotovljena popolna sinhronizacija. Ta vzorec se je izkazal kot zelo uporaben pri obravnavanju napak, ki zahtevajo "okrevanje" modula pred nadaljevanjem izvajanja, in pri preprečevanju ciklov med asinhronimi procesi, kar je bila pogosta težava pri starejših verzijah Nicehash Miner aplikacije.

Kot primer uporabe vzorca si lahko predstavljamo modul, ki upravlja z aplikacijo za rudarjenje kripto-valut. Predpostavimo naslednje: modul lahko zažene in zaustavi zunanjo aplikacijo na podlagi zahteve uporabnika; modul preko http povezave, vsakih deset sekund, pridobiva podatke o hitrosti rudarjenja; vsakič, ko se zagnana aplikacija ugasne brez zahteve uporabnika, jo mora modul kar najhitreje ponovno zagnati; vsakič, ko je pridobivanje podatkov neuspešno, mora modul ponovno vzpostaviti povezavo, ter v primeru, da je tudi vzpostavitev neuspešna, ugasniti in ponovno zagnati zunanjo aplikacijo; v primeru, da uporabnik ugasne Nicehash Miner aplikacijo, mora modul kar najhitreje ugasniti zunanje aplikacije in zaključiti svoje izvajanje.

```
go func() {
	for {
		select {
		case msg <-ch_1_1:
			handleMsg11(msg)
		...
		case msg <-ch_1_n:
			handleMsg1n(msg)
		default:
			select {
			case msg <-ch_2_1:
				handleMsg21(msg)
			...
			case msg <-ch_2_m:
				handleMsg2m(msg)
			default:
				select {
				... ...
				default:
					select {
					case msg <-ch_1_1:
						handleMsg11(msg)
						...
					case msg <-ch_1_n:
						handleMsg1n(msg)
					case msg <-ch_2_1:
						handleMsg21(msg)
						...
					case msg <-ch_2_m:
						handleMsg2m(msg)
						...
					case msg <-ch_z_m:
						handleMsgzm(msg)
}}}}}}()
```

Tabela 1: Predloga za vzorec Prioritizator.

Elegantna rešitev za opisani problem bi lahko vsebovala pet funkcij: "start", "stop", "connect" (oz. "vzpostavi povezavo"), "disconnect" (oz. "prekini povezavo") in "fetchData" (oz. "pridobi podatke"). Najvišjo prioriteto bi imela naloga, ki povzroči izklop modula (glej "nhmClose" v Tabeli 2), drugo najvišjo bi imele vse naloge, ki so zadolžene za okrevanje po napaki (glej "errorChan" v Tabeli 2), tretjo najvišjo prioriteto bi imele naloge za zagon in izklop (glej "userCommand" v Tabeli 2), najnižjo pa pridobivanje podatkov (glej "dataTimer" v Tabeli 2). V primeru neuspešne izvedbe bi vsaka funkcija vstavila sporočilo v primeren kanal. Implementacija Prioritizator vzorca v opisanem primeru je prikazana v Tabeli 2.

```
go func() {
	for {
		select {
		case <-nhmClose:
			stop()
			return
		default:
			select {
			case err := <-errorChan:
				handleError(err)
			default:
				select {
				case cmd := <-userCommand:
					executeUserCommand(cmd)
				default:
					select {
					case <-dataTimer:
						fetchData()
					case cmd := <-userCommand:
						executeUserCommand(cmd)
					case err := <-errorChan:
						handleError(err)
					case <-nhmClose:
						stop()
						return
}}}}}}()
...
func handleError(err error) {
	switch err.Error() {
	case "app": start()
	case "http": disconnect(); connect()
}}

func executeUserCommand(cmd int) {
	switch cmd {
	case 0: stop()
	case 1: start()
}}
```

Tabela 2: Primer implementacije za vzorec Prioritizator.

3.2 Sinhronizator

Drugi vzorec je namenjen hkratnemu izvajanju operacij, ob tem da zagotavlja končanje le-teh pred nadaljevanjem izvajanja glavnega procesa. Ker prejšnji vzorec ne omogoča asinhronosti na nivoju modula in dodeljenih nalog, je ta vzorec komplementarna rešitev za pohitritev izvajanja posamezne naloge, v primerih, ko je nalogo možno razdeliti na ustrezne manjše operacije.

Predloga vzorca je predstavljena v Tabeli 3. Bistvo vzorca je, da (skoraj) sočasno požene zbirko Go rutin, ki ob koncu izvajanja pošljejo sporočilo v kanal, ki je namenjen detekciji zaključka rutin in potencialni kolekciji rezultatov operacij. Nujni elementi vzorca so torej kanal (glej "responseChannel" v Tabeli 3), ki je lahko poljubnega tipa; ovojna funkcija (glej "operationWrapper" v Tabeli 3), ki mora izvesti operacijo in poročati po kanalu; in zanka (glej "for i := 0; i < len(operations); i++" v Tabeli 3), ki zagotavlja čakanje na izvedbo vseh operacij. Zunaj teh okvirov je predviden cel spekter možnih variacij, npr.: ovojna funkcija lahko vključuje poslovno logiko, za katero je najbolj primerna lokacija implementacije prav v tej funkciji; kanal lahko prenaša sporočila, ki jih shranimo v poljubno podatkovno strukturo znotraj zadnje zanke, analiziramo pa jih po prejemu vseh sporočil, ali pa se poslovna logika za analizo teh sporočil izvede že znotraj zanke, kar zagotavlja sekvenčno analizo; v ovojno funkcijo lahko pošljemo različne parametre, tudi funkcije itd.

```
func handleMsg() {
	responseChannel := make(chan interface{})
	for o := range operations {
		go operationWrapper(operationSpecificArgs..., responseChannel)
	}
	for i := 0; i < len(operations); i++ {
		<-responseChannel
	}
}

func operationWrapper(operationSpecificArgs..., responseChannel chan interface{}) {
	responseChannel <- operation(operationSpecificArgs...)
}
```

Tabela 3: Predloga za vzorec Sinhronizator.

Ta vzorec se je izkazal kot zelo uporaben v primerih, ko modul podpira nalogo, ki zahteva izvedbo več operacij, ki so si med seboj podobne, npr: prenos shranjenih podatkov iz diska v pomnilnik, iskanje najbližjega stratum strežnika, pridobivanje podatkov o hitrosti rudarjenja itd. Še posebej uporaben je takrat, ko so omenjene operacije implementirane v obliki čistih funkcij[1], spreminjanje teh funkcij pa je neželjeno ali nemogoče.

V Tabeli 4 je predstavljen primer nalaganja konfiguracije v pomnilnik, kjer nas zanima le, če so se operacije zaključile uspešno. Npr., "loadGeneralSettings" (oz. "naloži osnovne nastavitve"), "loadMiningSettings" (oz. "naloži nastavitve za rudarjenje") in "loadDeviceSettings" (oz. "naloži nastavitve za naprave") so tri funkcije, ki ne sprejemajo argumentov in vrnejo "error" (oz. "obvestilo o napaki"), ki ga izpišemo, ko ni prazen (oz. "nil"). Kot je razvidno iz primera, implementacija vzorca Sinhronizator omogoča zelo enostavno dodajanje novih operacij istega tipa v isto nalogo.

```
func loadSettings() {
	checkingChannel := make(chan bool)
	go performLoadingTask(loadGeneralSettings, checkingChannel)
	go performLoadingTask(loadMiningSettings, checkingChannel)
	go performLoadingTask(loadDeviceSettings, checkingChannel)
	for i := 0; i < 3; i++ {
		<-checkingChannel
	}
}

func performLoadingTask(task func() error, checkingChannel chan<- bool) {
	if err := task(); err != nil {
		fmt.Println(err)
	}
	checkingChannel <- true
}
```

Tabela 4: Primer implementacije za vzorec Sinhronizator.

3.3 Tester

Tretji vzorec je namenjen testiranju enot. Podobno kot prejšnji, temelji na enostavni strukturi, ki omogoča zelo hitro dodajanje testnih primerov. Pri razvoju Nicehash Miner aplikacije ga uporabljamo za testiranje modulov, ki se izvajajo avtonomno. Kot smo že omenili, so vsi vmesniki preko katerih moduli prejemajo sporočila iz okolja, implementirani skladno z vzorcem Prioritizator, kar pomeni, da je znotraj modula ločena, asinhrona Go rutina, s katero lahko komuniciramo le preko kanalov. Ta omejitev vpliva tudi na zasnovo testov.

[1] Čista funkcija je funkcija, ki bo pri enakih vhodnih podatkih vedno vrnila enake izhodne podatke, ob tem pa nima opaznih stranskih učinkov [9].

Predloga vzorca je predstavljena v Tabeli 5. Bistvo vzorca je, da imajo vsi testni primeri isto strukturo, ki jo definiramo na začetku. Ta struktura vključuje seznam vhodnih sporočil, seznam pričakovanih izhodnih sporočil in dolžino čakanja na izhodna sporočila. Ker je vrstni red izvajanja v modulih predvidljiv, lahko s primernim časovnim intervalom zagotovimo, da je imel modul dovolj časa za pošiljanje vseh sporočil, ki jih pričakujemo, ob tem pa lahko še zagotovimo, da ni poslal nobenega nepričakovanega sporočila. Ta vzorec omogoča tudi preverjanje vrstnega reda sporočil, ki mora biti pravilen.

```
func Test(t *testing.T) {
	type test struct {
		inSequence     []messages.IncomingMessage
		expOutSequence []messages.OutgoingMessage
		timeout        time.Duration
	}
	var tests = []test{
		{[]messages.IncomingMessage{message1,...,messageN},
		 []messages.OutgoingMessage{message1,..., messageM}, time.Second},
		...
		{[]messages.IncomingMessage{message1,...,messageL},
		 []messages.OutgoingMessage{message1,..., messageK}, time.Second},
	}

	in := make(chan messages.IncomingMessage, 100)
	out := make(chan messages.OutgoingMessage, 100)
	Run(in, out)
	for i, tt := range tests {
		t.Run(fmt.Sprintf("test %d", i), func(t *testing.T) {
			responsesDeadline := time.Now().Add(tt.timeout)
			for _, inMsg := range tt.inSequence {
				in <- inMsg
			}
			retOutSequence := make(messages.OutgoingMessage, 0)
		ChannelLoop:
			for {
				select {
				case msg := <-out:
					retOutSequence = append(retOutSequence, msg)
				case <-time.After(100 * time.Millisecond):
					if time.Now().After(responsesDeadline) {
						break ChannelLoop
			}}}
			if !reflect.DeepEqual(retOutSequence, tt.expOutSequence) {
				t.Errorf("test failed")
}}}}}
```

Tabela 5: Predloga za vzorec Tester.

Vzorec Tester je zelo prilagodljiv. V eni instanci lahko pošljemo poljubno število vhodnih sporočil in preverimo poljubno število izhodnih. Na ta način lahko definiramo pakete testov že znotraj ene same instance, ki jih po želji združujemo v večje pakete znotraj enega testa enote (glej spremenljivko "tests" v Tabeli 5). Tako lahko z enostavnim dodajanjem testnih primerov simuliramo kompleksne scenarije izvajanja modula, ki so primerljivi s primeri iz prakse.

Pri vsaki testni instanci se bodo najprej poslala vsa sporočila v modul (glej "in <- inMsg" v Tabeli 5), potem se bodo prebrala vsa izhodna sporočila, ki bodo poslana v roku (glej "select" v Tabeli 5), na koncu pa se bo preverilo še, če so izhodna sporočila enaka pričakovanim (glej "if !reflect.DeepEqual(retOutSequence, tt.expOutSequence)" v Tabeli 5).

Primer paketa testov za testiranje modula, ki skrbi za detekcijo in osveževanje hitrosti različnih algoritmov, je predstavljen v Tabeli 6. Predstavljen je samo paket, ker so ostali deli testa identični predlogi v Tabeli 5. Kot

je razvidno iz primera, po štirih sporočilih, ki obveščajo modul o stabilnih hitrostih algoritma "24" na kartici "NVD-1", ne pričakujemo še nobenega odgovora od modula, ki ga testiramo (glej "AddHashesToHistory" in "[]messages.OutgoingMessage{}" v Tabeli 6). Po petem sporočilu, ki tudi vsebuje hitrost blizu 100, pa mora modul odgovoriti s poročilom o stabilnih hitrostih (glej "StableHashesReportReady" v Tabeli 6).

```
var tests = []test{
	{[]messages.IncomingMessage{
		messages.AddHashesToHistory{DeviceId:	"NVD-1",	AlgorithmId{24},
Hash{100}},
		messages.AddHashesToHistory{DeviceId:	"NVD-1",	AlgorithmId{24},
Hash{100.1}},
		messages.AddHashesToHistory{DeviceId:	"NVD-1",	AlgorithmId{24},
Hash{99.9}},
		messages.AddHashesToHistory{DeviceId:	"NVD-1",	AlgorithmId{24},
Hash{100.2}},
	},[]messages.OutgoingMessage{}, 100 * time.Millisecond},
	{[]messages.IncomingMessage{
		messages.AddHashesToHistory{DeviceId:	"NVD-1",	AlgorithmId{24},
Hash{100.2}},
	},[]messages.OutgoingMessage{messages.StableHashesReportReady{HashesPerDevice:
	map[StabilityAnalyzerSetting]Hash{{"NVD-1", AlgorithmId{24}: Hash{100}}}}}, 200
* time.Millisecond},
}
```

Tabela 6: Primer implementacije za vzorec Tester.

4 ZAKLJUČEK

V tem prispevku smo s pomočjo primerov iz prakse predstavili tri vzorce za implementacijo in testiranje sočasnosti v programskem jeziku Go. Vzorci, ki se med seboj dopolnjujejo, so občutno zmanjšali najpogostejše težave preteklih Nicehash Miner aplikacij, to so programski hrošči, ki so povezani s cikli v katere občasno zapadejo asinhronimi procesi. Takšni hrošči vodijo v preobremenitev sistema, nekontrolirano zapiranje in odpiranje zunanjih aplikacij in povezav, zamrznitev uporabniškega vmesnika ipd. Konsistentnost in preglednost izvorne kode, ki je implementirana na podlagi treh opisanih vzorcev, je omogočila lažje razumevanje kompleksnih stanj, v katerih je lahko Nicehash Miner aplikacija. Posledično je postalo vzdrževanje in nadgrajevanje aplikacije lažje in hitrejše, saj se je za ceno nekaj dodatnih vrstic (podvojene) kode in nekaj milisekund počasnejšega izvajanja kot bi bilo optimalno, zmanjšala zahtevnost nadzora nad sočasnim izvajanjem. Implementacija opisanih vzorcev je omogočila tudi testiranje enot, ki pri prejšnjih generacijah Nicehash Miner aplikacije ni bilo izvedljivo v praksi.

5 LITERATURA

[1] https://golang.org/doc/faq#history, Frequently Asked Questions: What is the history of the project?, obiskano 30.4.2019.

[2] https://golang.org/doc/effective_go.html#introduction, Effective Go: Introduction, obiskano 30.4.2019.

[3] https://blog.golang.org/gos-declaration-syntax, Go's Declaration Syntax, obiskano 30.4.2019.

[4] https://golang.org/doc/effective_go.html#concurrency, Effective Go: Concurrency, obiskano 30.4.2019.

[5] HOARE Charles A. R. "Communicating sequential processes", Communications of the ACM, letnik 21, številka 8, avgust 1978, str. 666-677.

[6] https://golang.org/doc/effective_go.html#goroutines, Effective Go: Goroutines, obiskano 30.4.2019.

[7] https://tour.golang.org/concurrency/5, A Tour of Go: Select, obiskano 30.4.2019.

[8] https://blog.golang.org/pipelines, The Go Blog: Go Concurrency Patterns: Pipelines and cancellation, obiskano 30.4.2019.

[9] https://mostly-adequate.gitbooks.io/mostly-adequate-guide/ch03.html, Professor Frisby's Mostly Adequate Guide to Functional Programming: Pure Happiness with Pure Functions, obiskano 3.5.2019.

OTS 2019
SODOBNE INFORMACIJSKE TEHNOLOGIJE IN STORITVE
M. Heričko in K. Kous (ur.)

Univerzitetna založba
Univerze v Mariboru

PRIHODNOST JAVE V LUČI ZADNJIH VELIKIH SPREMEMB

ANDREJ KRAJNC, CIRIL PETR, MITJA SKUHALA, GREGA RAMŠAK

Povzetek: *Uporaba Jave je že veliko let eden od vodilnih pristopov za razvoj informacijskih rešitev. Java ni le programski jezik, temveč je skupaj z različnimi specifikacijami in rešitvami postala vodilna razvojna platforma, še posebej na strežniški strani. V zadnjem času se je na področju Jave zgodilo veliko sprememb, ki so postavila uporabo Jave na prelomnico, na kateri ni bila še nikoli. Spremembe v upravljanju specifikacij, spremembe v verzioniranju in licenciranju ter številne druge novosti postavljajo razvijalce v situacijo, ko morajo na novo premisliti, na kakšen način bodo uporabljali Javo v bodoče. Za reševanje sodobnih izzivov pogosto samo Java ni več dovolj, temveč je potrebno za doseganje optimalnih rešitev Javo kombinirati z drugimi tehnologijami. Java tudi v bodoče ostaja pomembna razvojna platforma, vendar bo vse skupaj izgledalo precej drugače kot do sedaj.*

Ključne besede: *Java, OpenJDK, programski jezik, platforma, razvoj*

NASLOVI AVTORJEV: mag. Andrej Krajnc, IskraTEL d.o.o., Maribor, Slovenija, e-pošta: a.krajnc@iskratel.si. dr. Ciril Petr, IskraTEL d.o.o., Maribor, Slovenija, e-pošta: petr@iskratel.si. Mitja Skuhala, IskraTEL d.o.o., Maribor, Slovenija, e-pošta: skuhala@iskratel.si. Grega Ramšak, IskraTEL d.o.o., Maribor, Slovenija, e-pošta: ramsak@iskratel.si.

https://doi.org/10.18690/978-961-286-282-4.24 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 UVOD

Živimo v časih, ko se veliko stvari zelo hitro menjava in le redke stvari ostajajo aktualne skozi daljše časovno obdobje. Programski jezik Java je že od samega začetka leta 1995 zelo popularen in že dlje časa najbolj uporabljen programski jezik. Java je še vedno najbolj iskan programski jezik s strani delodajalcev. Da je Java postala tako popularna, je vplivalo več faktorjev [1].

Java je bila prepoznana kot »pravi« objektno orientirani jezik, čeprav so že pred njo obstajali drugi objektno orientirani jeziki kot je npr. Smalltalk. Čeprav so dandanes ponovno na pohodu značilnosti funkcionalnega programiranja, pa so bili veliki tektonski premiki v programiranju narejeni prav preko objektno orientiranih konceptov, ki jih je v praksi najbolj promovirala prav Java.

Že od začetka je bil glavni slogan Jave to, da si program napisal le enkrat in vse skupaj se je potem avtomatsko izvajalo na številnih različnih platformah (»Write once, run anywhere«). To je mogoče, ker javanska platforma temelji na javanskem navideznem stroju (ang. »Java Virtual Machine« oz. JVM). Java je tako dandanes prisotna na najrazličnejših napravah, še posebej je močna na strežniški strani, nekoliko manj pa ji je uspelo na odjemalski strani, kjer lahko opažamo zaton tehnologij, kot so Applet, AWT, Swing, SWT, JavaFX, Java ME itd. Na odjemalski strani so v ospredju spletne aplikacije (temelječe na HTML5, JavaScript itd.), na mobilnih napravah pa prevladujejo mobilne aplikacije, kjer je najpopularnejša platforma Android, ki temelji na Javi. Javanski navidezni stroj se ne uporablja le za programski jezik Java, temveč ga kot izvajalno okolje uporabljajo tudi drugi programski jeziki, kot so Scala, Kotlin, Clojure, Groovy. JRuby, Jython itd. Java je v okviru javanskega navideznega stroja tudi kot prva ponudila avtomatsko upravljanje s pomnilnikom (ang. »Garbage Collection«).

S številnimi optimizacijami je Java skozi leta postala performančno zelo uspešna platforma, ki se lahko marsikje kosa s C/C++ implementacijami. Obstaja veliko različnih implementacij javanskega navideznega stroja (HotSpot, OpenJ9, Zing itd.) in veliko različnih implementacij upravljanja s pomnilnikom (G1, Shenandoah, ZGC, Epsilon itd.), tako da lahko uporabniki izberejo svojim potrebam najustreznejšo varianto. Najnovejši javanski prevajalnik se imenuje Graal, ki prinaša optimizirane performanse z uporabo vnaprejšnjega prevajanja (Ahead Of Time (AOT) Complication).

V praksi je zelo zaživela Java EE (Enterprise Edition), ki prinaša številne koristne programske vmesnike (ang Application Programming Interface – API), kot so vmesnik za strežniška javanska zrna (Enterprise JavaBeans – EJB), vmesnik za vrivanje konteksta in odvisnosti (Contexts and Dependency Injection – CDI), vmesnik za dostop do baze JPA (Java Persistence API), različni vmesniki za spletne storitve (Java API for XML Web Services - JAX-WS, Java API for RESTful Web Services - JAX-RS itd.), vmesnik za sporočilne storitve (Java Message Service - JMS), vmesnik za gradnjo spletnih splikacij (JavaServer Faces - JSF). Vzporedno z Java EE so nastajala številna druga ogrodja, ki olajšajo izgradnjo javanskih aplikacij, verjetno najbolj znano ogrodje je Spring.

Java ni bila nikoli odgovor na vsa vprašanja, zato so se v praksi uveljavili še drugi programski jeziki, ki so na svoj unikaten način določali smer razvoja izgradnje informacijskih rešitev (npr. C#, Go, PHP, JavaScript/Node.JS itd.). Zato ne čudi odločitev arhitektov Jave, da so pričeli v Javo vgrajevati stvari, ki so jih lahko programerji našli v drugih programskih (funkcionalnih) jezikih. Java 8 je naredila velik korak v tej smeri z uvedbo Lambda izrazov (Lambda Expressions). Java 9 se je precej razvijala tudi na drugih področjih, npr. uvedla module (Modules) in naredila prve korake v smeri reaktivnega programiranja (Reactive Programming), Java 10 je naredila dodaten korak k funkcionalnemu programiranju (npr. podpora za »var«), Java 11 je uvedlo novo sintakso za definiranje spremenljivk z Lambda parametri (Local-Variable Syntax for Lambda Parameters) itd.

Okoli Jave se je skozi leta razvijal javanski ekosistem (Java Ecosystem). Sun in Oracle sta kot kreatorja Jave ponudila številna orodja (NetBeans, JDevelopers, VisualVM, WebLogic), ob njima pa so k razvoju Jave doprinesli še številni drugi. Med najpomembnejše sodijo Apache (Commons, Log4J, Maven, Tomcat, Kafka, Mesos, Lucene), Eclipse Foundation (Eclipse, Jetty), Red Hat (WildFly/JBoss, Hibernate, Infinnispan, Thorntail), Pivotal (Spring, RabbitMQ), JetBrains (IntelliJ IDEA), itd.

Četudi je Java stara že skorajda 25 let, je ob nenehnem razvoju jezika in razvoju ekosistema tudi dandanes prepoznana kot konkurenčna rešitev za sodobna okolja, kjer vse bolj prevladujejo mikroservisi

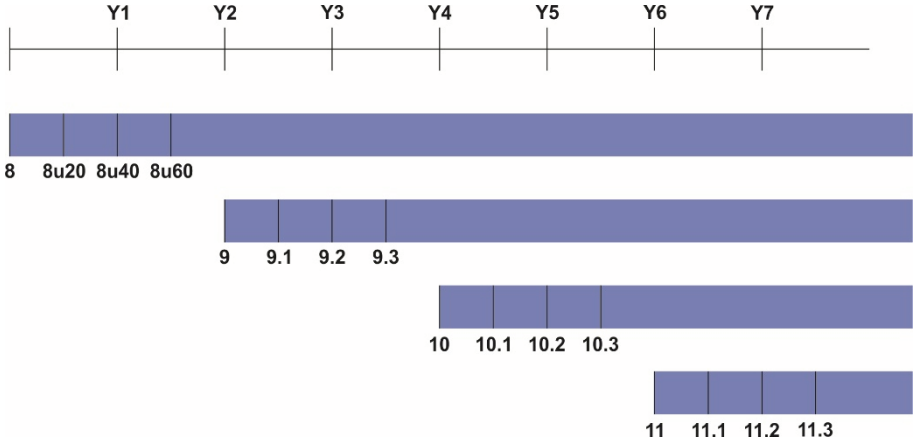
(Mikroservices), vsebniki (Containers), računalništvo v oblaku (Cloud Computing), internet stvari (Internet of Things – IoT), reaktivno programiranje (Reactive Programming) in strojno učenje (Machine Learning).

2 SPREMEMBE V JAVI

V zadnjem času je Java doživela veliko sprememb [2, 3]. Gre predvsem za spremembe v verzioniranju in podpori verzijam, spremembe pri licenciranju Jave, spremembe v upravljanju specifikacij Jave ter številne druge novosti. Sprememb v Javi je v zadnjem času res veliko in tako lahko rečemo, da je sprememb v zadnjih 2 letih več kot jih je bilo prej v 24 letih.

2.1 Spremembe v verzioniranju Jave

Dolga leta je imela Java vzpostavljen sistem upravljanja verzij, kjer so nove verzija Jave prihajale na nekaj let. V povprečju je prišla nova verzija Jave na približno vsake 2-3 leta (JDK Beta 1995, JDK 1.0 leta 1996, JDK 1.1 leta 1997, J2SE 1.2 leta 1998, J2SE 1.3 leta 2000, J2SE 1.4 leta 2002, J2SE 5.0 leta 2004, Java SE 6 leta 2006, Java SE 7 leta 2011, Java SE 8 leta 2014, Java SE 9 leta 2017 itd.). Po eni strani je bilo izdajanje nove verzije na nekaj let dobra zadeva, saj je prišla ven stabilna Java, po drugi strani pa so se pojavljali očitki, da vse skupaj traja prepočasi.

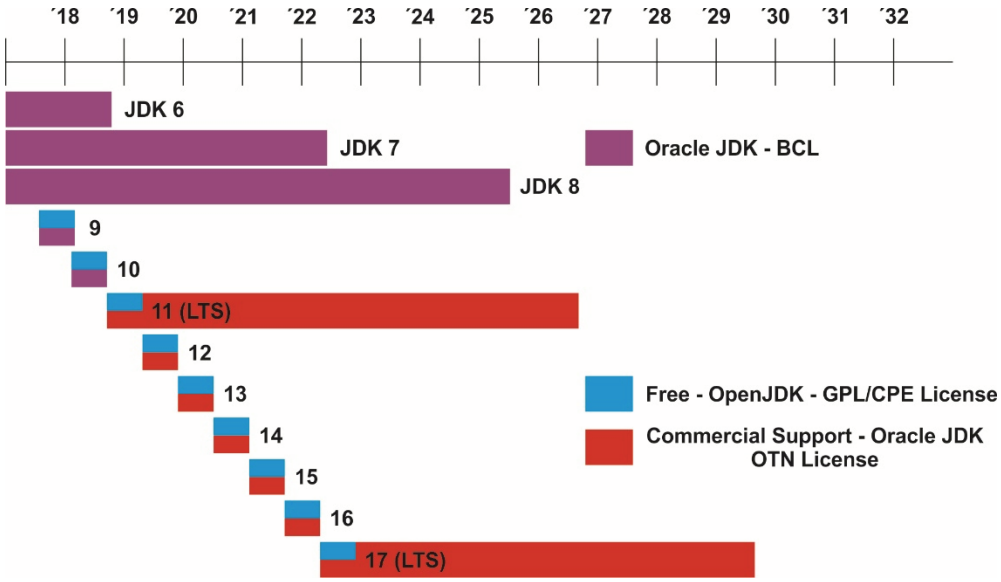


Slika 1: Staro verzioniranje Jave [8]

Septembra leta 2017 je glavni arhitekt za javansko platformo Mark Reinhold napovedal, da se bodo v bodoče nove verzije Jave izdajale vsakih 6 mesecev, tipično marca in septembra vsak leto. Eden glavnih argumentov za hitrejše izdajanje novih verzij je, da bo na ta način Java ostala konkurenčna, saj tudi drugi ponujajo nove funkcionalnosti veliko prej.

Živimo v času, ko razvoj programske opreme temelji na nenehni integraciji (Continuous Integration - CI), nenehni dostavi (continuous delivery - CD), avtomatiziranih testih (Automated Tests), vsebnikih (Containers), virtualizaciji (Virtualization) itd. Tako lahko razvijalci dandanes hitreje kot kadarkoli prej svoje rešitve preverijo na različnih verzijah Jave v vseh fazah razvoja (od prevajanja programske kode do izvajalnega okolja).

Tako so po novem zadnje verzije izhajale na 6 mesecev. Java SE 10 je prišla marca 2018, Java SE 11 je prišla septembra 2018, marca 2019 pa smo dobili Java SE 12, izdaja Java SE 13 je planirana za september 2019 itd. Vnaprej znani predvidljivi datumi izdaj novih verzij Jave omogočajo podjetjem, da temu primerno planirajo svoje izdaje produktov, da bodo prehodi na nove verzije Jave lažji. Nove izdaje vsebujejo tudi manj sprememb kot prejšnje velike izdaje, tako da prehod na novo verzijo Jave predstavlja tudi manjši riziko. V Javo 8 je bilo vključenih več 50 razširitev (Java Enhancement Proposal – JEP), v Javi 9 je bilo celo čez 80 razširitev, v novejših verzijah Jave je takšnih razširitev precej manj, nekje med 5 in 20 na izdajo.



Slika 2: Novo verzioniranje Jave [9]

Bo pa pogosto prehajanje na nove verzije Jave vseeno predstavljal precejšnji izziv za podjetja, ki doslej večinoma niso tako redno in hitro prehajala na nove verzije Jave. Praksa kaže, da je veliko podjetij preskočilo določene verzije Jave in da je še dandanes najbolj pogosto uporabljena verzija Jave 8, čeprav imamo zunaj že verzijo 12.

2.2 Spremembe v podpori verzijam

Veliko sprememb je tudi na področju podpore verzijam Jave [4]. Oracle se je odločil, da ne bo več podpiral vsake verzije Jave kot smo to poznali vsa ta leta. Vsaka verzija je bila še nekaj let po izdaji deležna posodobitev, tako je nastalo znotraj neke verzije veliko izdaj s temi posodobitvami. Pri izdaji Jave na vsakih 6 mesecev je to prevelik zalogaj za Oracle. Če bi izdajali novo verzijo Jave na vsake 6 mesecev in bi vsako morali podpirat 3 leta, bi to pomenilo, da bi Oracle moral vzdrževati 6 verzij Jave, kar je seveda preveč, zato so precej spremenili podporo v novih verzijah Jave.

