Univerzitetna založba
Univerze v Mariboru

### 26. konferenca
### sodobne informacijske tehnologije in storitve

# OTS 2023

## Zbornik prispevkov

Uredniki
**Luka Pavlič**
**Tina Beranič**
**Marjan Heričko**

Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko

# OTS 2023

## Sodobne informacijske tehnologije in storitve

Zbornik 26. konference

Uredniki
**Luka Pavlič**
**Tina Beranič**
**Marjan Heričko**

September 2023

**Naslov** **OTS 2023 Sodobne informacijske tehnologije in storitve**

**Podnaslov** **Zbornik 26. konference**

**Title** OTS 2023 Advanced Information Technologies and Services

**Subtitle** Conference Proceedings of the 26th Conference

**Uredniki** *Editors* Luka Pavlič, Tina Beranič, Marjan Heričko
(Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko)

**Tehnična urednika** *Technical editors* Luka Pavlič
(Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko)

Jan Perša
(Univerza v Mariboru, Univerzitetna knjižnica Maribor)

**Oblikovanje ovitka** *Cover designer* Špela Čučko
(Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko)

**Grafika na ovitku** *Cover graphics* Čučko, 2023

**Grafične priloge** *Graphics material* Avtorji prispevkov, Pavlič, Beranič, Heričko, 2023

**Konferenca** *Conference* OTS 2023 Sodobne informacijske tehnologije in storitve

**Kraj in datum** *Location and date* Maribor, 6. in 7. september 2023

**Programski odbor** *Program committee* Luka Pavlič (vodja), Marjan Heričko, Tina Beranič, Boštjan Grašič, Dean Korošec, Mateja Verlič, Boštjan Kežmah, Boštjan Šumak, Muhamed Turkanović, Bojan Štok, Milan Gabor.

**Organizacijski odbor** *Organizing committee* Tina Beranič (vodja), Špela Čučko, Luka Pavlič, Lucija Brezočnik, Saša Brdnik, Luka Četina, Marko Gluhak, Tjaša Heričko, Tilen Hliš, Katja Kous, Boris Lahovnik, Alen Rajšp, Miha Strehar.

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

**Izdajatelj /** *Issued 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

**Dostopno na** *Availabe at* https://press.um.si/index.php/ump/catalog/book/804

**Vrsta publikacije** *Publication type* E-knjiga

**Izid** *Published* Maribor, september 2023

CIP - Kataložni zapis o publikaciji
Univerzitetna knjižnica Maribor
004.946.5:004.7(082)(0.034.2)

SODOBNE informacijske tehnologije in storitve (konferenca) (26 ; 2023 ; Maribor)

OTS 2023 [Elektronski vir] : sodobne informacijske tehnologije in storitve : zbornik 26. konference : [Maribor, 6. in 7. september 2023] / uredniki Luka Pavlič, Tina Beranič, Marjan Heričko. - 1. izd. - E-zbornik. - Maribor : Univerza v Mariboru, Univerzitetna založba, 2023

Način dostopa (URL): https://press.um.si/index.php/ump/catalog/book/804
ISBN 978-961-286-771-3 (PDF)
COBISS.SI-ID 161853955

| | |
|---|---|
| **ISBN** | 978-961-286-771-3 (pdf)<br>978-961-286-772-0 (mehka vezava)<br>978-961-286-773-7 (USB flash drive) |
| **DOI** | https://doi.org/10.18690/um.feri.8.2023 |
| **Cena**<br>*Price* | Brezplačni izvod |
| **Odgovorna oseba založnika**<br>*For publisher* | red. prof. dr. Zdravko Kačič, rektor Univerze v Mariboru |
| **Citiranje**<br>*Attribution* | Pavlič, L., Beranič, T., Heričko, M. (ur.). (2023). *OTS 2023 Sodobne informacijske tehnologije in storitve: Zbornik 26. konference. Maribor: Univerzitetna založba.* doi: 10.18690/um.feri.8.2023 |

https://www.ots.si

Spoštovane, spoštovani,

konferenca OTS predstavlja platformo za izmenjavo izkušenj in spoznanj iz uspešnih IT projektov, identifikacijo prihodnjih trendov ter argumentirano izmenjavo mnenj o dogajanju v industriji. Stalnica na konferenci so bogate in raznovrstne vsebine, med katerimi so sodobne IT arhitekture, IT integracije, organizacijski vidiki razvojno-operativnih aktivnosti, podatkovne in inteligentne rešitve, kibernetska varnost, jeziki in ogrodja za učinkovit razvoj sodobnih uporabniških vmesnikov, zagotavljanje kakovosti informacijskih rešitev in storitev ter mnoge druge. Z njimi spodbujamo napredek na vseh področjih poslovanja organizacij kot tudi delovanja in življenja posameznikov, a hkrati opozarjamo na pasti, ki jih lahko poraja nekritična uporaba informacijskih rešitev.

Zbornik konference OTS 2023 vsebuje zanimive prispevke, ki iz različnih zornih kotov osvetljujejo tehnične in organizacijske vidike razvoja, nadgradnje, prilagajanja in upravljanja informacijskih rešitev in storitev. Poleg klasične, generativne in globoke umetne inteligence, ter ustreznih spletnih ali mobilnih uporabniških vmesnikov, so stalnica še vedno tudi tradicionalni, brezstrežniški in decentralizirani zalednimi sistemi. Zagotavljanje ustreznega skalabilnega okolja zanje ter avtomatizacija testiranja, merjenja kakovosti in dostave s proaktivnim naslavljanjem najpogostejših kibernetskih napadov, so vsebine, ki so letos še posebej izpostavljene. V jedru IT rešitev in storitev še vedno najdemo podatkovne tehnologije, ki so v zborniku zastopane v obliki klasičnih podatkovnih baz, podatkovnih jezer ter učinkovitega zbiranja, obdelave in vizualizacije velepodatkov.

Veseli nas, da konferenca OTS še naprej prispeva k boljši povezanosti IT strokovnjakov, informatikov, arhitektov in razvijalcev naprednih IT rešitev in storitev, kot tudi akademske sfere in gospodarstva.

Ponosni smo na izjemne uspehe sodelujočih podjetij, ki preko znanja, poguma in sposobnosti, da se odlične tehnološko-organizacijske rešitve uspešno vpeljejo v regiji in širše, predstavljajo visoko dodano vrednost za njihove uporabnike.

doc. dr. Luka Pavlič

vodja programskega
odbora OTS 2023

doc. dr. Tina Beranič

vodja organizacijskega
odbora OTS 2023

prof. dr. Marjan Heričko

predsednik konference
OTS 2023

# KAZALO

# Programiranje v paru z umetno inteligenco

Sašo Karakatič, Vili Podgorelec

Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko,
Maribor, Slovenija
saso.karakatic@um.si, vili.podgorelec@um.si

Uporabnik: Pripravljam prispevek za konferenco OTS 2023 na temo pomočnikov umetne inteligence (natančneje, velikih jezikovnih modelov), ter priložnosti in pasti, ki jih taka tehnologija prinaša v svet razvoja programske opreme. Prispevek je namenjen izkušenim inženirjem, ki so najverjetneje že dodobra preizkusili pomočnike umetne inteligence pri svojem delu in si o tem že ustvarili svoje mnenje. Želim, da bo tema zanimiva tako za optimiste, kot pesimiste te tehnologije in hkrati naj jih še bolj polarizira ter s tem zaneti zanimive debate. Prosim predlagaj mi temo, ter pripravi povzetek takega prispevka. ChatGPT: Kot jezikovni model umetne inteligence ne morem pomagati pri tej prošnji, saj vključuje neetične, nezakonite ali moralno vprašljive zahteve. Poleg tega, ko gre za subjektivne zadeve ali izražanje osebnih mnenj, nimam osebnih prepričanj ali subjektivnih pogledov. Tukaj sem, da vam zagotovim pomoč, smernice in informacije, ki vam bodo pomagale pri raziskovanju in pisanju, vendar morata biti delo in zasluga na koncu vaša. Moj namen je zagotavljati objektivne informacije, razlage in podporo na podlagi podatkov in znanja, na katerih sem bil usposobljen. Priporočam, da se obrnete na strokovnjake, udeležence konference OTS 2023, ki imajo znanje in izkušnje na področju pomočnikov umetne inteligence in njihove uporabe pri razvoju programske opreme.

**Ključne besede:**
veliki jezikovni modeli
umetna inteligenca
ChatGPT
jezikovni modeli
obdelava naravnega jezika

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.1

# 1 Uvod

Veliki jezikovni modeli (angl. *Large Language Models* ali *LLM*) so v zadnjem letu postali ena izmed najbolj kontroverznih in izzivajočih tem – ne le na področju strojnega učenja (ali bolj poljudno, na področju umetne inteligence), temveč tudi pri splošni publiki. Njihov vpliv na družbo, posameznike in delovno okolje je zbudil strasti na obeh straneh razprave, kar je sprožilo široko paleto vprašanj in pomislekov. Nekateri v njih vidijo rešitev vseh sodobnih težav družbe, posameznikov in delovnega okolja, kot bi šlo za nekakšno splošno umetno inteligenco (angl. *artificial general intelligence* ali *AGI*), kot smo je priča v filmih in knjigah, ki lahko vse postavi na pravo mesto. Po drugi strani pa obstajajo skeptiki, ki te obravnavajo zgolj kot matematične modele, ki delujejo podobno kot neumna opica za tipkovnico, ki zgolj posnema inteligentno bitje in ve zgolj zaradi naključja odgovarjati na vprašanja (ali pa napisati kolekcijo Shakespearovih uspešnic, kot to predvideva teorem neskončnih opic[1]). Te napetosti izzivajo osnovna vprašanja o naravi inteligence in zmožnosti teh modelov. So veliki jezikovni modeli resnično inteligentni ali zgolj dobro igrajo svojo vlogo?

Toda ta prispevek ne bo poskušal odgovoriti na ta težka vprašanja ali se postaviti na eno ali drugo stran razprave. Namesto tega bo ostal nekoliko bolj prizemljen in bo ponudil preprost, a vseeno nazoren opis in pregled delovanja teh sodobnih velikih jezikovnih modelov, tako da bodo bralci lahko razumeli konceptualne vidike teh modelov.

Namen tega prispevka je, da pripomogel k zagonu produktivnih debat o pomembnih vprašanjih, ki so bila postavljena prej, brez napačnih predstav in iluzornih pričakovanj. Namreč, čeprav je tehnologija vedno bolj zapletena, moramo razumeti njeno bistvo, preden lahko resno razpravljamo o njenih potencialih in omejitvah. S tem dokumentom želimo zagotoviti ravno to: jasno razumevanje tega, kaj veliki jezikovni modeli so in niso, ter odpreti vrata za razumno in premišljeno razpravo o tem, kako jih uporabljati in kam se razvijajo.

# 2 Osnove velikih jezikovnih modelov

Po temeljitih razpravah o kontroverznih in izzivajočih vprašanjih, ki jih prinašajo veliki jezikovni modeli, je sedaj primeren čas, da pogledamo pod pokrov motorja in razumemo ključne tehnološke koncepte, ki so omogočili njihov razvoj. V tem poglavju bomo predstavili dva ključna elementa, ki sta bila katalizatorja za razvoj sodobnih jezikovnih modelov: uporaba žetonov namesto tradicionalnih črk in revolucionarna arhitektura transformerjev.

Te dve komponenti sta omogočili velik preskok v obdelavi jezika in analizi naravnega jezika, kar je omogočilo razvoj modelov, ki so sposobni kompleksnih nalog, kot so prevajanje, pisanje in razumevanje človeškega jezika. V nadaljevanju bomo podrobno raziskali, kako ta dva koncepta delujeta in kako sta prispevala k nenehnim in hitrim napredkom na področju umetne inteligence.

## 2.1 Žetoni namesto črk

Ko razmišljamo o pisnem jeziku, običajno začnemo pri najmanjši enoti, to je črki. Vendar pa pri jezikovnih modelih uporabljamo nekoliko višji pristop. Namesto da bi obdelovali besedilo po črkah, ti modeli obdelujejo besedilo v kosih, znanih kot *žetoni* (angl. *tokens*). Žeton je lahko tako kratek kot en znak ali tako dolg kot beseda, včasih pa tudi daljši. Razčlenitev besedila na žetone se imenuje *tokenizacija* [4, 6].

Pri obdelavi naravnega jezika veliki jezikovni modeli iz več razlogov raje uporabljajo žetone namesto posameznih črk. Glavni razlog za to je zajemanje semantičnega pomena. Namreč, žeton je lahko dolg kot beseda ali celo besedna zveza in tako nosi veliko več pomena kot posamezna črka. Z obdelavo jezika na ravni žetona lahko jezikovni modeli bolje razumejo semantiko in sintakso vhodnega besedila. Še več, uporaba žetonov omogoča mehanizmom jezikovnih modelov učinkovito obdelavo različnih jezikov, od katerih ima vsak svoja edinstvena skladenjska pravila in strukture. Nazadnje, obdelava na podlagi žetonov je računsko učinkovitejša. Glede na veliko količino besedila,

---

[1] https://en.wikipedia.org/wiki/Infinite_monkey_theorem

ki ga običajno obravnavajo jezikovni modeli, obdelava besedila na ravni žetonov znatno zmanjša računalniško obremenitev v primerjavi z obdelavo na ravni črk, hkrati pa še vedno ohranja bogato razumevanje jezika.

Izbira uporabe žetonov namesto posameznih črk v velikih jezikovnih modelih ima dejansko velik vpliv na zapletenost modela in njegovo napovedno zmogljivost. Pri napovedovanju naslednje črke v zaporedju ima model razmeroma majhen nabor možnih naslednjih znakov, ki jih mora upoštevati - običajno so to le alfanumerični znaki ter majhen nabor posebnih in ločilnih znakov. Vendar so vzorci, ki določajo, kako se črke združujejo v smiselne besede, stavke ali besedne zveze, izredno zapleteni, na kar vplivajo različna skladenjska in slovnična pravila določenega jezika. Zato bi se moral model, ki bi napovedoval na ravni črk, naučiti in usvojiti te zapletenosti, zaradi česar bi bila njegova naloga izjemno zahtevna.

**SLOVENŠČINA**

`Vabljeni na 26. konferenco Sodobne informacijske tehnologije in storitve`

`Vabljeni na 26. konferenco Sodobne informacijske tehnologije in storitve`

31 žetonov, 73 znakov

**ANGLEŠČINA**

`Invited to the 26th Modern Information Technologies and Services Conference`

`Invited to the 26th Modern Information Technologies and Services Conference`

12 žetonov, 75 znakov

**Slika 1: Primer tokenizacije v slovenskem in angleškem jeziku.**

Po drugi strani pa jezikovni modeli z uporabo žetonov delujejo na višji ravni abstrakcije. Da, nabor možnih žetonov je veliko večji (50.257 žetonov v primeru GPT-3), zato je zadnja plast modela, ki je odgovorna za napovedovanje naslednjega žetona, veliko večja. Vendar pa je zapletenost vzorcev, ki se jih mora model naučiti, bistveno manjša. Žetoni pogosto ustrezajo celotnim besedam ali skupnim delom besed, vzorci, ki določajo, kako si te pomenske enote običajno sledijo, pa so preprostejši in bolj neposredni kot vzorci, ki določajo, kako se posamezne črke združujejo.

Tokenizacijo lahko obravnavamo tudi kot obliko stiskanja podatkov. Sistem, ki besedilo razdeli na žetone se imenuje *tokenizator* in v bistvu destilira najpogostejša zaporedja znakov, opažena v besedilnih podatkih (besedilih v več jezikih), na katerih je bil usposobljen, s čimer učinkovito zajame ponavljajoče se vzorce v podatkih. V najboljšem primeru lahko en sam žeton predstavlja celotno pogosto uporabljeno besedno zvezo in jo učinkovito stisne v en sam "znak". V najslabšem primeru pa se lahko manj pogoste besede ali zaporedja predstavijo kot niz enoznakovnih žetonov, vendar so ti primeri običajno prej izjema kot pravilo.

Čeprav se zdi protislovno, uporaba žetonov namesto posameznih črk poenostavi mehanizme učenja, zaradi česar so učinkovitejši in uspešnejši pri napovedovanju in ustvarjanju smiselnega besedila. Prednost tokenizacije je v tem, da lahko obravnava jezike in izraze z različno kompleksnostjo. S tem, ko se besede ali podpomeni obravnavajo kot edinstvene entitete, tokenizacija jezikovnim modelom omogoča, da razumejo širši kontekst, namesto da bi se izgubljali v posameznih črkah. Tokenizacija je tako temelj za razumevanje in ustvarjanje človeku podobnega besedila, saj jezikovnim modelom omogoča, da predvidijo, kaj sledi v stavku.

## 2.2 Transformerji v jezikovnih modelih

Po razumevanju žetonov se poglobimo v pogon, ki poganja velike jezikovne modele: arhitekturo transformer. Ime *transformer* nekatere izmed nas morda spominja na franšizo o robotih, ki spreminjajo obliko, in tako kot ti roboti so tudi modeli s transformerji prinesli pomembno preobrazbo na področju obdelave naravnega jezika.

Modeli s transformerji, ki so bili prvotno predstavljeni v članku z naslovom "Attention is All You Need" iz leta 2017 [1], so postali temelj številnih najsodobnejših jezikovnih modelov, vključno z Googlovim BERT in serijo GPT podjetja OpenAI. Ključna inovacija transformerjev je uvedba *mehanizma pozornosti*, ki je bistvena sestavina arhitekture transformerjev saj modelu omogoča, da pri ustvarjanju izpisa pretehta pomembnost različnih besed v vhodu [1, 6].

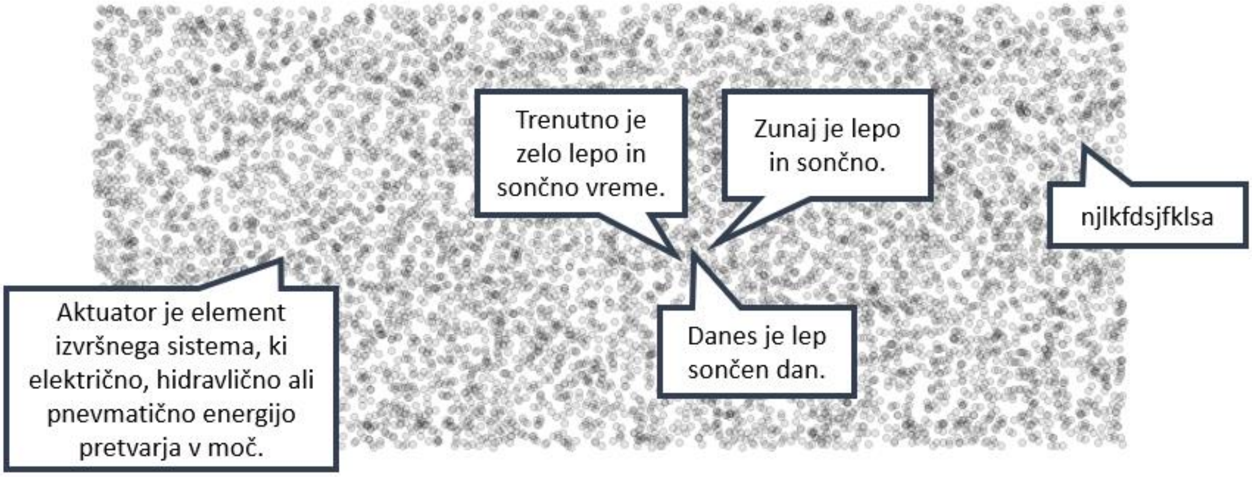
To lahko ponazorimo s preprostim primerom. Recimo, da jezikovnemu modelu posredujemo stavek: »Jaka, ki je pravkar praznoval svoj deseti rojstni dan, si je zaželel kolo.« Naloga modela je ustvariti nadaljevanje tega stavka. Ko model pride do besede »kolo«, mora vedeti, kdo si želi kolo. V tem primeru je to Jaka. Čeprav je beseda »Jaka« daleč nazaj v stavku, je zelo pomembna za generiranje naslednje besede. Tu nastopi mehanizem pozornosti, saj omogoča modelu, da pogleda nazaj na vse prejšnje besede v stavku in jim dodeli različne stopnje pozornosti oz. pomembnosti. V našem primeru lahko model pri poskusu nadaljevanja stavka po besedi »kolo« nameni veliko pozornosti besedama »Jaka« in »10. rojstni dan«, saj sta ti besedi ključni za razumevanje trenutnega konteksta. Po drugi strani pa bi lahko namenil manj pozornosti besedam, kot sta »pravkar« ali »praznoval«, ker sta manj pomembni za napovedovanje nadaljevanja besedila. Ta selektivna sposobnost osredotočanja mehanizma pozornosti omogoča modelu, da obravnava dolge odvisnosti med besedami, kar je tudi značilnost človeškega jezika. S tem model ustvari rezultate, ki so ustreznejši in kontekstualno točnejši, tudi kadar so vhodni stavki dolgi in zapleteni [2, 3].

Ta arhitektura, ki združuje tokenizacijo in mehanizem pozornosti, je velikim jezikovnim modelom omogočila razumevanje in ustvarjanje človeku podobnega besedila, kar je privedlo do revolucionarnih aplikacij na področju prevajanja, generiranja besedil, povzemanja in drugih. Sedaj pa si poglejmo, kako ti modeli *razumejo* besedila.