Večina verzij bo imela le kratko šestmesečno podporo, kar pomeni, da po izdaji naslednje verzije (po 6 mesecih) s strani Oracla ne bo več na voljo posodobitev za to starejšo verzijo. Oracle bo dolgotrajno podporo (Long Time Support - LTS) ponudil le za določene verzije Jave. Že sedaj ima dolgotrajno podporo verzija 8, zagotovo bo takšno dolgotrajno podporo imela verzija 11, naslednja verzija z dolgotrajno podporo pa bo verjetno šele verzija 17. Oracle bo dolgotrajno podporo vseeno ponujal le določen čas, verjetno bo še daljša dolgotrajna podpora prišla s strani drugih javanskih distribucij.

Podpora verzijam Jave je pomembna, saj se v Javo vključujejo številne pomembne posodobitve, še posebej so pomembne varnostne posodobitve. Java je imela v zadnjim letih kar nekaj varnostnih pomanjkljivosti, zaradi česar jo vedno bolj omejujejo pri uporabi na odjemalskih računalnikih (ukinitev v brskalniku, vedno večje omejitve pri namizni verziji Jave itd.), pa tudi na strežniških računalnikih so posodobitve zelo pomembne, saj se Java ne uporablja več toliko v zaprtih okoljih brez dostopa od zunaj, temveč je velikokrat prisotna v oblačnem okolju skupaj z drugimi aplikacijami in v takšnih okoljih sta varnost ter posodobljena Java še posebej pomembna.

Trenutno se zdi, da bo največ uporabnikov uporabljala verzije Jave z dolgotrajno podporo. Torej sedaj jih je največ na Javi 8, nato bodo prešli na verzijo 11 in potem na naslednjo dolgotrajno verzijo (verjetno Java 17). Vmesne verzije Jave se bodo v produkciji uporabljale le izjemoma, bo pa večina preverjala svoje projekte interno tudi s temi vmesnimi verzijami Jave (prevajanje in avtomatski testi na internih strežnikih). Na ta način bodo podjetja zmanjšala rizike ob prehodu na novo verzijo Jave z dolgotrajno podporo.

2.3 Spremembe v licenciranju

Že od začetka je za Javo veljalo, da so bile verzije Jave s posodobitvami brezplačno podprte le določeno obdobje (tipično najmanj 3 leta), za tem pa so imeli še vedno možnost uporabe plačljive podpore. Oracle (in pred tem že Sun) sta to urejala z licencami, ki pa so se v zadnjem času precej spremenile v »škodo« uporabnikom.

Oracle je sedaj veliko bolj zaostril pogoje uporabe Jave na dolgi rok. Na kratek rok lahko uporabljamo Javo brezplačno, za dolgotrajno podporo pa bo potrebno Oraclu plačevati. Če ne želimo plačevati Oraclu na dolgi rok, moramo vsakih 6 mesecev nadgraditi Javo na novo verzijo. To je velika novost za podjetja, ki doslej niso bila vajena tega, da jih nekdo sili v uporabo novih verzij Jave. Doslej je večina uporabljala Oraclovo Javo ne da bi jih kdo silil v nadgradnjo na nove verzije Jave. Razvijalci smo doslej bolj ali manj v vseh primerih enostavno šli na Oraclovo stran, prenesli novo verzijo Jave in jo uporabljali brez nekih zadržkov in skrbi. To se je sedaj s spremembami v licenciranju precej spremenilo.

Zaradi teh sprememb v licenciranju bo verjetno marsikdo opustil dosedanjo prakso uporabe standardnega Oraclovega JDK in se bo začel ozirati po odprtokodnih implementacijah.

Že sam Oracle ponuja Javo 11 pod dvema različnima licencama: obstaja plačljiva komercialna licenca in GPLv2+CPE odprtokodna licenca, ki nadomešča staro BCL (Binary Code License) licenco. Tako imamo sedaj dve Oraclovi distribuciji: plačljivi Oracle JDK in odprtokodni Oracle OpenJDK. Plačljivi JDK je možno zastonj uporabljati za razvoj in testiranje, v produkciji pa lahko le pod komercialnimi (plačljivimi) pogoji. Oraclov JDK se plačuje 25 dolarjev (lahko je tudi manj) na mesec na procesor na strežniški strani, na namizni strani pa se plačuje 2,5 dolarja na mesec za vsakega uporabnika.

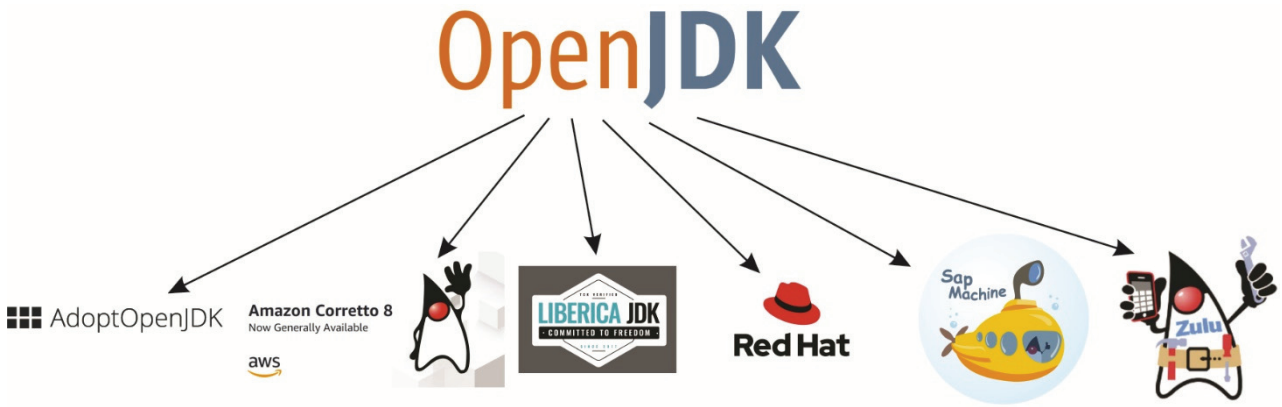
Do Jave 11 Oraclov JDK in OpenJDK nista bila nujno čisto enaka, od Jave 11 naprej pa sta Oracle JDK in Oracle OpenJDK zgrajena na isti osnovi in sta v osnovi identična prvih 6 mesecev. Razlika nastane po 6 mesecih, ko postane Oracle JDK plačljiv, Oracle OpenJDK pa se po 6 mesecih preneha posodabljat, lahko pa ga nadgrajuje kdo drug.

Oraclove nove spremembe pri licenciranju imajo tudi pozitivno stran. Oracle je večino svojih stvari ponudil v odprtokodno skupnost, med drugim tudi orodja, kot so Java Flight Recorder, Java Mission Control, ki doslej niso bila odprtokodna in tudi zaradi tega niso bila toliko uporabljena kot bodo verjetno v bodoče. Vse te spremembe ponujajo veliko možnosti pri razvoju odprtokodnih implementacij Jave in dejansko lahko v praksi opažamo veliko aktivnosti in konkurenčnih dejavnosti prav s strani odprtokodnih OpenJDK ponudnikov, kot so AdoptOpenJDK, Amazon, Azul. Bellsoft, IBM, jClarity, Red Hat, Linux distros itd. Na voljo je precej OpenJDK distribucij, večinoma so brezplačne in ponujajo posodobitve za daljša obdobja kot ponuja Oracle (podpora za dolgotrajne verzije bi naj bila vsaj 4 leta). Za razliko od Oraclove podpore je pri drugih distribucijah veliko bolj vpletena javanska skupnost, ki bo skrbela za posodobitve. Izgleda, da se bo večina podjetij namesto za Oraclovo implementacijo odločila za katero od drugih distribucij.

2.4 OpenJDK distribucije

OpenJDK distribucije je možno pridobiti na različne načine. Eden od načinov je, da si sami izgradimo OpenJDK implementacijo, pri čemer si lahko izvorno kodo prenesemo iz spletnih strani:

- Mercurial (http://hg.openjdk.java.net/),
- Tarballs (7+): (https://openjdk-sources.osci.io/) in
- AdoptOpenJDK (https://www.github.com/AdoptOpenJDK/openjdk-build).



Slika 3: OpenJDK distribucije

Lahko pa si prenesemo že izgrajeno binarno distribucijo od različnih ponudnikov

- AdoptOpenJDK
  - ponuja verzije 8 in 11 s posodobitvami najmanj do septembra 2023, podpira vse glavne arhitekture (Linux x85, Mac OS X, Windows x64) in različne druge platforme, komercialno podporo ponujata IBM in JClarify,
  - AdoptOpenJDK podpirajo Amazon, Azul, GoDaddy, IBM, jClarity, Pivotal, Red Hat in mnogi drugi,
  - distribucije vsebujejo Hotspot in Eclipse OpenJ9 navidezni stroj,
- Amazon Corretto
  - ponuja verzije 8 in 11 s posodobitvami najmanj do junija 2023, podpira vse glavne arhitekture (Linux x85, Mac OS X, Windows x64),
  - Amazon uporablja Correto na tisočih produkcijskih AWS strežnikih
- Azul Zulu
  - ponuja verzije 8 in 11, podpira vse glavne arhitekture (Linux x85, Mac OS X, Windows x64) in različne druge platforme, komercialno podporo ponuja Azul,
  - Azul ponuja srednjetrajno podporo (Medium Term Support – MTS) za Zulu JDK
  - za razliko od Oracla, ki komercialno podporo zaračunava za procesor, pa Azul komercialno podporo zaračunava za število sistemov (sistem je fizični ali virtualni strežnik),
- BellSoft Liberica JDK
  - ponuja verzije 8 in 11 s posodobitvami najmanj do leta 2023, podpira vse glavne arhitekture (Linux x85, Mac OS X, Windows x64) in različne druge platforme, komercialno podporo ponuja BellSoft,
  - cenovni model za podporo temelji na številu namiznih računalnikov, številu fizičnih in virtualnih strežnikov,
- Oracle OpenJDK
  - ponuja verzijo 11 s posodobitvami do marca 2019, podpira vse glavne arhitekture (Linux x85, Mac OS X, Windows x64), komercialno podporo ponuja Oracle,
- SapMachine
  - ponuja verzijo 11 podpira vse glavne arhitekture (Linux x85, Mac OS X, Windows x64),
- Red Hat OpenJDK
  - Red Hat ponuja OpenJDK za plaforme, na katerih teče Red Hat Enterprise Linux,
  - Red Hat ponuja OpeJDK RPMje za CentOS.

2.5 Druge spremembe v Javi

Na področju Jave so se zgodile številne druge spremembe. Med najzanimivejše sodi to, da je Oracle svoje razvojno orodje NetBeans predal organizaciji Apache. Prav tako je svojo glavno konferenco za Javo preimenoval iz JavaOne v CodeOne.

3 JAVA 8 ALI PREHOD NA JAVO 11

Podjetja so se znašla v precejšnji dilemi, kaj narediti glede uporabe verzij Jave. Večina podjetij uporablja Javo 8 in razmišlja o tem, ali naj ostane na tej verziji ali naj preide na novo verzijo, najverjetneje na verzijo 11 [5].

3.1 Nadaljnja uporaba Java 8

Če se bodo podjetja odločila ostati na verziji 8, morajo upoštevati nekatera dejstva.

Aprila 2019 bila izdana zadnja verzija Jave 8 po starih pogojih, nove verzije Jave 8 so na voljo le po komercialnih/plačljivih pogojih. Vse do decembra 2020 bo Oracle JDK na voljo zastonj za osebno namizno uporabo, za razvoj, testiranje, prototipiranje, demonstriranje in za uporabo z določenimi tipi aplikacij.

Po decembru 2020 se bodo morali uporabniki odločiti bodisi za plačljivi Java SE 8 ali pa OpenJDK 8 implementacijo enega od ponudnikov.

Če podjetja ne uporabljajo Oracle JDK 8, temveč neko drugo odprtokodno implementacijo Jave 8, za katero stojijo različni ponudniki (AdoptOpenJDK, Amazon, Azul. Bellsoft, IBM, jClarity, Red Hat itd.).

Posodobitve za OpenJDK 8 bodo na voljo do septembra 2023, razvoj OpenJDK 8 koordinira Red Hat, vse skupaj pa podpirajo zgoraj omenjeni ponudniki distribucij. Oracle na drugi strani ponuja plačljivo podporo za verzijo 8 vse do leta 2025. Plačljivo podporo za Java 11 ponujajo tudi ponudniki odprtokodnih implementacij.

3.2 Prehod na Javo 11

Če se podjetje odloči za prehod na Javo 11, lahko izbere plačljiv Oracle JDK 11, Oracle OpenJDK ali katero od omenjenih OpenJDK distribucij.

Posodobitve za OpenJDK 11 bodo na voljo do septembra 2025, razvoj OpenJDK 11 koordinira Red Hat, vse skupaj pa podpirajo omenjeni zgoraj ponudniki distribucij. Oracle na drugi strani ponuja plačljivo podporo za verzijo 11 vse do leta 2026. Plačljivo podporo za Java 11 ponujajo tudi ponudniki odprtokodnih implementacij.

S prehodom na Javo 11 je kar nekaj sprememb, katerih se moramo zavedati:

- v Javi 11 ni več JavaFX knjižnic, ki so sedaj ločene odprtokodne SDK komponente, za razvoj JavaFX pa skrbi OpenJFX,
- v Javi 11 ni več komponente javapackager za združevanje aplikacij in njihovih odvisnosti,
- v Javi 11 ni več Java Web Start, zato morajo tisti, ki uporabljajo Javo na odjemalcu preko protokola JNLP, začeti uporabljati alternative, kot sta npr. IcedTea-Web ali Karakun/OpenWebStart,
- v Javi 11 je umaknjena tudi podpora za protokol CORBA,
- v Javi 11 ni več implementacij za JAXB in JAX-WS, ki morata biti dodani kot zunanji knjižnici,
- že od Jave 9 naprej nimamo več Visual VM.

Obstaja kar nekaj orodij, ki nam olajšajo prehod na verzijo 11. Tako na primer obstaja jdeps za analizo odvisnosti ter orodje jdeprscan za statično analizo uporabe opuščenih (deprecated) stvari.

Java 11 ima kar nekaj novosti, ki so lahko že same po sebi razlog za prehod na verzijo 11, tu je navedenih le nekaj od njih:

- v Javo 11 je vgrajenih kar nekaj varnostnih posodobitev, npr. podpora za novo verzijo TLS 1.3,
    - vsako leto je vedno več ranljivosti v javanskih komponentah (predvsem odprtokodnih), zaradi česar je velika potreba po tem, da so že v samo Javo vgrajene najnovejše varnostne posodobitve,

- v Javo 11 je vgrajen nov HTTP odjemalec,
- Java 11 prinaša podporo za direkten zagon izvornih datotek z interpretorjem java (JEP 330 - Launch Single-File Source-Code Programs),
- v Javi 10 je bilo uvedeno deljenje aplikacijskih razredov v deljenih arhivih za različne javanske procese(Application Class-Data Sharing), zaradi česar je lahko pomnilniška zasedba (Memory Fooprint) manjša in zagon javanskih aplikacij hitrejši,
- od Jave 10 naprej je možno uporabljati spremenljivko var,
- Java 10 prinaša Graal VM in vnaprejšnje prevajanje (Ahead Of Time (AOT) Complication).
- v Javo 9 so bili dodani novi vmesniki za obdelavo tokov (Streams),
- Java 9 je prinesla nove tovarniške (factory) metode za kolekcije,
- Java 9 prinaša module,
- že od Jave 9 je na voljo interaktivno orodje za učenje in prototipiranje JShell, ki deluje na način REPL (Read-Evaluate-Print Loop), kjer se vpisane komande takoj evaluirajo, ob tem pa se takoj prikaže rezultat,
- na področju upravljanja pomnilnika (Garbage Collection) je Java 9 privzeto začela uporabljati G1 GC, Java 10 je prinesla paralelizem (Parallel Full GC for G1), Java 11 pa je pripeljala dva nova GC: Epsilon in ZGC.

Prehod na Javo 11 bo predstavljal tudi lažji kasnejši prehod na verzijo 12, ki med drugim prinaša stikalne izraze (switch expressions), mikro primerjalne meritve (microbenchmark), programski vmesnik za JVM konstante (JVM constants API), novosti so tudi pri upravljanju pomnilnika. Med projekte, ki so zanimivi za bodočnost Jave, sodijo tudi projekti Panama, Amber, Valhalla, Loom, v okviru katerih bodo v bodoče definirane številne zanimive novosti v Javi.

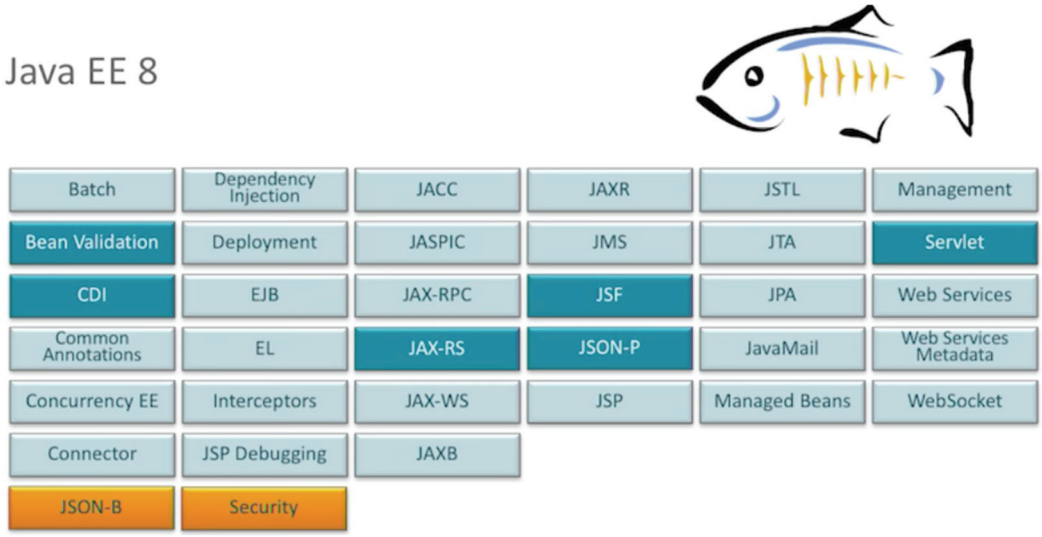
Nekateri menijo, da je eden od argumentov za prehod na najnovejšo verzijo Jave, naslednji: okolje, kjer se uporablja najnovejše verzije Jave, je lahko veliko bolj atraktivno okolje za programerje pri izbiri zaposlitve. Tista podjetja, ki bodo ostala na starejših verzijah Jave, bodo lahko smatrana kot podjetja, kjer se uporabljajo zastarele tehnologije. Včasih lahko takšne malenkosti odločajo pri tem, za katero ponudbo se bodo programerji odločili.

Največkrat se predlaga, da se migracija na Javo 11 izvede po naslednjih korakih:

- Javo 11 začnemo uporabljati za zagon obstoječe programske opreme
- posodobimo odvisnosti v programski opremi (starejše komponente morda ne delujejo na novi Javi, po potrebi dodamo nove odvisnosti, npr. JAXB, JAX-WS)
- posodobimo orodja za izgradnjo programske opreme (starejše verzije Maven in Gradle ne delujejo na Java 9)
- analiziramo opozorila izvajalnega okolja (zastarele/umaknjene GC komande, napačne uporabe itd.),
- zaženemo teste na Javi 11,
- programsko kodo prevedemo z Javo 11.

4 SPREMEMBE V JAVI EE

Od vseh področij je Java najbolj uspela na strežniški stran, še posebej je v praksi zaživela Java EE (Enterprise Edition), ki je uporabljena v številnih velikih projektih. Java EE vsebuje številne uporabne specifikacije za delo z bazami, spletnimi storitvami, spletnim uporabniškim vmesnikom itd. Razvijalcem so na voljo številne komponente in druge rešitve, ki temeljijo na Java EE specifikacijah. Precejkrat gre za komercialne/plačljive rešitve, še veliko več pa je odprtokodnih rešitev.



Slika 4: Komponente v specifikaciji Java EE 8 [10]

Na začetku se Java EE še ni imenovala tako, temveč je prva verzija pod imenom J2EE 1.2 nastala leta 1999. Leta 2001 je izšla J2EE 1.3, leta 2003 verzija J2EE 1.4, leta 2005 pa prvič pod imenom Java EE verzija 1.5. Java EE 6 je luč sveta zagledala leta 2009, Java EE 7 je bila izdana leta 2013, zadnja verzija pa je Java EE 8, ki je bila izdana 31.8.2017.

Implementacije Java EE ponujajo različni aplikacijski strežniki, kot so npr. WildFly (prej imenovan JBoss), IBM Websphere, Oracle WebLogic, GlassFish, OpenLiberty itd. Razlike med njimi v implementaciji Java EE so tako majhne, da so naše aplikacije v primeru uporabe standardnih Java EE komponent večinoma relativno enostavno prenosljive na druge aplikacijske strežnike.

12. septembra 2017 je Oracle najavil, da ne bo več skrbel v okviru svojega procesa JCP (Java Community Process), temveč bo Javo EE predal v upravljanje organizaciji Eclipse Foundation, ki skrbi za številne odprtokodne projekte. V okviru tega je nastal krovni projekt za Javo, ki so ga poimenovali Eclipse Enterprise for Java (EE4J). Oracle je zahteval preimenovanje Java EE, saj je želel ohraniti blagovno znamko Java. Tako so februarja 2018 sporočili, da bo novo ime za Java EE v bodoče Jakarta EE [6]. Podobno se tudi aplikacijski strežnik GlassFish, ki je tudi bil predan Eclipse Foundation, sedaj imenuje Eclipse GlassFish. Vlogo JCP pa je prevzela skupina Jakarta EE Working Group. Da je bil izbran Eclipse Foundation, ne čudi, saj gre za organizacijo, ki je že prej precej delovala na področju Java EE. Med drugim ponuja orodje Eclipse za razvoj EE aplikacije, v okviru Eclipse Foundation se razvija specifikacija za razvoj mikroservisov MicroProfile itd. Ustanovitelj fundacije Eclipse in še vedno glavnih podpornikov je IBM, ki je pred kratkim kupil Red Hat, ki pa je tudi eden najpomembnejših igralcev na področju Java EE. Eclipse Foundation veliko vlaga v »Cloud Native Java« in glavni projekti na tem področju so Eclipse IDE, Eclipse OpenJ9, Eclipse Vert.x, Eclipse Jemo, Eclipse Theia in Eclipse Che.

Slika 5: Logo Jakarta EE [11]

Čeprav je od Oraclove najave minilo že več kot leto in pol, žal še vedno ni pravega napredka. V zadnjem času je celo nasprotno, saj se pojavljajo novice, da sta Oracle in Eclipse Foundation v pogajanjih na več mestih na nasprotnih bregovih [7]. Oracle želi namreč v bodoče certificirati rešitve, kjer se bo uporabljala Jakarta EE, čemur pa pri neodvisni fundaciji Eclipse naprotujejo, saj bi Oracle na ta način kontroliral razvoj rešitev fundacije Eclipse.

Skorajda zagotovo bomo kmalu (predvidoma jeseni 2019) dočakali Jakarta EE 8, ki je v bistvu kopija Java EE 8, torej je kompatibilna z Java EE 8 specifikacijami, vsebuje enake programske vmesnike, uporablja isti imenski prostor (namespace) javax itd. Zaradi tega bodo vse obstoječe aplikacije, ki so bile kompatibilne z Java EE 8, istočasno kompatibilne tudi z Jakarta EE 8.

Veliko večja dilema pa je, kaj bo z Jakarta EE 8 v bodoče. Vse dokler Oracle in Eclipse Foundation ne dosežeta dogovora, se v Jakarta EE 8 brez Oraclovega soglasja ne sme spremeniti nič. Brez Oraclovega soglasja se nikjer v Jakarta EE ne sme uporabljati blagovna znamka Java. Še veliko večjo težavo pa predstavlja dejstvo, da Jakarta EE brez dogovora v spremenjenih verzijah Jakarta EE ne sme uporabljati imenskega prostora javax.

Če bo dogovor v kratkem vendarle sklenjen, si bodo vsi v svetu Jave oddahnili. Če pa dogovora ne bo, se bo skupnost, ki uporablja Java/Jakarta EE, znašla v veliki dilemi, kako razvijati EE v bodoče. Težko je pričakovati, da bo na lahek način zaživela takšna dopolnjena verzija Jakarta EE 8, v kateri bo spremenjen imenski prostor javax v nekaj drugega. Temu se želijo izogniti vsi, ki razvijajo rešitve na EE osnovi in bi morali predelati svoje rešitve, da bodo kompatibilne z novo spremenjeno verzijo Jakarta EE 8 z novim imenskim prostorom. Večina si želi, da bi tudi v bodoče ohranili to, da se program napiše le enkrat (»Write once, run anywhere«) in izvaja na različnih mestih brez dopolnitev.

Precej manj težav bodo imeli tisti, ki namesto Java EE uporabljajo katero od alternativnih rešitev. Daleč najbolj popularna alternativa za Java EE je ogrodje Spring, ki je v zadnjem času postalo vodilno orodje za razvoj strežniških aplikacij. Ogrodje Spring je že sedaj bil prva izbira za mnoge razvijalce, morebitno nadaljevanje težav na področju Java EE/Jakarta EE pa zna samo še povečati popularnost ogrodja Spring. Ogrodje Spring je zelo popularno že sedaj kljub dejstvu, da za njim stoji le eno podjetje (Pivotal). Orodje je veliko bolj enostavno za uporabo in v marsičem ponuja boljše rešitve kot klasična Java EE.

5 ZAKLJUČEK

Java platforma je že od samega začetka ena od najpopularnejših izbir za izgradnjo informacijskih rešitev. Tako ne čudi, da tudi mnogo internetnih velikanov, kot so Amazon, LinkedIn, Google, Facebook in mnogi drugi pri razvoju svojih rešitev temeljijo na uporabi Jave. Java se je skozi leta izkazala kot performančno učinkovita rešitev tudi za velike količine podatkov in ima vse osnove, da bo ostala popularna tudi v bodoče v sodobnih okoljih.

Spremembe v upravljanju specifikacij, spremembe v verzioniranju in licenciranju ter številne druge novosti postavljajo razvijalce v situacijo, ko morajo na novo premisliti, na kakšen način bodo uporabljali Javo v bodoče. Najverjetneje bo večina opustila doslej uporabljen Oracle JDK in začela uporabljati eno od OpenJDK distribucij.

Verjetno največ nejasnosti je na področju Java EE. Vsi razvijalci lahko upamo, da se bo našla rešitev, ki bo omogočala razvijalcem, da bodo tudi v bodoče uporabljali vse novosti v Jakarta EE na podoben način kot smo to bili vajeni v Java EE.

6 LITERATURA

[1] The future for Java, https://archbee.io/blog/the-future-for-java/, obiskano 17. 5. 2019.
[2] Trisha Gee, Life Beyond Java 8, https://www.infoq.com/presentations/java-8-plus, obiskano 17. 5. 2019.
[3] Java: New Developments and Features, https://dzone.com/guides/java-new-developments-and-features, obiskano 17. 5. 2019.
[4] Java is Still Free 2.0.3, https://medium.com/@javachampions/java-is-still-free-2-0-0-6b9aa8d6d244, obiskano 17. 5. 2019.
[5] It's time! Migrating to Java 11, https://medium.com/criciumadev/its-time-migrating-to-java-11-5eb3868354f9, obiskano 17. 5. 2019.
[6] Jakarta EE, https://jakarta.ee/, obiskano 17. 5. 2019.
[7] Negotiations Failed: How Oracle killed Java EE, https://headcrashing.wordpress.com/2019/05/03/negotiations-failed-how-oracle-killed-java-ee/, obiskano 17. 5. 2019.
[8] Dalibor Topić, Fast Forward to JDK 11, Make IT Konferenca, Portorož, 15.10.2018,
[9] Dalibor Topić, JDK Updates and OpenJDK, Make IT Konferenca, Portorož, 15.10.2018,
[10] Yolande Poirier, Java EE 8 Overview, https://blogs.oracle.com/java/java-ee-8-overview, obiskano 17. 5. 2019.
[11] Roxanne Joncas, A First Look at Jakarta EE, https://www.eclipse.org/community/eclipse_newsletter/2018/may/, obiskano 17. 5. 2019.

OTS 2019
SODOBNE INFORMACIJSKE TEHNOLOGIJE IN STORITVE
M. Heričko in K. Kous (ur.)

Univerzitetna založba
Univerze v Mariboru

RAZVOJ SPLETNIH APLIKACIJ V OGRODJU YESOD

DUŠAN FISTER, IZTOK FISTER JR.

Povzetek: *Razvita ogrodja za razvoj spletnih aplikacij lahko domala najdemo za vsak programski jezik. Dandanes prevladujejo ogrodja v programskih jezikih Python, Java, Go, PHP, Ruby. Yesod je spletno ogrodje, razvito v programskem jeziku Haskell, namenjeno za razvoj tako enostavnih kot tudi zahtevnih spletnih aplikacij. Glavni značilnosti, ki opisujeta to ogrodje sta varnost in zmogljivost delovanja. V tem prispevku kratko opisujemo programski jezik Haskell in z nekaterimi primerki programske kode predstavljamo ogrodje Yesod.*

Ključne besede: *Haskell, Yesod, funkcijski programski jeziki, spletno programiranje, varnost*

NASLOVA AVTORJEV: Dušan Fister, mladi raziskovalec, Univerza v Mariboru, Ekonomsko-poslovna fakulteta, Maribor, Slovenija, e-pošta: dusan.fister1@um.si. doc. dr. Iztok Fister Jr., asistent, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Maribor, Slovenija, e-pošta: iztok.fister1@um.si.

https://doi.org/10.18690/978-961-286-282-4.25 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 UVOD

Zgodovina računalništva prinaša mnogo različnih načinov programiranja, s tem pa tudi kopico najrazličnejših programskih jezikov. Programiranje je eden izmed najosnovnejših načinov komunikacije med človekom in računalnikom, kjer človek z vpisovanjem ukazov v različnih oblikah neposredno usmerja delovanje računalnika. Obstaja veliko načinov programiranja, med katerimi so najbolj pogosti imperativni, objektni in funkcijski.

Eden izmed razširjenih funkcijskih jezikov je programski jezik Haskell. Tega pogosto srečamo ne le v akademskih krogih, temveč tudi v industriji, kjer se njegova priljubljenost, sodeč po zadnjem trendu, močno povečuje. Zaradi zagotavljanja visoke stopnje varnosti in hitrosti izvajanja aplikacij, je Haskell priljubljen tudi pri razvijalcih spletnih aplikacij. Njegov prevajalnik namreč zagotavlja zavidljivo zmogljivost izvajanja programske kode, podobno kot programski jezik C, kakor tudi lastnost, da uporabnik morebitne napake v programski kodi opazi takoj med postopkom prevajanja, namesto da na njih naleti pri kasnejšem delovanju (angl. runtime bugs). Vse kakršne tovrstne napake utegnejo povzročati neželene motnje v delovanju ter tako dovoljevati varnostne luknje [4].

Programski jezik Haskell praktično srečamo pri razvoju visoko-zmogljivih aplikacij v finančni industriji [10], računalništvu v oblaku [8] ter uporabnih aplikacijah umetne inteligence [9], ipd. Nenazadnje srečamo Haskell tudi pri razvoju vgrajenih domensko specifičnih jezikov (angl. embedded domain-specific languages, krajše EDSL [5]).

Obstaja več spletnih ogrodij, razvitih v programskem jeziku Haskell, kot npr. Happstack, Snap, Scotty, Servant in Yesod. Izmed naštetih je v komercialnih aplikacijah najširše zastopan ravno Yesod. Za tega namreč obstaja obilo programskih knjižnic, ki poenostavljajo delo s programiranjem, dodatno pa je slednji že dlje časa izpostavljen ustaljenemu razvoju. K temu priča tudi mnogo zastavljenih vprašanj in odgovorov uporabnikov tega ogrodja na platformi StackOverflow.

V tem prispevku na kratko predstavljamo programski jezik Haskell, navajamo njegove prednosti glede na ostale programske jezike ter prikazujemo njegovo uporabno vrednost. Izpostavljamo bistvene značilnosti programskega jezika Haskell ter demonstriramo delovanje. Podrobneje predstavljamo in opisujemo spletno ogrodje Yesod in nekaj njegovih osnovnih primerov.

Struktura tega prispevka je sledeča: v poglavju 2 je na kratko predstavljen programski jezik Haskell, medtem ko je v poglavju 3 opisano ogrodje Yesod za razvoj spletnih strani. Navedenih je nekaj praktičnih primerov uporabe ogrodja Yesod. Postavitev (angl. deployment) aplikacije je prikazana v poglavju 4, v poglavju 5 pa podan povzetek prispevka.

2 PROGRAMSKI JEZIK HASKELL

Programski jezik Haskell spada med čiste funkcijske jezike (angl. purely functional languages), katerega začetek razvoja sega v osemdeseta leta prejšnjega stoletja. Tehnično gledano je programski jezik Haskell [1]:

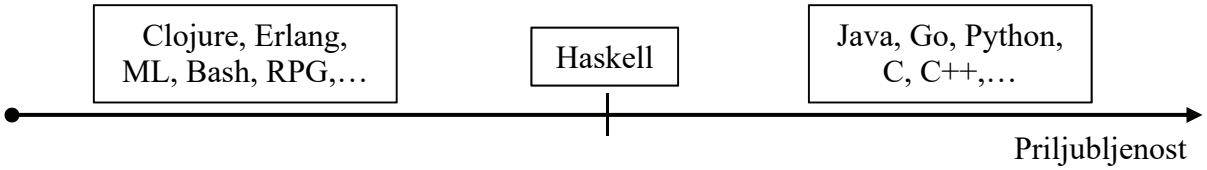
- močno in statično tipiziran jezik (angl. statically typed), kar pomeni da prevajalnik samodejno zaznava tip spremenljivk, npr. celo število, realno število, znak [16],
- uporablja zakasnjeni izračun (angl. lazy evaluation), kar pomeni da Haskell ne izračunava rezultatov, dokler ni prisiljen. Praktično to pomeni da se z definicijo funkcije ta ne izvede takoj, ampak šele dokler aplikacija ne potrebuje rezultatov te funkcije,
- eleganten in zgoščen jezik (angl. elegant and concise), ker visoko-nivojske koncepte opisuje smiselno, kratko ter jedrnato.

Funkcijski jeziki v splošnem zahtevajo popolnoma drugačno razmišljanje, kot npr. pri uporabi objektno orientiranih jezikov. Kakopak je zaradi tega drugačna tudi sintaksa jezika. V tabeli 1 je predstavljena osebna izkaznica programskega jezika Haskell [7,13].

Tabela 1. Osebna izkaznica programskega jezika Haskell [7,13]

Nastanek jezika	1990
Prevajalniki	GHC (Glasgow Haskell Compiler), Hugs
Nastal po vzoru sledečih programskih jezikov	Miranda, Lisp, Orwell (med drugimi)
Vplival na razvoj sledečih programskih jezikov	C++ 11, Python, Clojure, Elm, Rust, Scala, Swift (med drugimi)
Operacijski sistem	Deluje na vseh platformah
Končnice datotek	.hs, .hls
Github repozitorij	https://github.com/haskell
Aplikacije, ki uporabljajo Haskell [12]	xmonad, Monadius, Flippi, AMuZeD, Paradox, LPS, Knit, Truth, HWS-WP, Yi, Hoogle, EDSL

Priljubljenost programskega jezika dokazuje lestvica »Tiobe Index«, ki trenutno Haskell uvršča na 45. mesto med vsemi programskimi jeziki. Slika 1 prikazuje izvleček Tiobe lestvice, kamor umeščamo programski jezik Haskell.



Slika 1. Delna Tiobe lestvica [15]

Programski jeziki kot npr. Java, Python in C, so sicer prepričljivo pred Haskellom glede na popularnost, a se ta uvršča pred Clojure, Erlang in RPG. Glavni razlog za manjšo popularnost leži v dejstvu, da je Haskell čisti funkcijski jezik in nima veliko skupnega z objektno orientiranim programiranjem (angl. object oriented programming), ki dandanes predstavlja glavni standard razvoja aplikacij. Nekaj dodatnih razlogov za manjšo popularnost programskega jezika Haskell navaja spletni vir [17].

V nadaljevanju poglavja podajamo nekaj enostavnih in kratkih izsekov programske kode Haskell.

2.1 Uvodni program »Hello World«

Tipični uvodni primer programa je t.i. »Hello World«, katerega v programskem jeziku Haskell zapišemo kot prikazuje levi stolpec tabele 2 [3]:

Tabela 2. Hello World

'hello.hs' main = putStrLn ''Hello World''	>> ghc -o hello hello.hs [1 of 1] Compiling Main (hello.hs, hello.o) Linking hello ... >> ./hello >> Hello World

Programsko kodo v tem prispevku preizkušamo na operacijskem sistemu Linux Ubuntu, kamor namestimo prevajalnik GHC, npr. najlažje z »apt-get install ghc«. Programsko kodo prevedemo s sledečim ukazom v terminalnem oknu »ghc -o hello hello.hs«. S tem ustvarimo novo izvršljivo datoteko z imenom »hello«, katero nato poženemo z ukazom »./hello«. Ta nam v novi vrstici terminalnega okna izpiše »Hello World«.

2.2 Delo s funkcijami

V naslednjem primeru definiramo našo prvo funkcijo, poimenovano »sestevanje«. Funkcijo uredimo v urejevalniku besedil ter datoteko shranimo kot »sestej.hs«. Tabela 3 prikazuje primer funkcije seštevanja, kjer število, ki ga želimo sešteti s samim seboj, vnesemo kot vhodni parameter. Uporabnikovo posredovanje je, tako kot pri prejšnjem primeru, označeno z modro barvo v desnem stolpcu tabele 3.

Tabela 3. Funkcija seštevanja

'sestej.hs' sestevanje x = x + x	>> ghci GHCi, version 8.0.2: http://www.haskell.org/ghc/ :? for help Prelude> :l sestej [1 of 1] Compiling Main (sestej.hs, interpreted) Ok, modules loaded: Main. *Main> sestevanje 2 4 *Main>

V tabeli 3 upodabljamo interaktivno uporabo prevajalnika GHC, katerega zaženemo z ukazom »ghci« ali »ghc --interactive«. V interaktivnem prevajalniku skripto s sledečim ukazom najprej naložimo »:l sestej«. Ta nam jo sprva prevede, oz. definira, šele nato pa jo lahko uporabimo. H klicu funkcije dodamo še vhodni parameter, npr. 2, kot to prikazuje ukaz »sestevanje 2«. Vhodni parameter se v programskem jeziku Haskell zapiše pod spremenljivko »x«, katerega nato seštejemo s samim seboj, prevajalnik pa samodejno izpiše rezultat 4, tj. $2 + 2 = 4$. Vidimo, da funkcijo in vhodne parametre v programskem jeziku ustvarimo zelo kompaktno, tj. v eni vrstici (angl. one-liner). Podobno lahko sklepamo za nekoliko kompleksnejšo funkcijo, ki zraven matematičnih operacij množenja in seštevanja prinaša še pogojni (angl. »if«) stavek (tabela 4) [3].

Tabela 4. Matematične operacije in pogojni stavek

'kvadriraj.hs' kvadrirajMaloStevilo x = (if x > 100 then x else x*x) + 5	Prelude> :l kvadriraj *Main> kvadrirajMaloStevilo 3 14 *Main> kvadrirajMaloStevilo 101 106

Funkcija »kvadrirajMaloStevilo« kvadrira vhodni parameter, če je ta manjši ali enak 100. Ne glede na velikost vhodnega parametra (saj uporabljamo oklepaj), funkcija delnemu rezultatu prišteje število 5 in končni rezultat izpiše. Brez uporabljenega oklepaja bi funkcija število 5 prištela le, če bi bil vhodni parameter manjši ali enak 100. Kot zanimivost naj povemo, da je »else« stavek v programskem jeziku Haskell obvezen [3].

2.3 Delo s seznami

Nadalje prikazujemo tudi delo s seznami (angl. lists), ki so sodeč po [3] najbolj uporabljana podatkovna struktura v programskem jeziku Haskell. Seznami so v programskem jeziku Haskell homogeni, kar pomeni da morajo biti vsi vsebovani elementi seznama enakega tipa, npr. cela števila, karakterji, ipd. [3]. Seznam v programskem jeziku Haskell ustvarimo zelo preprosto, kot to prikazuje uvodna vrstica tabele 5.

Tabela 5. Seznami v programskem jeziku Haskell

```
Prelude> seznam=[1,2,4,5,6,8]
Prelude> seznam
[1,2,4,5,6,8]
Prelude> seznam !! 3
5
Prelude> head(seznam)
1
Prelude> tail(seznam)
[2,4,5,6,8]
Prelude> maximum(seznam)
8
Prelude> [x * 2 | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]
```

Seznam ustvarimo z ukazom »seznam=[1,2,4,5,6,8]«, pri čemer je »seznam« ime seznama, 1,2,4,5,6,8 pa so njegovi elementi. Slednje lahko v terminalnem oknu z ukazom »seznam« izpišemo. Izpišemo lahko tudi posamezni element seznama, npr. četrti element izpišemo z ukazom »seznam !! 3«, saj Haskell indeksiranje elementov začenja z ničlo. Nadalje lahko izpišemo prvi element seznama, tj. »head(seznam)«, preostali del seznama (razen prvega elementa), tj. »tail(seznam)« ter maksimalno vrednost seznama, tj. »maximum(seznam)«. Haskell omogoča tudi zahtevnejše operacije s seznami, tj. »pipe«. Zadnji primer prikazuje funkcijo, ki generira prvih deset števil ter jih pomnoži z 2, vse v eni vrstici.

Ker Haskell samostojno zaznava tip spremenljivk, uporabniku teh ni treba eksplicitno navajati. Kljub temu pa uporabnik tip spremenljivk lahko določi, npr. kot to prikazuje tabela 6. V levem stolpcu je prikazan primer za ročno določitev celega števila (angl. integer, krajše int) vhodni in izhodni spremenljivki. Prav tako lahko po vzoru desnega stolpca tip določimo tudi več vhodnim spremenljivkam.

Tabela 6. Ročno določanje tipa funkcije

'kvadriranje.hs'	'sestejStevila.hs'

`kvadriraj :: Integer -> Integer`
`kvadriraj x = x * x`
 	
`*Main> kvadriraj 3`
`9`
	`sestejStevila :: Int -> Int -> Int -> Int`
	`sestejStevila x y z = x + y + z`
	
	`*Main> sestejStevila 1 2 3`
	`6`

3 OGRODJE YESOD

Yesod je prosto dostopno spletno ogrodje, namenjeno za razvoj varnih in visoko-zmogljivih spletnih aplikacij [6]. Spletno ogrodje Yesod izvira iz programskega jezika Haskell, kar pomeni da vključuje vse ugodne značilnosti, naštete v drugem poglavju [7]. Avtor ogrodja Yesod Michael Snoyman navaja glavne posebnosti programskega ogrodja Yesod [1]:

- varnost tipov (angl. type safety): točno lahko določimo pričakovane vhodne in izhodne tipe,
- zgoščenost (angl. concise code): Yesod omogoča uporabo pred-pripravljenih orodij (angl. scaffolding tools) in knjižnic (npr. iz repozitorija Hackage [11]) za zmanjšanje količine programske kode in pohitritev razvoja,
- učinkovitost (angl. performance): Yesod zagotavlja prepričljivo učinkovitost, tako prevajalnika GHC, ki je podvržen neprenehnemu izboljševanju, kakor tudi same arhitekture. Slednja z uporabo voda (angl. conduit) in graditelja (angl. builder) dovoljuje, da se programska koda izvaja v pomnilniku in s tem izogiba dolgotrajnemu dostopu do trdega diska. K temu priča dejstvo, da je eden izmed Haskell-ovih spletnih strežnikov, t.i. Warp eden najhitrejših kar jih trenutno obstaja [2],
- modularnost (angl. modularity): Yesod z domiselnimi idejami in razvojem Yesod-a spodbuja razvoj novih programskih, medsebojno združljivih in konsistentnih paketov, ki so velikokrat uporabni tudi izven okvirjev Haskell-a,
- trdni temelji (angl. a solid foundation): Yesod je deležen vzporednega razvoja več različnih programskih paketov za enako stvar, kar mu omogoča izbiro najučinkovitejše rešitve za dan problem. Dodatno razvijalci Yesod-a in Haskell-a stremijo k nenehnemu izboljševanju in poenostavljanju programskih paketov,
- arhitektura (angl. architecture): posebnost Yesoda je v tem, da za ustvarjanje HTML, CSS ali JavaScript programske kode uporablja posebne domensko specifične jezike, ki so kot celotna družina znani pod skupnim imenom Shakespearske šablone (angl. Shakespearean Templates). Vsaka šablona je poimenovana po Shakespearskih akterjih, npr. šablona za HTML se imenuje »Hamlet«, šabloni za CSS sta »Lucius« in »Cassius«, šablona za JavaScript pa je poimenovana »Julius«. Družina Shakespearskih šablon bo v nadaljevanju prispevka predstavljena podrobneje.

3.1 Shakespearske šablone

Shakespearske šablone nudijo samodejno ustvarjanje HTML, CSS in JavaScript [1]. Hamlet je domensko specifični jezik, namenjen za ustvarjanje HTML. Nudi kompaktno in enostavno sintakso, brez uporabe zavitih oklepajev (angl. braceless), a s praznimi prostori (angl. white-spaces) za poravnavo (angl. indentation). Poleg ustvarjanja HTML kode omogoča tudi osnovne nadzorne bloke, kot npr. vejitve (tabela 8) in zanke. Hamlet je

najkompleksnejši izmed štirih naštetih domensko specifičnih jezikov [1], njegova sintaksa pa je prikazana v tabeli 7.

Tabela 7. Primerjava med HTML in Hamlet

HTML

```
<body>
<p>Primer nastevanja</p>
<ul>
<li>Prvi</li>
<li>Drugi</li>
</ul>
<p><h3>Vecji naslov</p>
</body>
```

Hamlet

```
<body>
    <p>Primer nastevanja
    <ul>
        <li>Prvi
        <li>Drugi
    <p>
        <h3>Vecji naslov
```

Tabela 8. Primer vejitve Hamlet

```
$if delovniDan
        <p>Gremo v službo.
$else
        <p>Ostanemo doma.
```

Lucius in Cassius sta domensko specifična jezika namenjena ustvarjanju CSS-a. Funkcionalno se slednja ne razlikujeta, sta si pa sintaktično različna. Prvi izmed njiju uporablja zavite oklepaje, medtem ko drugi prazne prostore za poravnavo. Tabela 9 prikazuje njuno osnovno sintakso.

Tabela 9. Primerjava med Lucius in Cassius [1]

Lucius (z zavitimi oklepaji)

```
section.blog {
    padding: 1em;
    border: 1px solid #000;
    h1 {
        color: #{headingColor};
        background-image: url(@{MyBackgroundR})
    }
}
```

Cassius (s praznimi prostori)

```
section.blog
    padding: 1em
    border: 1px solid #000
    h1
        color: #{headingColor}
        background-image: url(@
```

Julius je zadnji izmed domensko specifičnih jezikov, namenjen ustvarjanju JavaScript programske kode. V splošnem je zelo podoben samemu JavaScript-u, zato [1] navaja, da je Julius najenostavnejši izmed domensko specifičnih jezikov. Tabela 10 prikazuje njegovo sintakso.

Tabela 10. Prikaz domensko specifičnega jezika Julius [1]

```
$(function(){
    $("section.#{rawJS sectionClass}").hide();
    $("#mybutton").click(function(){document.location = "@{
    SomeRouteR}";});
    ^{addBling}
});
```

3.2 Uvodni program »Hello World« v ogrodju Yesod

Yesod je zelo obširno ogrodje, zato v prihajajočih podpoglavjih prikazujemo le njegove bistvene značilnosti. Sprva predstavljamo osnovni program »Hello World« (tabela 11), osnovan na čelnem ogrodju (angl. front-end framework) »Twitter Bootstrap« [19]. Za prevajanje programa uporabljamo prevajalnik GHC, lahko pa program zaženemo tudi z ukazom »runhaskell app.hs«, kjer »app.hs« predstavlja ime programa. S tem

zaženemo spletni strežnik Warp [2], čigar delovanje programa preizkusimo z obiskom na spletni strani http://localhost:3000, kot to prikazuje slika 2.

Tabela 11. Primer »Hello World« v spletnem ogrodju Yesod

```
1  {-# LANGUAGE OverloadedStrings #-}
2  {-# LANGUAGE QuasiQuotes       #-}
3  {-# LANGUAGE TemplateHaskell   #-}
4  {-# LANGUAGE TypeFamilies      #-}
5  {-# LANGUAGE MultiParamTypeClasses #-}
6  {-# LANGUAGE ViewPatterns      #-}
7
8  import           Yesod
9  import           Yesod.Static
10
11 data App = App
12     { getStatic :: Static
13     }
14
15 mkYesod "App" [parseRoutes|
16 / HomeR GET
17 /static StaticR Static getStatic
18 |]
19
20 instance Yesod App
21
22 getHomeR :: Handler Html
23 getHomeR = do
24     defaultLayout $ do
25         setTitle "Hello world"
26         $(whamletFile "hamlet/hello.hamlet")
27
28 main :: IO ()
29 main = do
30     static@(Static settings) <- static "static"
31     warp 3000 $ App static
```

Haskell language pragmas

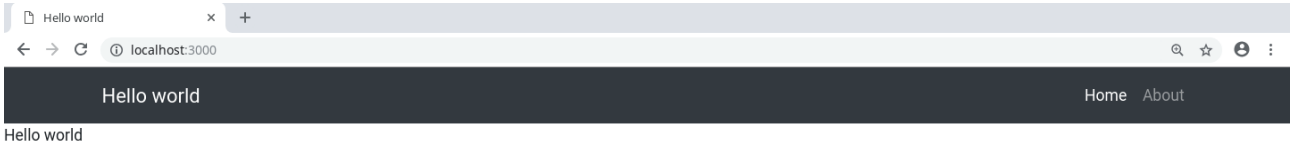
Vključitev Yesod knjižnic

Delo s statičnimi datotekami

Usmerjanje (angl. routing)

Rutina (angl. handler function)

Glavni program (zagon spletnega strežnika Warp)



Slika 2. Pogled v spletnem brskalniku

3.3 Gradniki

Nadaljujemo s pregledom Yesod gradnikov. Med razvojem spletne strani in aplikacij navadno usklajujemo tri različne tehnologije, tj. HTML, CSS in Javascript. Naštete se na spletni strani nahajajo na različnih lokacijah, npr. CSS v glavi in HTML v jedru spletne strani, kar dodatno otežuje organizacijo. Yesod za ta problem uvaja gradnike (angl. widgets). Ti omogočajo vključevanje različnih knjižnic, morebiti shranjenih lokalno, ali pa tudi oddaljeno. Primeri nekaterih elementov gradnikov so predstavljeni v tabeli 12.

Tabela 12. Primeri gradnikov

setTitle	Nastavi naslov spletne strani
addStylesheet	Poveži do lokalno shranjene slogovne predloge CSS
addStylesheetRemote	Poveži do oddaljeno shranjene slogovne predloge CSS
addCassius	Dodaj nekaj CSS značk
addHamlet	Dodaj Hamlet v značko body
addScript	Poveži do lokalno shranjene skripte
addScriptRemote	Poveži do oddaljeno shranjene skripte
addHtmlHead	Dodaj HTML v značko head

Tabela 13 prikazuje osnovni primer programske kode gradnikov, za več gradnikov pa je bralec povabljen k ogledu uradne dokumentacije [18]. Slika 3 predstavlja prikaz gradnikov v spletnem brskalniku.

Tabela 13. Prikaz programske kode za razvoj gradnika

```
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE QuasiQuotes       #-}
{-# LANGUAGE TemplateHaskell   #-}
{-# LANGUAGE TypeFamilies      #-}

import  Yesod

data App = App
mkYesod "App" [parseRoutes|
/ DomovR GET
|]
instance Yesod App

getDomovR = defaultLayout $ do
    setTitle "OTS 2019"
    toWidget [lucius| h3 { color: blue; } |]
    addScriptRemote "https://ajax.googleapis.com/ajax/libs/jquery
    /1.6.2/jquery.min.js"
    toWidget
        [julius|
            $(function() {
                $("h3").click(function(){
                    alert("Kliknili ste na naslov!");
                });
            });
        |]
    toWidgetHead
        [hamlet|
            <meta name=keywords content="haskell, yesod, spletna
    stran">
        |]
    toWidget
        [hamlet|
            <h3>Na tak nacin lahko vkljucimo vsebino
        |]
    [whamlet|<h4>In se en nacin |]
    toWidgetBody
        [julius|
            alert("JavaScript alert vkljucen v body");
        |]

main = warp 3000 App
```

Nastavi naslov spletne strani

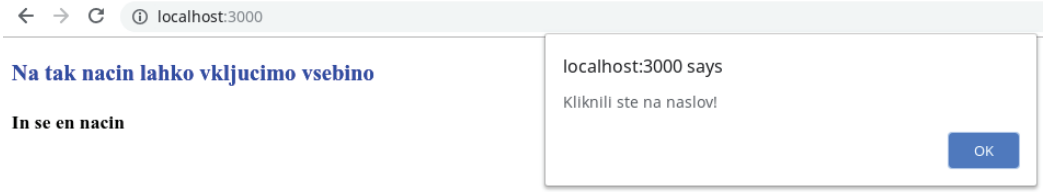
Poveži do oddaljene skripte

Vpelji pojavno okno

Določi meta-podatke spletne strani

Ustvari odstavek

Ustvari dodaten odstavek



Slika 3. Prikaz gradnika v spletnem brskalniku

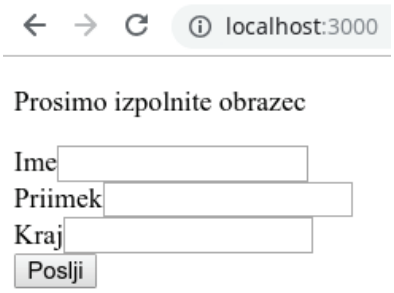
3.4 Obrazci

Poslednja zanimivost, ki jo predstavljamo v tem prispevku, so Yesod obrazci (angl. forms). Obrazci so interaktivni deli spletne strani, v katere uporabnik lahko vpisuje podatke. Tabela 14 prikazuje primer programske kode za razvoj Yesod obrazca, medtem ko slika 4 rezultat, ki se pri tem pojavi. Levi stolpec tabele 14 prikazuje določevanje lastnega podatkovnega tipa spremenljivk, s katerimi bo uporabnik spletne strani operiral. Desni stolpec tabele 14 prikazuje dodane gradnike, ki bodo določili izgled spletne strani.

Tabela 14. Prikaz programske kode za razvoj obrazcev

```
data Oseba = Oseba
    { osebaIme          :: Text
    , osebaPriimek      :: Text
    , osebaKraj         :: Text
    }
  deriving Show
```

```
osebaForm :: Html -> MForm Handler (FormResult Oseba, Widget)
osebaForm = renderDivs $ Oseba
    <$> areq textField "Ime" Nothing
    <*> areq textField "Priimek" Nothing
    <*> areq textField "Kraj" Nothing
```



Slika 4. Prikaz enostavnega obrazca Yesod v spletnem brskalniku

Z obrazci zaključujemo kratek pregled bistvenih značilnosti Yesod-a in v prihodnjem poglavju nadaljujemo na postavitev aplikacije.

4 POSTAVITEV APLIKACIJE

Postavitev končne aplikacije je navadno zadnji, a nadvse kompleksen korak razvoja spletnih aplikacij. Pri tem ni izjema niti spletno ogrodje Yesod, čeravno se celotna skupnost raziskovalcev zavzema za poenostavitev postopka postavitve [14]. V Yesod-u postavitev poteka večplastno. Sprva s prevajalnikom prevedemo vso programsko kodo in pridobimo izvedljivo datoteko. To nato prenesemo na spletni strežnik, kamor obvezno priložimo tudi statične datoteke. V drugem koraku namestimo enega izmed postavitvenih ogrodij (angl. deployment engine) ali spletni strežnik. V tabeli 15 navajamo nekaj najbolj pogostih postavitvenih ogrodij, njihove prednosti ter slabosti.

Tabela 15. Yesod postavitvena ogrodja

Ogrodje	Glavne značilnosti	Prednosti	Pomankljivosti
Keter	Sistem za postavitev spletnih aplikacij Yesod	Zagotavlja podporo SSL, samodejno zažene aplikacijo ob zagonu, spremlja procese kakor tudi ponovno zažene aplikacijo v primeru sesutja	Ni primerno ogrodje za manjše aplikacije
Warp	Visoko-zmogljiv spletni strežnik, v celoti napisan v Haskellu	Ni potrebe po namestniškem strežniku (angl. proxy server)	V primeru sesutja je potreben mehanizem za ponovni zagon aplikacije
Nginx	Vzvratni namestniški strežnik (angl. reverse proxy)	Lažja migracija spletnih aplikacij med več strežniki	Zahteva obilo konfiguriranja
Kombinacija	Kombinacija več postavitvenih ogrodij in tehnik, npr. Warp in Keter	Možnosti prilagajanja	Zahtevano napredno znanje sistemske administracije v Linuxu, oteženo vzdrževanje

5 ZAKLJUČEK

V tem prispevku smo na kratko predstavili programski jezik Haskell in spletno ogrodje Yesod. Yesod je ogrodje, ki omogoča hiter razvoj spletnih aplikacijah, katere odlikujeta varnost in zmogljivost. Varnost je zagotovljena z notranjo zasnovo programskega jezika Haskell, saj tovrstne aplikacije po uspešnem prevajanju ne povzročajo napak med sprotnim izvajanjem programske kode, zmogljivost izvajanja programske kode pa dosega približno raven programskega jezika C. Ocenjujemo, da je Yesod zato zelo primeren za časovno kritične spletne aplikacije in spletne aplikacije s poudarjenimi varnostnimi zahtevami.

Ugotavljamo, da je postavitev osnovne spletne aplikacije v spletnem ogrodju Yesod zelo kompleksna. Potrebne so veščine sistemske administracije v operacijskem sistemu Linux in poglobljeno znanje programskega jezika Haskell. Za vizualno privlačne spletne aplikacije je potrebno tudi dobro poznavanje Yesod gradnikov in obrazcev, nenazadnje pa tudi Shakespearskih šablon. Kljub temu pri zagonih osnovnih spletnih aplikacij opažamo nazorno pohitritev delovanja spletne strani.

6 LITERATURA

[1] SNOYMAN Michael. Developing web applications with Haskell and Yesod. O'Reilly Media, Inc., 2012.

[2] SNOYMAN Michael. "Warp: A Haskell web server." *IEEE Internet Computing* 15.3 (2011): 81-85.

[3] LIPOVAČA Miran. *Learn you a haskell for great good!: a beginner's guide*. No Starch Press, 2011.

[4] HUDAK Paul, FASEL H. Joseph, PETERSON John. "A gentle introduction to Haskell." *Sigplan Notices* 27.5 (1992): T1-T53.

[5] HUDAK Paul. "Building domain-specific embedded languages." *ACM Computing Surveys* 28.4es (1996): 196.

[6] DVOŘÁK Pavel. Web applications in Haskell. Diss. Masarykova univerzita, Fakulta informatiky, 2012.

[7] SNOYMAN Michael. Developing Web Apps with Haskell and Yesod: Safety-driven Web Development. "O'Reilly Media, Inc.", 2015.

[8] EPSTEIN Jeff, BLACK P. Andrew, PEYTON-JONES Simon. "Towards Haskell in the Cloud." *ACM SIGPLAN Notices*. Vol. 46. No. 12. ACM, 2011.

[9] IZBICKI Michael. "Hlearn: A Machine Learning Library for Haskell." Proceedings of The Fourteenth Symposium on Trends in Functional Programming, Brigham Young University, Utah. 2013.

[10] ANNENKOV Danil, ELSMAN Martin. "Certified Compilation of Financial Contracts." *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*. ACM, 2018.

[11] https://hackage.haskell.org/, Welcome to Hackage!, obiskano 13. 5. 2019.

[12] https://wiki.haskell.org/Haskell_in_practice, Haskell in practice, obiskano 10. 5. 2019.

[13] https://en.wikipedia.org/wiki/Haskell_(programming_language)), Haskell (programming language), obiskano 10. 5. 2019.

[14] https://www.yesodweb.com/book, Yesod Web Framework, obiskano 9. 5. 2019.

[15] https://www.tiobe.com/tiobe-index/, TIOBE Index for May 2019, obiskano 13. 5. 2019.

[16] https://en.wikipedia.org/wiki/Yesod_(web_framework), Yesod (web framework), obiskano 10. 5. 2019.

[17] https://www.forbes.com/sites/quora/2017/07/11/why-arent-more-programmers-using-haskell/, Why Arent't More Programmers Using Haskell?, obiskano 10. 5. 2019.

[18] https://hackage.haskell.org/package/yesod-core-1.1.1/docs/Yesod-Widget.html, Yesod Widget, obiskano 14. 5. 2019.

[19] https://github.com/firefly-cpp/Yesod-bootstrap-skeleton, GitHub Yesod-bootstrap-skeleton, obiskano 14. 5. 2019.

OTS 2019
SODOBNE INFORMACIJSKE TEHNOLOGIJE IN STORITVE
M. Heričko in K. Kous (ur.)

Univerzitetna založba
Univerze v Mariboru

KAKO RAZVIJATI V VUE/NUXT, ČE SI NAVAJEN/A OBJEKTNIH JEZIKOV, KOT SO C++, C# ALI JAVA?