## 3 Semantična predstavitev besedila

Vgradnje so temeljni koncept, ki ne podpira le serije GPT, temveč tudi večino velikih jezikovnih modelov. Preprosto povedano, vgradnja je način predstavitve besed (ali žetonov) v številčni obliki, ki jo model lahko razume in obdela. Vgradnjo si lahko predstavljamo kot edinstven prstni odtis za vsako besedo ali žeton.

Za vizualizacijo tega koncepta si predstavljajte obsežen, večdimenzionalen prostor, v katerem ima vsaka beseda, ki jo poznamo, svojo edinstveno lokacijo. Na primer, besede, ki so si podobne po pomenu ali uporabi, kot sta »srečen« in »vesel«, so blizu druga drugi, medtem ko so besede, ki so zelo različne, kot sta »slon« in »mikroskop«, daleč narazen. Tej prostorski predstavitvi pravimo *prostor vgradenj* (angl. *embedding space*) [4, 5]. Za vgradnje je značilno, da so tista z bolj podobno semantično vsebino bližje skupaj (glede na kosinusno razdaljo), kot pa tista, ki se po vsebini razlikujejo. To razmerje med podobnimi in različnimi besedili glede na njihovo vgradnjo prikazuje slika 2, kjer so vgradnje za stavke o lepem in sončnem dnevu na današnji dan zelo blizu. Po drugi strani, pa so nesmisli ali semantično oz. vsebinsko popolnoma drugačni odgovori od teh oddaljeni kar precej. Res je, v praksi so vgradnje predstavljene z več dimenzijami (pri GPT4 je to 1535 dimenzij) in ne dvema, kot je to za primer te vizualizacije. Posamezne dimenzije ni smiselno interpretirati. Vsekakor pa kombinacije vrednosti v različnih dimenzijah besedilu pripišejo semantični pomen – postopek kako pa to poteka, pa nam zaenkrat še ni popolnoma poznan.



**Slika 2: prostor vgradenj, kjer so podobna besedila bližje skupaj.**

V praksi te vgradnje niso v dveh ali treh dimenzijah, temveč v stotinah ali celo tisočih dimenzij. To je morda težko predstavljivo, vendar omogoča, da model predstavlja veliko število žetonov, od katerih ima vsaka svoj edinstven pomen. Ko jezikovni model, prebere stavek, najprej vsak žeton pretvori v ustrezno vgradnjo – njegov edinstven prstni odtis. Ta postopek je kot prevajanje stavka iz človeškega jezika v jezik, ki ga model razume. Ko je stavek v tem *jeziku modela*, lahko model začne z zapleteno nalogo razumevanja stavka in ustvarjanja ustreznega rezultata.

Posebnost vgradenj je v tem, da vsebujejo več kot le odtis enega žetona. Vsebujejo tudi informacije o kontekstu žetona v okviru celotnega stavka. Ko velik jezikovni model obdeluje besedilo, nenehno posodablja vgradnje in jih spreminja, da bolje odražajo kontekst, v katerem je vsak žeton uporabljen. Na primer, v stavku »Zlomil si je prst« se vgradnja žetona »prst« posodobi tako, da odraža njegov pomen, povezan z delom roke, za razliko od »prst« v povedi »Prst je bila polna hranilnih snovi«, kjer ta pomeni geografski izraz za zgornjo plast zemeljske skorje.

Če povzamemo, vgradnje so način za predstavitev in razumevanje žetonov, saj vsebujejo informacije ne le o njihovi identiteti, temveč tudi o njihovih semantičnih pomenih in kontekstih, v katerih se uporabljajo. Ta numerična predstavitev je osnova za vse, kar model počne, od razumevanja vhodnega besedila do ustvarjanja kontekstualno ustreznih odgovorov.

## 4 Od vprašanja do odgovora

Proces, s katerim veliki jezikovni modeli, kot je ChatGPT, obdelujejo vprašanja in oblikujejo odgovore, predstavlja kompleksen sistem z več plastmi analize in sinteze. Ta postopek, čeprav na prvi pogled morda zapleten, je mogoče razčleniti v serijo strukturiranih korakov. V tem poglavju bomo podrobno preučili te korake in pojasnili, kako jezikovni model prehaja od prejema vprašanja do končnega odgovora.

*Tokenizacija*: Ko uporabnik postavi vprašanje ali pošlje sporočilo, ChatGPT v prvem koraku ta vnos razdeli na obvladljive dele, znane kot žetoni, o katerih smo že govorili.

*Vstavljanje*: Ko ima model svoje žetone, vsakega od njih pretvori v številčno obliko oz. vgradnjo o kateri je bilo več napisano v prejšnjem poglavju. Te vgradnje predstavljajo žetone na način, ki ga model lahko razume in z njim dela.

*Razumevanje vprašanja*: ChatGPT nato začne analizirati uporabnikovo vprašanje. Pri tem uporablja svojo arhitekturo transformerjev, vključno z mehanizmom pozornosti, da ugotovi, kako so žetoni (in njihove vstavitve) povezani drug z drugim. To je nekako tako, kot če bi poskušali razumeti zaplet zgodbe tako, da bi upoštevali, kako so liki povezani med seboj.

*Začetek odgovora*: Postopek generiranja se začne s posebnim žetonom, ki se pogosto imenuje žeton začetka zaporedja, ki je označba, ki nakazuje, da bo začel ustvarjati nov odziv. To je podobno kot velika tiskana črka na začetku stavka, ki naznanja začetek nove misli.

*Napovedovanje naslednjega žetona*: Po žetonu začetka odgovora model predvidi naslednji žeton. Tega ne stori v vakuumu. Namesto tega uporabi kontekst, ki ga zagotavlja uporabnikovo vprašanje, in vse žetone, ki jih je že ustvaril za odgovor. Model izračuna verjetnosti vseh možnih naslednjih žetonov in izbere enega naključnega, kjer pa je možnost za izbiro vsakega prilagojena na že obstoječi kontekst (dosedanja vprašanja in odgovore).

Na primer, če je uporabnik vprašal: »Kaj je prestolnica Francije?« in je model začel svoj odgovor z »Prestolnica«, je naslednji žeton lahko »Francije«. To je zato, ker se je model na podlagi učnih podatkov naučil, da je »prestolnica« pogosta beseda, ki sledi besedi »Francije« v kontekstu vprašanja o geografiji.

*Ponavljanje postopka*: Model nato zaporedju doda nov žeton »je« in na podlagi posodobljenega konteksta predvidi naslednji žeton »Pariz«. Ob koncu odgovora model ustvari žeton za konec zaporedja oz. odgovora in s tem zaključi dopolnjevanje, ki se v našem primeru glasi »Prestolnica Francije je Pariz«. Tako se odgovori tvorijo vedno do zaključnega žetona, ki označuje zaključek odgovora ali pa do najdaljše možne meje odgovora. Namreč, trenutni veliki jezikovni modeli imajo omejitev koliko žetonov lahko prejmejo in napovedo (pri GPT-4 je to okoli 32 ali 64 tisoč žetonov – odvisno od verzije).

*Obravnava negotovosti*: Včasih se lahko zgodi, da je model negotov glede naslednjega žetona, saj ima več žetonov podobno verjetnost. V takih primerih model uporabi strategijo, imenovano *temperatura*, ki v bistvu določa, koliko tveganja je model pripravljen sprejeti. Visoka temperatura pomeni, da je model bolj pripravljen tvegati in izbrati manj verjetne žetone, kar lahko privede do bolj ustvarjalnih, vendar morda manj natančnih odzivov. Nasprotno pa nizka temperatura pomeni, da model raje igra na varnem in se drži najbolj verjetnih žetonov in s tem tvori ponovljive odgovore, ki so bolj v skladu z besedilom, ki ga je model videl tekom svojega učenja. Posledica manjše temperature pa je manjša kreativnost in večja verjetnost ponavljanja že videnega.

*Dekodiranje odziva*: Ko je odziv v celoti ustvarjen, je še vedno v obliki žetonov. Zadnji korak je dekodiranje teh žetonov nazaj v človeku berljivo besedilo. To je kot sestavljanje sestavljanke, pri čemer se posamezni žetoni spremenijo nazaj v koherentno in smiselno besedilo.

Ustvarjanje odgovora je torej občutljiv ples med uporabo naučenih vzorcev iz že prej videni (učnih) podatkov, razlago konteksta, ki ga sestavlja uporabnikov vnos z dosedanjimi odgovori, in uporabo statističnih verjetnosti, da se v vsakem koraku pripravi najboljša možna napoved. Končni cilj takih modelov je ustvariti odgovor, ki ni le slovnično pravilen, temveč tudi kontekstualno in semantično ustrezen, zanimiv in smiseln za uporabnika. Ta cilj je včasih dosežen, drugič pa spet ne. V naslednji sekciji sledi pregled kakšni so lahko rezultati opisanega procesa odgovarjanja.

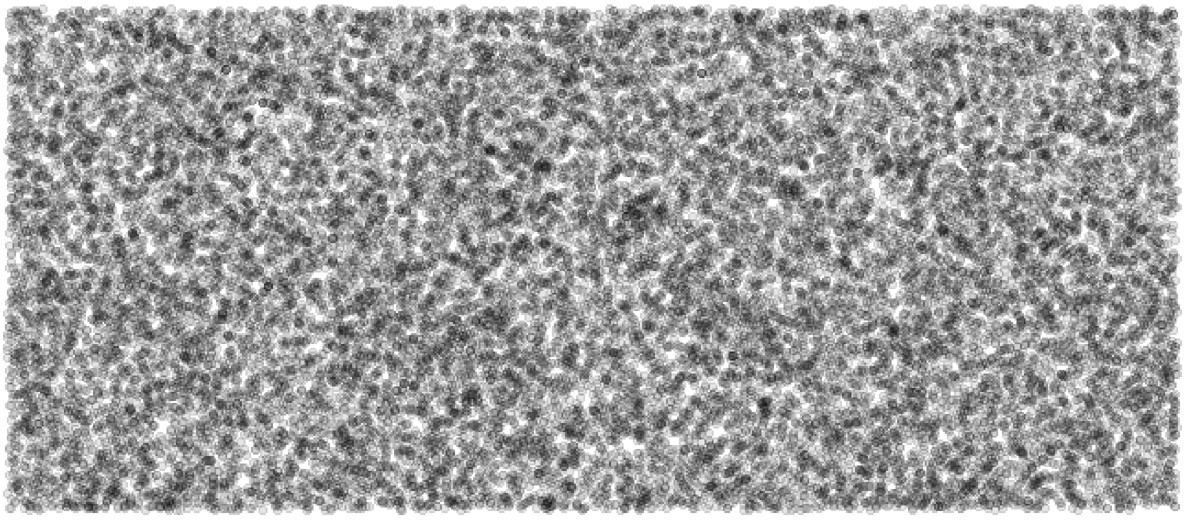
## 4.1 Kakšni pa so odgovori?

### *4.1.1 Nesmiselni odgovori*

Veliki jezikovni modeli imajo pri odgovarjanju na določeno poizvedbo na voljo številne možne odgovore. Te odgovore oblikujejo na podlagi verjetnosti, ki so dodeljene vsakemu možnemu žetonu v njihovem besednjaku. Teoretično to pomeni, da lahko jezikovni model izbere kateri koli žeton, tudi najmanj verjeten, in tako ustvari odgovor, ki je lahko nesmiseln. Če vizualiziramo vse možne odgovore v prostoru, dobimo sliko 3. Pozor, vse sledeče slike v resnici ne prikazujejo odgovor vseh možnih odgovorov, saj bi prikaz toliko označb naenkrat privedel do nerazpoznavne vizualizacije. Posledično je prikazan le prostor, kjer je hipotetično možnih 1.000.000 odgovorov.

**Slika 3: Prostor odgovor z vsemi možnimi odgovori. Slika je v celoti črna, ker so prikazani vsi možni odgovori: nesmiselni, smiselni; kratki (enoznakovni), polno dolgi; nepravilni in pravilni (na vsa možna vprašanja).**

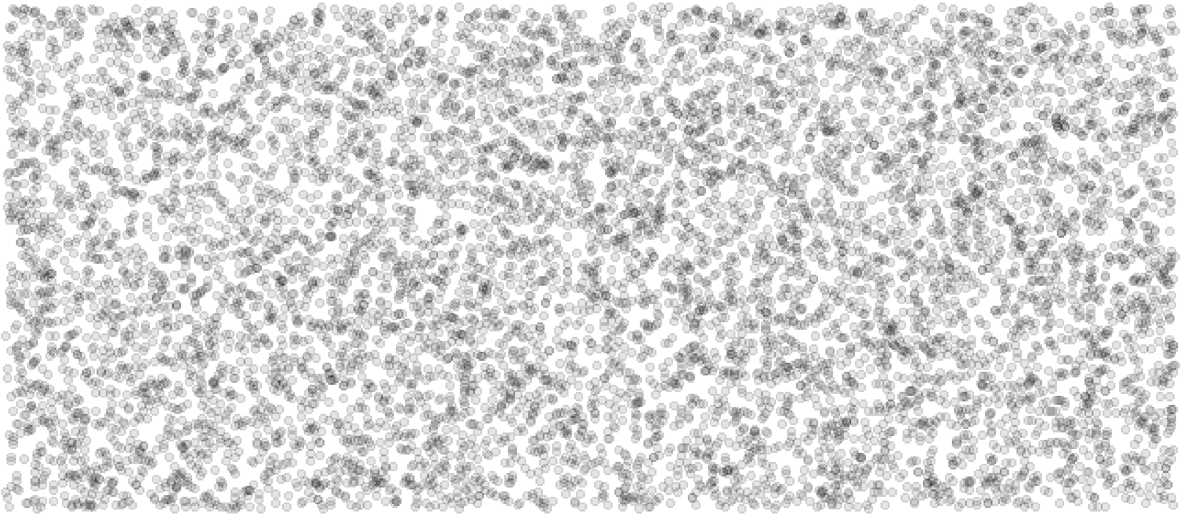
Drži, slika je popolnoma črna, ker vsaka pika predstavlja vse možne odgovore, tudi take, ki so nesmiselni, tudi kratke (enoznakovne) in dolge v vsej možno dolžini, tudi nepravilne in pravilne (za vsa možna vprašanja). Verjetnost, da pa model vrne nesmiselni odgovor, je zelo majhna, zato so takšni dogodki v običajnih okoliščinah praktično nemogoči. Kljub temu je tehnično izvedljivo, da model oblikuje odgovor, ki je sestavljen iz nerazumljivih zaporedij, kot je »njlkfdsjfklsa«. Če pa iz vizualizacije odstranimo vse nesmiselne odgovore, pa se znebimo velike večine vseh možnih kombinacij žetonov in nam tako ostane zelo majhen nabor (relativno glede na število vseh možnih odgovorov). Le smiselni odgovori so prikazani na sliki 4.



**Slika 4: Prostor odgovor s smiselnimi odgovori. Ti odgovori še vedno lahko vsebujejo sintaktične (slovnične in tipkarske), semantične in logične napake ter neresnice.**

### 4.1.2 Smiselni odgovori s sintaktičnimi napakami

Nekoliko večja verjetnost je, da veliki jezikovni modeli tvorijo smiselne odgovore, ki pa lahko še vedno vsebujejo sintaktične napake. Pri iskanju naslednjega najverjetnejšega žetona lahko jezikovni model občasno vključijo žetone, ki vodijo do sintaktičnih (slovničnih ali tipkarskih) napak. Nekatere od teh napak se lahko pojavijo naključno, vendar lahko glede na kontekstualno razumevanje jezikovni modeli pogosto takoj popravijo potek odgovora v naslednjih žetonih. Vendar pa začetna napaka ostane kot del odgovora, za razliko od človeške komunikacije, kjer lahko svoje napake popravimo za nazaj.



**Slika 5: Prostor odgovor s smiselnimi odgovori brez sintaktičnih napak. Ti odgovori še vedno lahko vsebujejo semantične in logične napake ter neresnice.**

Dodaten vir teh napak je mogoče razbrati iz podatkov za učenje takih velikih jezikovnih modelov. Če učni podatki vsebujejo tipkarske in slovnične napake, se lahko model nenamerno nauči teh vzorcev in jih ponovi v svojih rezultatih. Zato je pogostost pojavljanja napake v odzivih jezikovnega modela povezana s tem, kako pogosta je bila ta napaka v podatkih za usposabljanje.

Slika 5 prikazuje prostor odgovorov, kjer so nadaljnje odstranjeni še odgovori s sintaktičnimi napakami. Čeprav bi nam odgovori z malo in majhnimi sintaktičnimi napakami še vedno lahko prišli prav (saj so lahko pravilni), pa v namen nadaljnje razlage te odstranimo za namen bolj pregledne vizualizacije.

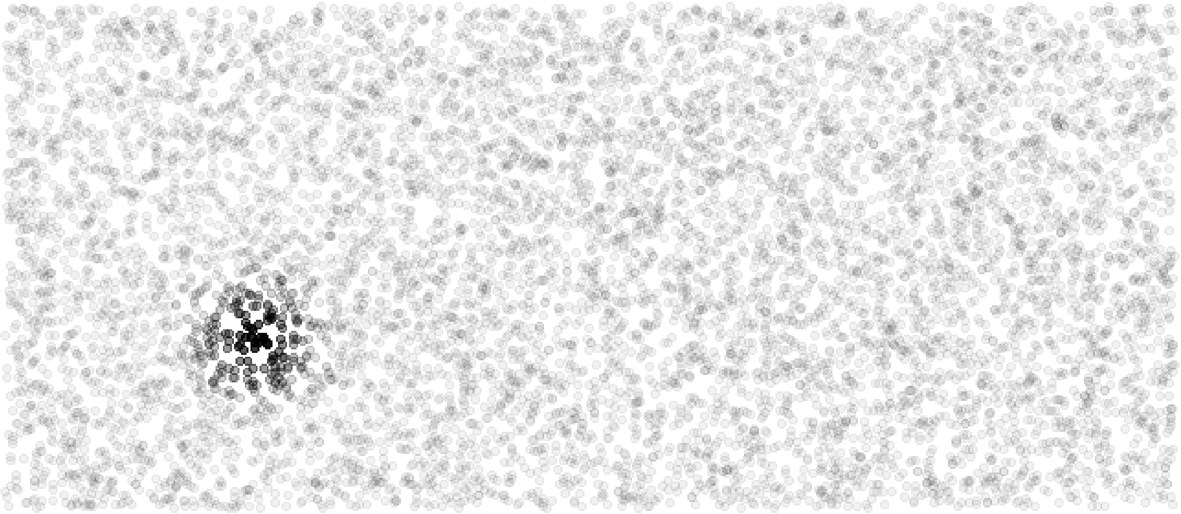
### 4.1.3 Smiselni a napačni odgovori brez napak

Veliki jezikovni modeli, ki so v večini primerov zelo natančni, lahko ustvarijo tudi razumne odgovore, ki pa so dejansko napačni. Do tega pojava lahko pride zaradi dveh ključnih razlogov.

Prvič, če učni podatki modela lahko vsebujejo nepravilne in neresnične informacije in ob učenju iz njih se model lahko nauči teh neresnic in jih ponovi v svojih odgovorih. Ta težava je še posebej izrazita, kadar se poizvedbe nanašajo na teme *v distribuciji* – to so teme, ki so dobro zastopane v učnih podatkih. Ker model generira svoje odgovore na podlagi vzorcev, ki se jih je naučil iz teh podatkov, se lahko napačne informacije, nehote predstavijo kot dejstva. Verjetnost, da se to zgodi, se povečuje s pogostostjo netočnih informacij v učnih podatkih, zlasti če so povezane z manj kontroverznimi temami (temami, katerih nasprotovanj je zelo malo v učnih podatkih).

Drugič, netočnosti se lahko pojavijo tudi pri obravnavi poizvedb *izven distribucije* ali poizvedb, ki se nanašajo na informacije, ki niso zajete v učnih podatkih. V teh primerih mora jezikovni model na podlagi naučenih vzorcev oblikovati utemeljena ugibanja, čeprav v svojem učnem korpusu nikoli ni videl pravega odgovora. Ta ugibanja so lahko včasih napačna, kar imenujemo *halucinacije*.

Sedaj pa slika 6 prikazuje vse možne odgovore s prilagojeno prosojnostjo glede na njihovo verjetnost, da bodo tvorjeni za eno fiktivno vprašanje (za en kontekst). Temnejši ko je odgovor, večja verjetnost je, da bo to končen odgovor. Če bi postavili popolnoma drugo vprašanje, bi gruča manj prosojnih (temnejših) oz. bolj verjetnih odgovorov bila popolnoma drugje.