MATJAŽ PRTENJAK

Povzetek: *VUE je samo knjižnica, napisana v programskem jeziku JavaScript in kot takšna sploh ni neposredno primerljiva s programskima jezikoma C++/Java. Težava pri preklopu torej ni VUE, temveč JavaScript. Za C++/Java programerja/programerko pa je JavaScript nevaren jezik, saj je prvima tako podoben, da daje napačen signal. Za uspešen razvoj v spletnih tehnologijah je torej nujno dobro spoznati programski jezik JavaScript. Ob tem pa seveda še HTML in CSS, ki sta pač osnova spleta.*

Ključne besede: *JavaScript, Vue, razvoj, splet, front-end*

NASLOV AVTORJA: mag. Matjaž Prtenjak, Matjaž Prtenjak s.p., Celje, Slovenija, info@mnet.si.

https://doi.org/10.18690/978-961-286-282-4.26 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 UVOD

Mnogi razvijalci, ki smo začenjali profesionalno pot razvoja programske opreme konec devetdesetih let prejšnjega stoletja, smo tedaj padli v intenzivno objektno usmerjeno programiranje. Objekti so bili povsod, vsi so govorili samo o objektno usmerjenem razvoju in vsi smo hiteli iz jezika C na C++ ter kasneje na Javo [1].

Do devetdesetih let smo uporabljali računalnike brez posebnega poudarka na grafiki in grafičnih uporabniških vmesnikih. Unix je sicer imel X-Windows GUI, v ozadju pa se je prebijal Microsoft z Windows 3.1 in kasneje z Windows 95 [2]. Devetdeseta leta so bila torej leta, ko smo intenzivno prehajali iz ukazne vrstice ter ASCII (znakovnih) uporabniških zaslonov na grafične uporabniške zaslone – GUI.

Jezika C++ in Java nimata vgrajenih grafičnih sposobnosti. To sta splošna programska jezika in v njih – z uporabo objektov – pač razvijamo programe, ki niti nimajo nujno uporabniškega vmesnika. Lahko gre za strežniško infrastrukturo, storitve in podobno.

Šele nekatere dodatne (objektne) knjižnice nam prinesejo možnost izdelave grafičnih uporabniških vmesnikov. S pomočjo takšnih knjižnic, recimo MFC za C++ in Windows, lahko razvijamo grafične uporabniške vmesnike. Konec stoletja torej vsi razvijamo grafične uporabniške vmesnike, nakar se zgodi bum spleta in naenkrat gre vse v HTML.

2 VUE

Hitro previjemo 15-20 let naprej in leta 2014 se pojavi Vue [3]. Dolgi uvod je potreben, saj Vue sploh ne moremo primerjati s C++ ali Javo. Vue je tista dodatna knjižnica, ki nekomu omogoča izdelavo grafičnega uporabniškega vmesnika, le da tokrat ne na operacijskih sistemih kot so Linux, Windows, Android ali Mac, temveč na spletu oz. natančneje, v spletnem brkljalniku, ki je kot takšen neodvisen od operacijskega sistema, na katerem teče.

2.1 Vue je knjižnica, napisana v JavaScript programskem jeziku

Vue torej nikakor ne moremo neposredno primerjati s C++ ali Java. Slednja sta splošno namenska programska jezika, Vue pa je JavaScript [4] programsko ogrodje (*JavaScript framework*), ki je namenjen izključno izdelavi interaktivnih HTML (spletnih) vmesnikov. Vue je za nekoga, ki je začel s programiranjem v devetdesetih, navadna knjižnica, kot sta WinForms ali QT. Pravzaprav ni niti to, saj kombinaciji HTML in CSS ne dodaja nič novega, le omogoča njuno lažjo in hitrejšo uporabo.

Preden se razvijalec sploh začne ukvarjati z Vue, mora (dobro) spoznati JavaScript. Ime JavaScript je zelo podobno imenu Java in tudi izgled tipične JavaScript kode je zelo zelo podoben izgledu Java ali C++ kode, vendar pa gre tukaj za popolnoma drug jezik!

Ena izmed mojih največjih strokovnih napak je bila, da se dolga leta sploh nisem poglobil v JavaScript, temveč sam ga vedno obravnaval kot nek čuden programski jezik z okusom po Javi/C++.

2.1.1 *Bistvene razlike med C++/Java in JavaScript*

- Java uporablja statične podatkovne tipe, JavaScript dinamične. To pomeni, da je lahko neka spremenljivka različnega podatkovnega skozi tok programa
- Java uporablja razredno hierarhijo, JavaScript uporablja prototipno hierarhijo
- Javanski program se naloži iz prevedene programske kode (ki jo razume računalnik), JavaScript se naloži iz kode, zapisane v »človeku razumljivi obliki« (v izvorni obliki). Javansko kodo je torej potrebno predhodno prevesti, JavaScript kode ne
- V JavaScriptu so funkcije enakovredne objektom in zato je lahko JavaScript tudi funkcijski jezik, Java ne more biti
- Java je obširnejši, kompleksnejši programski jezik z dobro premišljenimi osnovami, JavaScript je dokaj preprost, odprt, skriptni jezik, ki je nastal v 10 dneh in ima mnogo nekonsistenc
- V JavaScriptu podpičja niso obvezna
- Java definira spremenljivke na nivoju blokov, JavaScript na nivoju funkcij
- Moč JavaScript se pogosto skriva v zaprtjih spremenljivk (closures), v Javi tega sploh nimamo

- V JavaScript lahko prototip objekta spremenimo in s tem takoj spremenimo vse objekte, ki bazirajo na tem prototipu. V Javi česa podobnega nimamo

Seveda je razlik še kar nekaj, kakor je tudi kar veliko podobnosti in ravno to je past, v katero se ujame mnogo izkušenih programerjev [5].

2.2 Nekaj JavaScript primerov, ki so C++/Java programerja/programerko preprosto »čudni«

2.2.1 *Domet spremenljivk (scope)*

Primer 1

```
1  var i = 10
2
3  console.log("[ 3] == ",i)
4  test()
5  console.log("[ 5] == ",i)
6
7  function test() {
8    console.log("[ 8] == ",i)
9    // var i = 20
10   console.log("[10] == ",i)
11 }
12
13 console.log("[13] == ",i)
14 test()
15 console.log("[15] == ",i)
```

Rezultat

```
[ 3] ==  10
[ 8] ==  10
[10] ==  10
[ 5] ==  10
[13] ==  10
[ 8] ==  10
[10] ==  10
[15] ==  10
```

Primer 2

```
var i = 10

console.log("[ 3] == ",i)
test()
console.log("[ 5] == ",i)

function test() {
  console.log("[ 8] == ",i)
  var i = 20
  console.log("[10] == ",i)
}

console.log("[13] == ",i)
test()
console.log("[15] == ",i)
```

Rezultat

```
[ 3] ==  10
[ 8] ==  undefined
[10] ==  20
[ 5] ==  10
[13] ==  10
[ 8] ==  undefined
[10] ==  20
[15] ==  10
```

Primer 3

```
var i = 10

console.log("[ 3] == ",i)
test()
console.log("[ 5] == ",i)

function test() {
  console.log("[ 8] == ",i)
  i = 20
  console.log("[10] == ",i)
}

console.log("[13] == ",i)
test()
console.log("[15] == ",i)
```

Rezultat

```
[ 3] ==  10
[ 8] ==  10
[10] ==  20
[ 5] ==  20
[13] ==  20
[ 8] ==  20
[10] ==  20
[15] ==  20
```

Tukaj so trije zelo podobni primeri, ki se razlikujejo samo v vrstici 9 (odebeljena), hkrati pa so rezultati dokaj različni in praktično vsi nepričakovani!

Ker to ni predavanje o JavaScript, se v razloge za rezultate ne bom spuščal, naštel pa bom nekaj tipičnih vprašanj, ki se porajajo:

1. Kako lahko izvedemo funkcijo test v vrstici 4, če tam sploh še ni definirana ali vsaj deklarirana?
2. *primer1*: Zakaj je spremenljivka i vidna v funkciji test?
3. *primer2*: Zakaj vrednost spremenljivke v vrstici 8 sploh ni znana? Prej je bila vidna, zdaj pa sploh ne?
4. *primer3*: Zakaj je vrednost spremenljivke i v vrstici 13 enaka 20?
5. *... in še kar nekaj je vprašanj ...*

2.2.2 *Funkcije so enakovredni člani*

```
1  function test() {
2    console.log('To je test')
3  }
4  test.i = 10
5  test.funkcija1 = function() {
6    console.log('To je funkcija 1')
7  }
8  test()                      // rezultat je »To je test«
9  test.funkcija1()            // rezultat je »To je funkcija 1«
10 console.log(test)           // rezultat je »{[Function:test] i: 10,funkcija1:[Function]}«
11 console.log(test.i)         // rezultat je 10
```

Spet nekaj vprašanj, ki se porajajo:

1. *vrstica 5*: Kako smo lahko dostopali do lastnosti *i* funkcije *test*, če *test* sploh ni objekt, temveč funkcija in če *i* sploh nikjer ni definiran?
2. *vrstica 6:* Kako smo lahko definirali novo funkcijo kot lastnost neke druge funkcije?
3. *vrstica 12:* Kaj je ta čuden zapis objekta/funkcije?
4. *... in še kar nekaj je vprašanj ...*

2.2.3 *Novi objekti nastajajo brez eksplicitnih »konstruktorjev« in imajo dinamične lastnosti*

```
1  function Oseba(ime, priimek) {
2    this.ime = ime
3    this.priimek = priimek
4
5    this.izpisi = function() {
6      console.log(`oseba = ${ime} ${priimek}`)
7    }
8  }
9
10 Oseba("James", "Bond")
11 var peter = new Oseba("Peter", "Veliki")
12 var aleksander = new Oseba("Aleksander", "Veliki")
13
14 peter.izpisi()         // rezultat je »Peter Veliki«
15 aleksander.izpisi()    // rezultat je »Aleksander Veliki«
16
17 peter.izpisi = function() {
18   console.log(`oseba = ${ime} Mali`)
19 }
20
21 peter.izpisi()         // rezultat je »Peter Mali«
22 aleksander.izpisi()    // rezultat je »Aleksander Veliki«
```

No, tu pa je vprašanj že toliko, da jih niti ne bom pisal…

2.2.4 *Težava kazalca 'this'*

Vsakemu razvijalcu/razvijalki v C++ in Javi je kazalec '*this*' domač in vsakdo vedno ve, kaj točno predstavlja kazalec *this*. Nikoli ni v dilemi! V primeru JavaScript temu ni tako in to je ena izmed najtežjih lekcij, ki se je mora vsakdo naučiti, ko začne s programiranjem v JavaScript-u.

```
1  function Fun1() {
2    this.ime = 'fun1';
3    this.func = function() {
4      console.log(this.ime)
5    }
6  }
7
8  function Fun2() {
9    this.ime = 'fun2';
10 }
11
12 var f1 = new Fun1();
13 var f2 = new Fun2();
14
15 f1.func();            // rezultat je »fun1«
16 f1.func.call(f2);     // rezultat je »fun2«
```

2.2.5 *Pa še nekaj matematike*

Tukaj podajam samo programsko kodo in rezultate, ki so vsakemu C++/Java programerju/programerki preprosto nedojemljivi…

```
"10" - 4       // rezultat je 6
"10" + 4       // rezultat je "104"
"20" - "5"     // rezultat je 15
"20" + "5"     // rezultat je "205"
"20" - - "5"   // rezultat je 25
"ma" + "ma"    // rezultat je "mama"
"ma" + + "ma"  // rezultat je "maNaN"
"6" - 3 + 3    // rezultat je 6
"6" + 3 - 3    // rezultat je 60
```

2.3 JavaScript je tukaj in bo ostal

Kot že omenjeno, je Javascript programski jezik, ki je nastal v specifičnih okoliščinah, s strani enega samega razvijalca v 10 dneh. Glede na zapisano je jasno, da to ni programski jezik, ki bi bil domišljen, sofisticiran in logičen. JavaScript ima vse polno čudnih, smešnih in neverjetnih lastnosti, a ravno zaradi teh je tudi zelo močan in zaradi mnogih vedno bolj popularen programski jezik.

JavaScript že dolgo ni več »divji« jezik, ki bi kar rasel in pridobival vedno bolj čudne lastnosti. Ravno obratno! JavaScript je jezik, ki se že desetletje načrtno razvija, dopolnjuje in nadgrajuje, vendar pa je v njem napisano tako veliko programske kode, da osnov ne sme in ne more nihče več spremeniti. Vse, kar je čudnega, bo torej za vedno ostalo čudno, nadgradnje pa so že bolj »logične« [6].

2.4 JavaScript ~ Vue

Odgovor na naslovno vprašanje: »*Kako razvijati v Vue/NUXT, če si navajen/a objektnih jezikov, kot so C++, C# ali Java?*« se torej sploh ne skriva v Vue, temveč v JavaScript. Iz tega sledi tudi ključna misel tega predavanja:

Dandanes obstaja množica JavaScript knjižnic in objektno orientirano programiranje ali »neobjektno« orientirano programiranje ni v domeni teh dodatnih knjižnic, temveč v domeni JavaScript programskega jezika.

2.4.1 *JavaScript in objektno orientirano programiranje*

Z rastjo popularnosti programskih jezikov, kot so JavaScript, Python, Ruby in podobnih, se je počasi začel zaton ideje objektnega programiranja. Zelo znana je izjava Joe Armstronga, očeta programskega jezika Erlang [7]:

»I think the lack of reusability comes in object-oriented languages, not functional languages. Because the problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.«

Njegova izjava vsekakor drži in ji ni moč oporekati. Iz objektne knjižnice namreč ne moremo vzeti nekega objekta, ne da bi zraven vzeli tudi njegovega očeta in dedka in pradedka…, torej celotne hierarhije objektov.

Hkrati pa to ne pomeni, da je bilo oz. da je objektno orientirano programiranje napaka ali slepa ulica v razvoju. Tudi to tezo je namreč moč zaslediti med mladimi programerji in programerkami. Ne, objektno orientirano programiranje je pomembno vplivalo (in vpliva) na razvoj računalništva!

2.4.2 *Kaj sploh je »objektno orientirano programiranje«?*

Vseskozi uporabljam izraz »objektno orientirano programiranje« (OOP), ne da bi ga sploh definiral ali razložil [1]. Tukaj torej zares na hitro podajam neke osnovne lastnosti:

- OOP je koncept, zgrajen na ideji »objektov«, ki vsebujejo podatke (imenovane atributi/lastnosti) in kodo (imenovano metode) za manipulacijo teh podatkov
- Razred – abstraktna definicija razreda
- Objekt – konkretna instanca razreda.

Prava moč objektnih jezikov pa pride do izraza, ko tovrstne razrede združimo v hierarhije, ko uporabljamo virtualne funkcije, enkapsulacijo in podobne zasnove.

JavaScript tudi spada med »objektne« jezike in v JavaScript-u lahko brez težav programiramo s poznanimi konstrukti jezikov C++ in Java. Ravno tako, kot lahko C++ uporabljamo čisto proceduralno (brez objektov, razredov…), lahko tudi JavaScript uporabimo kot:

- »navaden« proceduralni jezik (kot C, Pascal)
- kot »objektno orientiran jezik« (kot C++, JavaScript)
- kot »funkcijski« jezik (kot F#).

Ravno ta izbira možnosti daje programskemu jeziku JavaScript takšno moč in popularnost. Razvijalec lahko namreč uporablja programerske vzorce, ki jih pozna. Lahko pa jih seveda tudi meša oz. uporablja enega in drugega hkrati.

Po neki računalniški logiki množic bi lahko rekli, da je JavaScript »nad množica« Jave ali C++, saj omogoča vse, kar omogočata slednja + nekaj več, recimo funkcijsko programiranje. Je pa tudi res, da čeravno govorimo o objektnih jezikih in hierarhijah med objekti, je izgradnja takšne hierarhije v JavaScript-u drugačna kot v C++ ali Javi (prototipna hierarhija ~ razredna hierarhija).

2.5 Odgovor na vprašanje iz naslova

Ko programirate v JavaScript programskem jeziku ali katerikoli knjižnici na njegovi osnovi, uporabite tiste značilnosti, ki so za dani program najprimernejše. Kot rečeno, lahko JavaScript uporabljate kot OOP jezik, kot funkcijski jezik, ali pa kot preprost proceduralni jezik.

Vedno uporabite tisto, kar je za rešitev danega problema najprimernejše in da se boste lahko pravilno odločili, se morate naučiti JavaScript.

3 VUE

Vue je torej programska knjižnica, napisana v programskem jeziku JavaScript. Namenjena je prikazu podatkov v brkljalniku, kar pomeni, da za delovanje nujno potrebuje brkljalnik. Vue je strogo vezan na spletne tehnologije oz. na brkljalnik ter HTML opisni jezik.

3.1 Njegovi konkurenti

Vue vedno in povsod primerjajo vsaj z njegovima glavnima konkurentoma Angular ter React. Na spletu je mnogo strani s primerjavami omenjene trojice in celoten članek bi lahko posvetili samo primerjavi omenjenih produktov [8]. Vsi trije namreč rešujejo isti problem, vsi temeljijo na JavaScript-u in so si v resnici zelo podobni. Najkrajša primerjava med omenjenimi bi bila sledeča (sredi leta 2019):

- Vue je najmlajši in najmanjši (zasede najmanj prostora v pomnilniku)
- Angular je najstarejši in izgublja popularnost, React ter Vue jo pridobivata
- Vue je programska knjižnica, ki jo lahko na spletni strani samo odpremo (naložimo) in takoj uporabimo, Angular in React pa sta kompleksnejša
- Z Vue pa lahko, v povezavi z Webpack, ravno tako kot z Angular in React, razvijamo kompleksne programske module oz. aplikacije.

3.2 Primer uporabe Vue na spletni strani

Ena izmed prednosti Vue knjižnice je torej preprosta uporaba znotraj HTML dokumenta in tukaj predstavljam preprosto aplikacijo, ki prikazuje interaktivnost spletne strani.

```
<!doctype html>
<html lang="sl">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
```

```
    <link rel="stylesheet"
          href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css">
    <title>Črpalke in cene goriv</title>
  </head>
  <body>
    <div id="app" class="container">
      <div class="alert alert-primary">
        <h3>Črpalke in cene goriv (10km ob izbranem kraju)</h3>
      </div>

      <div v-if="error" class="alert alert-danger">{{ error }}</div>

      <div class="alert alert-light">
        <div class="form-inline">
          <input type="text" class="form-control" placeholder="Kraj..."
                v-model="city" @keyup.enter="doSearch">
          <button type="button" class="btn btn-primary" @click="doSearch">Išči</button>
        </div>
      </div>

      <div v-if="data" class="alert alert-success">
        <table class="table table-hover table-sm">
          <thead><tr>
            <th v-for="(c, i) in captions" :key="i">{{ c }}</th>
          </tr></thead>
          <tbody>
            <tr v-for="(row, rdx) in data" :key="rdx">
              <td v-for="(cell, cdx) in row" :key="cdx">{{ cell }}</td>
            </tr>
          </tbody>
        </table>
      </div>
    </div>

<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/2.6.10/vue.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/axios/0.18.0/axios.min.js"></script>
<script>
const CORS_URL = 'https://cors-anywhere.herokuapp.com/'

let app = new Vue({
  el: '#app',
  data: {
    city: null,
    error: null,
    captions: ['#', 'Naziv', 'Naslov', '95', '100', 'Dizel', 'LPG'],
    data: null,
  },
  methods: {
    doSearch() {
      if ((!this.city) || (this.city.trim().length < 3)) {
        this.error = "vpišite vsaj prve tri črke kraja, ki ga iščete"
        return
      }
      this.error = null
      this.city = this.city.trim()

      let goriva_url = "https://goriva.si/api/v1/search/"
      let goriva_data = { 'position': this.city, 'radius': 10000 }

      axios.get(CORS_URL + goriva_url, { params: goriva_data })
        .then(response => {
          let petrol_stations = response.data.results
          this.data = petrol_stations.map((station, idx) => {
            return [idx, station.name, station.address, station.prices['95'] || '-',
                    station.prices['100'] || '-', station.prices['dizel'] || '-',
```

```
                    station.prices['avtoplin-lpg'] || '-']
          })
          console.log(this.data)
        })
        .catch(e => { this.error = e })
    }
  }
})
</script>
  </body>
</html>
```

Predstavljena programska koda je stisnjena, da zasede manj prostora, zato izgled ni »lep«, a že teh par vrstic programske kode omogoča izdelavo interaktivne spletne strani za izpis črpalk v nekem kraju in ustreznih cen goriva. Aplikacija je interaktivna in preko REST protokola uporablja podatke s spleta in to celo iz druge domene.

3.3 Vue, npm in Webpack

Predstavljen primer je izjema v razvoju interaktivnih spletnih strani, saj se dandanes večino razvoja opravi s pomočjo upravljalcev paketov, kot je npm in modulov za povezovanje, kot je webpack.

Pred razvojem neke rešitve torej s pomočjo upravljalca paketov naprej s spleta pridobimo vse potrebne knjižnice (recimo za dostop do podatkov preko REST protokola, za zaščito podatkov, za lepši prikaz podatkov, za grafe…), na kar produkt razvijemo in ga na koncu s pomočjo WebPack-a stisnemo v knjižnico, ki jo lahko prepišemo na spletni strežnik.

Tovrsten razvoj ima obilico prednosti, med katerimi bi izpostavil samo dve:

- Preko npm imamo dostop do sto tisoče različnih JavaScript programskih knjižnic
- Webpack (lahko) ob združitvi rešitve poskrbi tudi za kompatibilnost med različnimi spletnimi brkljalniki in stiskanje programske kode in s tem hitrejše nalaganje.

3.4 Kakšna znanja potrebujem za razvoj spletnih aplikacij v VUE

Na spletu je množica tečajev za programski jezik JavaScript in množica tečajev za knjižnico Vue, zato tukaj o tem ne bom več izgubljal preveč besed. Potrebujete torej sledeča znanja:

- JavaScript - za programiranje
- HTML in CSS za prikaz na spletu
- Spletne storitve/REST za dostop do podatkov.

3.5 Kaj pa je Nuxt? V naslovu predavanja je omenjen tudi Nuxt?

Kot že omenjeno, lahko Vue uporabljate kot knjižnico znotraj vaše spletne strani (*preprosta uporaba)* ali pa s pomočjo npm/webpack razvijete aplikacijo. V slednjem primeru lahko uporabite ogrodje Nuxt, ki vam s pomočjo par vprašanj vzpostavi npm/webpack projekt, s pomočjo katerega lahko razvijete aplikacijo.

Nuxt je torej skupek pomožnih funkcij/procedur, ki vam vzpostavijo osnovno strukturo Vue aplikacije. Nuxt je samo nadgradnja Vue, za lažjo uporabo.

4 KAKO SE TOREJ SPOPRIJETI S SPLETNIM RAZVOJEM S POMOČJO VUE KNJIŽNICE?

4.1 Naučiti se morate spletnega opisnega jezika HTML v povezavi s CSS

To je danes osnovno znanje in kot smo morali v devetdesetih letih znati uporabljati ukazno vrstico (in to znanje je zelo pomembno tudi danes!), tako je za spletni razvoj nujno potrebno znanje opisnega jezika HTML.

HTML je bil nekoč namenjen tako opisu vsebine, kot tudi oblike, kasneje pa je obliko prevzel CSS. Prišli smo namreč do spoznanja, da je bolje ločiti vsebino od oblike in zato danes uporabljamo HTML oz. XHTML za vsebino spletne strani, obliko posameznega elementa pa določa CSS.

4.2 Potrebno se je naučiti JavaScript

To je jasno že iz dosedanjih opisov in tukaj bi vse C++/Java programerje/programerke še posebej opozoril, naj ne omalovažujejo preprostosti JavaScript programskega jezika in naj bodo prepričani, da ga ne poznajo!

Programskega jezika JavaScript se je potrebno naučiti. Znanje Jave/C++ ni nujno koristno. Pri začetnih korakih spoznavanja JS je dobrodošlo, kasneje pa je marsikdaj celo ovira, saj 10, 15, 20 let razmišljanja v Java/C++ pusti posledice in predsodke.

4.3 Potrebno je spoznati node / npm (yarn) / webpack

Podobno, kot je nekoč VisiCalc pomenil revolucijo za prodor osebnih računalnikov, je nekaj let nazaj Googlova programska oprema '*node*' naredila revolucijo za popularizacijo JavaScript programskega jezika. *npm* / *yarn* sta upravljalca JavaScript paketov (knjižnic) in za delovanje potrebujeta *node*.

Kot razvijalec morate torej najprej instalirati *node* in potem lahko s pomočjo omenjenih upravljalcev pridete do »neskončnega« nabora odprtokodnih knjižnic.

Kot poznamo v razvojnih orodjih za Java/C++ koncept projektov, tako imamo v svetu JavaScript razvoja projekte, za katere skrbi *webpack.* *webpack* je namreč eno izmed JavaScript/node orodij, ki ga uporabite, da množico vaših .js, .html, .css, .gif… in kar je še raznoraznih datotek, združi v paket (ali pakete), ki jih nato prepišete na spletni strežnik kot aplikacijo.

4.4 In šele sedaj pride na vrsto Vue

Ko torej poznate prej omenjena orodja, lahko začete razvoj tudi z Vue.

4.5 Še posebno opozorilo

Svet JavaScript je tako ploden, da orodja nastajajo na tedenski bazi. To je zagotovo prekletstvo, a v tem trenutku izhoda ni. Ko stopite v ta svet, se pripravite na vrtinec novosti, izmed katerih se mnoge uveljavijo, zasvetijo in potem hitro ugasnejo. Nikoli pa ne veste, kdaj je pravi čas, da »skočite na vlak« in kdaj je dobro počakati, ter videti, ali bo sprememba uspela, ali ne. To torej pomeni, da se lahko nabor orodij npm/webpack že jutri zamenja za kaj boljšega, novejšega.

Zagotovo pa se lahko zanesete na dejstvo, da bodo ostali *HTML*, *CSS*, *JavaScript* in *node*. Vse ostalo je lahko že jutri zgodovina, vključno z Vue.

5 LITERATURA

[1] Object-oriented programming, https://en.wikipedia.org/wiki/Object-oriented_programming, obiskano 5.5.2019

[2] Microsoft Windows history, https://www.computerhope.com/history/windows.htm, obiskano 27.4.2019

[3] Vue.js - The Progressive JavaScript Framework, https://vuejs.org/, obiskano 5.5.2019

[4] A Brief History of JavaScript, https://auth0.com/blog/a-brief-history-of-javascript/, obiskano 5.5.2019

[5] Difference between Java and JavaScript, https://www.geeksforgeeks.org/difference-between-java-and-javascript/, obiskano 27.4.2019

[6] JavaScript Versions, https://www.w3schools.com/js/js_versions.asp, obiskano 27.4.2019

[7] You wanted a banana but you got a gorilla holding the banana , https://www.johndcook.com/blog/2011/07/19/you-wanted-banana/, obiskano 27.4.2019

[8] Angular vs React vs Vue: Which Framework to Choose in 2019 , https://www.codeinwp.com/blog/angular-vs-vue-vs-react/, obiskano 27.4.2019

OTS 2019
SODOBNE INFORMACIJSKE TEHNOLOGIJE IN STORITVE
M. Heričko in K. Kous (ur.)

Univerzitetna založba
Univerze v Mariboru

VIZUALIZACIJA PODATKOV V KNJIŽNICI REACT, PRAKTIČNE IZKUŠNJE

ALEN RAJŠP, GREGOR JOŠT, VIKTOR TANESKI, SAŠA KUHAR, LUKA PAVLIČ

Povzetek: *React je ena izmed treh najbolj priljubljenih knjižnic za razvoj enostranskih spletnih aplikacij. Omogoča tudi implementacijo ponovno uporabnih gradnikov uporabniškega vmesnika. Dodatne izzive in omejitve med razvojem predstavlja uporaba zunanjih knjižnic in ogrodij. To še posebej velja v primeru, ko knjižnico React uporabljamo v kombinaciji z enim izmed meta-programskih JavaScript jezikov (npr. TypeScript). Prispevek predstavlja možnosti uporabe ogrodja React, s tem povezane dobre prakse in možnosti učinkovite povezave s tretjimi knjižnicami. Podaja praktične izkušnje in izzive pri implementaciji tipične aplikacije za vizualizacijo podatkov. Omejili smo se na knjižnice za prikaz grafov. Demonstrirali smo postopek izbora knjižnice, njihove vključitve ter naše izkušnje in dobre prakse pri uporabi izbrane odprtokodne knjižnice JavaScript. Prispevek opisuje specifike, ki jih morajo izpolnjevati povezani in zaledni sistemi, s katerimi se aplikacije s knjižnico React povezujejo.*

Ključne besede: *React, vizualizacija podatkov, grafi, integracija knjižnic, spletne aplikacije*

NASLOVI AVTORJEV: Alen Rajšp, asistent, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Maribor, Slovenija, e-pošta: alen.rajsp@um.si. dr. Gregor Jošt, asistent, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Maribor, Slovenija, e-pošta: gregor.jost@um.si. Viktor Taneski, asistent, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Maribor, Slovenija, e-pošta: viktor.taneski@um.si. Saša Kuhar, asistentka, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Maribor, Slovenija, e-pošta: sasa.kuhar@um.si. dr. Luka Pavlič, docent, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Maribor, Slovenija, e-pošta: luka.pavlic@um.si.

https://doi.org/10.18690/978-961-286-282-4.27 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 UVOD

Splet je postal de-facto vodilna platforma prijaznih uporabniških vmesnikov. Spletne aplikacije niso več okorne utripajoče spletne strani, ki ob vsakem kliku uporabniku preko protokola za prenos hiperbesedila pretočijo celotno spletno stran. Konceptov, ki so bili mišljeni za spletne strani, več ne zlorabljamo za uporabniško prijazne spletne aplikacije. Namesto tega so spletne aplikacije postale točno to – aplikacije. Le-te delujejo znotraj okolja spletnega brskalnika ter samostojno komunicirajo z zalednim sistemom. Tehnološke rešitve, kot so jezik JavaScript, vmesniki REST in format JSON so osnovni gradniki takšnih aplikacij. A za hiter in relativno vzdržen razvoj spletnih aplikacij je ta nabor premajhen. Brez uporabe specializiranih knjižnic (npr. React.js) ali ogrodij (npr. Angular) se danes skoraj več ne moremo lotiti razvoja t.i. enostranske spletne aplikacije. Knjižnice in ogrodja razvijalcem ponujajo in olajšajo razvoj celotnih delov enostranskih spletnih aplikacij. Hkrati pa prinašajo tudi dodatne izzive, kot je npr. sobivanje »klasičnih« HTML ali JavaScript vsebin in komponent v širši aplikaciji, narejeni na osnovi tretjih knjižnic in ogrodij.

V članku predstavljamo naše izkušnje z gradnjo enostranske spletne aplikacije. Ena izmed glavnih funkcionalnosti aplikacije je ustrezna vizualizacija podatkov, ki jih aplikacija prejema iz zalednega sistema. Vizualizacijo podatkov smo dosegli s pomočjo tretje knjižnice, ki jo je bilo potrebno prilagoditi gostitelju – knjižnici React. V tem članku povzemamo s tem povezane izzive ter zaznane dobre prakse. V naslednjem poglavju predstavimo glavne značilnosti knjižnice React. V poglavju 3 predstavimo postopek izbora in izbrano knjižnico za vizualizacijo podatkov. Pred zaključkom članka povzamemo še glavne izzive in rešitve pri vključevanju knjižnice v končno rešitev.

2 KNJIŽNICA REACT.JS

React.js (v nadaljevanju React) je JavaScript knjižnica za razvoj uporabniških vmesnikov. Za razvoj in vzdrževanje skrbijo v podjetju Facebook. Podjetje je knjižnico prvič uporabilo leta 2011 za prikaz novic na njihovem družabnem omrežju [1]. Leta 2012 so jo vključili tudi v družabno omrežje Instagram [1], ki prav tako spada pod okrilje podjetja Facebook. Knjižnica je postala odprtokodna leta 2013, od takrat naprej pa za njen razvoj poleg razvijalcev omrežij Facebook in Instagram skrbi tudi skupnost uporabnikov. Knjižnica React je kmalu postala zelo priljubljena med razvijalci uporabniškega vmesnika spletnih aplikacij [2]. Anketa na spletni strani Stack Overflow [3] za leto 2019 kaže, da je knjižnica React na prvem mestu popularnosti znotraj družine izdelkov. Bolj popularna je tudi od npr. ogrodja Angular.

React tehnološko umeščamo na predstavitveni nivo spletnih aplikacij. Temelji na komponentah. Vsaka komponenta predstavlja samostojen modul, ki upodobi vnaprej definirano strukturo. Module lahko ponovno uporabljamo, npr. gnezdimo. Ker je knjižnica React deklarativna, običajno opredelimo tudi stanje, v katerem se komponenta nahaja in glede na to knjižnica izriše ustrezno strukturo [4]. V tem kontekstu predstavlja stanje vrednosti skozi življenjski cikel posamezne komponente. Vrednosti stanj lahko prenesemo tudi na vse gnezdene komponente [5]. S trenutno različico knjižnice React (16.8.0) je možno uporabiti stanja znotraj komponent, ki so napisane v obliki razredov ali funkcij. Komponente, implementirane kot funkcije, stanj v prejšnji različici niso podpirale in razvijalci so bili primorani definirati razrede [6].

Za razliko od ostalih JavaScript knjižnic ali ogrodij, React ne deluje neposredno nad objektnim modelom dokumenta (angl. Document Object Model, v nadaljevanju DOM). DOM predstavlja vmesnik, ki skriptam dovoljuje dinamični dostop in posodobitev vsebine, strukture ter stila dokumenta. Ker je DOM manipulacija neučinkovita, deluje React nad ti. virtualnim DOM (v nadaljevanju VDOM) [7]. VDOM hrani virtualno predstavitev uporabniškega vmesnika v pomnilniku in ko se spremeni stanje komponente, se posodobi tudi VDOM, React pa primerja prejšnjo in trenutno različico VDOM-a. Glede na rezultat primerjave se VDOM nato sinhronizira s strukturo DOM, kar posodobi uporabniški vmesnik. Na takšen način se pri izrisu posodobijo samo deli uporabniškega vmesnika, pri katerih so bile zaznane spremembe. Le-to običajno pomeni minimalno poseganje v DOM, zaradi česar je manipulacija VDOM hitrejša in bolj učinkovita kot manipulacija DOM [5, 8].

Poglavitna značilnost knjižnice React je tudi neločevanje logike aplikacije od njene predstavitve v različne datoteke. V ta namen se uporablja razširitev JSX, ki predstavlja sintaktično razširitev programskega jezika JavaScript. Namenjen definiranju elementov. Izsek kode v nadaljevanju demonstrira JSX sintakso (Izvorna koda 1: Primer sintakse JSX).

```
const naslov = <h1> Vizualizacija podatkov v knjižnici React</h1>
```

Izvorna koda 1: Primer sintakse JSX

Kot je razvidno, JSX torej omogoča pisanje kode JavaScript, ki je vizualno zelo podobna označevalnemu jeziku HTML. Ker je to še vedno JavaScript, uporaba rezerviranih besed ni dovoljena (npr. *for* ali *class*; alternativi sta *htlmFor* in *className*). Vsa koda, napisana v JSX, se nato s pomočjo Babel.js (prevajalnik JavaScript) prevede v sintakso React.createElement, ki skrbi za ustvarjanje novih elementov. Razvoj elementov React je sicer možen tudi brez uporabe JSX, vendar je uporaba JSX priporočljiva. Omogoča namreč lažji pregled kode, obenem pa React tudi lažje prikaže morebitne napake ali opozorila glede sintakse [9]. Slika 1 prikazuje sintakso JSX (levi del slike) in prevedeno kodo (desni del).

```
<div>
  <h4 id="vizualizacijaPodatkov">
    <strong>Vizualizacija podatkov v knjižnici React, praktične
izkušnje</strong>
    <br />
      Alen Rajšp, Gregor Jošt, Viktor Taneski, Saša Kuhar, Luka
Pavlič
  </h4>
  <p>
    React je ena izmed treh najbolj priljubljenih knjižnic za
razvoj enostranskih spletnih aplikacij (angl. frontend).
Omogoča tudi implementacijo ponovno uporabnih gradnikov
uporabniškega vmesnika. Dodatne izzive in omejitve predstavlja
uporaba zunanjih knjižnic in ogrodij. To še posebej velja v
primeru, ko knjižnico React uporabljamo v kombinaciji z enim
izmed meta-programskih Javascript jezikov (npr. TypeScript).
  </p>
</div>
```

```
"use strict";

React.createElement("div", null, React.createElement("h4", {
  id: "vizualizacijaPodatkov"
}, React.createElement("strong", null, "Vizualizacija podatkov
v knji\u017Enici React, prakti\u010Dne izku\u0161nje"),
React.createElement("br", null), "Alen Raj\u0161p, Gregor
Jo\u0161t, Viktor Taneski, Sa\u0161a Kuhar, Luka Pavli\u010D"),
React.createElement("p", null, "React je ena izmed treh najbolj
priljubljenih knji\u017Enic za razvoj enostranskih spletnih
aplikacij (angl. frontend). Omogo\u010Da tudi implementacijo
ponovno uporabnih gradnikov uporabni\u0161kega vmesnika.
Dodatne izzive in omejitve predstavlja uporaba zunanjih
knji\u017Enic in ogrodij. To \u0161e posebej velja v primeru,
ko knji\u017Enico React uporabljamo v kombinaciji z enim izmed
meta-programskih Javascript jezikov (npr. TypeScript).");
```

Slika 1: Babel.js pretvorba JSX kode v React.createElemet() klice

Logiko in prikaz komponente definiramo v eni datoteki, kot je razvidno iz primera v nadaljevanju (Izvorna koda 2).

```
const App = () => {
  const [show, setShow] = useState(false);
  useEffect(() => {
    setTimeout(() => {
      setShow(true);
    }, 3000);
  }, []);
  return (
    <>
      {!show && <div>Nalagam...</div>}
      {show &&
        <div>
          <h4 id="vizualizacijaPodatkov">
            <strong>
              Vizualizacija podatkov v knjižnici React, praktične izkušnje
            </strong>
            <br />
            Alen Rajšp, Gregor Jošt, Viktor Taneski, Saša Kuhar, Luka Pavlič
          </h4>
          <p>
            React je ena izmed treh najbolj priljubljenih knjižnic za razvoj enostranskih
spletnih aplikacij (angl. frontend). Omogoča tudi implementacijo ponovno uporabnih
gradnikov uporabniškega vmesnika. Dodatne izzive in omejitve predstavlja uporaba zunanjih
knjižnic in ogrodij. To še posebej velja v primeru, ko knjižnico React uporabljamo v
kombinaciji z enim izmed meta-programskih JavaScript jezikov (npr. TypeScript).
          </p>
        </div>}
    </>
  );
}
```

Izvorna koda 2: Logika in prikaz komponente v React-u

Primer uporablja aktualen pristop k opredelitvi stanja komponente (funkcija *useState*), ki v tem primeru najprej izpiše »Nalagam…« in čez 3 sekunde posodobi uporabniški vmesnik tako, da izpiše povzetek članka. Vsa sintaksa v *return* delu funkcije predstavlja JSX.

React-a ne uvrščamo med ogrodja, ampak knjižnice. Zato moramo ob uporabi običajno vključiti še dodatne knjižnice, ki skrbijo npr. za usmerjanje med posameznimi stranmi, pridobivanje podatkov, vodenje stanja skozi celotno aplikacijo in nenazadnje tudi za vizualizacijo podatkov. Poseben izziv zato predstavlja tudi odločanje, katero izmed množice knjižnic komplementarno uporabiti pri razvoju s pomočjo knjižnice React. Proces izbire in praktične izkušnje pri tem so predstavljeni v nadaljevanju članka.

3 PRIMERJAVA IN IZBOR KNJIŽNIC ZA VIZUALIZACIJO PODATKOV

Praktična rešitev, iz katere izhaja ta članek, omogoča kompleksne vizualizacije povezanih podatkov. Posledično je prikaz podatkov centralna funkcionalnost. Izbira pravih orodij je zato zelo pomembna, saj je bilo med drugim potrebno zagotoviti čim boljšo uporabniško izkušnjo.

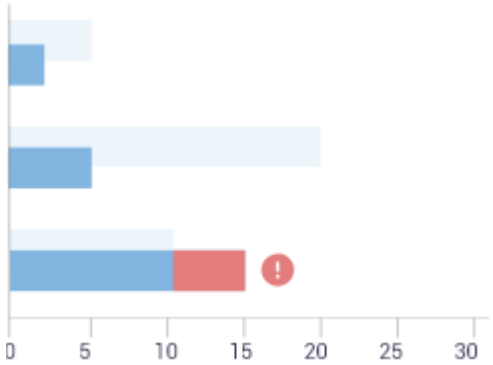
Iz vidika razvijalca je zmožnost prikaza podatkov enako koristna, kot izdelava interaktivnih spletnih strani, saj ti pogosto sodita skupaj. Ker jezik JavaScript še naprej pridobiva na popularnosti na področju vizualizacije podatkov, je trg že dokaj bogat s knjižnicami za ustvarjanje spletnih grafov za prikaz podatkov.

Za naše potrebe je bilo razumevanje delovanja in načina uporabe najbolj popularnih vizualizacijskih JavaScript knjižnic ključno. Izbrati je bilo potrebno tisto, ki bi najbolje ustrezala zahtevam končne rešitve. Pri tem smo upoštevali več dejavnikov. Med drugimi: (1) nabor grafov, ki jih potrebujemo za prikaz podatkov, (2) količino podatkov, ki jo je potrebno prikazati, (3) pravilen prikaz grafov na različnih napravah, (4) podpora brskalnikov, (5) potreba po dodatnem prilagajanju grafov itn.

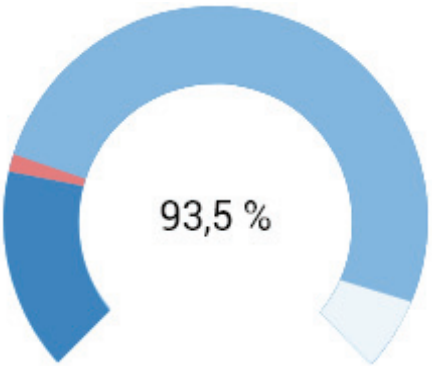
V nadaljevanju tega poglavja podrobneje opišemo zahteve glede prikazovanja podatkov, metodo izbora ustrezne knjižnice ter primerjavo konkretnih knjižnic.

3.1 Zahteve

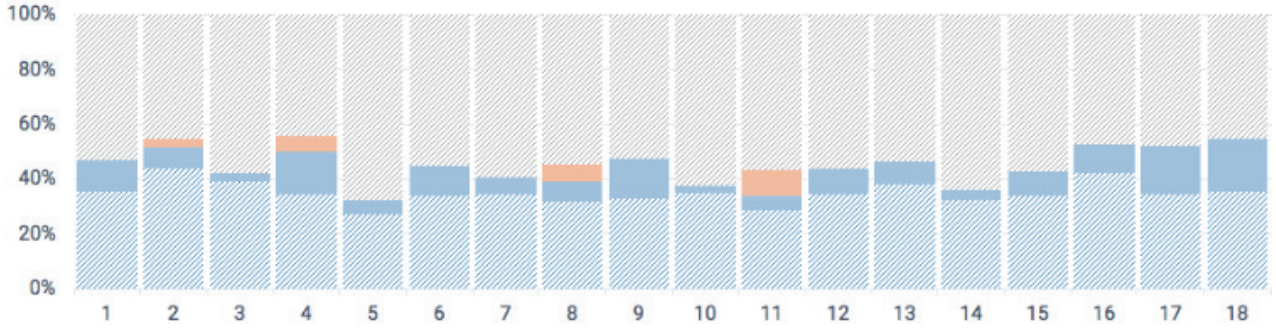
Zaradi specifičnosti naše rešitve in večje količine vizualiziranih podatkov je bila raba aplikacije predvidena predvsem na napravah z večjim zaslonom (npr. računalnikih ali tablicah in ne na mobilnih telefonih). Zahteve glede vizualizacije podatkov so bile podane vizualno s pomočjo narisanega prototipa uporabniškega vmesnika. Ob nekaterih dokaj klasičnih stolpčnih, vrstičnih, linijskih in naloženih diagramih, so bili predvideni še takšni z visoko stopnjo prilagajanja: prekrivajoči se stolpčni oz. vrstični diagram (Slika 2), stiliziran diagram merilnika hitrosti (Slika 3), naložen stolpčni (Slika 4) in površinski grafikon (Slika 5), stolpčni in površinski diagrami z vizualno in številčno označenimi dodatnimi območji (Slika 6) ter grafikoni z dodanimi ikonami in zapisi na območjih izrisa podatkov (Slika 7).



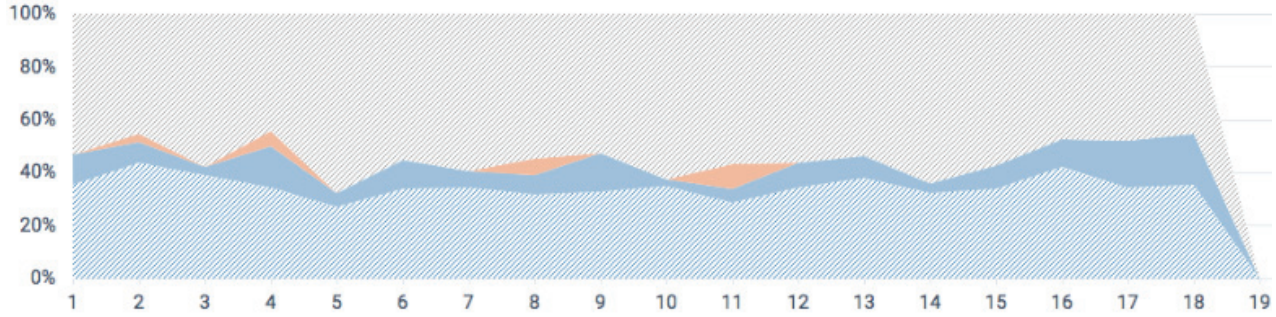
Slika 2: Prekrivajoči vrstični diagram z dodano ikono



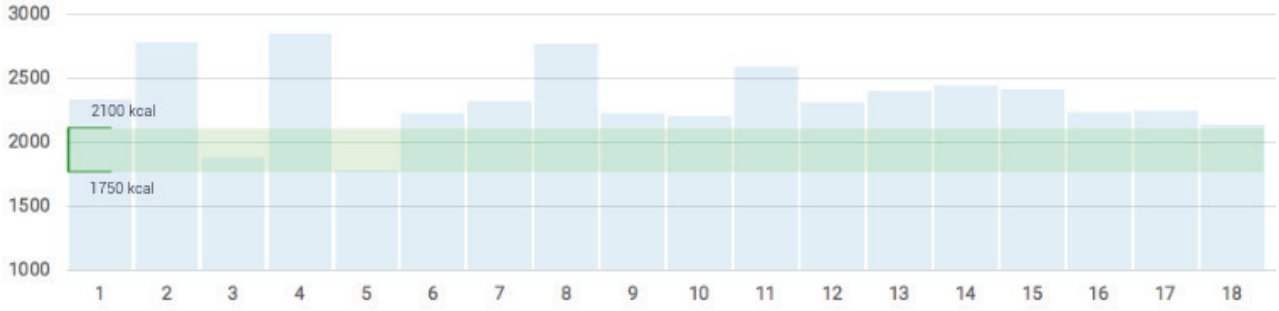
Slika 3: Stiliziran diagram merilnika hitrosti



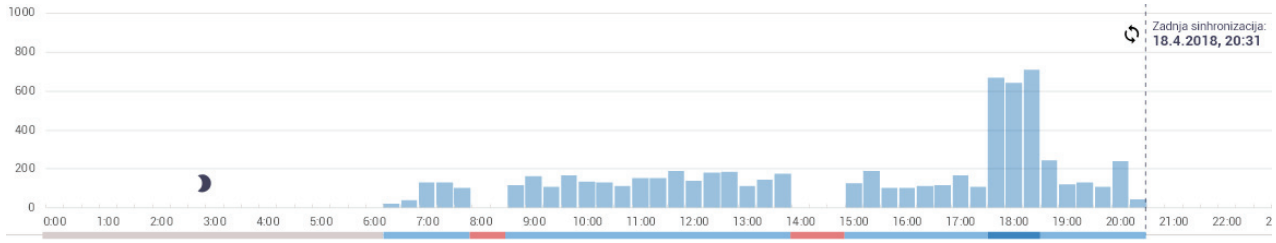
Slika 4: Naložen stolpčni diagram



Slika 5: Naložen površinski diagram



Slika 6: Stolpčni diagram z dodatno označenim območjem

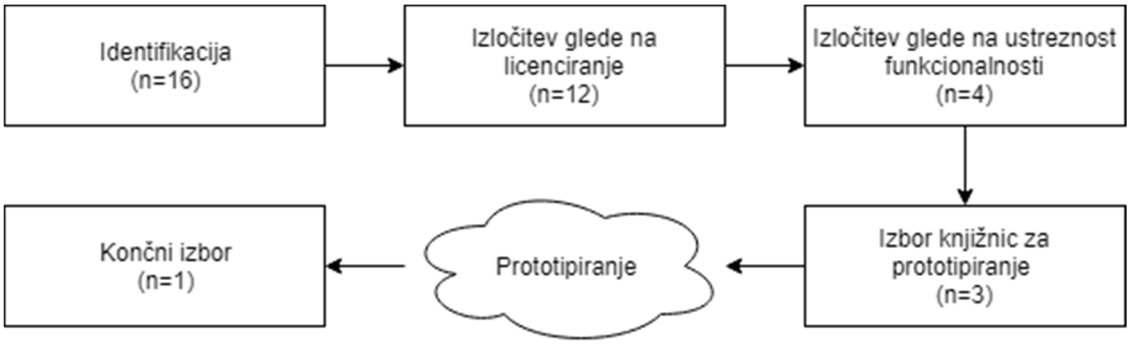


Slika 7: Diagram z dodanimi ikonami in zapisi na območju izrisa podatkov

3.2 Metoda izbora in primerjava izbranih knjižnic

Ker smo želeli dopustiti možnost posodabljanja sistema in vnašanja morebitnih novih funkcionalnosti, je bil cilj temu primerno izbrati tudi knjižnico za vizualizacijo podatkov. Pomembno je bilo, da bo izbrana knjižnica omogočala zahtevane tipe grafov, bo odprtokodna, brezplačna za uporabo v komercialni aplikaciji, dobro dokumentirana, redno posodobljena, čim preprostejša za uporabo ter, da bo omogočala prilagoditve.

Kot je prikazano na sliki (Slika 8), smo sledili šest-stopenjskemu načinu izbora knjižnice:



Slika 8: Postopek izbora primernega orodja

Identifikacijo knjižnic smo opravili s pomočjo repozitorija programske kode Github [10]. Identificirali smo 16 različnih knjižnic za vizualizacijo grafov. Začetni izbor smo zožili tako, da smo najprej izločili knjižnice, pri katerih koda ni prosto dostopna. Nato smo odstranili vse, pri katerih je potrebno plačati rabo v komercialnih aplikacijah.

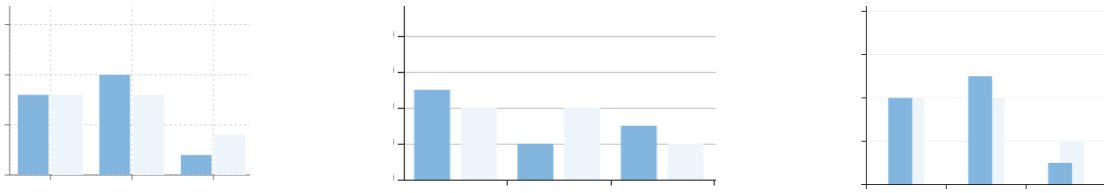
Začetni izbor smo nato primerjali glede na podprte tipe grafov, kjer smo se pri identifikaciji zgledovali po [11]. Identificirali smo 13 osnovnih tipov grafov, ki so se pojavljali v knjižnicah (podčrtani so tisti, ki so bili nujni za izris v konkretnem primeru): merilnik, stolpčni (navaden, naložen), črtni (navaden, površinski), radar, kolobarni, tortni, raztreseni, lijak, ganttov, kvartilni, polarni in mehurčni. Glede na zahteve, smo identificirali tudi 5 specifičnih funkcionalnosti, ki smo jih potrebovali: prekrivajoče regije, ikone na grafih, kombinacije grafov in vertikalne / horizontalne črte z označbami. Tako smo izbrali 4 potencialno primerne knjižnice (C3.js, D3.js, Recharts in Incubator echarts), kot je prikazano spodaj (Tabela 16).

Tabela 16: Identificirane knjižnice

	Knjižnica	Št. Github zvezd (11. 2018)	Licenca	Zahtevnost implementacije	Prilagodljivost	Št. manjkajočih tipov grafov	Št manjkajočih funkcionalnosti
	Chart.JS [12]	40361	MIT [13]	nizka	nizka	3	4
	Chartist.JS [14]	11231	MIT	nizka	nizka	2	5
✔	**C3.js** [15]	8089	MIT	nizka	srednja	2	1
✔	**D3.js** [16]	80246	BSD-3 [17]	visoka	visoka	0	0
✔	**Recharts** [18]	10318	MIT	nizka	srednja	2	2
	Plotly.js [19]	9282	MIT	nizka	srednja	2	2
	NVD3.js [20]	6730	Apache 2.0 [21]	nizka	srednja	3	5
	Victory [22]	6548	MIT	nizka	nizka	3	4
	Vx [23]	5311	MIT	srednja	visoka	3	5
	React-Vis [24]	4696	MIT	nizka	nizka	3	4
	Britecharts [25]	3184	Apache 2.0	nizka	nizka	3	5
✔	**Incubator echarts** [26]	31145	Apache 2.0	srednja	srednja	2	2
	AnyChart [27]	188	lastniška	Knjižnice plačljive za komercialno rabo.			
	Amcharts [28]	373	lastniška				
	CanvasJS [29]	9	lastniška				
	Google Charts [30]	763	lastniška				

V fazo prototipiranja smo vključili vse razen D3.js, ki se je izkazala za preveč kompleksno za začetno uporabo.

Namen te faze je bila implementacija kompleksnejših tipov grafov na statičnih spletnih straneh. Primer poizkusa izrisa prekrivajočega se stolpčnega grafa je viden na Slika 9. Tudi v preostalih primerih je bilo s knjižnico C3.js mogoče doseči največji približek zahtevanemu izgledu, glede na zahteve.



Slika 9: Primerjava izrisov prekrivajočega stolpčnega grafa v treh izbranih knjižnicah

4 VIZUALIZACIJA PODATKOV S KNJIŽNICO C3.JS

C3.js [15] je knjižnica za izris grafov, ki je izdana je pod licenco MIT [13]. To omogoča, da jo brez nadomestila uporabimo tudi v komercialnih projektih. Temelji na knjižnici D3.js [16]. Knjižnica je bila prvič izdana 2014 in je do sedaj doživela že 104 izdaje, kar nakazuje, da je aktualna in sprotno vzdrževana. Grafi so, kot pri knjižnici D3.js, izrisani v formatu SVG. Za uporabo moramo vključiti njeno stilno predlogo (*css*), knjižnico D3.js ter JavaScript datoteko knjižnice.

Knjižnica upodobljene podatke tematsko podrobneje loči. Npr. na podatke (data), osi (axis), legendo (legend), mrežo (grid), regije (region) idr., pri čemer vsaka kategorija predstavlja objekt, ki je vključen v krovni JSON objekt grafa.

Obstoječe grafe je mogoče tudi posodabljati, brez ustvarjanja novega primerka grafa. Preprost primer je viden v nadaljevanju (Izvorna koda 3).

```
<div id="graf"></div>
<script>
    var graf = c3.generate({
        data: {
            columns: [['naravna stevila', 1, 2, 3, 4, 5], ...]
        }
        bindto: '#graf',
    });
    graf.load({
        columns: [['postevanka 2', 2, 4, 6, 8, 10], ...]
    });
</script>
```

Izvorna koda 3: Preprost primer C3.js

C3.js pri izrisu grafov neposredno dostopa do DOM. V povezavi s knjižnico React (ki dostopa do VDOM) to predstavlja določeno tveganje, saj obe knjižnici manipulirata z istim dokumentom. V nadaljevanju (poglavje Izzivi in rešitve) bomo naslovili tudi ta tveganja in podali nekaj usmeritev.

4.1 Prilagoditve grafov

Osnovne grafe je bilo potrebno prirediti, da bodo le-ti vizualno in funkcionalno ustrezali zahtevam. Prilagoditve so se nanašale tako na stil kot tudi funkcionalnosti. Stilske prilagoditve grafov smo dosegli večinoma z modifikacijo stilnih predlog (*css*).

Izris npr. grafa merilnika (ki v C3.js ni predviden) je zahteval (Slika 10), da del tortnega grafa napolnimo s prosojno barvo (2), kar smo storili tako, da smo vanj vstavili navidezne podatke, ki imajo isto barvo kot ozadje, nato s pomočjo CSS transformacij graf rotirali (3), napise pa rotirali v obratni smeri, da tako ohranijo normalno orientacijo (4).

(1) začetni graf brez prilagoditev | (2) vstavljanje navideznih podatkov | (3) CSS rotacija celotnega grafa | (4) CSS rotacija napisov in oznak v obratni smeri

Slika 10: Transformacija tortnega grafa v merilnik

Na mestih, kjer prilagoditve CSS niso zadoščale, smo prilagodili še metode knjižnice C3.js (npr. *init* – se proži ob inicializaciji, *onrendered* – se proži ob ponovnem risanju grafa, *onmouseover* – se proži ob vstopu kurzorja v območje grafa, *onmouseout* – se proži ob zapustitvi kurzorja iz območja grafa, *onresize* – se proži ob spreminjanju velikosti okna brskalnika). Pri slednjem načinu uporabe smo v metodah uporabljali funkcionalnosti krovne knjižnice D3.js. Transformacije CSS so bile zmogljivostno manj zahtevne saj so se izvedle ob izrisu grafov. Po drugi strani pa je bila raba metod veliko zahtevnejša. Naknadne transformacije na grafih so se lahko izvedle šele po prvotnem izrisu grafa, kar pomeni, da se je vsak graf za prikaz moral vsaj dvakrat deloma izrisati. V primeru merilnika smo potrebovali naknadne transformacije, saj zapis besedila na zgornjih grafih v primeru merilnika ni statičen in je odvisen od podatkov (njegov namen v prvotnem tortnem grafu je, da služi kot naslov, ki se ne spreminja).

4.2 Izzivi in rešitve

Za vključitev knjižnice C3.js v našo rešitev, smo potrebovali komponento, ki predstavlja ovojnico, saj tako React kot C3.js dostopata do istega DOM. Zaradi uporabe meta-programskega jezika TypeScript smo pri uporabi obstoječih ovojnic [31, 32] naleteli na izzive, saj ovojnice niso nudile podpore TypeScript. Kljub temu, da smo omejitev zaobšli z izključitvijo TypeScript preverjanja pri vključitvi ovojnic, smo naleteli na težave, povezane z verzijam knjižnic C3.js in D3.js. Ovojnici sta bili namreč narejeni za zastareli različici omenjenih knjižnic, posodobitev zgolj teh odvisnosti na novejše različice pa je povzročila napake pri prikazovanju grafov. To nas je med drugim privedlo do odločitve, da smo razvili lastno ovojnico za izris grafov, kjer smo zagotovili podporo TypeScript in najnovejšim različicam knjižnic za izris grafov.

C3.js je namenjen klasičnim spletnim stranem, ki uporabljajo DOM. V knjižnici je predvideno, da se veže na DOM element (npr. *div*), katerega *id* je znan. V Reactu takšen način naslavljanja elementov ni v celoti podprt; v ta namen je predvidena uporaba *useRef*, ki omogoča sklic na komponento (v našem primeru predstavlja *div* vsebnik za prikaz grafa). Pristop z rabo referenc pri razvoju lastne ovojnice je viden v nadaljevanju (Izvorna koda 4).

```
// Kreiranje reference
const refNaGraf: React.RefObject<HTMLDivElement> = useRef<HTMLDivElement>(null);
...
graf.current = c3.generate({
                bindto: refChart.current, //referenca nadomešča privzeti način
pripenjanja preko id-ja
...
});
...
// Pripenjanje reference na komponento
return <div ref={refNaGraf} />;
```

Izvorna koda 4: Kreiranje reference in pripenjanje reference za direkten dostop do elementa DOM ob nastanku grafa

Nepazljivost pri nalaganju in sproščanju podatkov v knjižnici C3.js lahko v primeru nepravilne rabe proži napako. Zato je bilo potrebno pozorno upravljati z življenjskim ciklom grafa, ki je vezan na življenjski cikel vsebnika (komponente – ovojnice). Potrebno je bilo zagotoviti, da se ob kreaciji/priklopu (angl. mount) komponente, ustvari celoten graf, ob posodabljanju (angl. updating) pa spremenijo le podatki. In sicer na takšen način, da se sprostijo stari podatki (angl. unload) in naložijo novi (angl. load). Ob odklapljanju (angl. unmount)

komponente je bilo potrebno zagotoviti, da se podatki iz grafa najprej sprostijo, šele nato se lahko komponenta odklopi.

Nenazadnje pa smo se soočili tudi s težavo, vezano na obliko podatkov iz končnih točk REST. Struktura podatkov ni bila prilagojena uporabi v grafih, saj smo podatke prikazovali tudi v tabelarični obliki. V ta namen smo bili primorani strukturo podatkov prilagoditi glede na tip grafov, kar je predstavljalo svojevrsten izziv.