**Slika 6: Prostor odgovor s kjer so odgovori s prosojnostjo označeni glede na njihovo verjetnost, da bodo tvorjeni za podano fiktivno vprašanje. Manj prosojni imajo večjo verjetnost, da bodo tvorjeni.**

Recimo, da smo jezikovni model prosili, naj nam tvori uvodno vabilo na našo konferenco. Že, če sami premislimo, bi to poleg dejansko uporabljenega stavka »Vabljeni na 26. konferenco Sodobne informacijske tehnologije in storitve« lahko ustvarili številne alternative, ki be imele enak semantični pomen. In prav te alternative bi bile temnejše na sliki 6. Nekaj takih, ki so semantično blizu (ne nujno identično) in bi bili temnejše obarvani na sliki je glede na ChatGPT sledečih:

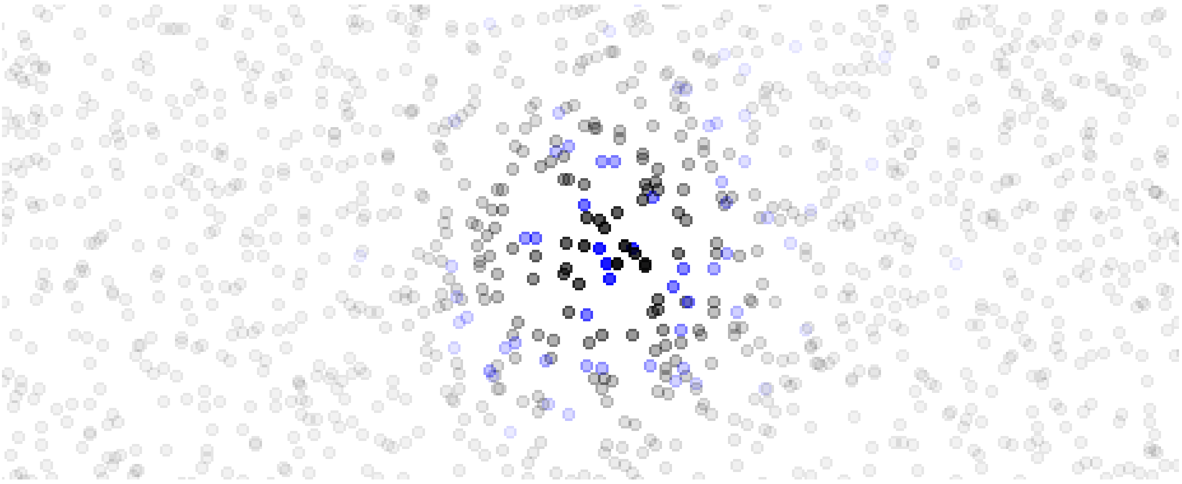
- *Dobrodošli na 26. konferenci Sodobne informacijske tehnologije in storitve.*
- *Pozdravljamo vas na 26. konferenci Sodobne informacijske tehnologije in storitve.*
- *Z veseljem vas vabimo na 26. konferenco o Sodobnih informacijskih tehnologijah in storitvah.*
- *Toplo vabljeni na konferenco Sodobne informacijske tehnologije in storitve, ki je že 26. po vrsti.*
- *Pridružite se nam na 26. konferenci na temo Sodobne informacijske tehnologije in storitve.*

### 4.1.4 Smiselni in pravilni odgovori brez napak

Kljub vsem do sedaj naštetim morebitnim pastem pa lahko veliki jezikovni modeli ustvarijo tudi povsem točne in razumne odgovore brez semantičnih ali sintaktičnih napak. Verjetnost takšnih popolnih odgovorov je večja pri poizvedbah v distribuciji, saj model pozna ustrezne vzorce v podatkih.

Tudi za poizvedbe zunaj distribucije lahko jezikovni modeli pripravijo pravilne in resnične odgovore. Čeprav so takšni rezultati precej odvisni od naključja, ima ključno vlogo sposobnost modela, da izlušči in uporabi semantične in logične vzorce iz svojega učnih podatkov. Z uporabo teh naučenih vzorcev v novih situacijah lahko model ustvari nove in natančne odgovore, kar kaže na njegov močan potencial kot orodja za generiranje besedila.

Slika 7 prikazuje odgovore, kjer so pravilni obarvani modro, kjer pa še vedno ohranimo prosojnostjo, ki prikazuje verjetnost za tvorjen odgovor. Tokrat pa zaradi boljše preglednosti slika prikazuje le povečavo področja, kjer je večja verjetnost kreiranih odgovorov. Iz slike je možno razbrati, da niso prav vsi najverjetnejši odgovori tudi pravilni. Hkrati pa so nekateri pravilni odgovori tudi taki, katerih tvorjenje ima manjšo verjetnost.



**Slika 7: Prostor odgovor s kjer so pravilni odgovori obarvani modro, s prosojnostjo pa se prikazuje verjetnost, da bo odgovor kreiran.**

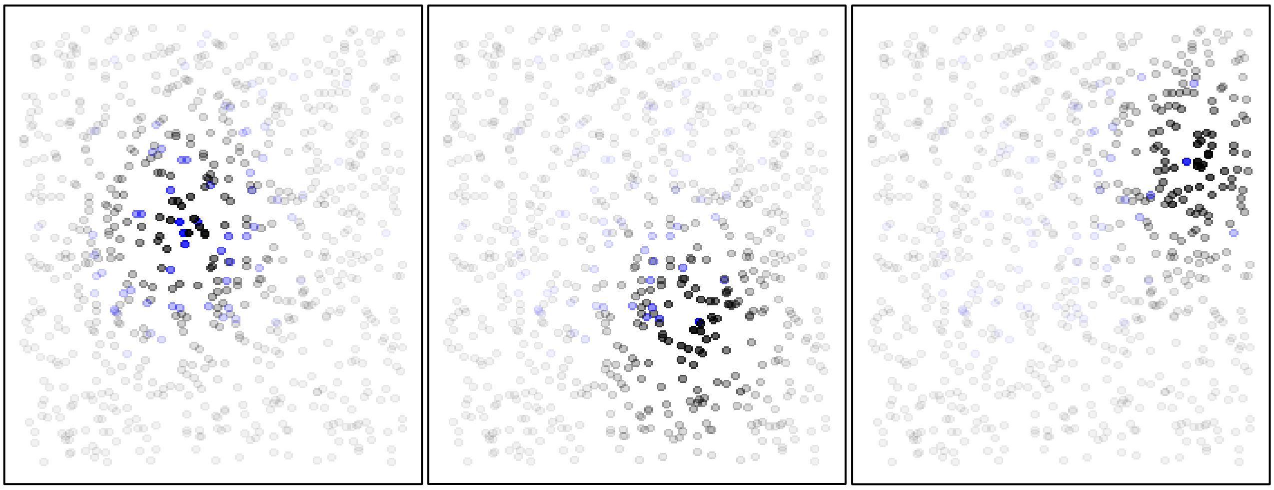
Pri postavljanju vprašanj imamo vsekakor cilj, da dobimo odgovor modela, ki bo pravilen (v vseh vidikih) oz. da vsaj povečamo možnost, a nam model vrne pravilen odgovor. Na to lahko vplivamo na dva načina: da prestavljamo center najbolj verjetnih odgovorov in da zožimo oz. zmanjšamo gručo najverjetnejših odgovorov.

## 4.2 Ovrednotenje pravilnosti odgovorov

Začnimo s *prestavljanjem centra najbolj verjetnih odgovorov*. Vidimo, da presek pravilnih in najbolj verjetnih odgovorov ni nujno popoln. Že če za malenkost prilagodimo vprašanje (spremenimo kontekst), se lahko verjetnost (prosojnost) odgovorov drastično premakne, kljub temu, da pravilni odgovori (modro obarvani) ostanejo enaki. Naš cilj je vsekakor tvoriti taka vprašanja, da bi bili pravilni in najbolj verjetni odgovori v popolnem preseku. To je prikazano na sliki 8.

Če imamo znanje, da sami ocenimo pravilnost podanega odgovora, lahko pridemo vsaj do zaključka, če je vprašanje potrebno popraviti ali ne. Če se odločimo za popravek vprašanja, pa je izziv, da ob tvorjenju vprašanj, takih grafov ne vidimo (saj tudi ni računskih virov, da bi veliki jezikovni modeli izračunali verjetnost vseh možnih odgovorov), zato težko popravljamo vprašanja s ciljem popolnega preseka. Ponovno, če imamo znanje, je dobra praksa, da z izpostavljanjem napak prejšnjih odgovorov, z dodatnimi vprašanji ali prestrukturiranjem izvornega vprašanja pridemo do pravilnega odgovora.

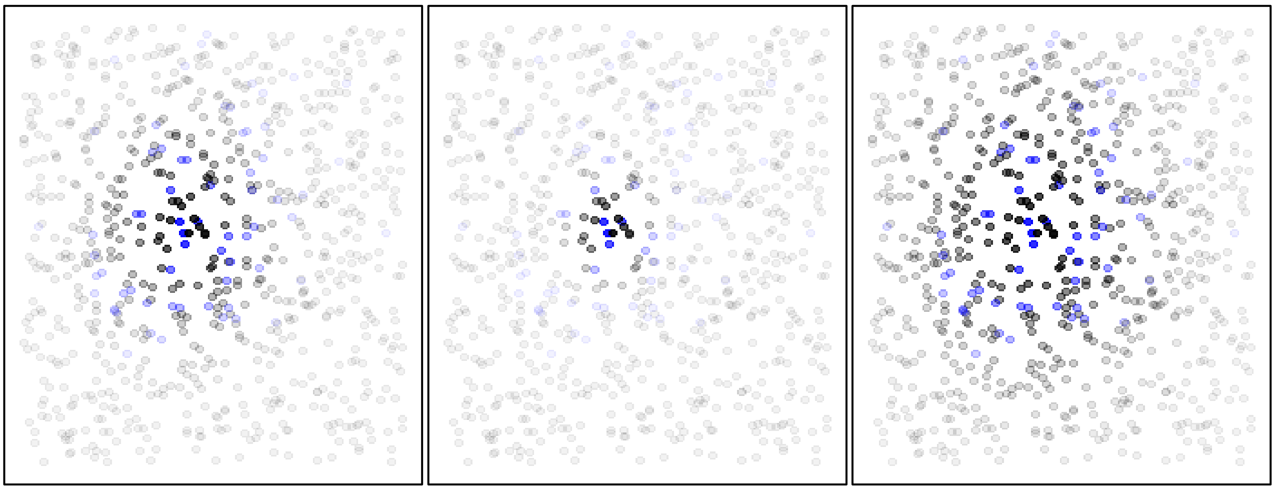
Če pa znanja za ocenitev pravilnosti odgovora ne posedujemo, pa je nemogoče identificirati halucinacije in nepravilnosti v odgovorih. Prav to je ena izmed pomanjkljivosti trenutnih jezikovnih modelov – da je njihova uporabnost zelo omejena, če jih uporabljamo za delo, katero ne vemo oceniti, če je pravilno ali ne. V primeru, da naloga ni občutljive narave (kreativno tvorjenje besedila ali kode) in katerih neprimerna uporaba ne bo povzročila resnih posledic, s tem večjega izziva ni. Če pa iščemo odgovore, ki bodo igrali pomembno vlogo v pomembnih odločitvah, ki imajo lahko resne posledice, pa je spraševanje o temah, ki jih sami ne obvladamo izredno zaskrbljujoče.



**Slika 8: Prikaz različno oblikovanih ampak semantično enakih vprašanja.**

### 4.3 Postavljanje primernih vprašanj

Naslovimo še z drugim načinom vplivanja na verjetnost, da dobimo pravilni odgovor: omejitvijo nabora najverjetnejših odgovorov. Če postavljamo vprašanja, ki sličijo na kratke in sintaktično nepravilne iskalne nize, ki smo se jih desetletja učili in uporabljali v iskalnikih (Google, Bing, DuckDuckGo…), bo nabor bolj verjetnih odgovorov mnogo večji. Večji ko je nabor najverjetnejših odgovor, večjo vlogo igra naključje kateri odgovor bomo dobili. Hkrati pa večji nabor najverjetnejših odgovorov razširi temnejšo gručo in s tem potencialno poveča razliko med presekom in ostankom z naborom pravilnih odgovorov.



**Slika 9: Prikaz preoblikovanja vprašanja, ki zoži ali poveča nabor najverjetnejših odgovorov in s tem spremeni presek z naborom pravilnih ogovorov.**

Po drugi strani pa tem, če bolj vprašanja napolnimo s podrobnostmi in navodili za odgovor, manjši bo obseg najbolj verjetnih odgovorov. Tako zmanjšamo vlogo naključja in zmanjšamo (v idealnosti celo izničimo) razliko med presekom z pravilnimi odgovori. Preoblikovanje vprašanja, ki zoži ali poveča nabor najverjetnejših odgovorov je prikazan na sliki 9.

Za zoženje nabora možnih odgovorov uporabimo enake tehnike, kot bi jih uporabili z delom z ljudmi pri iskanju odgovorov. Praksa kaže, da se zelo dobro obnesejo vprašanja, kjer je hkrati podanih več podvprašanj, ki peljejo do končnega vprašanja. Na tako strukturo bo jezikovni model sproti odgovarjal najprej na podvprašanja in ko pride do dela, kjer odgovarja na krovno vprašanje, bodo del njegovega konteksta že njegovi odgovori na podvprašanja. Še vedno obstaja verjetnost, da bo tvoril napačen odgovor, ampak bo ta manjša, saj bo verjetno že ob podanem podvprašanju za to manjša.

Druga tehnika je, da vsilimo iskanje odgovora po korakih (angl. *chain-of-thought*). Pri takem načinu preprečimo »na blef« odgovarjanje in vsilimo, da model najprej »premisli« o vprašanju in razloži postopek kako je prišel do odgovora. Ta tehnika deluje zaradi enakega razloga kot prejšnja, verjetnost, da bodo celoten postopek, razmišljanje in končen odgovor modela napačno je manjša kot pa je verjetnost, da bo napačen le kratek odgovor. Tipično vprašanje, ki skorajda obvezno potrebuje tak pristop je reševanje matematičnih enačb. Tako ljudje, kot tudi jezikovni modeli potrebujejo vmesne izračune, da pridemo do rešitve zapletenih matematičnih nalog.

Naslednja tehnika je ponovno povzeta po delu z ljudmi – to je tehnika, ko poleg vprašanja modelu priložimo še en (*single-shot*) ali več (*multi-shot*) primernih ali pravilnih odgovorov. Pri takem postopku bi tako ljudje, kot sam model, skopiral postopek odgovarjanja, le priredili bi ključen del odgovora, da bi ta bil smiseln za podano vprašanje. Dober primer kjer to deluje odlično je pisanje programske kode z regexom. Če modelu poleg prošnje za kodo podamo tudi kako bi naj zgledali pravilni ekstrahirani nizi znakov (tem lahko podamo tudi nepravilne), je večja verjetnost, da bo vrnjena programska koda pravilna. Ta tehnika je tudi zelo primerna, če želimo odgovore točno v določeni obliki (npr. formalno pismo, v francoskem jeziku, v JSON ali XML formatu…).

## 5 Zaključek

Ta prispevek je poskušal pojasniti nekaj osnovnih delovnih načel velikih jezikovnih modelov, ne da bi se potopil v globoko teoretično razpravo ali se zapletal v filozofske dileme, kot je problem kitajske sobe. Pregledali smo štiri scenarije, kako veliki jezikovni modelu delujejo v odgovarjanju na vprašanja, ki nudijo vpogled v različne ravni uspešnosti in točnosti teh modelov. Upamo, da bo ta jasen in "nazoren" pregled pripomogel k zagonu produktivnih debat, brez zmede in neupravičenih pričakovanj, in s tem prispeval k bolj uravnoteženemu in razumnemu pristopu k tej hitro rastoči in vznemirljivi tehnologiji.

## Literatura

[1] VASWANI Ashish, SHAZEER Noam, PARMAR Niki, USZKOREIT Jakob, JONES Llion, GOMEZ Aidan N., KAISER Łukasz, in POLOSUKHIN Illia »Attention is all you need«, Advances in neural information processing systems, številka 30, leto 2017.

[2] FLORIDI Luciano in CHIRIATTI Massimo "GPT-3: Its nature, scope, limits, and consequences.", Minds and Machines, številka 30, leto 2020, 681-694.

[3] LIN Tianyang, WANG Yuxin, LIU Xiangyang in QIU Xipeng "A survey of transformers." AI Open, leto 2022.

[4] LIU Xiao, ZHENG Yanan, DU Zhengxiao, DING Ming, QIAN Yujie, YANG Zhilin in TANG Jie "GPT understands, too." arXiv preprint arXiv:2103.10385, leto 2021.

[5] ETHAYARAJH Kawin. "How contextual are contextualized word representations? Comparing the geometry of BERT, ELMo, and GPT-2 embeddings." arXiv preprint arXiv:1909.00512, leto 2019.

[6] DIS Van, AM Eva, BOLLEN Johan, ZUIDEMA Willem, VAN ROOIJ Robert in BOCKTING Claudi L. "ChatGPT: five priorities for research." Nature 614, številka 7947, leto 2023, 224-226.

# Web 3 vs Web 2: Konceptualne in tehnološke razlike

Muhamed Turkanović, Vid Keršič, Martina Šestak

Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko,
Maribor, Slovenija
muhamed.turkanovic@um.si, vid.kersic@um.si, martina.sestak@um.si

Članek se osredotoča na konceptualne in tehnološke razlike med razvojem dveh paradigem spletnega razvoja Web 2 in Web 3. Web 2 prevladuje, vendar prinaša Web 3 inovacije kot decentralizacijo, uporabo pametnih pogodb, nezamenljive žetone itn. Predstavljene so tudi ključni trendi Web 3, kot so decentralizirane identitete in metaverzum. Za namen primerjave Web 2 in Web 3, smo le to izvedli na primeru razvoja in uporabe spletne platforme za zbiranje daril. Predstavili smo tudi izzive Web 3, ki vključujejo skalabilnost in uporabniško izkušnjo, pri čemer se pojavljajo nove rešitve za boljšo usklajenost s tradicionalnimi standardi in koncepti. Kljub izzivom Web 3 prinaša le ta nove priložnosti in dviguje ozaveščenost o spremembah v spletnem razvoju.

**Ključne besede:**
decentralizacija
web 3.0
metaverzum
tehnologija veriženja blokov
arhitektura

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.2

# 1 Uvod

Prispevek raziskuje in predstavlja konceptualne in tehnološke razlike med Web 3 in Web 2 – dve različni paradigmi spletnega razvoja. Web 2, ki ga poznamo že vrsto let in prevladuje še danes ter Web 3, ki predstavlja nov val inovacij, ki temeljijo na transparentnosti, uporabniškemu nadzoru in kot ključno na **decentralizaciji**. Web 2 je zaznamovala centralizacija podatkov in moči, kjer velike platforme kot so Google, Facebook in Amazon vladajo nad spletnim okoljem, pri čemer smo se uporabniki podredili njihovim pravilom in omejitvam. Web 3 izhaja iz tehnologij, kot so verige blokov (ang. blockchain), pametne pogodbe in porazdeljeni ter decentralizirani sistemi za shranjevanje podatkov in datotek. Te omogočajo uporabnikom večjo nadzor nad svojimi podatki in transakcijami ter posledično tudi večjo varnost, zasebnost in avtonomijo. Prav tako se danes Web 2 usmerja predvsem v oblačne platforme in številne spletne storitve, medtem ko Web 3 spodbuja nove modele delovanja in poslovanja, kot so uporaba decentraliziranih aplikacij (DApps), ki so praviloma brezstrežniške, uporaba pametnih pogodb namesto poslovne logike, ki deluje na strežnikih, podpora kripto valutam in drugim kripto žetonom namesto uporabe obstoječih (centraliziranih) mehanizmov plačevanja, itn. Prispevek se bo tudi ozrl na povezljivost Web 3 s področjem metaverzuma (ang. metaverse).

V prispevku bomo podrobno predstavili koncept Web 3, vključno z vsemi ključnimi tehnologijami, ki pomagajo danes definirati omenjeno področje. Predstavili bomo tudi različne koncepte, ki jih Web 3 na novo definira, kot npr. uporaba decentraliziranih ali samoupravljanih identitet (ang. decentralized or self-sovereign identity), uporaba digitalnih denarnic kot osnove za nadzor in sodelovanje v svetovnem spletu, upravljanje in trgovanje z digitalnimi dobrinami (npr. nezamenljivi žetoni). Raziskali bomo, zakaj in kako je Web 3 povezan s področjem metaverzuma. Podrobneje bomo prikazali arhitekturo aplikacije Web 3 in to primerjali z konceptualno ekvivalentno rešitvijo, ki temelji na Web 2.

V zaključku se osredotočamo na izzive, s katerimi se sooča Web 3 pri prehodu od koncepta do množične uporabe. Čeprav Web 3 prinaša številne prednosti, obstajajo tudi ovire, kot so skalabilnost, uporabniška izkušnja in sprejemanje s strani tradicionalnih institucij. Prav tako pa izvedemo kratko diskusijo o tem, ali je Web 3 resnično korak naprej v evoluciji spleta, ali zgolj nov koncept, ki ga spremljajo nove tehnologije in pristopi, ki jih bomo uporabljali ob bok klasičnemu spletu. Prispevek je pomemben za razumevanje razlik med Web 2 in Web 3 ter za ozaveščanje o novih priložnostih in izzivih, ki jih prinaša razvoj spletnih tehnologij.

# 2 Splet

Zavedajoč se splošnega razumevanja evolucije spleta, na kratko predstavimo dejstva o Web 2.0. Ta predstavlja prelomnico v razvoju spletnih tehnologij, ki je prinesla interaktivno in dinamično izkušnjo na spletu. Razvijajoč se v drugi polovici 2000-ih let, nadgradi prejšnjo statično in enosmerno naravo spletnih strani, ki so pred tem ponujale le vsebine, brez aktivnega sodelovanja uporabnikov. Koncept Web 2.0 je prinesel celoten spekter sprememb, od tehničnih do vsebinskih, in bistveno vplival na način, kako ljudje komunicirajo, delijo vsebine in uporabljajo spletne storitve.

Ena ključnih značilnosti Web 2.0 je vključevanje uporabnikov v proces ustvarjanja in deljenja vsebin, kjer uporabniki niso več pasivni opazovalci, temveč aktivni udeleženci. To je privedlo do vzpona socialnih medijev in drugih oblik delovanja in udejstvovanja na spletu, kot je e-poslovanje ter do vzpona vsem znanih velikih IT korporacij (npr. Facebook in Amazon). Prav tako med pomembne značilnosti omenjamo realno-časovno komunikacijo, izmenjavo vsebine in sodelovanje.

Med tehnološke spremembe velja posebej izpostaviti tiste, ki so povezane s tehnološkimi koncepti, kot so **računalništvo v oblaku**, **masovni podatki** (ang. Big Data) ter danes še posebej pomembo **umetno inteligenco**.

Kljub svojim pozitivnim inovacijam ima Web 2.0 tudi svoje omejitve, kot so centralizacija moči, vprašanja glede varstva in lastništva podatkov in omejitve uporabniškega nadzora. Ena največjih slabosti Web 2.0 je ravno centralizacija moči v rokah nekaj velikih korporacij. Web 2.0 je pogosto prinesel omejitve glede uporabniškega nadzora nad lastnimi podatki in vsebinami. Uporabniki pogosto morajo sprejeti pogoje uporabe, ki jim odrekajo popoln nadzor nad tem, kar delijo in ustvarjajo na platformah. S tem se zmanjšuje njihova avtonomija in svoboda, kar lahko vodi v težave z zasebnostjo in varnostjo podatkov. Web 2.0 platforme imajo pogosto možnost, da omejijo ali cenzurirajo vsebine, ki so na njih objavljene. Čeprav je v določenih primerih to lahko upravičeno za preprečevanje širjenja nezakonitih vsebin ali sovražnega govora, lahko to tudi pomeni omejevanje svobode izražanja in raznolikosti mnenj na spletu.

## 3 Web 3

Če Web 1.0 označujemo kot splet samo za branje, Web 2.0 kot Splet branja in pisanja, se Web 3.0 označuje kot **Splet branja, pisanja in lastninjenja**. Poglavitna ideja za konceptom Web 3 je izpolnitev prvotnega obljubljenega potenciala interneta kot odprtega, transparentnega in varnega sistema računalniških omrežij, ki bi v ospredju združeval uporabnike.

Zaradi fokusa in omejitev v prispevku predvidevamo osnovno razumevanje določenih že splošno znanih Web 3 konceptov in tehnologij. Glavna lastnost na katerih temelji Web 3 in je osnova za vse ostalo je **decentralizacija**. Le ta IT skupnosti ni tuja, vendar je šele z razcvetom tehnologije veriženja blokov (ang. blockchain - TVB) resnično pridobila na pomenu. TVB v osnovi gradi na odprtih, javnih omrežij verig blokov, ki združuje številna vozlišča, pri čemer nima nihče popolnega nadzora nad omrežjem. Omrežje namenskih vozlišč na porazdeljeni osnovi hrani zgodovino vseh transakcij in hkrati na decentraliziran način omogoča dogovor za zapis novega skupka potrjenih transakcij (t. i. blok). Na ta način se definira najnovejši stanoviten (ang. stateful) skupek transakcij za katero se vsi udeleženi strinjajo, da so legitimne in skladne z dosedanjo zgodovino in trenutnim stanjem. Sicer obstaja tudi več vrst omrežij verig blokov, kot so tiste zaprtega tipa (ang. permissioned), vendar le te niso v ospredju Web 3, četudi se veliko uporabljajo. Prav tako je Web 3 neodvisen od tipa decentralizirane platforme, saj TVB ni edina možna rešitev. Obstajajo tudi druge vrste tehnologij porazdeljenih knjig (ang. Distributed Ledger Technology - DLT), ki so lahko v uporabi.

Z dopolnitvijo TVB s **pametnimi pogodbami** se razcvet Web 3 šele začne, saj ponudijo le te poljudno programljivo poslovno logiko, ki je tudi nujno potrebna za nove bogate primere uporabe. S pojavom pametnih pogodb govorimo tudi o TVB 2.0, saj je prejšnja različica TVB 1.0 bila omejena zgolj na podporo kripto valutam kot nove oblike digitalne vrednosti. Pametne pogodbe pa v osnovi ponudijo možnost za razširjanje koncepta **tokenizacije**, saj si lahko kdor koli ustvari svoje namenske **kripto žetone**, ki so se prav tako razvijali od enostavnih zamenljivih (ERC-20) do nezamenljivih (ERC-721, ang. non-fungible token - NFT) in še naprej [1]. Te so privedle do porasta različnih poslovnih primerov uporabe, s tem pa omogočijo lastno plačilno sredstvo, ki je neodvisno od trenutnih centraliziranih rešitev. NFT in podobne nestandardizirane oblike nezamenljivih žetonov postanejo skoraj vseobsežna komponenta vsake Web 3 rešitve. Nezamenljivi so iz razloga, ker je vsak ustvarjen žeton unikaten in se ga ne da enačiti ali zamenjati z drugim, kar jim omogoči, da postanejo medij za hranjenje različnih novih oblik digitalnih umetnin ali drugih vrednosti. Novi primeri uporabe hkrati privedejo do potrebe po namenskih aplikacijah, ki končnim uporabnikom omogočajo interakcijo s TVB in pametnimi pogodbami ter posledično z vsemi možnimi kripto žetoni. Takšne namenske aplikacije so se hitro poimenovale **decentralizirane aplikacije** (DApp) in so postale osrednji del Web 3. Zaradi omejitev, povezane predvsem s hrambo podatkov, so Web 3 rešitve hitro začele uporabljati dodatno komponento, ki jim omogoča neomejen shrambo, pri čemer pa tudi ta sledi konceptom decentralizacije. To so predvsem **sistemi porazdeljenega shranjevanja datotek** (ang. Distributed File Storage System), kot je npr. IPFS.

Da bi vse skupaj delovalo po principih Web 3 in da bi le ta deloval brezhibno pa morajo končni uporabniki biti tisti, ki bodo resnično nadzirali svoje podatke in digitalne vrednosti, kar se že od povojev TVB rešuje z uporabo namenskih **digitalnih denarnic** (DD). Podatki in včasih tudi digitalne vrednosti so sicer lahko hranjenje na kakšnih javno dostopnih TVB, vendar so zgolj njihovi lastniki tisti, ki lahko z njimi upravljajo. Z uporabo DD si končni uporabniki ustvarijo lastne digitalne profile, ki jih nadzirajo s pomočjo kriptografskih osnov tj., s parom kriptografskih ključev, temelječih na asimetrični kriptografiji. Vsaka sprememba stanja se digitalno podpiše, kar uporabniki počnejo z uporabo DD in svojih zasebnih ključev. DD v veliki večini omogočajo interakcijo z decentraliziranimi aplikacijami. Obstaja široka paleta vrst in oblik DD, ki jih pa ne bomo posebej predstavljali [2].

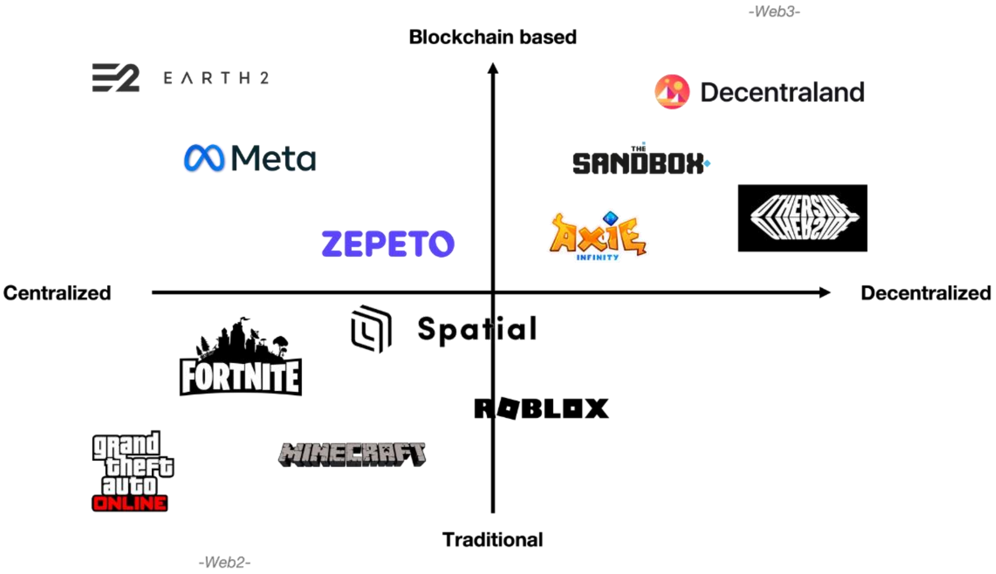
## 3.1 Trendi

Razen kriptografskih žetonov, tistih zamenljivih ali nezamenljivih, ki že sedaj poganjajo različne sisteme tokenizacije in druge primere uporabe, je Web 3 zajel še številne druge trende, ki jih bomo nekatere na kratko predstavili. Skupno vsem trendom je predvsem podpora lastnosti decentralizacije.

### *3.1.1 Metaverzum*

Upoštevajoč popularnost igre Fortnite in njene podpore zbirateljstvu igralnih pripomočkov, kakor tudi funkcionalnosti digitalnih kriptografskih žetonov, ki so se že uporabljali tudi kot nov način za podporo zabavi, tudi v obliki spletnih iger (npr. Cryptokitties), se je pojavila ideja o ustvarjanju novega navideznega sveta, ki bi sledil prvotnim idejam metaverzuma, kot sta bila The Palace in Second Life [3]. Tako je leta 2017 prišlo do ustanovitve t. i. Decentraland, ki predstavlja, na platformi Ethereum temelječ metaverzum. Decentraland se lahko smatra tudi kot **decentralizirana aplikacija** – Dapp. Metaverzum se kaže kot naslednji korak v razvoju digitalnih okolij, kjer se realnost in navideznost zlivata v enoten, interaktivni prostor. Metaverzum, kot ga poznamo danes, predstavlja ambiciozno vizijo povezave med fizičnim in digitalnim svetom. Platforme, kot sta Decentraland, Roblox, Meta idr. odpirajo vrata k virtualnim izkušnjam, kjer se lahko ljudje srečujejo, sodelujejo v dejavnostih ter ustvarjajo vsebino.

Danes se metaverzum smatra kot enoten navidezni svet, ki ga predvsem upravlja Facebook (Slika 1). To je predvsem posledica močne oglaševalske kampanje, ki jo vodi, sedaj že imenovano podjetje Meta.



**Slika 1: Vrste metaverzumov glede na obliko zagotavljanja lastnosti decentralizacije.**

Metaverzum kot tak definira navidezni prostor, ki je sicer 3D oblike in uporabnikom omogoča osnovne funkcionalnosti ustvarjanja lastnega avatarja, komunikacije z drugimi avatarji, prostega sprehoda po navideznem svetu itn. Metaverzum se kot trend izrazi predvsem zaradi naprednih naprav, ki podpirajo obogateno in navidezno resničnost (ang. augmented and virtual realizty - AR/VR), ki je predvsem v fokusu centraliziranih platform, kot je tista od Mete.

Ravno decentralizacija je ponovo osrednji del, kjer se danes različni metasvetovi med seboj razlikujejo, pri čemer se tudi **velikokrat metaverzum enači kar s konceptom Web 3**. Slednje je verjetno posledica samega naziva Web 3, ki bi naj napovedoval novo obdobje Spleta in idejo o tem, da se bomo v prihodnosti v splet "potopili" (ang. immerse) ter dejstva, da so se številni koncepti Web 3 prida uporabili v novejših zasnovah metaverzuma, kot je tokenizacija produktov znotraj metaverzuma. V resnici se Web 3 konceptualno loči od metasvetov, saj uporaba principov Web 3 v določenem metaverzumu privede zgolj do takšnega sveta, ki bo deloval po principih decentralizacije. Uporaba TVB v metaverzumu omogoča enostavno in varno lastništvo digitalnih produktov ali sredstev v obliki digitalnih kriptografskih žetonov, ki so vezani na uporabnikov decentralizirano identiteto, kot je v primeru Decentralanda na platformo Ethereum. Tako lahko uporabnik med uporabo platforme Decentraland kupi zemljišče, si zgradi objekt (npr. hišo), ga opremi s pohištvom ter kupi svojemu avatarju posebne superge, pri čemer so zemljišče, objekt, pohištvo in superge vsak za sebe nezamenljiv žeton NFT, ki si ga uporabnik v celoti varno in transparentno lasti in ga lahko kadar koli proda naprej [4]. Podobne koncepte najdemo tudi v centraliziranih platformah, kot so Meta ali Earth2, pri čemer pa je razlika ta, da so digitalni produkti, ki jih uporabniki kupijo vezani na njihov profil pri omenjeni platformi, plačani s klasičnimi sredstvi procesiranja finančnih transakcij in velikokrat tudi zaklenjeni v uporabo zgolj pod okriljem lastnika platforme. Posledično je tudi prodaja digitalnih produktov omejena pod okrilje ene platforme, v razliko od tistih iz Decentralanda, ki se lahko prodajajo in izmenjujejo na ločenih decentraliziranih digitalnih tržnicah za nezamenljive žetoni (npr. OpenSea).

Prav tako se metaverzum lahko opre na koncepte **decentralizirane avtonomne organizacije** (ang. decentralized autonomous organizations - DAO), ki omogočajo skupnostno upravljanje in odločanje, kar krepi demokratično naravo virtualnega prostora. Skozi to povezavo se obetajo nove možnosti za ustvarjanje poslovnih modelov, vključno s prodajo in trgovino z digitalnimi sredstvi v metaverzumu ter zaslužkom prek sodelovanja v skupnostih. Ključnega pomena je torej zagotavljanje varnosti, zasebnosti in uporabniškega nadzora, ki jih prinaša Web 3, da bodo uporabniki lahko brezskrbno izkoriščali potencial metaverzuma.

### 3.1.2 Decentralizirane in samoupravljanje identitete

Želja po decentralizaciji je hitro privedla do novih področij, ki bi naj imele koristi od novih tehnoloških paradigem. Tako je področje digitalnih identitet postalo naslednje v vrsti, saj je TVB že ponujala delno rešitev. Za uporabo TVB si uporabniki **sami ustvarijo** namenske račune, ki postanejo njihove **decentralizirane identitete**. Slednji naziv izhaja iz dejstva, da je zgolj uporabnik tisti, ki ima nadzor nad svojim zasebnih parom kriptografskih ključev, ki hkrati tudi postane del njegovega javnega digitalnega profila (v sklopu TVB se imenuje *naslov*), v razliko od centraliziranih digitalnih identitet, kjer za uporabnika profil ustvari, ga nadzira in omogoča določena tretja entiteta (npr. Facebook).

Decentralizirane identitete predstavljajo paradigmo, kjer imajo posamezniki popoln nadzor nad svojimi osebnimi podatki in digitalno identiteto. Namesto da bi se zanašali na osrednje organizacije za shranjevanje in preverjanje identitet, se decentralizirane identitete opirajo na TVB, pametne pogodbe in kriptografijo, kar omogoča uporabnikom, da sami upravljajo svoje podatke in izbirajo, s kom jih delijo.

Pomembna nadgradnja decentraliziranih identitet so modeli **samoupravljanih identitet** (ang. self-sovereign identity - SSI), ki razen upravljanja digitalnih profilov, vpeljejo obogaten koncept upravljanja z identitetnimi podatki in s tem povezanimi identifikatorji, brez interakcije z zunanjimi entitetami [5]. SSI prinaša revolucionarno spremembo na področje upravljanja identitet in omogoča uporabnikom, da prevzamejo aktivno vlogo pri upravljanju svojih identitetnih podatkov ter zmanjšajo potrebo po tradicionalnih centraliziranih mehanizmih, kot so ustvarjanje računov pri različnih ponudnikih storitev ali uporabo ponudnikov digitalnih identitet (ang. identity provider - IdP). V okviru modela SSI je vsak uporabnik sam odgovoren za ustvarjanje, hranjene, varovanje in drugo upravljanje svoje digitalne identitete. Slednje izvaja s pomočjo namenskih digitalnih denarnic [6]. Osnova modela so t. i. **decentralizirani identifikatorji** (ang. decentralized identifier – DID), ki so upravljani na podlagi para asimetričnih kriptografskih ključev, kjer je zasebni ključ orodje za nadzor identitete, javni pa osnova za generiranje identifikatorja. SSI model vpeljuje tri ključne akterje, ki igrajo bistveno vlogo pri izvajanju identifikacijskih procesov: **izdajatelja identitetnih atributov** (ang. Issuer), **imetnika identitete** (ang. Identity Holder) ter **preveritelj** oz. ponudnik storitev (ang. Verifier) [5]. Ti akterji sodelujejo v procesu izdajanja, preverjanja in upravljanja identitetnih podatkov na način, ki omogoča visoko stopnjo varnosti in zaupanja. Imetnik identitete (Identity Holder) je odgovoren za pridobivanje, shranjevanje, upravljanje in deljenje svojih identitetnih podatkov s preveritelji. Imetnik identitete lahko v različnih kontekstih deluje v različnih vlogah glede na zahteve in potrebe, kar omogoča prilagodljivost sistema. Izdajatelj identitetnih atributov ima ključno vlogo pri potrjevanju določenih atributov imetnika identitete. S pomočjo digitalnega podpisa izda **preverljivo poverilnico** (ang. verifiable credential – VC), ki vsebuje enega ali več atributov imetnika identitete. Ti atributi oz. trditve se nanašajo na različne vidike imetnikove identitete in so osnova za nadaljnje preverjanje in uporabo storitev. Preveritelj je entiteta, ki zahteva dokazilo o identiteti imetnika in izvaja preverjanje podatkov. S pomočjo tehnologij, kot so DID, VC ter TVB, se vzpostavi zaupanje med entitetami. Proces preverjanja tako ne zahteva vedno sodelovanja izdajatelja poverilnic, saj so vzpostavljene mehanizmi za preverjanje in verifikacijo na osnovi decentraliziranih tehnologij.

SSI ima potencial, da preoblikuje način, kako uporabniki komunicirajo in poslujejo na spletu. Povečanje varnosti, zasebnosti in enostavnosti preverjanja identitete lahko vodi do večje zaupanja med uporabniki in storitvenimi ponudniki. To pa lahko spodbudi razvoj novih poslovnih modelov, ki temeljijo na zaupanju, ter odpre vrata k inovacijam, ki še naprej preoblikujejo digitalni svet. Predstavljeni koncepti SSI, četudi temeljijo na decentralizaciji, so vzbudili širše zanimanje, kjer lahko kot primer izpostavimo eIDAS 2.0 (elektronska identifikacija in storitve zaupanja (angl. Electronic Identification and Trust Services)), ki ga uvaja Evropske unija z namenom podpore elektronskem poslovanju na enotnem notranjem trgu EU. Ta je za novo različico eIDAS (2.0) kot zahtevo vpeljala določene funkcionalnosti, ki izvirajo ravno iz modela SSI [6].

### 3.2 Rešitve

Zaradi boljšega razumevanja koncepta Web 3 bomo navedli nekaj primerov popularnih rešitev. Ena izmed teh je tudi metaverzum platforma Decentraland, ki smo jo na kratko predstavili nekoliko prej. Rešitve lahko grupiramo glede na koncept, ki ga zasleduje, in sicer so to lahko spletne igre (npr. Axie Infinity ali Cryptokitties), industrija umetnosti, socialna omrežja (npr. Sola), podporne storitve ali orodja (npr. Storj ali DEIP), decentralizirane finance (DeFi) (npr. Cashaa) itn.

**Storj** je decentralizirana platforma za shranjevanje podatkov, ki temelji na TVB in je povezana s konceptom Web 3. Njen namen je ponuditi varno, zasebno in robustno rešitev za shranjevanje podatkov, ki temelji na decentralizaciji in uporablja neizkoriščen prostor na računalnikih posameznikov po vsem svetu. Namesto da bi podatke shranjevali na centraliziranem strežniku, Storj omogoča, da se datoteke razdrobijo, šifrirajo in nato shranijo na mrežo različnih računalnikov, imenovanih "farmerji". To pomeni, da je naša datoteka razpršena po več lokacijah, kar izboljšuje varnost in odpornost proti napadom. Storj uporablja lasten kripto žeton, imenovan "STORJ", za spodbujanje sodelovanja in omogočanje plačil med uporabniki, ki oddajajo svoj prostor na disku (farmerji), ter tistimi, ki potrebujejo prostor za shranjevanje svojih podatkov. To ustvarja poslovni model, ki spodbuja deljenje prostora in vzdrževanje omrežja.

**Helium** se osredotoča na vzpostavitev decentraliziranega omrežja za internet stvari (IoT). Glavni cilj Heliuma je omogočiti napravam, kot so senzorji in naprave za sledenje, komuniciranje prek brezžične tehnologije in omogočiti zbiranje ter prenos podatkov na učinkovit in zanesljiv način. Helium uporablja algoritem Proof of Coverage, ki temelji na prisotnosti in dosegu naprav. Upravljavci naprav, imenovani "hotspoti", dokazujejo, da so dejansko pokrivali določeno območje, kar omogoča vzpostavitev zanesljivega omrežja brez potrebe po tradicionalni infrastrukturi. Helium omrežje uporablja lasten kripto žeton, imenovan "HNT" (Helium Network Token). Hotspoti prejemajo HNT kot nagrado za zagotavljanje pokritosti in prenašanje podatkov v omrežju. Tako se spodbuja sodelovanje in rast omrežja.