5 ZAKLJUČEK

React je trenutno najbolj priljubljena knjižnica za razvoj enostranskih spletnih aplikacij. Omogoča tudi implementacijo ponovno uporabnih gradnikov uporabniškega vmesnika. Dodatne izzive in omejitve med razvojem predstavlja uporaba zunanjih knjižnic in ogrodij. V članku smo predstavili možnosti uporabe ogrodja React, s tem povezane dobre prakse in možnosti učinkovite povezave s tretjimi knjižnicami. Podali smo praktične izkušnje in izzive pri implementaciji tipične aplikacije za vizualizacijo podatkov, kjer smo se omejili na knjižnice za prikaz grafov. Demonstrirali smo postopek izbora knjižnice, njihove vključitve ter naše izkušnje in dobre prakse pri uporabi izbrane odprtokodne JavaScript knjižnice.

6 LITERATURA

[1] Education Ecosystem, „React.js History - Education Ecosystem“, 2018. [Na spletu]. Dostopno: https://www.education-ecosystem.com/guides/programming/react-js/history. [Dostopano: 22-maj-2019].

[2] O. Williams, „Making the business case for React in 2019 – LogRocket“, 2019. [Na spletu]. Dostopno: https://blog.logrocket.com/making-the-business-case-for-react-in-2019-74463bbb22de. [Dostopano: 22-maj-2019].

[3] Stackoverflow, „Stack Overflow Developer Survey 2019“, 2019. [Na spletu]. Dostopno: https://insights.stackoverflow.com/survey/2019. [Dostopano: 22-maj-2019].

[4] Facebook Inc., „React – A JavaScript library for building user interfaces“. [Na spletu]. Dostopno: https://reactjs.org/. [Dostopano: 22-maj-2019].

[5] M. Abe, „React is a JavaScript Library for Building User Interfaces; Not a Framework Like Angular“. [Na spletu]. Dostopno: https://www.codemag.com/Article/1809041/Demystifying-React. [Dostopano: 22-maj-2019].

[6] Facebook Inc., „Introducing Hooks – React“. [Na spletu]. Dostopno: https://reactjs.org/docs/hooks-intro.html. [Dostopano: 22-maj-2019].

[7] Facebook Inc., „Virtual DOM and Internals – React“. [Na spletu]. Dostopno: https://reactjs.org/docs/faq-internals.html. [Dostopano: 22-maj-2019].

[8] M. Hamedani, „React Virtual DOM Explained in Simple English - Programming with Mosh“, 2018. [Na spletu]. Dostopno: https://programmingwithmosh.com/react/react-virtual-dom-explained/. [Dostopano: 22-maj-2019].

[9] Facebook Inc., „Introducing JSX – React“. [Na spletu]. Dostopno: https://reactjs.org/docs/introducing-jsx.html. [Dostopano: 22-maj-2019].

[10] Microsoft, „GitHub“. [Na spletu]. Dostopno: https://github.com/. [Dostopano: 27-maj-2019].

[11] Wikipedia, „Comparison of JavaScript charting libraries“, 2019. [Na spletu]. Dostopno: https://en.wikipedia.org/wiki/Comparison_of_JavaScript_charting_libraries. [Dostopano: 27-maj-2019].

[12] „Chart.js | Open source HTML5 Charts for your website“. [Na spletu]. Dostopno: https://www.chartjs.org/. [Dostopano: 27-maj-2019].

[13] Opensource.org, „The MIT License | Open Source Initiative“. [Na spletu]. Dostopno: https://opensource.org/licenses/MIT. [Dostopano: 22-maj-2019].

[14] „Chartist - Simple responsive charts“. [Na spletu]. Dostopno: https://gionkunz.github.io/chartist-js/. [Dostopano: 27-maj-2019].

[15] „C3.js | D3-based reusable chart library“. [Na spletu]. Dostopno: https://c3js.org/. [Dostopano: 22-maj-2019].

[16] M. Bostock, „D3.js - Data-Driven Documents“. [Na spletu]. Dostopno: https://d3js.org/. [Dostopano: 22-maj-2019].

[17] „The 3-Clause BSD License | Open Source Initiative“. [Na spletu]. Dostopno: https://opensource.org/licenses/BSD-3-Clause. [Dostopano: 27-maj-2019].

[18] „Recharts“. [Na spletu]. Dostopno: http://recharts.org/en-US/. [Dostopano: 27-maj-2019].

[19] „plotly.js | JavaScript Graphing Library“. [Na spletu]. Dostopno: https://plot.ly/javascript/. [Dostopano: 27-maj-2019].

[20] „NVD3“. [Na spletu]. Dostopno: http://nvd3.org/. [Dostopano: 27-maj-2019].

[21] ASF, „Apache License, Version 2.0“. [Na spletu]. Dostopno: https://www.apache.org/licenses/LICENSE-2.0.html. [Dostopano: 27-maj-2019].

[22] „Victory | VictoryChart“. [Na spletu]. Dostopno: https://formidable.com/open-source/victory/. [Dostopano: 27-maj-2019].

[23] „vx | visualization components“. [Na spletu]. Dostopno: https://vx-demo.now.sh/. [Dostopano: 27-maj-2019].

[24] „react-vis“. [Na spletu]. Dostopno: https://uber.github.io/react-vis/. [Dostopano: 27-maj-2019].

[25] „Britecharts - D3.js based charting library of reusable components“. [Na spletu]. Dostopno: http://eventbrite.github.io/britecharts/. [Dostopano: 27-maj-2019].

[26] „ECharts“. [Na spletu]. Dostopno: http://echarts.apache.org/. [Dostopano: 27-maj-2019].

[27] „AnyChart is a lightweight and robust JavaScript charting library“. [Na spletu]. Dostopno: https://www.anychart.com/. [Dostopano: 27-maj-2019].

[28] „JavaScript Charts & Maps - amCharts“. [Na spletu]. Dostopno: https://www.amcharts.com/. [Dostopano: 27-maj-2019].

[29] „Beautiful HTML5 JavaScript Charts | CanvasJS“. [Na spletu]. Dostopno: https://canvasjs.com/. [Dostopano: 27-maj-2019].

[30] „Charts | Google Developers“. [Na spletu]. Dostopno: https://developers.google.com/chart/. [Dostopano: 27-maj-2019].

[31] C. X. Su, „GitHub - bcbcarl/react-c3js: React component for C3.js“. [Na spletu]. Dostopno: https://github.com/bcbcarl/react-c3js. [Dostopano: 22-maj-2019].

[32] C. Reichert, „GitHub - CodyReichert/react-c3: A React.js Component wrapper for creating graphs using c3.js“. [Na spletu]. Dostopno: https://github.com/CodyReichert/react-c3. [Dostopano: 22-maj-2019].

OTS 2019
ADVANCED INFORMATION TECHNOLOGY AND SERVICES
M. Heričko in K. Kous (eds.)

University of Maribor Press

AN INSIDE LOOK INTO FLUTTER DEVELOPMENT

BLAGOJ SOKLEVSKI, ANDREJ KLINE

Abstract: *What is Flutter and how does it work? What is Dart? How to get started with Flutter? These are just some of the questions that will be answered in this paper which will help you understand this new framework and will provide a solid ground to continue learning Flutter. Here we share the inside knowledge of Flutter development which we acquired while developing our iAgent mobile application for car insurance, and last but not least we will share our piece of mind about this new modern framework that shook the mobile development world.*

Key words: *Flutter, Dart, hybrid mobile development, Android, iOS*

CORRESPONDENCE ADDRESS: Blagoj Soklevski, msg life odateam d.o.o., Maribor, Slovenia e-mail: blagoj.soklevski@msg-life.com. Andrej Kline, msg life odateam d.o.o., Maribor, Slovenia e-mail: andrej.kline@msg-life.com.

https://doi.org/10.18690/978-961-286-282-4.28 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 INTRODUCTION

In designing and developing mobile applications we face a lot of challenges, such as wanting to make them with a high-quality look and feel, extending the reach (building and releasing them for the two major platforms iOS and Android), minimizing the time to market and making them affordable in the whole process. Now, we are faced with the dilemma of mobile app development, should we go hybrid and have only one project to maintain but sacrifice performance and usability. Or, should we take the more expensive route and create separate native apps with separate code bases but gain in performance and overall usability but pay the price in maintaining them. Through the years, many frameworks for hybrid mobile development have surfaced, some created by big names, some gained temporary popularity, while others lasted through the years, the list does not stop (Xamarin, PhoneGap, Ionic Framework, React Native, etc.). However, native development always seems to have the upper hand until a truly big player decided to enter the game. With the creation of Flutter (by Google), hybrid development finally stood a chance and seeing how major players like Alibaba and Google have adopted it (just as Google rewriting its Google Ads with Flutter), it seems that this framework has a bright future and is here to stay.

After having an eye on Flutter for some time now, and with the rapid growth of our new ecosystem of insurance suites and accompanying mobile applications under the brand name of "iAgent", we took the opportunity and got our hands dirty by starting the development of one of the mobile application for "iAgent" in Flutter. The application we are developing gives the end user the opportunity to easily purchase a car insurance policy regardless whether the user is on a lunch break, sitting on the bus, or relaxing on the couch at home.

In this article, we will cover everything we have learned regarding Flutter from the first line of code to the release procedure, including which IDE we tried, the pros and cons of developing in Flutter, the learning curve of Dart and the difficulty of changing the mindset from native to hybrid development of mobile applications.

With the new release version of Flutter 1.5 and Dart 2.2 and the idea that you can cover two platforms (Android and iOS) with a single codebase, the temptation becomes even bigger to leave all of the time consuming native programming and just focus on this new open-source mobile application development framework created by Google which has the perks of fast development, expressive and flexible UI, and native performance. Now, lets not fly too close to the sun, we do not live in a perfect world, every now and then you have to look under the hood and write some native code if needed, depending on your use case so do not get rusty in Kotlin, Swift, or Java too fast.

2 FLUTTER

This paragraph will answer questions such as what is Flutter, how does it work, what is the anatomy of the Flutter project, what is Dart and last but not least how to get started. The answers to these questions will help you understand this new framework and will provide a solid background for you to continue learning Flutter.

2.1 What is Flutter and how does it work?

Right off the bat, Flutter is an open-source framework for creating mobile applications, first appears by the name "Sky" in 2015 and ran on the Android operating system. Flutter allows you to build beautiful native apps on iOS and Android from a single codebase [1]. Google markets its creation by these three key points: expressive and flexible UI, fast development and native performance. Native performers may be the best feature because the first hybrid mobile applications that came out were nothing but web pages shown in a Webkit (browser rendering engine), this is a very expensive way of manipulating the DOM paid in performance by the app. Building the JavaScript bridge became a solution for some platforms but not the ideal one. That bridge became the bottleneck because JavaScript code had to be compiled while the application is running which is a big performance issue. Flutter, on the other hand, does not have such problems because it compiles directly to ARM code. A note to take in is that Dart has an AOT (Ahead of Time) and JIT (Just in Time) compilers so in Flutter when you release your app in production the AOT is used but when you are developing JIT is being used [2]. The "hot reload" is responsible for the rapid development, a feature that you cannot leave without after you try it. This feature speeds up the development because you can make changes on the UI and you can see them on the fly, this works by injecting the changed source code files into the running Dart Virtual

Machine (VM), Flutter rebuilds the widget tree, allowing you to quickly view the effects of your changes [3]. Flutter has layered architecture which gives the opportunity for full customization, it is also well stock on UI element but the biggest feature regarding the design is that Flutter provides Material and Cupertino design libraries. With these two libraries, you can make beautiful applications for both Android and iOS users, also, both styles are well documented and have really nice guidelines.

When you look at application programmed in Flutter the first thing you will notice is the style in which it is programmed. In the beginning, it was difficult to change the mindset because UI is typically programmed in an imperative style, so programming in a declarative style came as a little shock. A high chance is that most of the developers who would start to learn Flutter are mobile developers and are used to the imperative style of programming in native Android and iOS.

In the declarative style, view configurations (such as Flutter's Widgets) are only some sort of lightweight blueprints and to change the UI, a Widget triggers a rebuild on itself and constructs a new Widget subtree [4]. In Flutter the UI is created programmatically it does not have a separate file like a ".xml" that Android has or ".xib" for iOS, this means that the state and the view are tightly intertwined which is another fact that helps explain why the declarative style was chosen. In Flutter everything is a Widget (views, buttons, layout, inputs, labels, etc.) and widgets are just Dart classes that know how to describe the view. Knowing this, we can see why Flutter prefers composition over inheritance because by creating new widgets you can reuse them throughout the application [2]. We can divide the Widgets in two main groups ones that extend the StatelessWidget class and the others that extend the StatefullWidget class. However, all widgets regardless of which class they extend must have a build method and should be immutable except for the state objects. Let's start with the StatelessWidget, as the name implies this type of widget can't hold any state and is destroyed when removed from the widget tree. This kind of widget is used when the part of the user interface you are describing does not depend on anything other than the configuration information in the object [5]. On the other hand, you have the StatefullWidget which has a State object and has control over rebuilding the widgets using this methods:

- initState
- didChangeDepencies
- build
- widgetDidUpdate
- setState
- dispose

Flutter has dynamic routing which makes the routing more flexible and fluid, the navigator is the one that allows you to create routes on the fly in your code. Flutter also supports static routing using named routes that need to be defined in your MatherialApp widget [2].

2.2 What is dart Dart?

Dart is an open source programing language which first appeared in 2011, developed by Google with the intended use for creating web apps, hoping to boost performance and productivity. Fast forward to 2019 with its stable release 2.2.0 (as of 26 February 2019) Dart became a language used for developing desktop, server, web, and mobile applications, the accent falling on the mobile development area with the emergence of Flutter.

Dart is easy to learn. A wide range of developers can learn Dart quickly. It is an object-oriented language with classes, single inheritance, lexical scope, top-level functions, and a familiar syntax. Most developers are up and running with Dart in just a few hours [6].

It supports interfaces, mixins, abstract classes, reified generics, static typing, and a sound type system [7].

In Dart, the static typing is optional but it would be better practice to use types because it makes the code safer and IDEs tend to give better warnings and error messages. Besides, if you are coming from a programming language that is statically typed (Java for example) it will be instinctual to use types. An interesting fact is that Dart 2 made the "new" keyword optional, another cool feature is the optional parameters. You can make named optional parameters by wrapping them in curly brackets or positional by wrapping them in box brackets, but you cannot use them both at the same time. Everything you can place in a variable is an object, and every

object is an instance of a class. Even numbers, functions, and null are objects. All objects inherit from the Object class [8].

2.3 What is the anatomy of a Flutter project?

Every framework has a unique anatomy (structure) inside its project, looking inside of a Flutter project anatomy may be a bit confusing because it contains a whole Android and iOS project. If you look at a Flutter project you can see the basic structure in which you will find the subprojects for Android and iOS**.** In some cases, you probably will not even open these subprojects depending on your use cases for the application you are writing. But, if for example, you need to write a platform channel this is the place to do it. For the most part, plugins fix these issues when you need some native power (like accessing the camera) in your application, but Flutter is a young framework and at the moment of writing this article, not everything is covered by plugins, so you may need to get your hands dirty and write some native code. Platform channels and method fix this issue and send messages from Flutter to the native code. Be cautious when you create your Flutter project because the default language for Android is Java and for iOS is Objective C and for example, if you need to write a method channel and you only know Swift you will have a problem. Another example in which you will need to access the Android and iOS project is if you need to add changes to the configuration files ("AndroidManifest.xml" or "Info.plist") like a new permission. The “lib” folder is where you will be spending most of your time because here is where we write dart code for our Flutter app. The last important part of the project structure is the configuration file "pubspec.yaml". Here we specify the name of the app, the description, the version. Also, here we add dependencies and assets such as images or custom funds. Have in mind if you add new dependence you should run the "flutter packages get" command to retrieve them or "flutter packages upgrade" command to update them, but modern IDEs like VS Code or Android Studio do this for you after saving the file or they supply dedicated buttons for these actions.

2.4 How to get started? From installation to release (.apk/.ipa)

This paragraph will cover in short summary the whole process from installing Flutter to the releasing procedure.

Flutter is available for Linux, macOS, and Windows. The installation process is pretty straight forward and well-documented on the Flutter official website. After the installation, you can use the Flutter command-line tools. The first command you should run is flutter doctor, this command checks your environment and displays a report to the terminal window. The Dart SDK is bundled with Flutter; it is not necessary to install Dart separately [9]. Some defenses may be missing so just run flutter doctor and follow the instruction you get, this process may have to be repeated several times. Next comes the platform specific setup such as installing Xcode for iOS and the latest Android SDK, Platform-Tools, and build-Tools for Android. After completing the installation process you can create a project by running the command "flutter create my_app_name", this command creates a template project provided by Flutter and with this you are testing whether everything is working right. Now, either start an emulator or connect a device, navigate to the project and run the command "flutter run". If everything is ok you should see the app load. Here, you should take note that by running the “flutter create” command without providing additional flags the default template creates a project which for Android uses Java and for iOS uses Objective C. For example, if you want to use Kotlin or Swift you should run the command with the flags "i" for iOS and "a" for android as such "flutter create -i swift -a kotlin my_app_name". For IDE you can choose from Android Studio, IntelliJ IDEA (Community / Ultimate) or Visual Studio Code, whichever option you choose you cannot be wrong. They all have good plugins for Dart and Flutter, but from our experience we suggest if the goal is to learn Flutter, VS Code is the better choice because you do not get that much help from the IDE. However, in terms of better productivity, we suggest Android Studio because of the additional tooling supplied such as auto import, Flutter Inspector, Flutter Outline, Flutter Performance and let’s not forget that if you are working on macOS you can even open the iOS Simulator from the IDE. Now comes the growing task of creating your software which we will not cover so let‘s fast forward to the release procedure. For Android it is pretty straight forward, just run "flutter build apk" in the terminal and an “.apk” file will be created. For iOS let’s not forget that without a macOS machine, Xcode and a valid Apple developer account you cannot complete the process. Let's assume we have all of that covered, the next step would be to run the "flutter build clean" that "flutter build ios" and now you have to crack open

the Xcode. In Xcode select "Generic iOS Device" then click "Product>Archive" when that is finished a modal dialog will appear with your archives, select you latest archive and click the "Distribute App" button and from there onward you choose your way of distribution. In a short summary, you will need to get started with Flutter if you want to learn more, just open the official flutter web page and read up.

3 IAGENT APP

As always after toying around and creating side projects while testing the features of a new framework or programming language, it's time to see what it can bring to the table so you have to get your hands dirty with the real deal and create a fully functional app. So we decided to create our iAgent app for car insurance with Flutter, so you can say the development of this app is responsible for the insight we got. This application is still a work in progress and it is going through a lot of changes, now we will describe the current state. Only Material design has been used for the current version and our goal is to have a unique design that does not strictly follow the Material design guidelines but still provides an excellent user experience both for Android and iOS. The app is primarily optimized for mobile phones but works fine on tablets as well, through the development process it is being tested in parallel on virtual iOS and Android devices. One segment of our app uses the camera, this segment has been tested on an iPad Air and on OnePlus 5T. The idea behind the application is to give the end user the opportunity to easily purchase a car insurance policy regardless whether the user is on a lunch break, sitting on the bus, or relaxing on the couch at home. For this purpose, we have created a custom wizard which with just five steps gets the user's personal information, consent, the information about the vehicle, chooses an insurance package and in the final step gets the insurance policy. For this, we have used a lot of widgets that are provided out of the box by Flutter but also created some custom ones, for user interaction we are using gestures detector and inputs. In the future we hope to enrich the application with animation and expand the user's experience by adding a 360 view for the user. In short, we are happy with the progress we have made and the app will always have a special place in our hearts because it helped us to better understand the Flutter development process.

4 CONCLUSION

Google is going strong shipping several Flutter updates in the last year (the currently active version of Flutter is 1.5.4 released on May 7, 2019) with adding new functionalities, reducing the compile time, expanding to new fronts like web, embedded systems and desktop, also improving the plugins for VS Code and Android studio making Flutter more and more tempting for the developers. According to LinkedIn Data, Flutter is the fastest-growing skill among software engineers and after trying it for ourselves it is not that surprising [10]. To put it shortly, just try it for yourself chances are you will fall in love with this remarkable modern framework.

A good analogy for our firm is a tree that has strong roots build upon Smalltalk, that are still growing, a robust trunk of Java that supports everything, branches that cannot stop spreading Spring, Microservices, Dockers, DevOps, Angular, Android, iOS and some of the branches even reach the Clouds (Amazon ECS) in the sky. Flutter now is nothing but a new twig that we hope will grow into a strong branch that could bear the fruit of our creative work in the mobile department.

All that glitters is not gold but that is not case with Flutter.

5 REFERENCES

[1] "Flutter - Beautiful native apps in record time." https://flutter.dev/. Accessed 1 May. 2019.

[2] "Manning | Flutter in Action." https://www.manning.com/books/flutter-in-action. Accessed 13 May. 2019.

[3] "Hot reload - Flutter." https://flutter.dev/docs/development/tools/hot-reload. Accessed 15 May. 2019.

[4] "Introduction to declarative UI - Flutter." https://flutter.dev/docs/get-started/flutter-for/declarative. Accessed 13 May. 2019.

[5] "StatelessWidget class - widgets library - Dart API - Flutter." https://docs.flutter.io/flutter/widgets/StatelessWidget-class.html. Accessed 14 May. 2019.

[6] "1. Quick Start - Dart: Up and Running [Book] - O'Reilly Media." https://www.oreilly.com/library/view/dart-up-and/9781449330880/ch01.html. Accessed 23 Apr. 2019.

[7] "Dart's Type System | Dart." https://www.dartlang.org/guides/language/sound-dart. Accessed 23 Apr. 2019.

[8] "Dart: Up and Running: A New, Tool-Friendly Language for Structured" https://books.google.com/books?id=STrDtHUQv2oC&pg=PA12&lpg=PA12&dq=+dart+Everything+you+can+place+in+a+variable+is+an+object,+and+every+object+is+an+instance+of+a+class.+Even+numbers,+functions,+and+null+are+objects.+All+objects+inherit+from+the+Object+class.&source=bl&ots=hOP7YeqB40&sig=ACfU3U1IC9ezb53eRi5K4PTVsUSgxLflCQ&hl=en. Accessed 25 Apr. 2019.

[9] "Linux install - Flutter." https://flutter.dev/docs/get-started/install/linux. Accessed 25 Apr. 2019.

[10] "The Fastest Growing Skills Among Software - LinkedIn Learning." 25 Mar. 2019, https://learning.linkedin.com/blog/tech-tips/the-fastest-growing-skills-among-software-engineers--and-how-to-. Accessed 13 May. 2019.

OTS 2019
SODOBNE INFORMACIJSKE TEHNOLOGIJE IN STORITVE
M. Heričko in K. Kous (ur.)

Univerzitetna založba
Univerze v Mariboru

MYPARK – MOBILNA APLIKACIJA ZA PREVERJANJE PARKIRNIH MEST

MARK BERDNIK, GREGOR GRILC, MATIC STRAJNŠAK

Povzetek: *Prispevek predstavlja mobilno aplikacijo, ustvarjeno s sodobnim ogrodjem Flutter ter programskim jezikom Dart. S pomočjo omrežnega ogrodja DarkNet in knjižnice za računalniški vid OpenCV smo ustvarili video analitiko parkirišča, ki uporabnikom v aplikaciji izpiše, koliko je prostih parkirnih mest. Za administracijo celotne mobilne aplikacije smo kot nadzorno ploščo ustvarili spletno stran. Stran smo naredili s tehnologijo MERN (MongoDB, Express.js, React, Node.js) stack. Aplikacija ima tudi dva mikroservisa, to sta video analitika in strežnik (Node.js). Navedena mikroservisa smo virtualizirali z uporabo tehnologije Docker.*

Ključne besede: *Flutter, Umetna inteligenca, video analitika, mikroservisi, MERN*

NASLOVI AVTORJEV: Mark Berdnik, dijak, Srednja elektro-računalniška šola, Maribor, Slovenija, e-pošta: mark.berdnik@sers.si. Gregor Gril, dijak, Srednja elektro-računalniška šola, Maribor, Slovenija, e-pošta: gregor.gril@sers.si.

NASLOV MENTORJA: dipl. inz. rac. in inf. teh. Matic Strajnšak (un), Alcad d.o.o., Zgornja Bistrica, Slovenija, e-pošta: matic.strajnsak@alcad.si

https://doi.org/10.18690/978-961-286-282-4.29 ISBN 978-961-286-282-4
Dostopno na: http://press.um.si

1 UVOD

Živimo v modernem času in skoraj vsakdo že ima svoje osebno prevozno sredstvo, zaradi česar pa lahko hitro pride do stresa in težav pri iskanju prostih parkirnih mest. Rešitev za ta sodoben problem bi bila aplikacija MyPark, ki bi uporabniku prikazala prosta/zasedena parkirna mesta. Takšen pregled je mogoč z video analitiko parkirnih mest. Prepoznavanje slik v okviru strojnega vida je zmožnost programske opreme, da identificira objekte, mesta, ljudi, pisanje in dejanja v slikah.

Računalniki uporabljajo tehnologijo strojnega vida v kombinaciji s fotoaparatom/kamero za umetno inteligenco, da dosežejo svoj cilj. Prepoznavanje slik se uporablja za izvedbo velikega števila vizualnih nalog, kot je označevanje vsebine slik z meta-oznakami, izvajanje iskanja vsebine slike in vodenje avtonomnih robotov, samovoznih avtomobilov in sistemov za izogibanje nesrečam.

Čeprav človeški in živalski možgani prepoznajo predmete z lahkoto, imajo računalniki težave s to nalogo. Programska oprema za prepoznavanje slik namreč zahteva globoko strojno učenje. Učinkovitost reševanja problemov se zelo izboljša ob uporabi konvolucijskih nevronskih mrež in z uporabo namenske strojne opreme saj naloga zahteva veliko količino energije za intenzivne, kompleksne račune. Algoritmi za prepoznavanje slik so pogosto usposobljeni na milijonih predhodno označenih slik z vodenim računalniškim učenjem. Obdelava slik s Python3, OpenCV in Darknet je uporabljeno kot mikroservis.

V mikroservisni arhitekturi je aplikacija razdeljena na tako imenovanje »mikrostoritve«. Vsaka storitev izvaja edinstven proces in običajno upravlja svojo lastno bazo podatkov. Storitev lahko generira opozorila, podatke dnevnika, podpira uporabniške vmesnike, obravnava identifikacijo ali overjanje uporabnikov in opravlja različne druge naloge. Paradigma mikroservisov zagotavlja razvojnim skupinam bolj decentraliziran pristop k izgradnji programske opreme.

Mikrostoritve omogočajo, da je vsaka storitev obnovljena, prerazporejena in upravljana neodvisno. Na primer, če program ne ustvarja pravilno poročil, je lahko lažje izslediti težavo do določene storitve. Ta posebna storitev bi se lahko po potrebi preizkusila, ponovno zagnala, popravila in prerazporedila neodvisno od drugih storitev.

Kako bi aplikacija delovala?

Po svetu se uporabljajo kamere zaradi več razlogov, recimo varnost ljudi ali samo pregled določenega prostora. V našem primeru bi se uporabljale malo drugače, torej za implementacijo aplikacije bi bila potrebna kamera za pregled parkirišča. Kamera bi podatke o parkirišču posredovala aplikaciji in posledično tudi uporabniku. Uporabniku se na aplikaciji prikaže mapa določenega mesta in vsa vnesenega parkirišča, zraven pa še prostost le-teh. Ob kliku na izbrano parkirišče uporabnik vidi potrebne informacije o parkirišču kot so ime parkirišča, ulica parkirišča, število vseh parkirnih mest in število prostih ter zasedenih parkirnih mest.

Uporabljene tehnologije

Za izdelavo naše aplikacije smo uporabili sodobno ogrodje Flutter, s programskim jezikom Dart. Za dinamičnost aplikacije je uporabljena podatkovna baza MongoDB. Za pregled parkirnih mest je implementirano živčno omrežno ogrodje DarkNet (strojno učenje) in knjižnica OpenCV v programskem jeziku Python. Za administracijo spletne aplikacije smo izdelali spletno stran s tako imenovano MERN (MongoDB, Express, React, NodeJS) stack tehnologijo.

2 PREDSTAVITEV TEHNOLOGIJ

2.1 Flutter

Flutter je odprtokodno ogrodje za razvoj mobilnih aplikacij, ki ga je ustvaril Google leta 2017. Flutter ni jezik, temveč programski paket za razvoj oz. SDK (Software development kit). Za razvoj med platformami, ki uporabljajo Flutter, je DART uradni programski jezik. Vse komponente, ki se v Flutterju uporabljajo za delovanje aplikacije so tako imenovani gradniki oz. angleško widgets. Tako je tudi celotna aplikacija na koncu en widget

Zgrajeni so z uporabo sodobnega ogrodja, ki ga navdihuje React. Dart je nov jezik s C-stilom, ki ga je razvil Google, prvič se je pojavil leta 2007. Uporablja se za razvoj aplikacij Android in IOS pa tudi kot osnovna metoda za ustvarjanje aplikaciji v Google Fuchsia. Flutter smo izbrali, saj je novejša tehnologija, deluje na

dveh platformah (Android in IOS), ponuja hitrejše in zmogljivejše aplikacije in lepši izgled. Prav tako je želja letošnje konference uporaba te tehnologije.

2.2 Dart

Dart je splošen programski jezik, ki ga je prvotno razvil Google, kasneje pa ga je Ecma odobrila kot standard. Uporablja se za izdelavo spletnih, strežniških, namiznih in mobilnih aplikacij. Dart izgleda kot C in je objektno usmerjen programski jezik.

Ustvarjen je za hitro univerzalno razvojno kodo, zato je skoraj vse v Flutter-ju napisano s tem programskim jezikom. Omogoča lažje ustvarjanje animacij in prehodov, ki se izvajajo pri 60fps. V Dart-u se najpogosteje uporabljajo Widgeti, Materiali (mobilno UI ogrodje) in Cupertino (IOS UI ogrodje).

2.3 MongoDB

MongoDB je odprtokodni sistem za upravljanje podatkovnih baz, ki uporablja dokumentno usmerjen model baze podatkov, ki podpira različne oblike podatkov. Gre za eno od številnih ne-relacijskih tehnologij podatkovnih baz, ki so se pojavile sredi leta 2000 pod zastavico NoSQL.

Za uporabo v velikih podatkovnih aplikacijah in drugih obdelovalnih opravilih, ki vključujejo množice podatke, ki se ne ujemajo s trdim relacijskim modelom. Arhitektura MongoDB je namesto uporabe tabel in vrstic kot v relacijskih bazah podatkov sestavljena iz zbirk in dokumentov.