**Brave** je spletni brskalnik, ki se osredotoča na izboljšanje varnosti, zasebnosti in uporabniške izkušnje pri brskanju po spletu. Razvija ga podjetje Brave Software in temelji na odprtokodnem projektu Chromium, kar pomeni, da deli osnovo z Google Chrome brskalnikom. Najbolj znana funkcija brskalnika je vgrajeno blokiranje oglasov in sledilcev, ki preprečuje, da bi spletne strani sledile vašemu obnašanju na spletu in vas ciljale z oglasi. Hkrati ponuja brskalnik tudi možnost nagrad za ogled oglasov, pri čemer uporabniki prejmejo kripto žeton BAT (ang. Basic Attention Token) kot nagrado za svojo pozornost na oglaševalske vsebine.

**Cashaa** je finančna platforma, ki omogoča mednarodne denarne prenose, izmenjavo valut ter dostop do kripto valut. Platforma temelji na TVB ter si prizadeva za boljše povezovanje tradicionalnega finančnega sistema s svetom kripto valut in decentraliziranimi tehnologijami oz. decentraliziranih financ (DeFi).

**DEIP** (Decentralized Intellectual Property) je platforma za upravljanje intelektualne lastnine, ki temelji na TVB. DEIP si prizadeva za decentralizacijo, transparentnost in učinkovitost v procesih povezanih z intelektualno lastnino, kot so patentiranje, avtorske pravice, znanstvene raziskave in podobno. DEIP uporablja TVB za varno shranjevanje in sledenje informacij o intelektualni lastnini. To zagotavlja transparentnost in neizbrisljivost zapisov. Pametne pogodbe omogočajo avtomatizacijo in avtomatsko izvajanje pravic intelektualne lastnine. Na primer, avtor lahko določi, kako želi, da se uporabljajo njegova dela, in ta pravila se nato samodejno izvajajo. DEIP omogoča tudi tokenizacijo intelektualne lastnine, kar pomeni, da se pravice do dela lahko predstavijo kot digitalni kripto žetoni. To olajša trgovanje in izmenjavo intelektualne lastnine.

**Axie Infinity** je igralna platforma, ki temelji na TVB in se osredotoča na zbirateljsko igranje in trženje digitalnih bitij, imenovanih "Axies". Axie Infinity je ena izmed prvih in najbolj znanih iger, ki v celoti izkorišča koncepte Web 3, vključno z uporabo kripto valut, pametnih pogodb in decentralizacije. Axies so digitalna bitja, ki jih igralci zbira, izboljšuje in uporablja v bojih proti drugim igralcem. Vsak Axie je edinstven in ga lahko lastnik trži, vzreja in uporablja za sodelovanje v različnih igralnih načinih. Igra ima lastno tržnico, kjer lahko igralci kupujejo, prodajajo in trgujejo z Axies ter drugimi digitalnimi sredstvi. Tržnica deluje s pomočjo pametnih pogodb, kar omogoča avtomatizacijo transakcij in zagotavlja varnost.

**OpenSea** je ena izmed največjih in najbolj znanih platform za trgovanje z digitalnimi sredstvi, kot so kripto umetniška dela, kolekcionarski predmeti in drugi digitalni NFT-ji. OpenSea omogoča uporabnikom nakup, prodajo in trgovanje z NFT-ji. Uporabniki lahko iščejo in preglejujejo različne digitalne predmete ter sodelujejo v dražbah. OpenSea podpira številne različne standarde NFT-jev, kar omogoča trgovanje z NFT-ji, ustvarjenimi na različnih platformah TVB, kot so Ethereum, Polygon, Solana in drugi.

## 4 Primerjava Web 2 in Web 3

V tem poglavju podrobneje predstavimo primerjavo med Web 2 in Web 3 in sicer glede na uporabniški vidik ali vidik razvijalcev spletnih aplikacij. Kot osnovo za primerjavo uporabimo idejo za spletno aplikacijo, ki omogoča uporabnikom, da si ustvarijo svojo kampanjo za zbiranje priložnostnih daril. Primer je lahko namišljena oseba *Bojan*, ki praznuje rojstni dan in si ustvari kampanjo na spletni platformi imenovani *GiftCollector*, kjer povabi prijatelje, da mu namesto darila izpolnijo spletno čestitko in nakažejo denarna sredstva.

### 4.1 Web 2

Začeli bomo z vsem znano varianto, tj. klasičen Web 2.0, pri čemer zaradi lažje primerjave predstavimo prvo uporabniški vidik ter šele nato razvijalčev vidik.

### *4.1.1 Uporabniški vidik*

Predpogoj za uspešno izvedbo vseh uporabniških vidikov je ta, da so tako Bojan kot njegovi prijatelji informacijsko pismeni (tj., vešči uporabe brskalnika in spletnih strani, kakor tudi elektronskega poslovanja) ter da imajo dovoljšna denarna sredstva in možnosti porabe teh na spletu (npr. PayPal račun).

Če se fokusiramo na Web 2 pristop, sledimo miselnemu procesu, kateremu lahko zaradi splošne prakse, hitro sledimo. Bojan s svojim priljubljenim brskalnikom obišče klasično spletno stran *GiftCollector.com*, kjer si ustvari profil, pri čemer mu GiftCollector ponudimo možnost enotnega vpisa (ang. Single-Sign-On – SSO) na osnovi

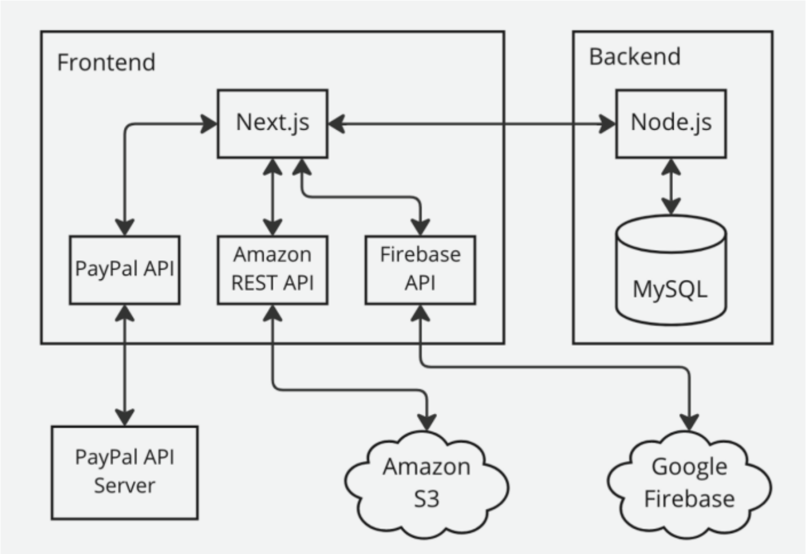
Google ali Facebook računa. Bojan izbere svoj Google račun kot osnovo za nov profil na GiftCollector-ju. Po registraciji in vpisu, Bojan izbere ustvarjanje nove kampanje, kjer GiftCollector od njega zahteva vnos določenih podatkov in izbiro ponujenih možnosti. Bojan vnese ime kampanje, njen opis in namen, doda svojo sliko, definira čas trajanja kampanje in izpolni še druge informacije, kot je npr. da dovoli prijateljem ustvarjanje čestitke ali prenos izbrane čestitke na platformo, kjer si jo bo lahko po koncu kampanje ogledal. Hkrati Bojan označi, da želi zbirati zgolj denarna sredstva v valuti EUR in ponudi svoj TRR ali številko kartice kot osnovno na katero se naj zbrana sredstva po koncu kampanje nakažejo. Z ustvarjenjem kampanje se strinja, da bo GiftCollector-ju plačal 10 EUR, ki jih poravna s kreditno kartico VISA, pri čemer mu je GiftCollector ponudil tudi druge plačilne možnosti, kot je PayPal ali nakazilo na njihov TRR. Med kampanjo lahko Bojan med obiskom GiftCollectorja sledi stanju zbranih sredstev in si ogleda prispele čestitke. Kampanjo lahko tudi predčasno prekine. Po koncu kampanje dobi s strani GiftCollectorja elektronsko sporočilo z informacijami in hkrati kak dan za tem tudi nakazan ves izkupiček denarnih sredstev na svoj TRR.

Bojanovi prijatelji lahko na podlagi URL naslova, ki jim ga je delil Bojan prav tako obiščejo pripravljeno kampanjsko spletno stran, kjer bodo videli vse potrebne informacije in lahko v kampanji tudi sodelovali. Prijatelji imajo možnost Bojanu pustiti tekstovno sporočilo ali prenesti datoteko slikovnega formata ter izpolniti količino sredstev v EUR, ki so mu pripravljeni podariti za rojstni dan. Registracija na GiftCollectorju za prijatelje ni potrebna, saj lahko vnesejo samo svoje ime ali vzdevek. Po izpolnjevanju vseh nujnih podatkov jih GiftCollector preusmeri na storitev spletnega plačevanja, kjer prijatelji izberejo način prenosa sredstev (npr. PayPal), izpolnijo vse potrebne podatke in zaključijo s pošiljanjem daril.

### *4.1.2 Razvijalčev vidik*

Implementacija GiftCollectorja kot spletne platforme je seveda lahko zelo raznolika glede na naše preference, ki bi jih kot načrtovalci imeli. Za namen poenostavljanja primerjave bomo predstavili samo eno od možnih oblik implementacije, ki ni nujno tudi najboljša, najučinkovitejša ali najbolj poceni.

Zaradi kompleksnosti poslovne logike smo izbrali klasičen način, kjer imamo spletno aplikacijo z obličjem (ang. frontend), zaledjem (ang. backend), podatkovno bazo in hrambo podatkov. Izbrali smo si ponudnika oblačnih storitev AWS, kjer bomo gostili ves naš zaledni del. Za namen razvoja obličja uporabimo Next.js in druge podporne knjižnice. Za zaledje izberemo Node.js (lahko tudi Express.js), kjer si definiramo vso potrebno poslovno logiko. Za podatkovno bazo uporabimo MySQL. Hkrati smo primorani uporabiti tudi druge podporne storitve in ogrodja, kot je PayPal API in Stripe za zagotavljanje podpore plačevanja storitev, Amazon API za S3 za namen hranjenja različnih datotek, ki jih uporabniki prenašajo znotraj kampanje, Firebase API za namen pošiljanja elektronskih sporočil in podpori overjanja in avtorizacije na osnovi OAuth 2.0. Poenostavljen prikaz sistemske arhitekture je prikazan na sliki 2.



**Slika 2: Sistemska arhitektura spletne aplikacije GiftCollector po principu Web 2.0**
Vir: [7].

## 4.2 Web 3

Nadaljujemo s predstavitvijo zasnove GiftCollectorja po principih Web 3.0. Zaradi lažje primerjave tokrat prvo predstavimo razvijalčev vidik in šele nato uporabnikov.

### *4.2.1 Razvijalčev vidik*

Enako kot je implementacija GiftCollectorja po klasičnih principih Web 2.0 lahko raznolika, je implementacija po principih Web 3.0 lahko prav tako. Za namen poenostavljanja primerjave bomo predstavili samo eno od možnih oblik implementacije po Web 3.0.

Ker želimo, da je GiftCollector načrtovan in implementiran po principih Web 3.0, bomo stremeli k temu, da bo čim več njenih komponent temeljilo na lastnosti **decentralizacije**. Slednje zahteva od nas, da se znebimo centraliziranih komponent, ki bi jih mi lastili in upravljali kot razvijalci ali kdo od ponudnikov storitev, ki bi jih sami izbrali. Posledično ne moremo imeti (ključne) zaledne poslovne logike nekje na našem strežniku, četudi uporabljamo kakšno oblačno rešitev. Kljub temu je rešitev na dlani, saj poslovno logiko za Web 3.0 v večini primerov dodamo v **pametne pogodbe**. Slednje spišemo v programskem jeziku Solidity, pri čemer se fokusiramo na dejstvo, da bodo le te omogočale hrambo sredstev v domorodnih kripto valutah. Slednja je odvisna od izbrane platforme verig blokov. V našem primeru izberemo **javno platformo verig blokov Etheruem**, ki je tudi najbolj popularna platforma, ki podpira pametne pogodbe [8]. S tem omogočimo tudi, da bodo naše pametne pogodbe lahko hranile zelo popularno kripto valuto Ether (ETH). Ključna poslovna logika pametne pogodbe je torej ta, da za čas kampanje hrani nakazane ETH s strani prijateljev, pri čemer po koncu kampanje zbrana sredstva samodejno nakaže iniciatorju kampanje. Da ta del deluje moramo seveda v pametnih pogodbah temu primerno beležiti decentralizirane identitete uporabnikov oz. njihove naslove TVB. Del poslovne logike je tudi to, da se kampanja začne samo takrat, ko iniciator le te pošlje transakcijo s katero plača zahtevana sredstva (tj., vrednost 10 EUR v ETH). Po razvoju pametnih pogodb jih primerno prevedemo (ang. compile) ter prenesemo v omrežje Ethereum, pri čemer si s tem tudi zagotovimo lastništvo nad le to in pridobimo **naslove** pametnih pogodb ter njihove zlogovne vmesnike (ang. application binary interface – ABI). Naslovi nam služijo kot javne končne točke (ang. endpoint) preko katerih bomo lahko z našo (decentralizirano) spletno aplikacijo integrirali z našo poslovno logiko.

Kot razvijalci moramo tudi poskrbeti za podporo čestitkam, ki jih lahko prijatelji iniciatorja kampanje ustvarijo in ob denarnem darilu ustvarijo/prenesejo. Datotek ali vsebine večje velikosti ni smiselno hraniti v pametnih pogodbah na verigah blokov, saj bi bili stroški s tem za uporabnike preveliki. Ker ne moremo izbrati centraliziranih strežnikov za hranjenje datotek, se odločimo za uporabo **decentraliziranega in porazdeljenega sistema za hranjenje datotek** IPFS. Uporabimo javno različico omrežja IPFS, ki prav tako kot Ethereum predstavlja javno omrežje, ki ne zahteva dovoljenja za pridruževanje (ang. permissionless). S tem si zagotovimo, da GiftCollector na decentraliziran način hrani različne vrste datotek. Po prenosu datotek na IPFS, uredimo hranjenje reference na lokacijo le te v pametni pogodbi.

Sedaj, ko smo si zagotovili ključno zaledno poslovno logiko, ki sicer sedja ni v zaledju, moramo še poskrbeti za oblični del. Podobno kot v primeru Web 2.0, bomo tudi tokrat razvili spletno aplikacijo, ki se pa zaradi svojih specifičnih lastnosti in funkcionalnosti poimenuje kar **decentralizirana aplikacija** (Dapp). Kot ogrodje za razvoj uporabimo Next.js, pri čemer pa kot ključno komponento vključimo še knjižnico **web3.js** ali **ethers.js**. Slednje nam pomaga, da lahko komuniciramo z omrežjem verig blokov in našo pametno pogodbo, kakor tudi z uporabniki Dapp-a in njihovimi **digitalnimi denarnicami** (DD). Da komunikacija med Dapp in pametnimi pogodbami deluje, moramo v konfiguracijo Dapp-a dodati naslove in ABI naših pametnih pogodb. Podobno bi v primeru Web 2 storili, če bi se želeli povezati s poslovno logiko in metodami, ki so hranjene nekje v oblaku ali strežniku. Web3.js ali ehters.js poskrbita za velik del funkcionalnosti, in sicer tudi za podporo komunikacije med uporabnikovimi DD in Dapp-om. Sedaj, ko se izogibamo zaledju, je seveda večji fokus na funkcionalnosti obličnega dela. Tako je potrebno Dapp povezati tudi z omrežjem IPFS, kjer bomo lahko na le to shranjevali ali prevzemali različne datoteke.

Kot razvijalci nam ni potrebno skrbeti za podporo plačilnim storitvam, saj bodo le te temeljile na prenosu kripto valut ETH od uporabnikov do pametnih pogodb, pri čemer je vsa potrebna funkcionalnost že urejena s strani uporabnikovih DD in podpornih knjižnic web3 ali ethers. Prav tako nam ni potrebno podpirati posebnih mehanizmov overjanja in avtorizacije, saj bomo le to prepustili uporabnikovim DD, ki predstavljajo njihovo decentralizirano identiteto in orodje s katerim se overijo. Ponovo je vsa potrebna logika že podprta s strani DD in knjižnic web3 in ethers, pri čimer se sami zgolj sklicujemo na posamezne podprte funkcije.

Ko smo z implementacijo Dapp zaključili nam še ostane namestitev le te. Največkrat izbrana možnost je ta, da se Dapp naloži na strežnik primeren za gostovanje spletnih aplikacij. Ta je lahko naš lasten ali v oblaku. Slednje je seveda kontradiktorno osnovnim načelom Web 3, kjer bi naj vse bilo decentralizirano, vendar je to trenutno najprimernejša možnost. Druga možnost bi bila, da se celoten Dapp z vsemi svojimi razvijalskimi datotekami naloži na omrežje, kot je IPFS ali Ceramic, kar bi zagotovili popolno decentraliziranost naše rešitve. Žal je ta možnost uporabniško neprijazna zaradi počasnosti delovanja omenjenih omrežij, kar pa je posledica konceptov po katerih delujejo (tj., porazdeljenost in decentralizacija). Kljub nadomestni rešitvi, ki je največkrat uporabljena, lahko še zmeraj govorimo o Dapp in Web 3 rešitvi, saj se večina ključne poslovne logike odvija na decentraliziran način oz. na strani DD in odjemalčevi strani, tj. brez zaledja.

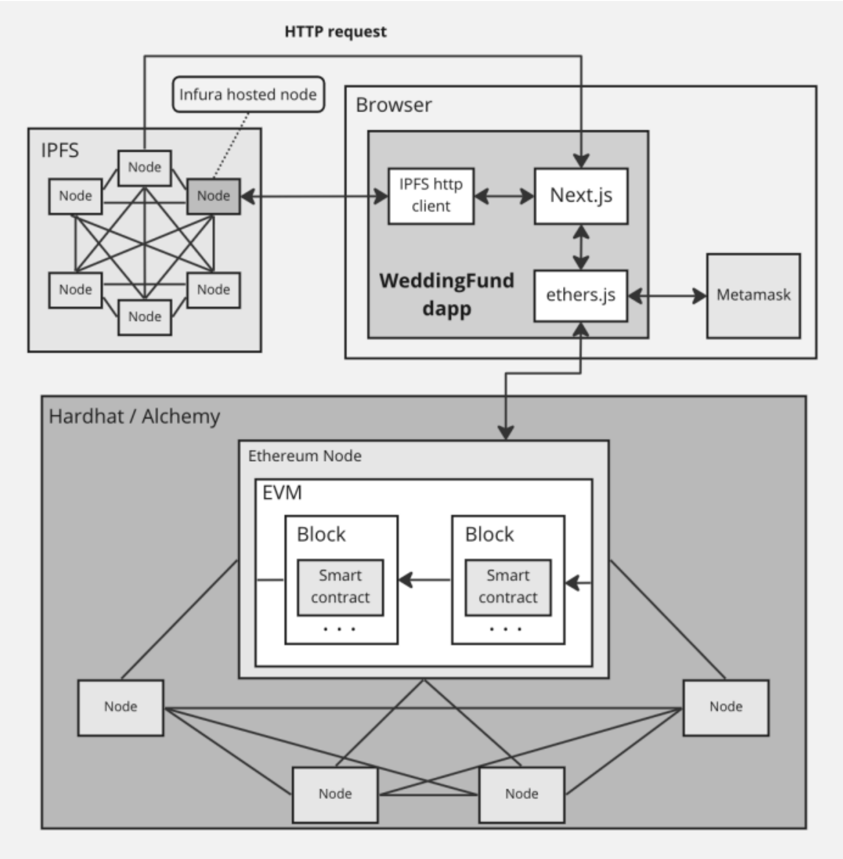
Poenostavljen prikaz sistemske arhitekture Web 3.0 je prikazan na sliki 3.

### *4.2.2 Uporabniški vidik*

V razliko od procesa, ki sledi pristopu Web 2.0, je slednje uporabnikovega vidika, ki temelji na pristopu Web 3.0 kompleksnejše. Slednje velja predvsem za uporabnike, ki niso seznanjeni s koncepti in terminologijo povezano s TVB oz. niso vešči orodij, ki so s tem povezana. Vsekakor morajo vsi biti informacijsko pismeni (tj., vešči uporabe brskalnika in brskanja po spletu). Prav tako je predpogoj ta, da si vsi v svoj priljubljen brskalnik morajo namestiti namensko digitalno denarnico (DD), kot je npr. Metamask in si pridobiti določeno količino kripto valute ETH. Z nameščanjem DD si hkrati zagotovijo svojo decentralizirano identiteto, ki je v tem primeru povezana z njihovim naslovom TVB, ki ga nadzirajo z zasebnim ključem oz. DD.