2.4 React

React je JavaScript knjižnica za izdelavo uporabniških vmesnikov. Vzdržuje jo Facebook in skupnost posameznih razvijalcev in podjetij. Vse v React-u so komponente, z gradnjo in združevanjem vseh teh komponent pa se ustvari dinamična spletna aplikacija. Največja prednost uporabe komponent je, da se lahko kadarkoli spremenijo, ne da bi to vplivalo na ostale aplikacije. Ta funkcija je najbolj učinkovita, če se izvaja z večjimi aplikacijami v realnem času, kjer se podatki pogosto spreminjajo.

Vsakič, ko so dodani ali posodobljeni katerikoli podatki, ReactJS samodejno posodobi določeno komponento. To prihrani brskalniku nalogo ponovnega nalaganja celotne aplikacije, za odražanje sprememb. Zaradi teh razlogov se nam je React zdel najprimernejša knjižnica za izdelavo administracijske spletne strani, saj se tako podatki pogosteje nabirajo.

2.5 NodeJS

Node.Js je odprtokodno JavaScript okolje za izvajanje tega programskega jezika zunaj brskalnika. Node.js je napisal Ryan Dahl leta 2009, približno trinajst let po uvedbi prvega JavaScript okolja na strežniški strani. Prvotna izdaja je podpirala le Linux in Mac OS X. Njegov razvoj in vzdrževanje je vodil Dahl, kasneje pa ga je sponzoriral Joyent.

Razvijalcem omogoča uporabo JavaScripta za pisanje ukazov za skriptiranje na strežniku - izvajanje skriptnih strani strežnika za izdelavo dinamične vsebine spletne strani, preden se stran pošlje v spletni brskalnik uporabnika. Node.js lahko na strežniku ustvarja, odpre, bere, piše, briše in zapira datoteke. Lahko tudi zbira podatke obrazcev in dodaja, briše, spreminja podatke v bazi. Uporabili smo NodeJS za dinamiko spletne strani in z njim ustvarili svoj API.

2.6 ExpressJS

ExpressJS je orodje za NodeJS kot brezplačna odprtokodna programska oprema pod licenco MIT. Zasnovan je za izdelavo spletnih aplikacij in API-jev. Ustanovil ga je TJ Holowaychuk, njegova prva izdaja po GitHub je bila 22. maja 2010. Express zagotavlja vmesnik za izdelavo spletnih aplikacij. Zagotavlja orodja, ki so potrebna za gradnjo aplikacije. Prilagodljiv je, saj so na voljo številni moduli, ki jih je mogoče neposredno priključiti v Express.

2.7 DarkNet

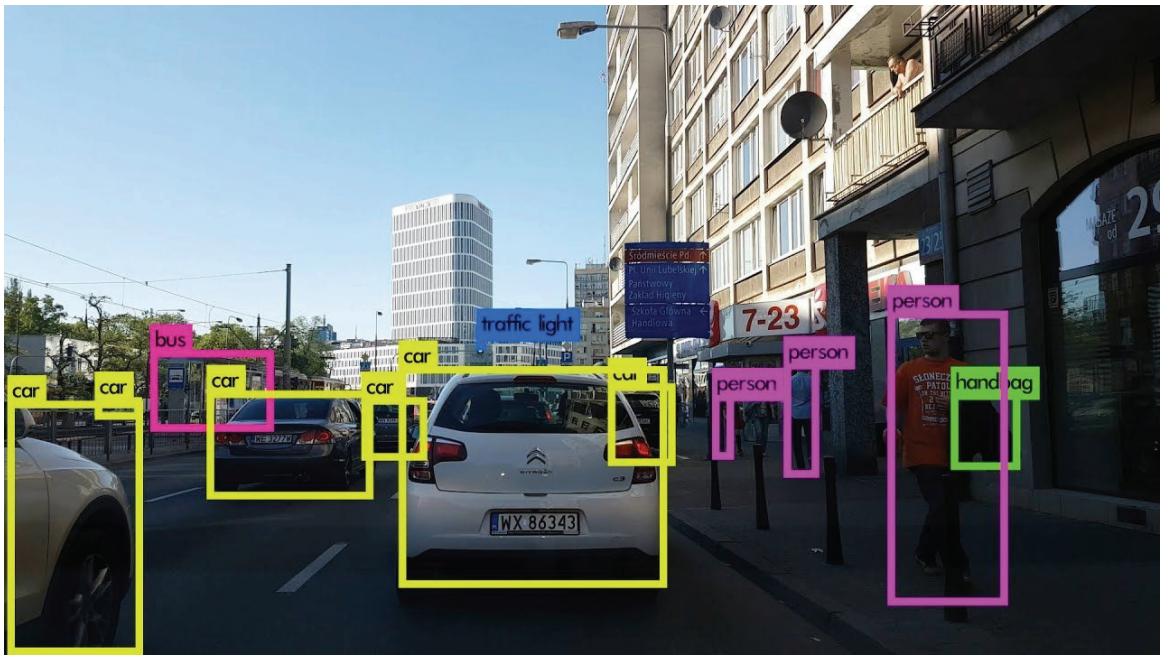
DarkNet je odprtokodno ogrodje, ki se uporablja za pripravo modelov ob uporabi strojnega učenja za potrebe problematike računalniškega vida.V zadnjem času pridobiva veliko pozornosti in to z dobrim razlogom. Z globokim učenjem se računalniški model nauči izvajati naloge razlikovanja iz slik, besedil ali zvoka.

Modeli učenja lahko dosežejo najboljšo natančnost, ki je včasih boljša kot na človeški ravni. Modeli so usposobljeni z uporabo velikega števila označenih podatkovnih in nevronskih omrežnih arhitektur, ki vsebujejo veliko slojev. Odločili smo se za DarkNet, saj z njim najlažje določimo parametre za parkirna mesta, s katerimi bo računalnik prepoznal ali je mesto prosto/zasedeno.

2.8 OpenCV

OpenCV je odprtokodna računalniška vizija in knjižnica programske opreme za strojno učenje. OpenCV je bil zgrajen, da bi zagotovil skupno infrastrukturo za aplikacije računalniškega vida in pospešil uporabo strojnega zaznavanja v komercialnih izdelkih. Knjižnica ima več kot 2500 algoritmov, ki vključujejo obsežen nabor klasičnih in najsodobnejših algoritmov za računalniški vid in strojno učenje.

Ti algoritmi se lahko uporabljajo za zaznavanje in prepoznavanje obrazov, identifikacijo objektov, klasifikacijo akcij npr. klasificiranje človeških dejanj iz videoposnetkov, sledenje premikajočim objektom, podobo celotnega prizora, odstranitev rdečih oči s slik posnetih z uporabo bliskavice, itd… Knjižnica se pogosto uporablja v podjetjih, raziskovalnih skupinah in vladnih organih. S pomočjo programskega jezika Python, smo z OpenCV implementirali model nevronske mreže iz ogrodja DarkNet v naš program.



Slika 1: Primer uporabe OpenCV [4]

https://blog.paperspace.com/how-to-implement-a-yolo-v3-object-detector-from-scratch-in-pytorch-part-5/

2.9 Docker

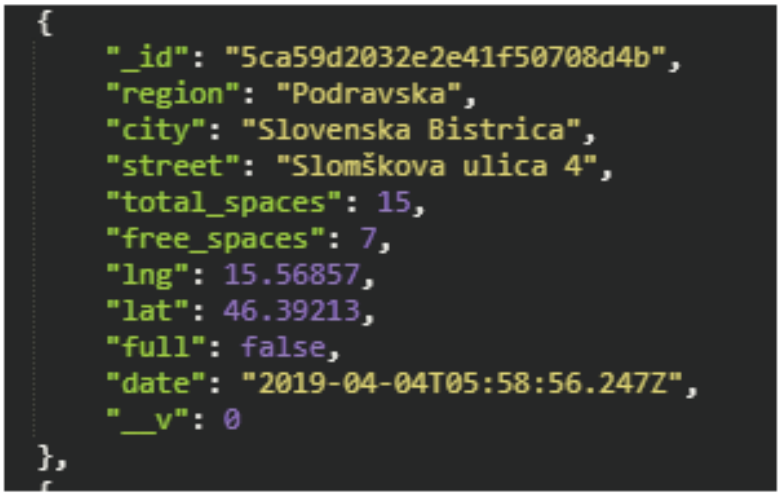
Docker je orodje zasnovano za lažje ustvarjanje, uvajanje in zagon aplikacij z uporabo virtualizacije in tako imenovanih docker kontejnerjev. Kontejnerji omogočajo razvijalcu, da v končno rešitev zapakira in prilagodi vse elemente, ki jih aplikacija potrebuje za pravilno in neodvisno delovanje, kot so knjižnice, pravilne verzije programske opreme in druge dejavnike, in jih zbere kot en paket. Za razvijalce to pomeni, da se lahko osredotočijo na pisanje kode, ne da bi se morali ukvarjati s sistemom, na katerem se bo program na koncu izvajal. Omogoča jim tudi, da uporabo enega izmed tisočih programov, ki so že izdelani za izvajanje v Docker Hubu, kot del njihove aplikacije.

Kontejner je standardna enota programske opreme, ki pakira kodo in vse njene odvisnosti, tako da aplikacija teče hitro in tekoče in je prenosljiva iz enega računalnika na drugega kjer imamo nameščeno programsko opremo Docker. Docker image oz. slika kontejnerja je lahek in samostojen, izvršljiv paket programske opreme, ki vključuje vse, kar je potrebno za zagon aplikacije: koda, izvajalno okolje, sistemska orodja, sistemske knjižnice in nastavitve.

3 PREDSTAVITEV APLIKACIJE

Pričeli smo z izdelavo API-ja, s katerim bi podatki potovali preko strežnika na podatkovno bazo oziroma na aplikacijo. Izdelali smo ga z zgoraj opisanim okoljem NodeJS, ki uporablja programski jezik JavaScript. V tem so bile kasneje dodane končne točke (end point), za določene CRUD klice iz aplikacije. Da ima takšen API namen se uporablja podatkovna baza MongoDB.

Baza služi shranjevanju vseh potrebnih podatkov za brezhibno dinamično delovanje aplikacije. API v podatkovno bazo shranjuje podatke v obliki JSON formata. JSON je odprta standardna oblika zapisa datotek, ki za branje podatkovnih objektov, ki jih sestavljajo pari atributov in vrednosti o nizu, uporablja človeku berljivo besedilo.



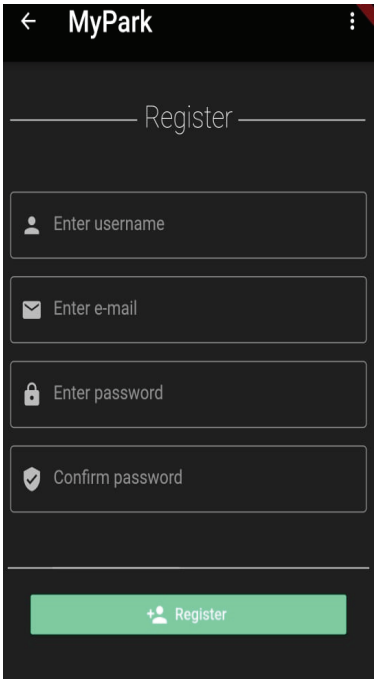
```
{
    "_id": "5ca59d2032e2e41f50708d4b",
    "region": "Podravska",
    "city": "Slovenska Bistrica",
    "street": "Slomškova ulica 4",
    "total_spaces": 15,
    "free_spaces": 7,
    "lng": 15.56857,
    "lat": 46.39213,
    "full": false,
    "date": "2019-04-04T05:58:56.247Z",
    "__v": 0
},
```


Slika 2: JSON Parks

Slika 4 prikazuje JSON zapis, v njem je prikazan tako imenovan Parks. API kliče podatke iz podatkovne baze in jih prikaže na aplikaciji. Nato smo pričeli izdelovati našo mobilno aplikacijo. Za varnosti in preglednost uporabnikov smo se lotili Login strani (prijava v aplikacijo). Menimo da je takšen Login sistem potreben za pregled statistike (aktivni uporabniki, število vseh uporabnikov) in lažjo administracijo celotne aplikacije.

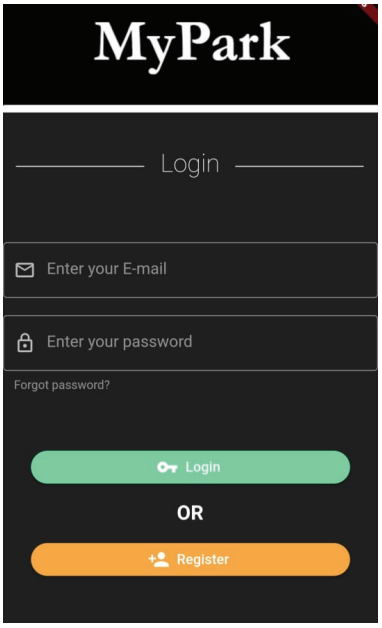
3.1 Login/Register

Da *Login* sistem deluje je potreben *register*, s katerim se podatki shranijo v podatkovno bazo. Oblikovali smo štiri potrebna polja za registracijo – uporabniško ime (*username*), elektronska pošta (*e-mail*), geslo (*password*), potrditev gesla (*confirm password*) in gumb za registracijo (*register*). Ko uporabnik pravilno izpolni vsa polja in pritisne gumb, se podatki preko API-ja shranijo v podatkovno bazo.



Slika 3: Register

Na prvi strani aplikacije (Login) smo ustvarili dva polja in dva gumba - elektronska pošta (*e-mail),* geslo (*password*) ter gumba login in register. Ob kliku na gumb Login aplikacija preveri če so vpisani podatki že v podatkovni bazi, če so, uporabnika vpiše in mu prikaže prvo stran aplikacije. Če uporabnik vpiše pravilne izbrane podatke v login sistem, se mu prikaže spodnji ekran:



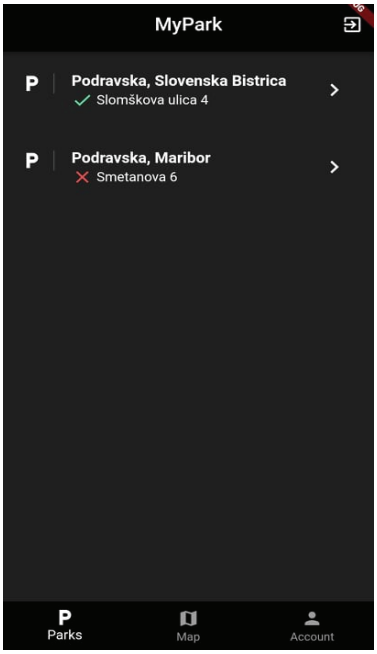
Slika 4: Login

3.2 Parks

Da pri aplikaciji pridemo do bistva in da uporabnik čimprej najde svoje parkirno mesto, smo se odločili da bodo na prvi strani aplikacije vsa parkirišča ki jih podpira aplikacija MyPark. Podatki se na aplikaciji prikažejo preko API-ja iz podatkovne baze, prikazani so samo najpotrebnejši podatki.

Za to smo ustvarili dinamično dodajanje parkirišč, na naši administracijski spletni strani jih je mogoče dodajati, urejati in izbrisati. Prikazani so podatki kot so - regija, mesto, ulica parkirišča in kljukica/križec ki povesta

uporabniku če ima parkirišče prosta oziroma zasedena parkirna mesta. S klikom na parkirišče nas aplikacija preusmeri na informacije o izbranem parkirišču.

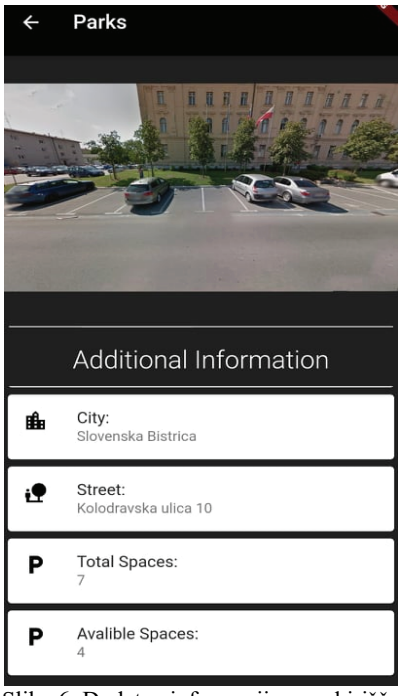


Slika 5: Parks

3.3 Dodatne informacije o parkiriščih

Stran 'dodatne informacije' uporabniku izpiše vse podatke o parkirišču, ki jih potrebuje. Prikazana je slika parkirišča, ki se vsako minuto posodobi, saj kamera vsako minuto slika parkirišče, ga predela z video analitiko in prvotno sliko shrani v podatkovno bazo in posledično se slika preko API-ja prikaže na aplikaciji.

Ostale dodatne informacije so – regija (*region*), mesto (*city*), ulica (*street*), število vseh parkirnih mest (*total spaces*), število prostih parkirnih mest (*available spaces*) in ID parkirišča. Število prostih parkirnih mest se predela z video analitiko, nato se podatki shranijo v podatkovno bazo in, enako kot pri sliki, preko API-ja prikažejo na aplikaciji.



Slika 6: Dodatne informacije o parkirišču

3.4 Zemljevid

Zemljevid je namenjen da uporabniku olajšamo oziroma izboljšamo vpogled na lokacijo izbranih parkirnih mest. Uporabili smo Googlov API in tako na aplikaciji prikazali njihove zemljevide in na njej so s puščicami označena naša izbrana parkirna mesta. Ob kliku na puščico se prikaže število vseh in število prostih parkirnih mest (primer 7/15). S klikom na parkirno mesto aplikacija uporabnika usmeri na stran o dodatnih informacijah.

3.5 Račun

Stran račun smo ustvarili za lažjo preglednost in administracijo za uporabnike. Predvsem je namenjena temu, da si uporabniki po želji spremenijo svoje uporabniško ime (*username*), elektronsko pošto (*e-mail*) in geslo (*password*). Navedene podatke si lahko na strani tudi ogledajo.

4 SPLETNA STRAN ZA ADMINISTRACIJO

Kot že rečeno, je stran namenjena administraciji celotne aplikacije in predvsem razvijalcem oziroma vzdrževalcem. Celotna spletna stran je narejena z MERN stack (MongoDB, Express, React in NodeJS). Vsebuje nadzorno ploščo, administracijo parkirišč in uporabnikov, celoten zemljevid in urejanje lastnega računa.

4.1 Nadzorna plošča

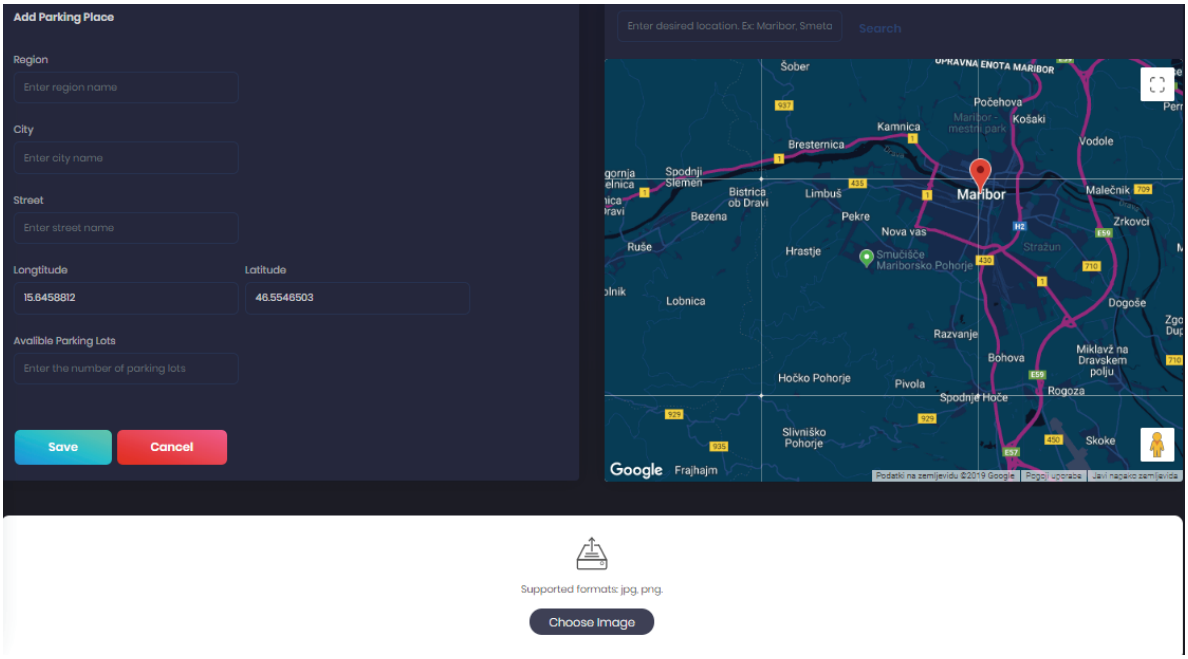
Nadzorna plošča oziroma *Dashboard* prikazuje potrebne podatke in statistiko o aplikaciji. Del statistike je dinamičen grafikon ki prikazuje število vseh API klicev, ki jih opravi aplikacija. Druga grafikona prikazujeta število vseh uporabnikov v vseh mesecih in število vseh parkirišč ter kje se nahajajo (mesto).

Slika 7: Nadzorna plošča

4.2 Parkirišča

Ker celotna aplikacija temelji na parkiriščih, jih je potrebno tudi dodati in imeti nadzor nad podatki. Prva stran je identična tisti v aplikaciji, torej prikazani so podatki kot so - regija, mesto, ulica parkirišča in kljukica/križec ki povesta uporabniku če ima parkirišče prosta oziroma zasedena parkirna mesta. Desno zgoraj smo ustvarili gumb »dodaj parkirišče« (*add park*), stran omogoča administratorju dodajanje vseh potrebnih informacij o

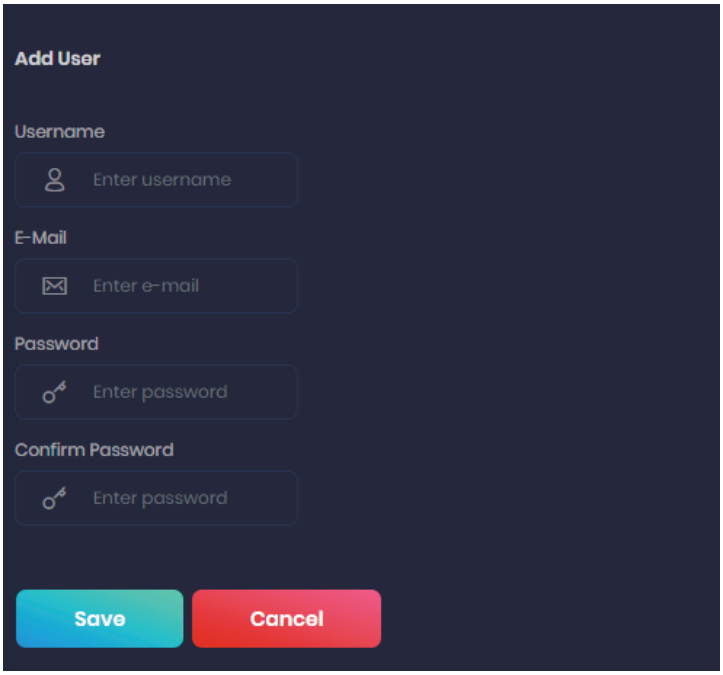
določenem parkirišču: Regijo (*Region*), mesto (*city*), ulico (*street*), zemljepisno dolžino (*longitude*), širino (*latitude*) in število vseh parkirnih mest. V desnem zgornjem kotu se nahaja iskalna vrstica. Če v njo vnesemo pravilno mesto in ulico, nam pokaže željeno parkirišče.



Slika 8: Dodajanje parkirišč

4.3 Uporabniki

Za administracijo uporabniških računov smo na strani dodali stran *Users,* ki je namenjena predvsem dodajanju oziroma spreminjanju računov. Na začetni strani so izpisani vsi uporabniki aplikacije, zraven njihovih uporabniških imen, elektronske pošte, skupine in datum ustvarjenega računa, pa je gumb za urejanje. Na strani se nahaja gumb *add users,* služi za dodajanje uporabniških računov. Add users je namenjen predvsem za dodajanje administracijskih računov, saj ima aplikacija svoj registracijski sistem.

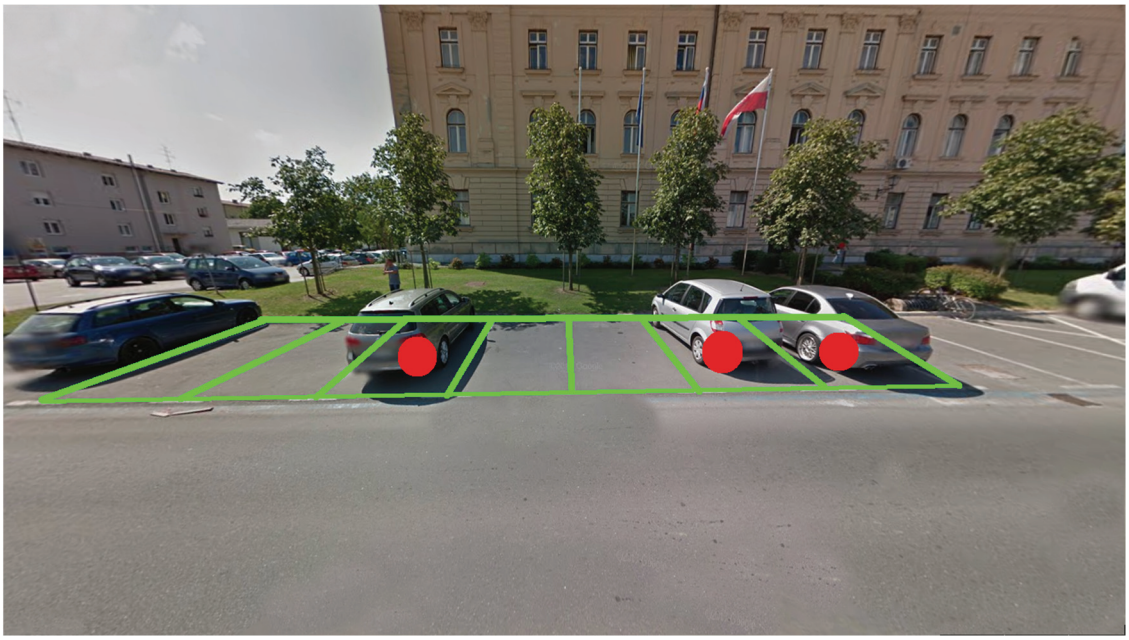


Slika 9: Dodajanje uporabnika

5 PYTHON - VIDEO ANALITIKA

Celotna aplikacija temelji na video analitiki, zato smo morali izdelati svojo delujočo video analitiko parkirišča. Polja smo določili tako, da kamera prepozna če je parkirno mesto zasedeno oziroma prosto. Rdeča pika nam pove da je parkirno mesto zasedeno, v nasprotnem primeru polje ostane prazno. Kamera vsako minuto zajame sliko parkirišča, jo z našo analitiko predela in posledično se podatki o predelani sliki preko strežnika prikažejo na aplikaciji.

Slika 10: Parkirišče pred video analitiko [9]



Slika 11: Parkirišče po video analitiki

6 ZAKLJUČEK

Začetna ideja celotnega projekta je bila izdelati mobilno aplikacijo z uvedbo nam precej neznane tehnologije Flutter. Tekom izdelave aplikacije smo se spoznavali s programskim jezikom Dart. Za izvedbo izbrane ideje smo uporabili prav tako precej neznano področje video analitike, ki temelji na principu klasifikacij slik in detekciji objektov. Prva izdaja tehnologije Flutter leta 2017 je bila narejena striktno za izdelovanje mobilnih aplikacij. V zadnjem času pa napreduje in se izpopolnjuje, da bo lahko v prihodnosti uporabljena tudi za izdelovanje spletnih strani.

Ob začetku izdelovanja arhitekture dela nismo bili prepričani, če bomo zmožni celotno aplikacijo uresničiti z vsemi tehnologijami, ki smo si jih izbrali. V poteku izdelave mobilne aplikacije smo se začeli zavedati, da bo projekt uspešen. Ustvarili smo celotno aplikacijo z novejšim ogrodjem, uporabili celotno tehnologijo MERN stack in uspešno uvedli eno izmed obetavnejših panog računalništva, video analitiko, in verjamemo, da bo s pomočjo nje človeštvo v prihodnosti pridobilo še veliko koristi. Globoko učenje se že dandanes uporablja v medicini in drugih strokovnih področjih.

Razvili smo aplikacijo MyPark, ki pomaga uporabnikom pri iskanju prostih parkirnih mest ter omogoča vpogled v osnovne potrebne podatke o njih. Razvijalcem smo s pomočjo MERN stack ustvarili spletno stran z nadzorno ploščo in celotno administracijo nad aplikacijo. V prihodnosti bi v aplikaciji izboljšali video analitiko za natančno detekcijo objektov, videz aplikacije in dodali še več naprednejših uporabniških funkcij (npr. vodenje od vhoda na parkirni prostor do prvega prostega parkirnega mesta). Menimo, da je naša ideja zelo praktičnega pomena in bi jo bilo mogoče implementirati v resničnem svetu.

7 LITERATURA

[1] http://lvelho.impa.br/ip08/reading/rt-ocv.pdf, Real-Time Computer Vision with OpenCV, 10.5.2019

[2] https://medium.com/javascript-in-plain-english/full-stack-mongodb-react-node-js-express-js-in-one-simple-app-6cc8ed6de274, Flutter Layout Cheat Sheet, 10.5.2019

[3] https://medium.com/javascript-in-plain-english/full-stack-mongodb-react-node-js-express-js-in-one-simple-app-6cc8ed6de274, Build a full stack MongoDB, React, Node and Express (MERN) app, 10.5.2019

[4] https://blog.paperspace.com/how-to-implement-a-yolo-v3-object-detector-from-scratch-in-pytorch-part-5/, Primer OpenCV slike, 10.5.2019

[5] https://cdn-images-1.medium.com/max/800/1*gqBLqChWtWLq33DvWm6Nog.png, Flutter logotip, 10.5.2019

[6] http://www.jsweet.org/wp-content/uploads/2016/04/react-logo-300x289.png, React logotip, 10.5.2019

[7] https://medium.com/@habibridho/docker-as-deployment-tools-5a6de294a5ff, How to Deploy App Using Docker, 10.5.2019

[8] https://link.springer.com/article/10.1007/s10278-017-9965-6, Toolkits and Libraries for Deep Learning, 10.5.2019

[9] https://www.google.com/maps/@46.3920561,15.5759222,3a,75y,214.13h,74.93t/data=!3m6!1e1!3m4!1szPQxRg98n-clm-oCBAce3Q!2e0!7i13312!8i6656, Google maps.

Blockchain Lab:UM®

- **Razvoj pilotnih rešitev**
- **Prilagojene delavnice in predavanja**
- **Podpora pri presoji, odločanju in uvajanju strategije sprejemanja BC**
- **Podpora pri analizi, načrtovanju in izbiri najprimernejših BC modelov, tehnologij in platform**

Projekt EduCTX je prejel eNagrado Slovenskega društva Informatika za **Najboljši projekt s področja računalništva in informatike v 2019**

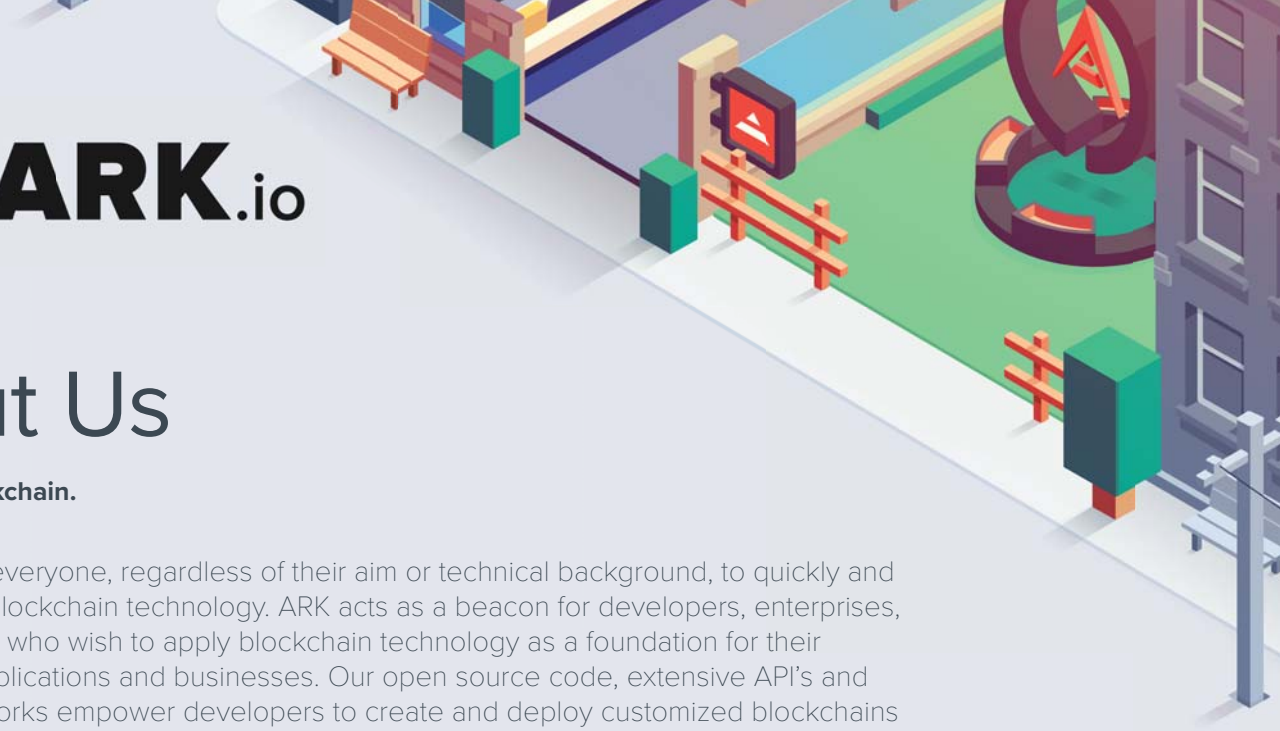
Projekti

EduCTX decentralizirana platforma za upravljanje digitalnih mikro-certifikatov
IOT&IOTA zbiranje realnočasovnih senzorskih podatkov s platformo IOTA
BC 24alife pilotna rešitev, temelječa na ogrodju Hiperledger Fabric
ECTA svetovanje, presoja in razvoj pametnih pogodb za ICO
SmartInsurTech presoja rešitve blockchain za mikro-zavarovanja
H2020 Concordia raziskave varnosti in razširljivosti BC omrežij

ethereum HYPERLEDGER IROHA HYPERLEDGER FABRIC IOTA

BLOCKCHAIN-LAB.UM.SI • 02 220 7351 • INFO@BLOCKCHAIN-LAB.UM.SI

ARK.io



About Us

Point. Click. Blockchain.

ARK empowers everyone, regardless of their aim or technical background, to quickly and easily leverage blockchain technology. ARK acts as a beacon for developers, enterprises, and communities who wish to apply blockchain technology as a foundation for their own projects, applications and businesses. Our open source code, extensive API's and powerful frameworks empower developers to create and deploy customized blockchains in minutes.

Simplicity

Our open source tools, frameworks and products are easy to understand and utilize by both developers and users within an ecosystem that reduces barriers and learning curves

Security

The modified Delegated-Proof-of-Stake (DPoS) consensus mechanism utilized by ARK features 51 delegates that maintain healthy network governance and decentralization

Speed

Eight second block times on the ARK Public Network coupled with the offloading of non-essential transactions to a network of bridgechains means the mainnet is blazing fast and bloat-free

Scalability

By using bridgechains to scale the ARK network, newly created chains are able to communicate with the ARK Public Network and interoperate with non-ARK blockchain networks

Sovereignty

ARK provides the technology to create completely custom blockchains with bespoke governance and settings meaning that your network operates to your own requirements

Tools & Resources

With ARK you'll get all the tools and resources you need to build. Our open source code, exhaustive documentation, easy to use products and SDK's provide you with everything you need to create your own custom blockchain and project

Developer Community

We empower our community of developers to collaborate and improve the ARK platform through bounties and the ARK Improvement Proposal (AIP) initiative. Developers are also able to submit a proposal to the ARK Community Fund (ACF) for project support

Enterprise Support

ARK blockchain technology and business services such as the Powered by ARK Program enable startups, enterprises and public institutions across many industries to unlock the true value of decentralized technology

Learn more at: **ark.io**

ARK Deployer

Create a Blockchain in Minutes

ARK Deployer is a free and easy-to-use blockchain creation tool that enables you to build, customize and deploy your own blockchain. In just a few minutes, you'll have your own scalable, efficient and decentralized blockchain network based on ARK's core code, but completely customized to your project and business requirements

Step 1

Prepare

There are a few things you'll need to know before you get started - just follow our Deployer preparation guide

Step 2

Customize

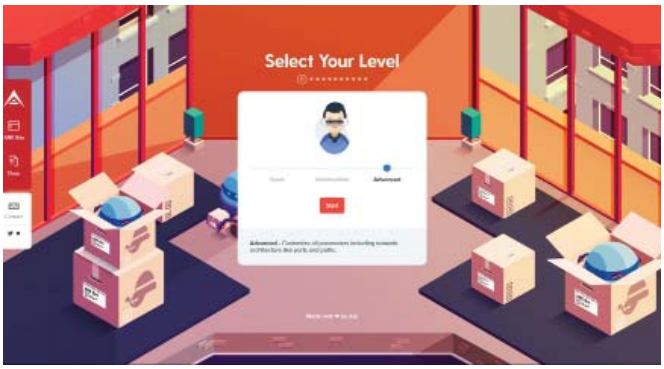
Create and customize your blockchain settings and options to your unique requirements

Step 3

Deploy

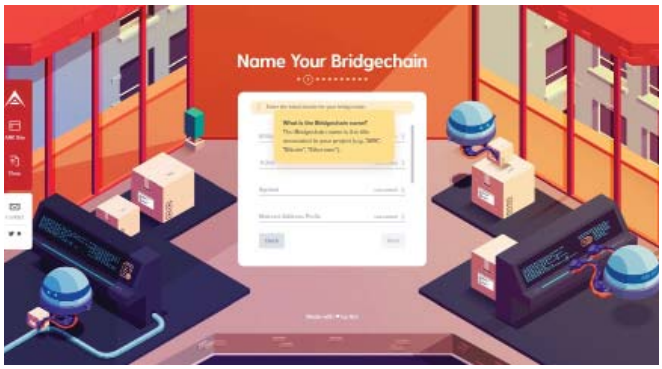
After customization, it's now time to deploy your blockchain with your genesis node - just follow our Deployer launch guide

Key Features



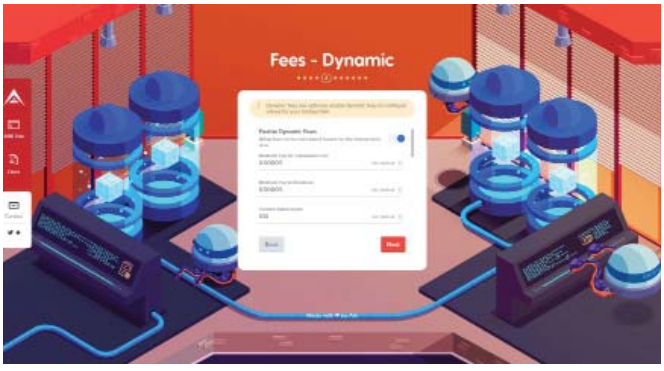
Level Selection

ARK Deployer caters to developers of all skill levels allowing you to choose the level that best represents your needs - providing a completely tailored experience



Help & Assistance

If you are unsure of any customization variable or parameter, use the help buttons for further information or select the 'use default' to automatically assign a default value



Complete Customization

ARK Deployer allows developers to create a blockchain in minutes that is completely unique to their project and business requirements

Advanced Option Selection

Customization options include: forging delegate count, block times, token count, chain prefixes, rewards, fee structures, network architecture and seed node setup among others

Learn more at: **ark.io/deployer**

Alcad

ALCAD d.o.o.
Mroževa ulica 5
2310 Slov. Bistrica

T: (0)2 8055655
F: (0)2 8055665
W: www.alcad.si

SAP Analytics Cloud

Managerska poročila prihodnosti!

Zakaj ne bi že ukrepali, medtem ko konkurenti še čakajo na informacije?

SAP Analytics Cloud prinaša moderno in dostopno rešitev za poslovno odločanje, ki jo odlikujeta hitra uvedba in preprosta uporaba. Odlična izbira za vse uporabnike SAP rešitev – na telefonu, tablici ali računalniku.

VEDNO NA TEKOČEM S POSLOM

POPOLNA SLIKA POSLOVANJA

PRILAGOJENE INFORMACIJE ZA VSE RAVNI ODLOČANJA

UKVARJATE SE SAMO ŠE Z VSEBINO

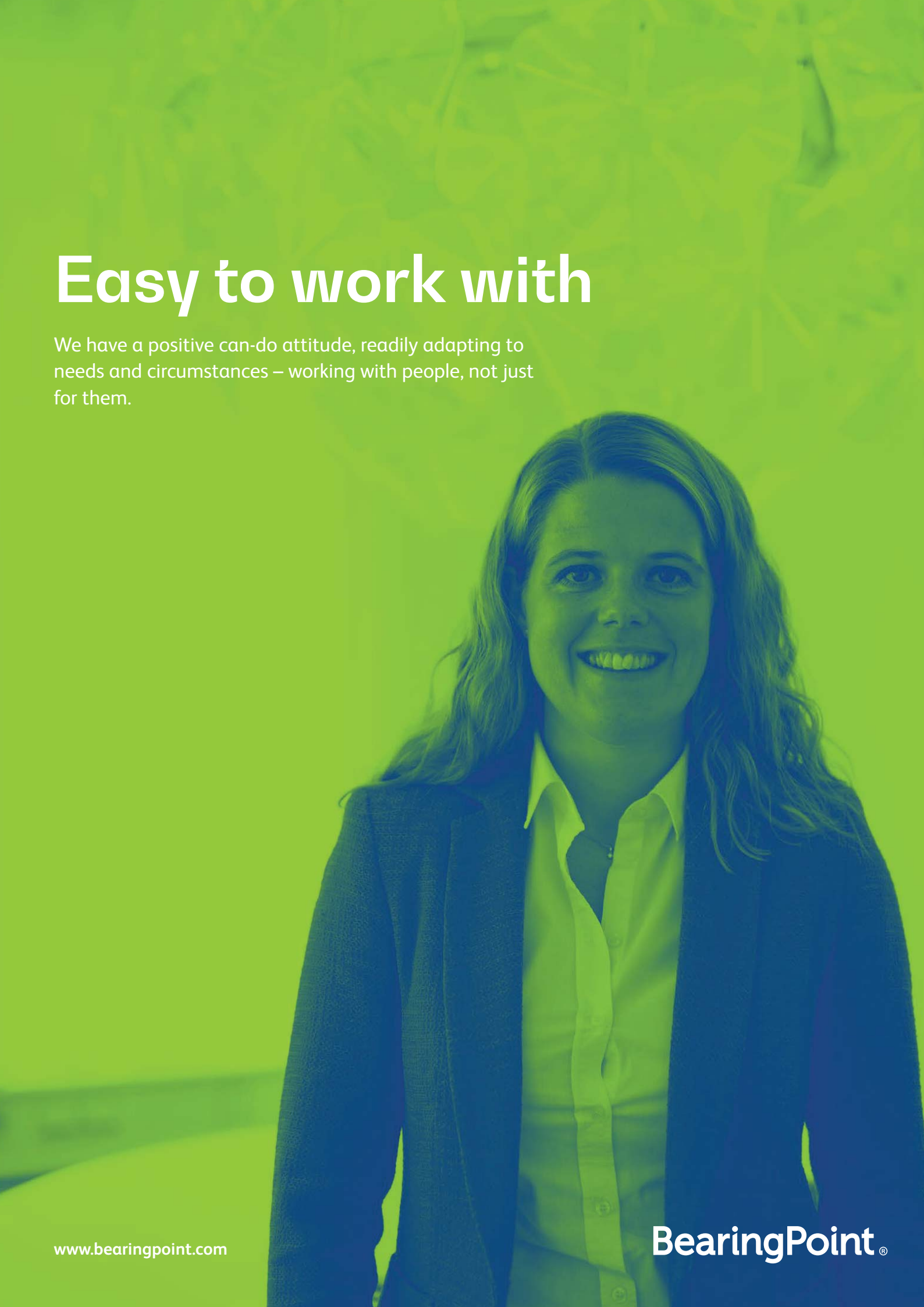
Uporabniki SAP Analytics Clouda so predvsem iz vrst vrhnjega managementa in uprav. Cenijo izjemno uporabniško izkušnjo, ki v podjetje prinaša tudi uspešnejše managiranje vseh procesov. Managerska poročila, dashboardi, prikazi KPIjev ... omogočajo sprotni pregled nad poslovanjem in zato boljše odločanje. Uvedba orodja je hitra, ko so podatki pripravljeni, jih naprej lahko oblikujejo kar sami.

Igor Kavčič, vodja prodaje SAP rešitev

www.avtenta.si sap@avtenta.si avtenta.

Easy to work with

We have a positive can-do attitude, readily adapting to needs and circumstances – working with people, not just for them.



www.bearingpoint.com

BearingPoint®

bintegra

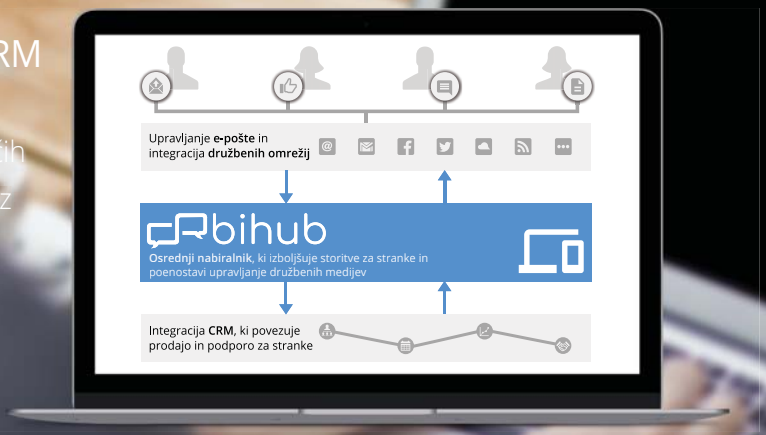
Svetovanje in IT rešitve

Integracija poslovnih aplikacij, podatkov in informacij

Pomagamo vam pri integraciji aplikacij, podatkov in informacij z namenom avtomatizacije procesov in izboljšanja zadovoljstva strank. Imamo bogate reference pri razvoju integracijskih rešitev tako s platformami proizvajalca TIBCO, kot z ogrodji temelječimi na odprtokodni Java arhitekturi.

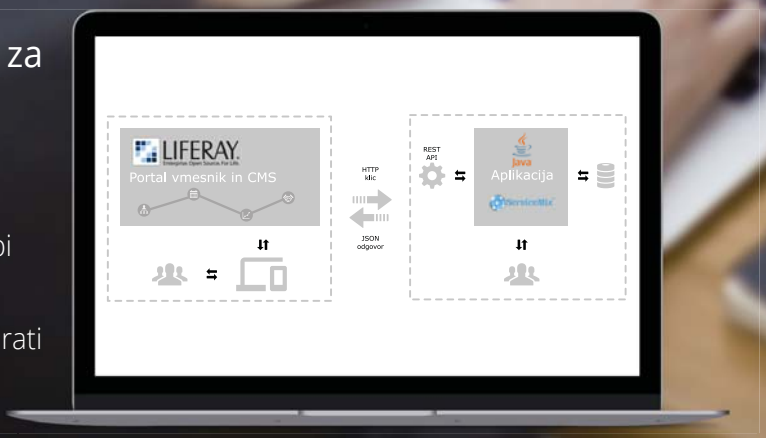
Upravljanje z družbenimi omrežji in CRM integracija

Družbena omrežja so eden izmed najhitreje rastočih kanalov, ki jih stranke uporabljajo za komunikacijo z znamkami. Z našo rešitvijo lahko poenostavite upravljanje z vašimi družbenimi omrežji, lažje sodelujete z vašo marketinško agencijo in se z njo integrirate z vašim CRM-jem.



Uporabniški portali in portalne rešitve za vaše stranke

Vaši uporabniki zdaj uporabljajo različne digitalne kanale in od vas pričakujejo štiriindvajseturno dosegljivost za interakcijo, sedem dni v tednu. Da bi izboljšali storitve za vaše stranke in integrirali marketinške iniciative, vam pomagamo implementirati spletne in mobilne rešitve.



Kontakt:

info@bintegra.com

+386 (0) 2 620 4861 /2

+386 (0) 2 620 4863

Glavna pisarna:

Železnikova 4

2000 Maribor

Slovenia

Pisarna v Ljubljani:

Stegne 23A

1000 Ljubljana

Slovenia

COMTRADE GAMING

An international company with 200+ software engineers that work on innovative projects within the gaming industry.

Hiring

- web developer
- mobile developer
- front-end developer
- business analyst
- software architect
- test automation engineer
- tester
- BI developer
- system engineer
- C++ developer
- DB developer
- .NET developer

Offering

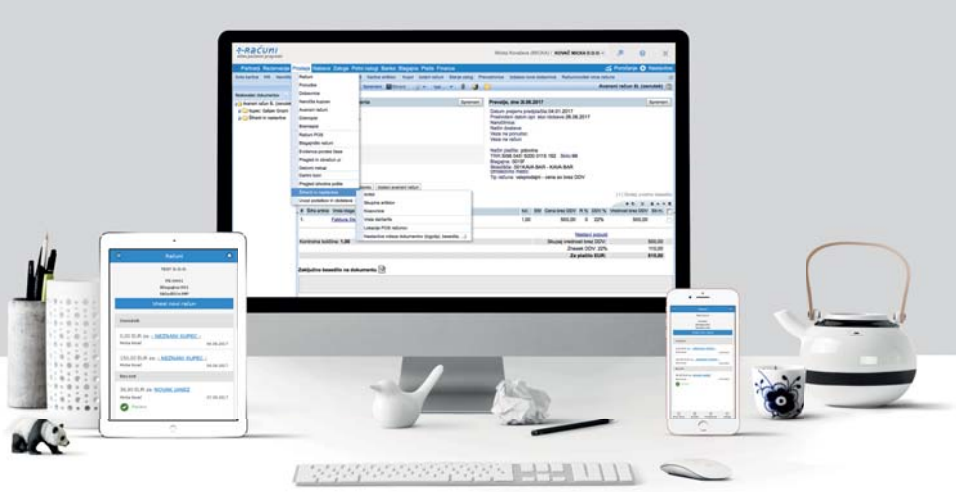
Excellent working conditions and relaxed atmosphere

An international working environment

Opportunities for professional and personal development

Expand your social circle

Apply now: comtradegaming.com

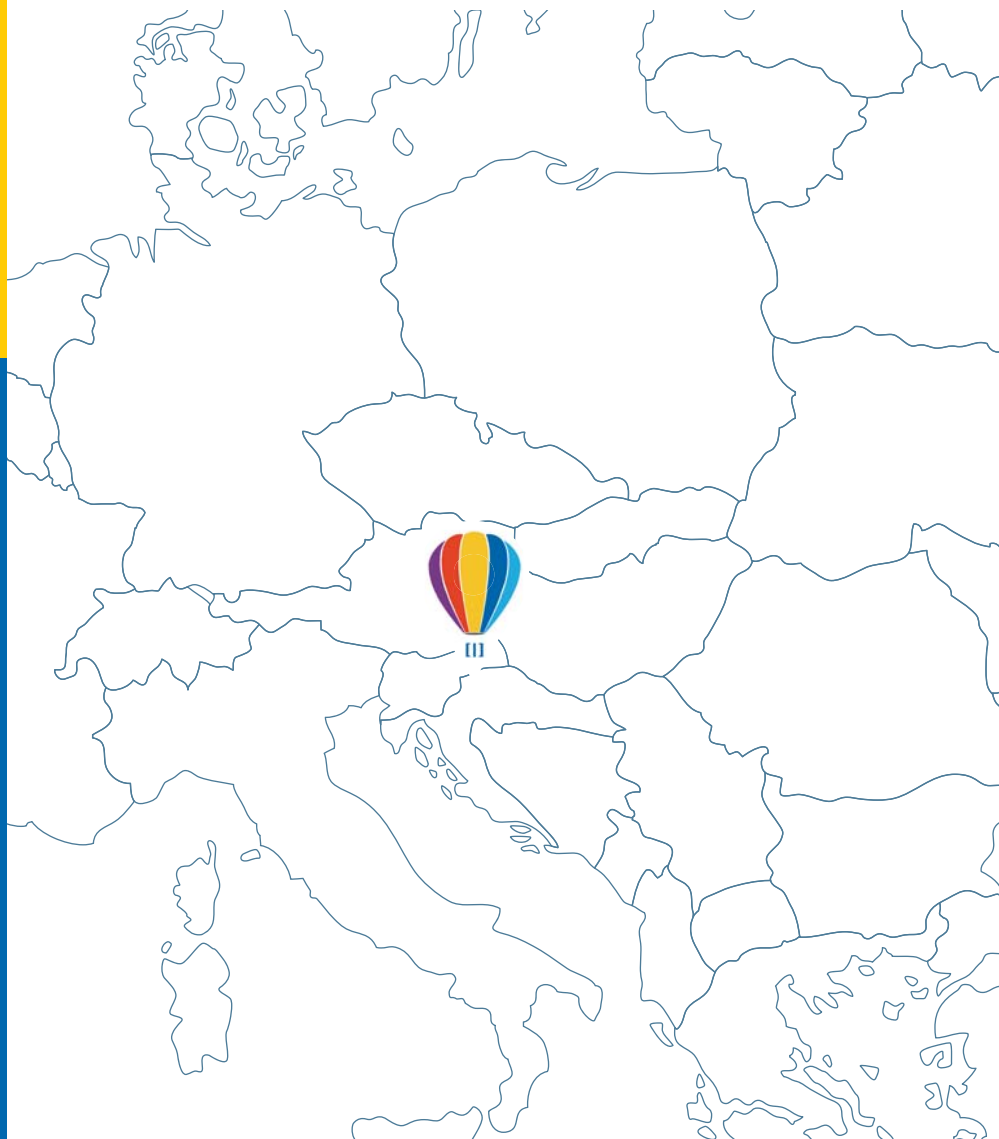


Cloud based ERP system
for Central Europe.

www.e-racuni.com
www.eurofaktura.com

Powered by Smalltalk objects since 2002.

5 countries
40.000+ users.



E-RAČUNI d.o.o.
Titova cesta 8
2000 Maribor

RC IRC Celje, d.o.o.

Ul. XIV. divizije 14, 3000 Celje

5. DESETLETJE SOUSTVARJAMO INFORMACIJSKO DOBO

- Informacijski sistemi za zdravstvene ustanove
- Informacijski sistemi za celovito podporo poslovnih procesov
- Prenova poslovnih procesov in racionalizacija poslovanja
- Informacijska podpora za poslovno odločanje in upravljanje
- Certifikata kakovosti informacijske tehnologije in informacijske varnosti
- Storitve svetovanja pri uvedbi standardov kakovosti in informacijske varnosti
- Poslovno in informacijsko svetovanje
- Celovito vzdrževanje informacijskih sistemov

Poslovna področja

- Zdravstvo
- Televizija
- Visoko šolstvo
- Telekomunikacije
- Industrija
- Banke

Informacijska cesta, ki povezuje slovenske zgodbe o uspehu.

Inovativnost v službi uspešnosti

RC IRC d.o.o.
Ulica XIV. divizije 14
3000 Celje

T. +386(0)3 427 42 00
F. +386(0)3 427 41 98
E. info@rcc-irc-si
W. www.rcc-irc.si

ISO 9001/TickIT BUREAU VERITAS Certification

ISO 27001 BUREAU VERITAS Certification

better
BY MARAND

OPENeP

Okolje, kjer je varnost pacientov na prvem mestu.

WWW.BETTER.CARE

.expertise .innovation .partnership

- razvoj celostnih informacijskih sistemov za zavarovalnice in njihove agente
- rešitve v oblačnih storitvah v mikrostoritveni arhitekturi z uporabo vsebnikov Docker
- digitalizacija procesov
- razvoj mobilnih aplikacij
- agilen razvoj
- več kot 20 let zanesljiv in inovativen partner

msg.Symass

Kontakt:
msg life odateam d.o.o.
Titova cesta 8
SI-2000 Maribor
Tel: + 386 2/ 2356 229
E-Mail: Andrej.Kline@msg-life.com

www.msg-life.si

myWorld

Find your perfect place to work!

#WEARE

CashbackWorld
Cashback**Solutions**
GreenfinityFoundation
ChildandFamilyFoundation

myworld-solutions.com

NOVUM

www.novum-online.si

ENTERPRISE JAVA APPLICATION DEVELOPMENT

AGILE SOFTWARE DEVELOPMENT

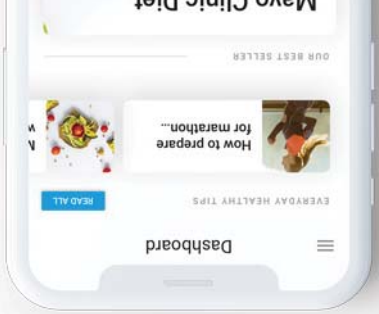
INTERNATIONAL PROJECTS

IT SOLUTIONS FOR INSURANCE BUSINESS

CLOUD SERVICES FOR INSURERS

Illnau Köln Maribor Nürnberg Salzburg Wien

24alife



POSEBNA PONUDBA!
PAKET PROGRAMOV

UČINKOVITI PROGRAMI PROMOCIJE ZDRAVJA V PODJETJIH

MAYO CLINIC RESILIENT MIND | **MAYO CLINIC DIET** | **12 HABITS** | **WHEEL OF LIFE**

24alife™ je celostna rešitev, ki s pomočjo znanstveno podprtih programov klinike Mayo vodi do bolj zdravega načina življenja in boljšega počutja. Izboljšajte zadovoljstvo in produktivnost zaposlenih z enostavnimi programi promocije zdravja pri delu, usmerjenimi v zdrav življenjski slog in občutek zadovoljstva na delovnem mestu.

ZNIŽAJTE Z ZDRAVJEM POVEZANE STROŠKE IN ZVIŠAJTE PRODUKTIVNOST.

Available on the App Store

ANDROID APP ON Google Play

Za vse informacije smo vam na voljo na
05 922 6554 ali na Monika.Zeleznik@24alife.com
www.24alife.com

Developed in collaboration with MAYO CLINIC

GROW Your Bussines with INNOVATION

ACCELERATE TIME-TO-MARKET

MONETIZATION

Monetize Anything-as-a-Service

Tridens Monetization is an innovative, real-time, convergent charging, and monetization platform for any industry, any business model, and any revenue stream in today's digital economy.

CHARGE & DRIVE

Electric Vehicle Charging

Tridens Charge & Drive is an all-in-one software solution for Charge Point Operators (CPO) and e-Mobility Service Providers (EMP).

DRIVE & PAY

Pay-As-You-Drive Insurance

Tridens Drive & Pay is a pay-as-you-go software solution for insurance companies. Enabling them to charge their customers for subscription and usage-based consumption services.

Contact Us

tridens technology

Tridens d.o.o.
Zagrebška cesta 22
2000 Maribor
Slovenia

www.tridenstechnology.com
+386 31 627 462
+386 59 010 975
info@tridenstechnology.com

RAČUNALNIŠKE NOVICE
udeležencem konference **OTS2019**
ponujajo POSEBNO PONUDBO!

12 ŠTEVILK revije **RAČUNALNIŠKE NOVICE**
plačate samo stroške pošiljanja **9,70 €** za vseh 12 številk, brez vezave.

Posebna ponudba velja samo do 31. 8. 2019!

Naročite lahko na: maja@stromboli.si 01 620 88 00

kjer navedete geslo OTS.

MARIBOR 100,6 | LJUBLJANA 99,5 | CELJE 100,8
VELENJE 100,8 | M. SOBOTA 100,6 | KRANJ 99,5

Ni nam lahko®

www.radiocity.si

Spletne zgodbe, ki so mi blizu!

Spremljajte nas na

VECER.com

Časnik Večer, Ulica slovenske osamosvojitve 2, 2000 Maribor

Revija avtomatika +ELEKTRONIKA

moderne tehnologije • sodobni pristopi
učinkovite rešitve • profesionalna elektronika

www.avtomatika.com

S5TEHNIKA.NET d.o.o.
Sostrska cesta 43C
1261 Ljubljana Dobrunje

Uredništvo:
Glavni urednik: 040 423 303
Odg. urednik: 040 423 302

Uredništvo:
info@avtomatika.com
www.avtomatika.com

GENERALNI POKROVITELJ

inova iT

POKROVITELJI

Alcad

ARK.io
Point. Click. Blockchain.

avtenta.

BearingPoint.

better
BY MARAND

bintegra
Svetovanje in IT rešitve

COMTRADE GAMING

www.e-racuni.com

RC IRC Celje, d.o.o. / Ltd.

HOLDING MIKROpis

msg ODATEAM

myWorld

NOVUM

RAZVOJNI CENTER IKTS ŽALEC

tridens technology

MEDIJSKI POKROVITELJI

Revija avtomatika
moderne tehnologije • sodobni pristopi • učinkovite rešitve

Računalniške novice
www.racunalniske-novice.com

Radio CITY 100,6 99,5 100,8 RADIO CITY

VEČER

ISBN 978-961-286-282-4

Univerza v Mariboru
Fakulteta za elektrotehniko, računalništvo in informatiko

INŠTITUT ZA INFORMATIKO