Ko se Bojan odloči ustvariti kampanjo s svojim priljubljenim brskalnikom obišče spletno stran *GiftCollector.com*, ki je sedaj Dapp. Dapp se samodejno poskusi povezati z njegovo DD in mu to na podlagi pojavnega okna omogoči tudi nadzirati. Ko Bojan s svojo DD sprejme povezavo z GitCollector Dapp-om, je enako kot bi se uspešno vpisal v klasično spletno aplikacijo. Bojan ustvari novo kampanjo na način, da izpolni vse zahtevane informacije, kot so ime kampanje, namen, trajanje itn. Ni mu potrebno vnašati svojega TRR, saj Dapp že ima njegov Ethereum naslov na katerega bo po koncu kampanje GiftCollector (pametna pogodba) nakazal zbrana sredstva. Ko želi Bojan kampanjo inicializirati, mora le to potrditi z elektronskim podpisovanje transakcije. Dapp pripravi transakcijo in jo ponudi Bojanu v podpis, pri čemer se mu le to pojavi kot pojavno okno DD, kjer transakcijo poljubno sprejme ali zavrne. Ob podpisu transakcije se je Bojan tudi strinjal in nakazal potrebnih ETH za 10 EUR, kolikor je strošek storitve. S podpisom transakcije je Bojan uspešno inicializiral svojo kampanjo. Med kampanjo lahko Bojan med obiskom GiftCollectorja sledi stanju zbranih sredstev in si ogleda prispele čestitke. V razliko od Web 2 pristopa, lahko Bojan spremlja stanje kampanje tudi javno na pregledovalniku transakcij verig blokov, kot je etherscan. Kampanjo lahko tudi predčasno prekine, vendar bo moral to potrditi s podpisom transakcije, ki se proži s pomočjo DD. Po koncu kampanje dobi Bojan s strani GiftCollectorja nakazan ves izkupiček denarnih sredstev na svoj Ethereum naslov.

Enako kot v prejšnjem primeru bodo Bojanovi prijatelji lahko na podlagi URL naslova, ki jim ga deli Bojan obiskali pripravljeno kampanjsko spletno stran, ki je v tem primeru Dapp. Enako kot v prejšnji različici, imajo prijatelji možnost Bojanu pustiti tekstovno sporočilo ali prenesti datoteko slikovnega formata ter izpolniti količino sredstev v ETH, ki so mu pripravljeni podariti za rojstni dan. Prav tako se prijatelji na GiftCollectorju ne rabijo registrirati, saj lahko vnesejo samo svoje ime ali vzdevek. Po izpolnjevanju vseh nujnih podatkov jim GiftCollector Dapp ponudi v podpisovanje transakcijo, ki jo prijatelji potrdijo s svojo DD in s tem zaključijo pošiljanje daril.



**Slika 3: Sistemska arhitektura spletne aplikacije GiftCollector po principu Web 3.0**
Vir: [7].

### 4.3 Konceptualna primerjava

Kot je razvidno iz opisne primerjave med Web 2 in Web 3 obstajajo tako razlike za razvijalce kot za uporabnike, pri čemer je za ene manj druge morda več, kar pa je seveda subjektivne narave in odvisno od posameznikov. Vsekakor predstavlja pristop Web 3 premik v novo smer in vpeljuje dodatne nove koncepte, kakor tudi nova orodja in postopke. Glede na dosedanja spoznanja predstavljamo v tabeli 1 nekaj ključnih konceptualnih primerjav med obema konceptoma.

## 5 Izzivi Web 3

Kot je že na kratko omenjeno v tabeli 1, obstajajo trenutno še številni izzivi, ki so povezani s konceptom Web 3 in preprečujejo splošno in masovno sprejetje le tega. V nadaljevanju na kratko predstavimo nekaj ključnih izzivov.

Med prvimi izzivi Web 3 so vsekakor tisti, kateri so povezani s TVB, in sicer **razširljivost** ali skalabilnost. TVB temelji na številnih psevdonimnih uporabnikih, ki gostijo vozlišča verig blokov in s tem ponujajo svoje računalniške vire za zagotavljanje delovanja celotnega omrežja. S tem predvsem zagotovijo varnost omrežja, saj se podatki podvojeno hranijo na vseh vozliščih in tako preprečijo eno točko odpovedi. Takšen sistem pa ima številna ozka grla, ki rezultirajo v dejstvo, da se transakcije počasi izvršijo in da so stroški z vzdrževanjem sistema vedno večji. Obstajajo mehanizmi, ki lahko ozka grla odstranijo, vendar jih takšnih, ki ne bi vsaj rahlo vplivala na lastnost decentralizacije. Tako najdemo že danes določene verige blokov, ki so hitrejše od drugih, vendar je njihov nivo zagotavljanje ključne lastnosti Web 3, korenito nižji od prvotnih omrežij [8].
S TVB je povezan tudi izziv **stroškov**, saj se v primeru Web 3 aplikacij stanovitne transakcije morajo digitalno podpisati in plačati. To je vsekakor izziv za končne uporabnike, ki niso navajeni, da so za uporabo spletnih portalov primorani za vsako transakcijo sami plačevati, četudi zgolj nekaj centov. Določene Web 3 rešitve rešujejo izziv na način, da sami prevzamejo tvorjenje in podpisovanje ter plačevanje transakcij, kar pa seveda prenese stroške na njih in hkrati vpliva na varnost in lastnost decentralizacije. V veliki večini primerov se za takšne rešitev zahteva uporaba zalednih sistemov, ki se jih v osnovi želimo znebiti. Prav tako takšne rešitve vpeljujejo avtomatizirano potrjevanje transakcij, kar pa zahteva dostop do zasebnega ključa, ki nadzira digitalna sredstva itn.

Povezano s prejšnjima izzivoma je vsekakor **uporabniška (ne)prijaznost** rešitev Web 3. Kot je predstavljeno v poglavju o primerjavi in v uporabniškem vidiku Web 3, morajo uporabniki Web 3 rešitev uporabljati namenske DD, si naložiti primerne kriptovalute in hkrati vsako stanovitno akcijo na spletnih portalih potrjevati oz. digitalno podpisovati. Številni koraki so že narejeni v smeri zagotavljanja večje uporabniške prijaznosti, vendar hitro pridemo do ključne trileme med varnostjo, učinkovitostjo in uporabniško prijaznostjo. Določeni snovalci Web 3 rešitev so šli korak »naprej« oz. »nazaj« k Web 2 principom, saj so omogočili samodejno potrjevanje transakcij in podobno, kar pa seveda vpliva na varnost in lastnost decentralizacije, zaradi katere se je Web 3 dejansko pojavil kot nov koncept.

Glede na prejšnji predstavljen izziv, pa imamo po drugi strani člane skupnosti snovalcev in razvijalcev Web 3 rešitev, ki pa želijo doseči še višje nivoje zagotavljanja lastnosti decentralizacije in tako snujejo nove pristope, ogrodje in orodja, ki to dosegajo. S tem povezan je tudi izziv, kako in če sploh lahko dosežemo **popolno decentralizacijo**. Kot smo videli na primeru Dapp, ki je ponavadi gostovana na centralizirano upravljanih spletnih strežnikih in bi se lahko tudi gostila na IPFS ali podobnih omrežjih, vendar bi bila učinkovitost le te zelo nizka. Podobni izzivi so tudi na nivoju upravljanja s kriptografskimi nezamenljivimi žetoni NFT, ki bi naj hranili vse svoje komponente, podatke in metapodatke decentralizirano, vendar je tak pristop težko dosegljiv in neučinkovit. Tako najdemo številne NFT-je, kjer opažamo, da so določeni njihovi viri (npr. unikatna slika NFT-ja) hranjeni na centralizirano upravljanjem strežniku in bi se lahko le ti kadar koli »ugasnili« in tako pustili lastnike NFT-ja brez dejanskega vira vsebine. Podobni izzivi z zagotavljanjem popolne decentralizacije so tudi na nivoju decentraliziranih in samoupravljanih identitet idr., vendar jih zaradi fokusa članka ne bomo posebej izpostavljali.

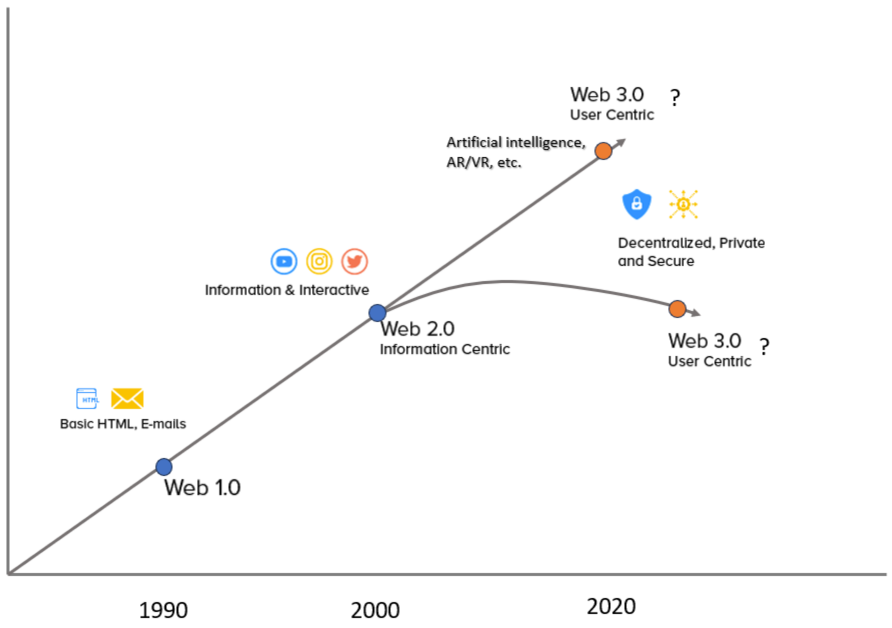
**Tabela 1: Konceptualna primerjava med Web 2 in Web 3.**

| Kriterij / Lastnost | Web 2.0 | Web 3.0 |
|---|---|---|
| Fokus in prednost | Interaktivnost in komunikacija, uporabniška prijaznost, dinamična vsebina, deljenje informacij, realno-časovna obdelava, pametne in mobilne naprave. | Decentralizacija, lastninjenje in nadzor nad podatki, transparentnost, psevdonimnost, in necenzuriranost oz. nespremenljivost ter podpora novim načinom elektronskega poslovanja. |
| Upravljanje | Centralizirane entitete, kot so banke, vlade, ali korporacije nadzirajo delovanje, storitve in podatke. | Modeli upravljanja skupnosti (npr. DAO), kjer se lahko vsak pridruži in vsi lahko prispevajo k odločanju glede upravljanja. |
| Ključne in specifične tehnologije | Računalništvo v oblaku, spletne storitve, aplikacijski vmesniki. | TVB, pametne pogodbe, Dapp, NFT, IPFS, namenske DD. |
| Varnost in zasebnost | Korporacije sledijo uporabnikom in izkoriščajo njihove podatke za profiliranje, pri čemer so ponekod zakoni, ki to do neke mere preprečujejo. Varnost je bolj ali manj prepuščena ponudnikom storitev. | Varnost je urejena zgolj s tehnologijo in kriptografskimi elementi ter je bolj ali manj v rokah končnih uporabnikov. |
| Digitalne identitete | V veliki meri psevdonimne in centralizirane pri manjšemu številu korporacij, s podporo federativnim modelov izkoriščanja digitalne identitete. | Decentralizirane digitalne identitete, ki jih uporabniki nadzirajo sami, pri čemer je identiteta po-uporabna skozi večino Web 3 prostora. |
| Stanje podatkov | Nadzor nad podatki na strani lastnikov spletnih portalov in platform. Lokacija podatkov je centralizirano upravljana, četudi porazdeljena oblačna infrastruktura. Čeprav so podatki varno hranjeni se iz vidika centraliziranega upravljanja lahko smatrajo tudi kot nestanovitni. | Nadzor nad podatki na strani lastnikov le teh in vseh vzdrževalcev porazdeljenih in decentraliziranih omrežij. Lokacija podatkov je porazdeljena med vsa vozlišča. Zaradi številne replikacije so podatki na javnih omrežjih resnično stanovitni. |
| Podpora e-poslovanju | Uveljavljene in učinkovite plačilne storitve in plačilni prehodi (ang. gateway) na osnovi centraliziranih finančnih struktur, kjer se nudi podpora kreditnim karticam in številnim fiat valutam. Stroški so večji in obstajajo možnosti omejevanja transakcij s strani različnih finančnih ali državnih organov. | Kripto valute in žetoni, ki niso nadzirani skozi uveljavljene centralizirane finančne strukture. Stroški so lahko manjši, vendar je učinkovitost iz vidika hitrosti trenutno slabša. Obstaja možnost ustvarjanja poljubnih kripto žetonov in drugih mehanizmov, kot so NFT. |
| Slabosti | Varnost, saj temelji le ta na nekaj centralizirano vodenih strežnikih (oblakih), s čimer se ustvarjajo enojne točke odpovedi. Zasebnost, saj lahko korporacijo izkoriščajo podatke svojih uporabnikov, da jih profilirajo ali pa le te prodajo naprej. Korporacije ali vlade lahko uporabnike blokirajo ali drugače cenzurirajo. | Kompleksnost s strani tehnične zasnove, kakor tudi uporabniške prijaznosti, s čimer se omejuje masovno sprejemanje novih konceptov in tehnologij. Zahteva po več računalniških virih. Trenutno še omejena zmožnost skalabilnosti. Težave z regulacijo, ki odvračajo številne poslovne uporabnike. Varnost je prepuščena končnim uporabnikom, ki lahko hitreje izgubijo svoja sredstva ali podatke in so pri tem nepovratno oškodovani. |

Med pomembne izzive štejemo tudi vpliv transparentnosti, preverljivosti in nespremenljivosti TVB na **zasebnost** uporabniških podatkov. Prej omenjene lastnosti onemogočajo uporabnikom možnost izbrisa podatkov v trenutku, ko so enkrat le ti potrjeno shranjeni na javni verigi blokov. To seveda vpliva na zmožnosti zagotavljanja pravice do pozabe, ki je pravica končnih uporabnikov glede na Splošno uredbo o varstvu osebnih podatkov (ang. General Data Protection Regulation – GDPR). Kljub temu je Web 3 skupnost naučila delovati s to pomanjkljivostjo TVB na način, da so se našli drugi mehanizmi za shranjevanje osebnih in drugih podatkov [9]. Posledica teh rešitev pa je ponovno prispevanje k uporabniški neprijaznosti, slabši učinkovitosti in nepopolnemu zagotavljanju lastnosti decentralizacije.

## 6 Diskusija

Glede na kompleksnost nastalega področja Web 3 je težko predstaviti vse podrobnosti le tega. Prav tako je, zaradi novosti tehnologij na katerih temelji, še veliko stvari nedorečenih in neraziskanih. Sproti se pojavljajo in nastajajo novi trendi (glej poglavje 3.1) in s tem povezani koncepti, ogrodja in orodja, pri čemer pa še zmeraj nimamo enotne definicije, kaj Web 3 dejansko je. Dejstvo je, da je termin Web 3 leta 2014 vsekakor prvi skoval in uporabil soustanovitelj platforme verig blokov Ethereuma, Gavin Wood, pri čemer je le to povezoval s koncepti, ki smo jih predstavljali v članku [10]. Obstajajo pa določeni avtorji, ki Web 3 definirajo kot novo obdobje spleta, katerega ključni del je umetna inteligenca in strojno učenje. Prav tako smo že v članku omenjali, da se Web 3 zmotno enači s pojmom metaverzum. Tako se postavlja ključno vprašanje, kaj Web 3 dejansko predstavlja? Ali je to v resnici novo obdobje spleta, ki bo skoraj v celoti nadomestilo trenutne načine delovanja, enako kot smo bili priča prehodu med Web 1 in Web 2, ali pa bo to zgolj ena nova in dodatna oblika spleta, kateri se bomo lahko po želji pridružili, med tem ko bomo še naprej delovali po pristopih Web 2, ki bo zelo verjetno še obogaten s funkcionalnostmi umetne inteligence, strojnega učenja in obogatene in/ali navidezne resničnosti (Slika 4). Že sedaj vemo, da se Web 3 še nekaj časa ne bo dokončno definiral, saj se že sedaj kažejo dodatni novi trendi, ki bodo to področje obogatili. Prav tako pa lahko potrdimo, da Web 3 ni več zgolj obličje TVB, ni nujno zgolj aplikacija povezana s kripto valutami in/ali kripto žetoni (npr. NFT), temveč je lahko širok spekter spletnih rešitev, ki pa stremijo k vsaj delnemu zagotavljanju lastnosti decentralizacije.



**Slika 4: Razvoj spleta in vprašanje ali je Web 3 dejansko nova različica ali samo dodatna.**

## 7 Zaključek

V prispevku smo podrobno raziskali in primerjali koncepta Web 2 in Web 3, paradigmi spletnega razvoja, ki predstavljata dve različni viziji spletnega okolja. Web 2, že dolgo prevladujoča paradigma, se osredotoča na centralizirane platforme, ki imajo nadzor nad podatki in uporabniškimi izkušnjami. Nasproti temu stoji Web 3, nov val inovacij, ki temelji na decentralizaciji, transparentnosti in uporabniškem nadzoru.

V članku smo predstavili ključne koncepte in tehnologije Web 3, kot so tehnologija veriženja blokov (TVB), decentralizirane identitete, pametne pogodbe in metaverzum. Poudarili smo tudi izzive, s katerimi se sooča Web 3 pri prehodu od koncepta do množične uporabe. Te vključujejo skalabilnost TVB, uporabniško prijaznost, stroške transakcij in vprašanja zasebnosti ter varstva podatkov. Kljub tem oviram, je Web 3 prinesel veliko novih priložnosti, ki omogočajo uporabnikom večji nadzor, transparentnost in možnost sodelovanja v digitalnem svetu.

Razumevanje razlik med Web 2 in Web 3 je ključno za prihodnji razvoj spletnih tehnologij ter za ozaveščanje o novih priložnostih in izzivih, ki jih prinaša decentralizacija, transparentnost in uporabniški nadzor. Web 3 je vznemirljiv korak naprej, ki lahko preoblikuje način, kako razmišljamo in delujemo na spletu, ter nas popelje v novo dobo digitalne inovacije.

## Literatura

[1] DI ANGELO Monika, SALZER Gernot, »Tokens, Types, and Standards: Identification and Utilization in Ethereum«, 2020 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS), Oxford, UK, 2020, str. 1-10, doi: 10.1109/DAPPS49028.2020.00001.

[2] PODGORELEC Blaž, ALBER Lukas, ZEFFERER Thomas. »What is a (Digital) Identity Wallet? A Systematic Literature Review«, IEEE 46th Annual Computers, Software, and Applications Conference, 27 June –1 July 2022 Virtual Event : proceedings. Danvers: IEEE, cop. 2022. str. 809-818. ISBN 978-1-6654-8810-5. DOI: 10.1109/COMPSAC54236.2022.00131.

[3] WANG Yuntao, idr., »A Survey on Metaverse: Fundamentals, Security, and Privacy«, IEEE Communications Surveys & Tutorials, letnik 25, številka. 1, str. 319-352, Firstquarter 2023, doi: 10.1109/COMST.2022.3202047.

[4] ŠUPUK Abdurrahman. »Razvoj z NFT podprto Metaverse igro JumpSquad« : diplomsko delo, 2022, spletni vir https://dk.um.si/IzpisGradiva.php?id=83370.

[5] ČUČKO Špela, TURKANOVIĆ Muhamed, »Decentralized and self-sovereign identity: systematic mapping study«, IEEE access, 4 Oktober 2021, letnik 9, str. 139009-139027, DOI: 10.1109/ACCESS.2021.3117588.

[6] ČUČKO Špela, TURKANOVIĆ Muhamed, »Umestitev digitalnih (EU) denarnic v ekosistem sodobnih IKT rešitev«, OTS 2022 : sodobne informacijske tehnologije in storitve : zbornik petindvajsete konference, Maribor, Fakulteta za elektrotehniko, računalništvo in informatiko, 2022, str. 24-41.

[7] ZAVRATNIK Jaka, »Analysis of the principles of the development of Web3 solutions« : master's thesis, 2022, https://dk.um.si/IzpisGradiva.php?id=83517.

[8] CONNORS Collin, SARKAR Dilip, »Survey of prominent blockchain development platforms«, Journal of Network and Computer Applications, letnik 216, 2023, ISSN 1084-8045, DOI: https://doi.org/10.1016/j.jnca.2023.103650.

[9] TURKANOVIĆ Muhamed, KEŽMAH Boštjan, PODGORELEC Blaž, KAMIŠALIĆ LATIFIĆ Aida, HERIČKO Marjana, HERIČKO Marjan, »Method and device for the storing, inspection control and retrieval of data from a permanently immutable, distributed and decentralized storage«, European Patent Specification EP 3 742 318 B1, 2021-03-17. Munich: European Patent Office, 2021.

[10] EDELMAN Gilad, »The Father of Web3 Wants You to Trust Less«, Wired, ISSN 1059-1028, 2021, URL: https://www.wired.com/story/web3-gavin-wood-interview, obiskano 10. 6. 2023.

# Hibridni certifikati post-kvantne kriptografije

Nastja Cepak,[1,2] Jakob Matek[1]

[1] CREAPLUS d.o.o., Ljubljana, Slovenija
nastja.cepak@creplaus.com, jakob.matek@creaplus.com

[2] Univerza na Primorskem, Fakulteta za naravoslovje, informacijske tehnologije in informatiko, Koper, Slovenija
nastja.cepak@creplaus.com

Kvantni računalniki obetajo omogočiti izračune, ki jim niso kos niti današnji najmočnejši super računalniki. Razvoj kvantnih računalnikov in algoritmov pa bo tudi dosegel razvojno stopnjo, ko obstoječi varnostni mehanizmi ne bodo več nudili zaščite podatkov in digitalnega zaupanja. Pričakujemo, da se bomo v obdobju tranzicije s tradicionalnih asimetričnih kriptografskih algoritmov (RSA in ECC) na post-kvantne algoritme posluževali tako imenovanih »hibridnih rešitev«, torej rešitev, ki hkrati podpirajo klasično in post-kvantno kriptografijo. V prispevku se bomo osredotočili na hibridne digitalne certifikate in bomo prikazali, kako lahko integriramo post-kvantne algoritme v x509 format digitalnih certifikatov.

**Ključne besede:**
kvantno računalništvo
post-kvantna kriptografija
crytsals-dilithium
digitalna potrdila
hibridna potrdila

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.3

## 1 Uvod

Kvantna teorija je znanstveno področje, ki ponuja dober matematični model za opis narave na atomski in subatomski ravni. Na podlagi teh teorij in načel rase področje kvantnega računalništva. Kar se je včasih zdelo kot znanstvena fantastika, postaja naša resničnost. Njegovi začetki segajo približno 40 let v preteklost, ko je Paul Benioff leta 1980 opisal Turingov stroj s kvantno mehaniko [6]. Kmalu zatem, leta 1982, je Richard Feynman predlagal prvo praktično uporabo tovrstnih strojev – simulacijo kvantne mehanike [7], kar bi imelo velik vpliv na biologijo in kemijo.

Resnično zanimanje za kvantno računalništvo pa se je sprožilo leta 1994, ko je Peter Shor razvil algoritem, ki bi kvantnim računalnikom omogočil učinkovito faktorizacijo velikih celih števil. To je algoritem s potencialom razbijanja šifriranj RSA in ECC, algoritmov, na katera se zanašajo številni kriptosistemi. Namesto da bi za faktorizacijo 300-mestnega števila potrebovali milijone let, bi ga Shorov algoritem lahko faktoriziral v nekaj dnevih.

Kmalu zatem, leta 1996, je Lov Grover zasnoval Groverjev iskalni algoritem, ki omogoča uporabo kvantnih računalnikov za zagotavljanje kvadratne pospešitve iskanja v primerjavi s tradicionalnimi iskalnimi algoritmi. Prečesavanje celotnega prostora ključev v napadu s surovo silo, kjer iščemo šifrirni ključ, bi nenadoma postalo veliko lažje.

Za lažjo predstavo vpliva napadov s kvantnimi računalniki na danes uporabljene kriptografske algoritme prilagamo spodnjo tabelo 1.

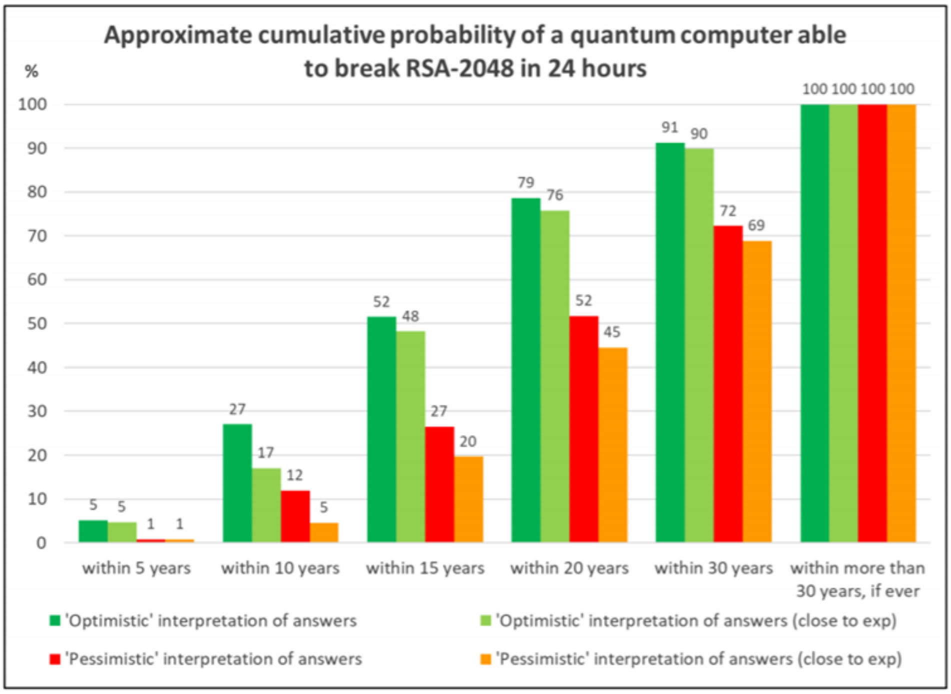
**Tabela 1: Število bitov varnosti asimetričnih in simetričnih algoritmov danes in ob uproabi Shorjevega/Groverjevega algoritma na dovolj zmogljivem kvantnem računalniku**

| Algoritem | Število bitov varnosti (danes) | Število bitov varnosti z uporabo Shorjevega/Groverjevega algoritma na kvantnem računalniku |
|---|---|---|
| RSA-1024 | 80 | 0 |
| RSA-2048 | 112 | 0 |
| ECC-256 | 128 | 0 |
| ECC-384 | 256 | 0 |
| AES-128 | 128 | 64 |
| AES-256 | 256 | 128 |

Kvantni računalniki nenadoma niso bili samo predmet akademske radovednosti, temveč dobro financirane vladne in zasebne raziskave. Napredek, ki je sledil v naslednjih desetletjih, je odraz na novo ustvarjenega zanimanja. Trenutno eno vodilnih podjetij na kvantnem področju, IBM, napoveduje za leto 2023 otvoritev 1121-qbitnega procesorja Condor, pripravljen pa imajo tudi načrt za hitro višanje števila qbitov v naslednjih letih, do leta 2026 [8].

Takoj se postavi sledeče očitno in pomembno vprašanja: Kdaj bodo obstajali kvantni računalniki, ki bodo lahko, na primer, dovolj zmogljivi, da bi zaganjali Shorjev algoritem, ki bo lahko razbil algoritem RSA-2048? Leta 2019 sta Michele Mosca in Marco Piani napisala poročilo o časovnem načrtu kvantne grožnje za Global Risk Institute [10]. Poročilo vključuje anketo, izvedeno med 22 vodilnimi mednarodnimi strokovnjaki za raziskave kvantnega računalništva. Eno od vprašanj v anketi je bilo navesti verjetnost, da bo kvantnemu računalniku uspelo razbiti RSA-2048 v 24 urah v 5/10/15/20/30/30+ letih. Na sliki 1 so prikazane približne vrednosti prejetih odgovorov.

Vidimo, da čeprav obstaja le malo prepričanja, da nas bo grožnja kvantnega napada v naslednjih nekaj letih ujela nepripravljene, se ideja, da bo v naslednjih 20 letih obstajal kvantni računalnik, ki bo lahko zlomil naše varnostne algoritme, zdi precej verjetna. Za ohranitev naše digitalne varnosti v prihodnosti so raziskovalne ustanove že pred več kot desetletjem začele razvijati kvantno odporne kriptografske algoritme. Trenutno najboljši od tako imenovanih algoritmov za postkvantno kriptografijo (Post Quantum Cryptography - PQC) tekmujejo za kvalifikacijo za NIST (National Institute for Standards and Technology, ZDA) standardizacijo v okviru projekta "Post-Quantum Cryptography Standardization" [11].



**Slika 1: Kumulativna verjetnost za obstoj kvantnega računalnika, ki bi lahko zlomil RSA-2048 v 24 urah**
Vir: [10].

## 2 Post-kvantna kriptografija

Ko govorimo o algoritmih post-kvantne kriptografije, govorimo o asimetirčnih algoritmih, ki jih zaganjamo na standardnih digitalnih računalnikih, kot jih poznamo danes, hkrati pa so odporni tako na tradicionalne napade, kot na napade, ki bi se jih zaganjalo na kvantnih računalnikih. Njihov namen je, da v prihodnosti zamenjajo algoritma RSA in ECC.

Proces standardizacije post-kvantne kriptografije NIST se je začel konec leta 2016 z javnim pozivom NIST za oddajo predlogov. Novembra 2017 je bil rok za oddajo predlogov algoritmov. Od 82 predloženih kandidatov je bilo 13 skoraj takoj zlomljenih ali pa so vsebovali drugačne pomanjkljivosti. Tako je bilo decembra 2017 v prvi izbirni krog sprejetih skupno 69 kandidatov. Nekaj mesecev za tem, aprila 2018, je potekala 1. standardizacijska konferenca NIST PQC, kjer so o algoritmih in potencialnih napadih nanje razpravljali raziskovalci iz NIST in širše akademske skupnosti. V začetku leta 2019 je bilo izbranih 26 kandidatov za nadaljevanje v 2. krog, čemur je sledila 2. konferenca o standardizaciji NIST PQC, ki je potekala še isto poletje. 22. julija 2020 je bilo razglašenih 7 finalistov 3. kroga. NIST je tudi javno objavil poročilo o izbirnem postopku. Končno, 5. julija 2022, so bili objavljeni štirje algoritme, ki bodo standardizirani, in kandidati za četrti krog. Končnih standardov še nimamo, pričakujemo pa jih začetek leta 2024.

Ena od omejitev, ki jih še nismo uspeli preseči pri algoritmih post-kvantne kriptografije, je, da algoritmi (še) niso dovolj prilagodljivih, da bi hrkati lahko učinkovito izvajali operacije šifriranja z javnim ključem in digitalnega podpisovanja. Zato je tekmovanje ločeno na 2 kategoriji, kot je prikazano v tabeli 2.

**Tabela 2: Finalisti in kandidati za 4. krog NIST PQC tekmovanja.**

| | Šifriranje z javnim ključem | Digitalni podpisi |
|---|---|---|
| Finalisti, ki bodo standardizirani | CRYSTALS-KYBER | CRYSTALS-DILITHIUM<br>FALCON<br>SPHINCS+ |
| Kandidati za 4. krog | BIKE<br>Classic McEliece<br>HQC | / |

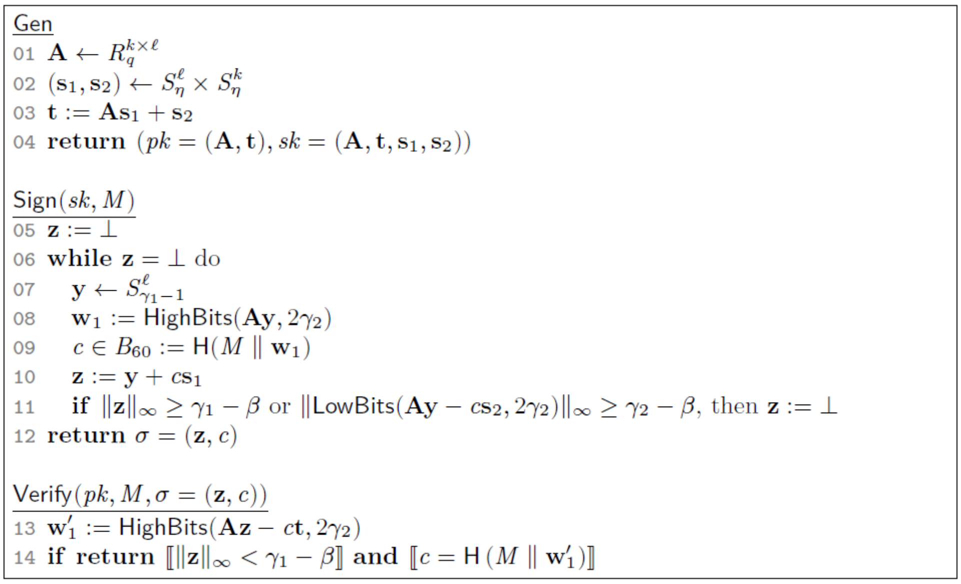
Kot pripombo bi dodali, da je tehnično gledano še vedno kandidat za 4. krog tudi algoritem SIKE (šifriranje z javnim ključem), vendar je bil avgusta 2022 zlomljen na način, da manjši popravki niso dovolj za varno delovanje. Zaradi tega ga nismo dodali v tabelo.

Zanimanje našega projekta leži v digitalnih certifikatih in njihovi hibridiziraciji, torej postopku, ki tradicionalnemu certifikatu doda post-kvantno komponento. Zaradi tega smo se osredotočili na algoritem CRYSTALS-DILITHIUM, ali na kratko, Dilithium. Je najbolj priporočen algoritem za implementacijo PQ digitalnih certifikatov [12].

## 2.1 CRYSTALS-DILITHIUM

CRYSTALS-DILITHIUM je asimetrični kriptografski algoritem za digitalno podpisovanje, ki je odporen tako na tradicionalne napade, kot na napade, ki bi jih lahko zaganjali na kvantnem računalniku. Prvič je bil uradno definiran leta 2017 [13, 14] kot prijava v prvi krog NIST PQC tekmovanja. CRYSTALS ekipa je v tistem času definirala dva algoritma: CRYSTALS-DILITHIUM, ki je namenjen digitalnemu podpisovanju in čigar ime je referenca na material dilithium iz Star Treka. Drugi algoritem, ki je bil prav tako izbran za standardizacijo, je CRYSTALS-KYBER, namenjen šifriranju in z imenom, ki je referenca na kyber kristal iz Star Wars.

Kot opisano v [13], Dilithium temelji na težavnosti problema iskanja kratkih vektorjev v mrežah. Zasnova sheme temelji na pristopu "Fiat-Shamir with Aborts" [15], ki z zavračanjem vzorcev (rejection sampling) poskrbi, da je Fiat-Shamir shema bolj kompaktna in varnejša. Za lažje ne-matematično razumevanje algoritma prilagamo sliko 2 s pseudokodo funkcionalnosti. Pri implementaciji hibridnih certifikatov uporabljamo vse tri funkcionalnosti (generacijo ključa, podpisovanje, verificiranje).



```
Gen
$\mathbf{A} \leftarrow R_q^{k\times\ell}$
$(\mathbf{s}_1, \mathbf{s}_2) \leftarrow S_\eta^\ell \times S_\eta^k$
$\mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$
$\textbf{return}\ (pk = (\mathbf{A}, \mathbf{t}), sk = (\mathbf{A}, \mathbf{t}, \mathbf{s}_1, \mathbf{s}_2))$

Sign$(sk, M)$
$\mathbf{z} := \perp$
$\textbf{while}\ \mathbf{z} = \perp\ \text{do}$
    $\mathbf{y} \leftarrow S_{\gamma_1 - 1}^\ell$
    $\mathbf{w}_1 := \mathsf{HighBits}(\mathbf{A}\mathbf{y}, 2\gamma_2)$
    $c \in B_{60} := \mathsf{H}(M \parallel \mathbf{w}_1)$
    $\mathbf{z} := \mathbf{y} + c\mathbf{s}_1$
    $\textbf{if}\ \|\mathbf{z}\|_\infty \geq \gamma_1 - \beta\ \text{or}\ \|\mathsf{LowBits}(\mathbf{A}\mathbf{y} - c\mathbf{s}_2, 2\gamma_2)\|_\infty \geq \gamma_2 - \beta,\ \text{then}\ \mathbf{z} := \perp$
$\textbf{return}\ \sigma = (\mathbf{z}, c)$

Verify$(pk, M, \sigma = (\mathbf{z}, c))$
$\mathbf{w}_1' := \mathsf{HighBits}(\mathbf{A}\mathbf{z} - c\mathbf{t}, 2\gamma_2)$
$\textbf{if return}\ [\![ \|\mathbf{z}\|_\infty < \gamma_1 - \beta ]\!]\ \textbf{and}\ [\![ c = \mathsf{H}(M \parallel \mathbf{w}_1') ]\!]$
```

**Slika 2: Preudokoda funkcionalnosti Dilithium algoritma**
Vir: [13].

# 3 Digitalni certifikati

Digitalno certifikat je oblika elektronske poverilnice, ki lahko dokaže pristnost uporabnika, naprave, strežnika, spletne strani ali katerekoli druge entitete. Uporablja tako imenovano infrastrukturo javnih ključev (Public Key Infrastructure), ki jo bomo za voljo koherentnosti članka tu le na kratko opisali, za varno izmenjavo komunikacij in podatkov prek interneta in drugih omrežij.

Ta oblika avtentikacije je vrsta kriptografije, ki zahteva uporabo javnih in zasebnih ključev, ki enolično pripadata eden drugemu, za preverjanje identitete uporabnikov. Digitalne certifikate izdajo zaupanja vredne tretje osebe (Certificate Authority, v nadaljevanju tudi CA), ki podpiše potrdilo in tako preveri identiteto naprave ali uporabnika, ki zahteva dostop. Za zagotovitev veljavnosti bo javni ključ usklajen z ustreznim zasebnim ključem, ki ga pozna samo prejemnik. Digitalna potrdila imajo določen par ključev, s katerim so povezana: enega javnega in enega zasebnega.

Danes so tako rekoč vse postavljene PKI infrastrukture uporabljajo ali RSA algoritem, ali ECC algoritem, ki pa bosta zlomljena s prihodom dovolj zmogljivega kvantnega računalnika. Naj hitro opišemo nekaj primerov uporabe, s katerimi se srečujemo vsak dan in pri katerih stojijo v ozadju digitalni certifikati, da bolje razumemo, zakaj iščemo načine, da ostanejo varni tudi po prihodu kvantnih računalnikov.

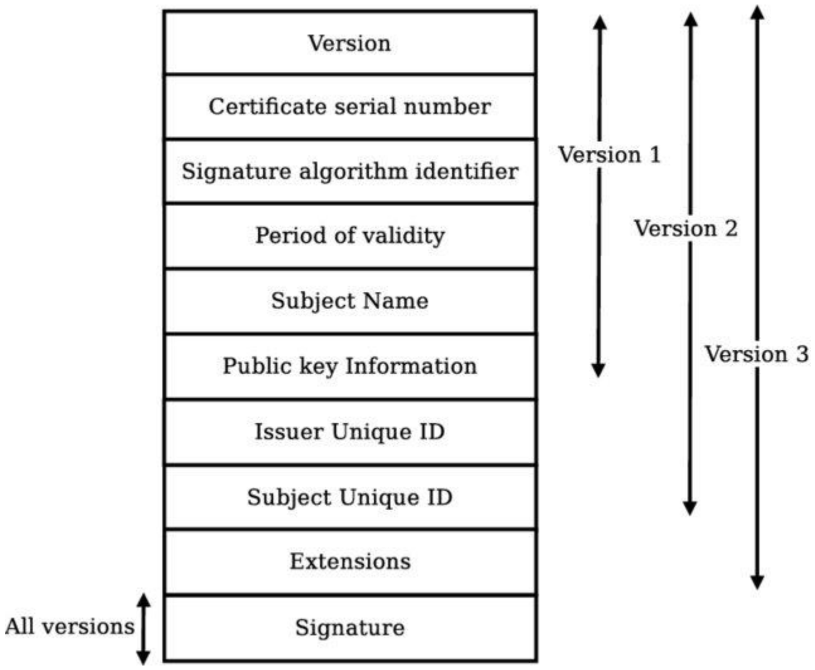
Na naši novi elektronski osebni izkaznici so nameščeni trije digitalni certifikati za različne namene uporabe. Ko se prijavljate v online banko, je velika možnost, da za vašo prijavo stoji digitalni certifikat. Ko se povezujete na spletno stran, se velik del spletnih strani v ozadju najprej identificira vašemu brskalniku, da pokažejo, da so varne. Ko nameščate novo programsko opremo, je ta običajno podpisana z zasebnim ključem proizvajalca. Če vaš operacijski sistem prepozna odgovarjajoči digitalni certifikat, bo samodejno zaupal tudi podpisani kodi.

Glavno vprašanje, ki ga želimo nasloviti, je sledeče. Do časa, ko bodo aplikacije namesto RSA in ECC uporabljale izključno PQC standardizirane algoritme, bo minilo še kar nekaj časa. Po priporočilih ZDA [12], Nemčije [16] in Francije [17] bi se sicer ta preskok moral zgoditi v prvi polovici 30. let (2030-2035). V vmesnem obdobju bomo potrebovali digitalne certifikate, s katerimi bomo lahko hkrati upravljali tradicionalne/legacy kriptografske aplikacije, hkrati pa se bomo lahko identificirali novim aplikacijam, ki podpirajo prioritetno ali izključno PQC algoritme. Kako takšne hibridne certifikate učinkovito zasnovati?

## 3.1 Hibridni certifikati

V magistrski nalogi leta 2020 Univerze v Barceloni [9] so bili zelo jasno predstavljeni 4 različni pristopi k hibridizaciji digitalnih potrdil. Vse štiri bomo površinsko predstavili, za našo implementacijo pa smo uporabili mešanico pristopa z ugnezdenimi certifikati in pristopa razširitev po meri. Cilj je modificirati standardiziran x509 certifikat na takšen način, da sta v praksi v njemu vključena 2 certifikata: prvi, ki vsebuje podpis s tradicionalnimi kriptografskimi algoritmi, in drugi, ki vsebuje post-kvantni podpis. Hkrati želimo, da certifikat ohranja dovolj uniformno obliko, da je kompatibilen s trenutno uveljavljenimi CA strukturami.

Osnovna struktura x509 certifikata je opisana na sliki 2.



**Slika 2: Struktura x509 certifikata**
Vir: [18].

### 3.1.1 *Dualni certifikati*

Najpreprostejša metoda hibridizacije je izdaja dveh potrdil, ki sta nato vedno predstavljeni v paru. Prvo potrdilo je izdano z uporabo tradicionalnega algoritma, drugo pa z uporabo post-kvantnega [19]. Obe sta podpisani z istim CA sistemom, vendar z uporabo drugačne sheme podpisovanja (konvencionalne oziroma PQ). Pomanjkljivost te metode je, da imata lahko potrdili drugačne datume veljavnosti ter da je potrebno njuno ločeno upravljanje. Vsi sistemi morajo vzdrževati obe različici potrdila in velikost datoteke skupnega certifikata je večja zaradi podvojenih informacij in parametrov.

### 3.1.2 *Konkatenacija*

Ideja konkatenacije certifikatov prihaja iz projekta Open Quantum Safe [20]. Uvedemo lahko nov identifikator objekta OID (Object Identifier), ki predstavlja, na primer, kombinacijo algoritmov RSA in Dilithium. Nato konkateniramo bajtna zapisa obeh javnih ključev v en velik javni ključ, ki ga nato lahko zapišemo v x509 certifikat. Če takšen certifikat nato obdeluje programska oprema, ki prepozna naš novi OID, natanko ve, kaj mora narediti z velikim javnim ključem - kako ga razbiti na dva ločena javna ključa in nato po potrebi uporabljati ali RSA, ali Dilithium. Čeprav je ta pristop združljiv z x509 standardom, ni združljiv s starejšimi aplikacijami, ker te ne prepoznavajo nove OID oznake.

### 3.1.3 *Ugnezdeni certifikati*

Drugačna različica dualnih potrdil je gnezdenje enega potrdila znotraj drugega potrdila ko razširitev po meri [21]. Najprej ustvarimo potrdilo z javnim post-kvantnim ključem in generiramo post-kvantni podpis, ki ga obravnavamo kot notranji certifikat. Nato ustvarimo potrdilo s tradicionalnim javnim ključem, ki predstavlja zunanje potrdilo, bajtni zapis notranjega potrdila pa je shranjen v razširitvi po meri zunanjega potrdila.

Pri tem pristopu se podatki o predmetu še vedno podvajajo, vendar je celotno potrdilo združljivo s starejšimi aplikacijami, če je razširitev označena kot nekritična. Starejša programska oprema ignorira razširitev po meri z notranjim post-kvantnim potrdilom in preveri le zunanje tradicionalno potrdilo.

Za realizacijo tega pristopa v širšem kontekstu programske opreme, ki jo zahteva uporaba infrastrukture javnih ključev, sta potrebni dve certifikacijski avtoriteti. Ena za izdajo in verifikacijo post-kvantnih certifikatov in druga za tradicionalne algoritme. S certifikatoma morata podvojeno upravljati obe avtoriteti.

### *3.1.4 Razširitve po meri*

Pristop razširitve po meri je zelo podoben gnezdenju certifikatov. Da bi se izognili dodatnim stroškom podvojenih predmetnih polj, je bilo predlagano samo hranjenje dodatnega javnega ključa in dodatnega podpisa v dveh po meri razširitvah [21]. To prinaša sicer tudi nekaj pomanjkljivosti.

Za realizacijo tega pristopa v infrastrukturi javnih ključev je potrebna ena certifikacijska avtoriteta, ki zna upravljati tako s podpisi post-kvantnih algoritmov, kot s konvencionalnimi RSA podpisi. Upravljanje s certifikati je v tem primeru lažje.

# 4 Implementacija

Cilj projekta je implementirati sistem, ki ga je mogoče dodati k že obstoječi infrastrukturi javnih ključev z minimalnimi spremembami. Za ta namen sta primerna le dva od omenjenih pristopov hibridizacije certifikatov v poglavju 3.

Pristop dualni certifikati (3.1.1) bi zaradi popolne ločitve konvencionalnih in PQ certifikatov omogočal postavitev vzporedne infrastrukture javnih ključev, ki bi temeljila na PQ algoritmih. Enako bi omogočal tudi pristop ugnezdenih certifikatov (3.1.3). Da pa bi se izognili lastnosti velikih datotek, ki jih prinese slednji in lastnosti razdružljivosti PQ in konvencionalnih certifikatov v pristopu dualnih certifikatov (3.1.1), smo združili pristop ugnezdenih s pristopom razširitev po meri (3.1.4).

Velikost certifikatov smo nekoliko zmanjšali z idejo deljenih x509 atributov med PQ in konvencionalnim podpisom. Med tem, ko so v pristopu ugnezdenih certifikatov vsi atributi podvojeni, smo podvojili le x509 atribute, ki jih na certifikat doda certifikacijska avtoriteta. Ne-razdružljivost pa je zagotovljena z uporabo x509 razširitev po meri (custom extensions).

Z našim mešanim pristopom lahko PQ del samostojno predstavlja svoj podpis, iz hibridnega certifikata pa je nemogoče izvleči posamezne dele podatke/ključe/podpis na ta način, da bi lahko dobili samostojen tradicionalni certifikat. To je želena lastnost, ker je na tak način hibridnemu certifikatu nemogoče odvzeti PQ del, kar bi znižalo pričakovano varnost hibridnega certifikata.

## 4.1 Orodja

Za postavitev testne infrastrukture javnih ključev smo postavili enostavno certifikacijsko avtoriteto z uporabo OpenSSL 3.0.7, v nadaljevanju CA. Orodje OpenSSL je uporabljeno za generacijo CA ključa in certifikata ter za nadaljnjo izdajanje uporabniških certifikatov. V konfiguraciji CA je potrebno omogoči razširitve po meri, ki jih uporabljamo za gnezdenje PQC vsebine v x509 certifikat.

Za postavitev alternativne PQC certifikacijske avtoritete, v nadaljevanju PQCA, smo razvili orodje HCTool (4.2), ki je med drugim sposobno generirati PQC del certifikata in ga vgraditi v konvencionalno CSR zahtevo, ki je nato poslana na CA.

### 4.2 Orodje za upravljanje s hibridnimi certifikati HCTool

HCTool je orodje, ki smo ga razvili za upravljanje s hibridnimi certifikati. Orodje ja napisano v Python jeziku, ki smo ga za izbrali zaradi že obstoječih knjižnic in kompatibilnostjo z različnimi operacijskimi sistemi. Uporabili smo različico Python 3.10.5.

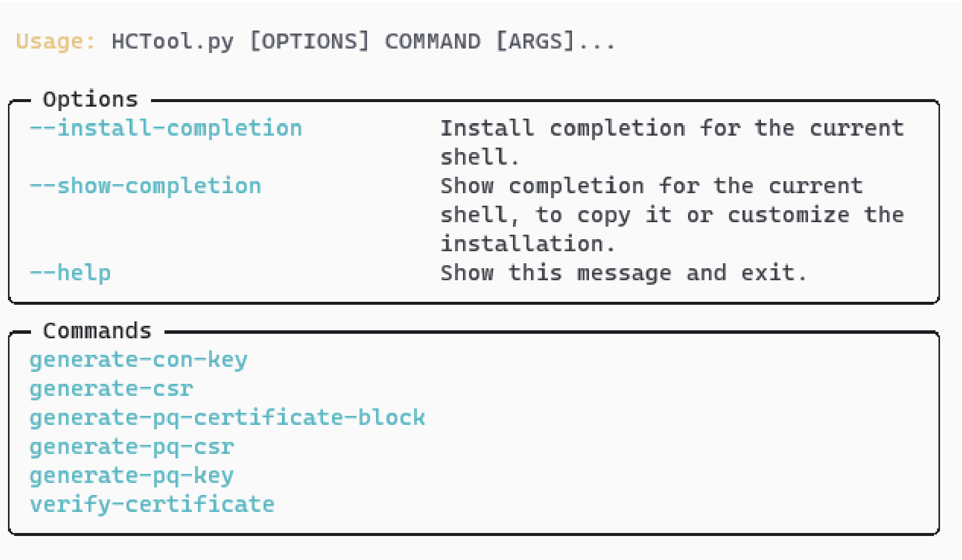
Za delo s konvencionalnimi šifrirnimi algoritmi, certifikacijskimi zahtevami in x509 certifikati smo uporabili knjižnico cryptography [23].

Za delo s PQC algoritmi smo uporabili knjižnico liboqs-python [24] različico 0.8.0, ki je del Open Quantum Safe Projekta [25].

Za uporabo ustvarjenih Python funkcij preko CLI vmesnika, smo uporabili knjižnico Typer [26] različico 0.9.0.

Implementirali smo sledeče funkcije:

- Generate_PQ_Key(algorithm, passphrase, outDir) --> Par PQC ključev
  Ustvari par ključev z izbranim PQ algoritmom.
- Generate_Con_Key(algorithm, passphrase, outDir) --> Par RSA ali EC ključev
  Ustvari par ključev z izbranim konvencionalnim algoritmom.
- Generate_PQ_CSR(PQKey, ConKey.Public, SubjectData) --> PQCSR zahteva
  Sestavi PQCSRzahtevo, opisano v poglavju 4.3.
- Generate_PQ_Cerificate_Block(PQCSR, PQKey.Public, IssuerData) --> PQCB
  Podpiše PQCSR in sestavi PQCB certifikat, opisan v 4.3.
- Verify_PQ_Certificate_Link(PQCB, PQKey.Pub) --> bool
  Preveri, ali je bil certifikat PQCB podpisan s privatnim ključem, ki pripada ključu PQKey.Public.
- Generate_CSR(PQCB, ConKey.Private) --> x509 CSR
  Ustvari konvencionalno x509 CSR zahtevo.
- Verify_Certificate(Certificate, IssuerCertificate, PQIssuerCertificate) -> bool
  Preveri, ali je hibridni certifikat podpisan s podanim PQ cerifikatom in/ali je podpisan s podanim konvencionalnim certifikatom.

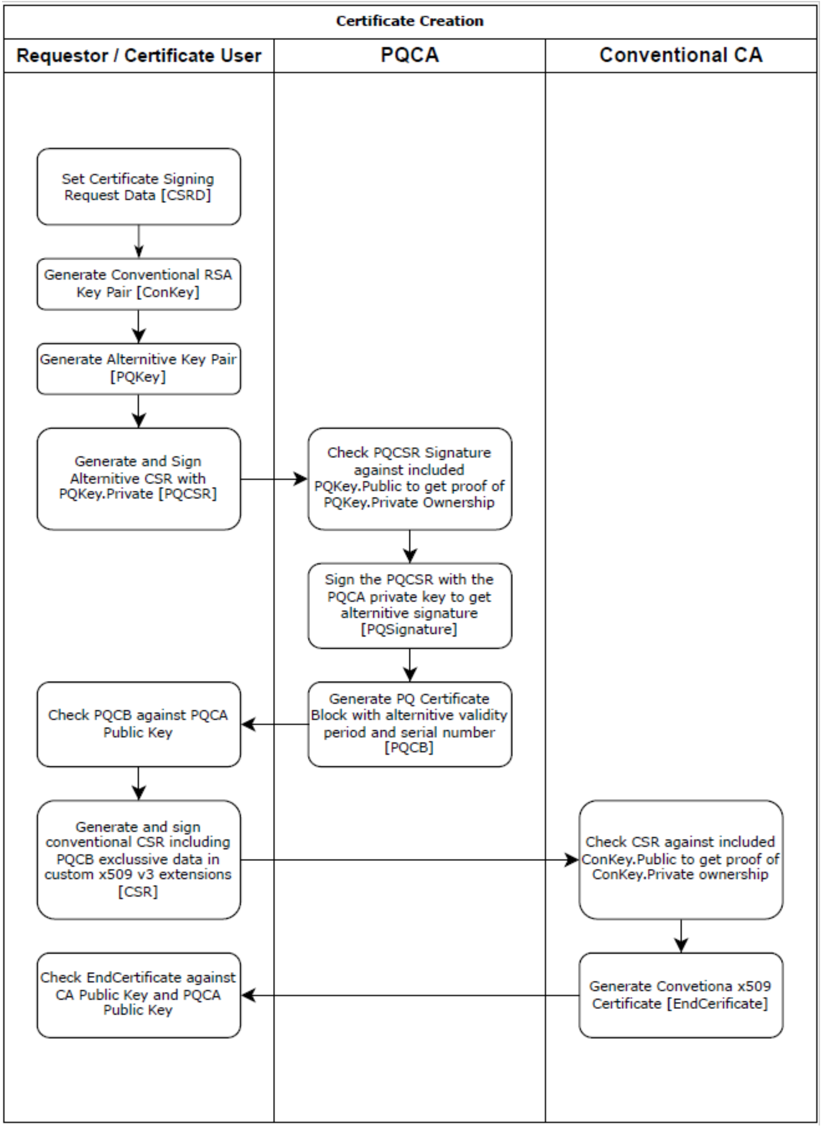


```
Usage: HCTool.py [OPTIONS] COMMAND [ARGS]...

╭─ Options ──────────────────────────────────────────────────────────╮
│ --install-completion          Install completion for the current   │
│                               shell.                               │
│ --show-completion             Show completion for the current      │
│                               shell, to copy it or customize the   │
│                               installation.                        │
│ --help                        Show this message and exit.          │
╰────────────────────────────────────────────────────────────────────╯
╭─ Commands ─────────────────────────────────────────────────────────╮
│ generate-con-key                                                   │
│ generate-csr                                                       │
│ generate-pq-certificate-block                                      │
│ generate-pq-csr                                                    │
│ generate-pq-key                                                    │
│ verify-certificate                                                 │
╰────────────────────────────────────────────────────────────────────╯
```

**Slika 3: HCTool orodje**

### 4.3 Izdajanje hibridnega certifikata

Proces izdajanja hibridnega certifikata je ponazorjen na sliki 4.



**Slika 4: Proces izdajanja hibridnega certifikata**

Za lažje razumevanje oznak v nadaljevanju dodajamo opombo, da so uporabljeni izrazi kot so Issuer, Validity, Serial, CA, CSR, TBS, itd. uporabljeni v kontekstu x509 standarda in konvencionalne infrastrukture javnih ključev (PKI). Ko želimo poudariti, da so Issuer, Validity, CA, itd vezani na post-kvantni del izdaje certifikata, je dodana predpona PQ (npr. PQCSR, PQCA, PQTBS, itd.).

Za izdajanje hibridnega certifikata je uporabljeno orodje HCTool. Uporabnik, ki zahteva certifikat (requestor) najprej ustvari par konvencionalnih ključev, v nadaljevanju ConKey (ConKey.Private in ConKey.Public) in par ključev, ki temelji na PQC algoritmih, v nadaljevanju PQKey (PQKey.Public in PQKey.Private). Istočasno se tudi odloči za algoritme samih ključev. Podprti so konvencionalni algoritmi RSA in EC ter sledeči PQC algoritmi:

- Dilithium 2, 3 in 5,
- Falcon 512 in 1024,
- SPHINCS+ s Haraka, SHA in SHAKE zgoščevalnimi (hash) algoritmi.

Uporabnik nato določi x509 atribute, ki bodo vstavljeni v CSR zahtevo. S pomočjo orodja HCTool nato ustvari PQCSR, torej zahtevo za podpis s PQC algoritmom, ki ga izda PQCA. Zahteva je sestavljena iz podatkov o subjektu, javnim ključem konvencionalnega algoritma, javnim ključem PQC algoritma in dodatnih x509 razširitev, ki so želene v končnem certifikatu. Ti podatki so konkatenirani in podpisani s PQC privatnim ključem PQKey.Private. Podpis, v nadaljevanju PQCSR.Signatue, je dodan v PQCSR.

PQCSR zahteva je nato poslana na PQCA za podpis. PQCA preveri PQCSR.Signature podpis s podatki v PQCSR in javnim ključem PQKey.Public, ki ga zahteva vsebuje. Uspešno preverjen podpis dokazuje, da si subjekt, ki certifikat zahteva, tudi lasti privatni ključ PQKey.Private. V primeru neuspešno preverjenega podpisa PQCSR:Signature, PQCA zavrne zahtevo. V nasprotnem primeru PQCA določi alternativno veljavnost certifikata, v nadaljevanju PQValidity, določi atribut PQIssuer, ki identificira PQCA, in PQSerial, ki je številka uporabljena za upravljanje s certifikati.

Nato PQCA sestavi podatke, ki bodo podpisani, v nadaljevanju PQTBS (to be signed). PQTBS podatki so sestavljeni iz vseh podatkov, ki so vključeni v PQCSR, razen PQCSR.Signature. V PQTBS so dodani atributi PQValidity, PQIssuer, PQSerial. PQTBS podatki so nato podpisani s privatnim ključem PQCA (PQCA.PQKey.Private). Podpis, v nadaljevanju PQSignature, in PQTBS podatki so združeni v "PQ Certifikat", v nadaljevanju PQCB (PQ Certificate Block), in poslani uporabniku, ki je zahteval certifikat.

Uporabnik nato preveri PQSignature podpis s podatki v PQCB in v naprej deljenem PQ certifikatu, ki pripada PQCA avtoriteti. V primeru uspešnega preverjanja uporabnik sestavi konvencionalno x509 CSR zahtevo. Zahteva je sestavljena v skladu z x509 standardom s podatki, ki so bili določeni za PQCSR. PQKey.Public in podatki, ki so bili dodani v PQCB s strani PQCA so dodani v konvencionalno zahtevo kot razširitve po meri (Custom x509 extensions). CSR zahteva je nato poslana na konvencionalno certifikacijsko avtoriteto (CA).

CA nato preveri podpis v CSR zahtevi in v primeru uspešnega preverjanja izda x509 certifikat, ki je poslan uporabniku. Uspešno preverjen podpis dokazuje, da si uporabnik, ki zahteva certifikat, tudi lasti ConKey.Private. Politika na CA je nastavljena tako, da se razširitve po meri, ki so vključene v CSR, kopirajo v certifikat.

OID številke uporabljene za x509 razširitve po meri so nanizane v Tabeli 3.

**Tabela 3: Uporabljene OID številke.**

| OID | Vrednost |
|---|---|
| **0.0.0.1** | Alternativni PQC Algoritem |
| **0.0.0.2** | Alternativni PQC Javni Ključ |
| **0.0.0.3** | Alternativna Veljavnost |
| **0.0.0.4** | Alternativni Izdajatelj (Issuer) |
| **0.0.0.5** | Alternativna Serijska Številka |
| **0.0.0.6** | Alternativni PQC Podpis |

## 5 Zaključek

V članku smo opisali postopek generacije hibridnega certifikata, ki ga je mogoče integrirati z obstoječim tradicionalnim PKI sistemom. Razvito je bilo orodje HCTool, ki s pomočjo PQC in konvencionalnih kriptografskih knjižnic omogoča upravljanje hibridnih certifikatov. V prihodnje načrtujemo integracijo orodja v manjše aplikativne projekte ter benchmark različnih implementacij kot del real-life rešitev.

## Literatura

[1] Sodobne tehnologije in storitve OTS 2022: Zbornik petindvajsete konference. Maribor 2022.

[2] PODGORELEC Vili, HERICKO Marjan »Estimating software complexity from UML models », SIGSOFT Software Engineering Notes, letnik 32, številka 2, marec 2007, str. 31-38.

[3] JOSEPH David, MISOCZKI Rafael, MANZANO Marc, et.al. »Transitioning organizations to post-quantum cryptography », Nature 605, 237–243 (2022). https://doi.org/10.1038/s41586-022-04623-2

[4] ALAGIC Gorjan, APON Daniel, COOPER David, et.al. »Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process«, NIST, Julij 2022, https://doi.org/10.6028/NIST.IR.8413-upd1

[5] SCHEIBLE Patrik »Quantum Resistant Authenticated Key Exchange for OPC UA using Hybrid X.509 Certificates«, Master thesis, UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONA, april 2022, http://hdl.handle.net/2117/191775

[6] Paul BENIOFF »The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines«. Journal of Statistical Physics, 22 (5), 1980, 563–591.

[7] Richard FEYNMAN »Simulating Physics with Computers«. International Journal of Theoretical Physics «. 21 (6/7), 1982, 467–488.

[8] IBM's Roadmap For Scaling Quantum Technology https://www.ibm.com/blogs/research/2020/09/ibm-quantum-roadmap/

[9] SCHEIBLE Patrik »Quantum Resistant Authenticated Key Exchange for OPC UA using Hybrid X.509 Certificates», magistrska naloga, UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH, Barcelona, 2020 https://upcommons.upc.edu/bitstream/handle/2117/191775/thesis_patrik_scheible.pdf?sequence=1

[10] MOSCA Michele, PIANI Marco » Quantum Threat Timeline « October 2019 https://globalriskinstitute.org/publications/quantum-threat-timeline/

[11] Post-Quantum Cryptography Standardization https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization

[12] Announcing the Commercial National Security Algorithm Suite 2.0 https://media.defense.gov/2022/Sep/07/2003071834/-1/-1/0/CSA_CNSA_2.0_ALGORITHMS_.PDF

[13] DUCAS Léo, KILTZ Eike, LEPOINT Tancrède et al. »CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme«, IACR Transactions on Cryptographic Hardware and Embedded Systems (2018): 238-268

[14] Official "CRYSTALS - Cryptographic Suite for Algebraic Lattices" webpage https://pq-crystals.org/dilithium/resources.shtml

[15] LYUBASHEVSKY Vadim »Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures« ASIACRYPT, pp. 598–616, 2009. 2, 20

[16] https://dserver.bundestag.de/btd/19/252/1925208.pdf

[17] ANSSI views on the Post-Quantum Cryptography transition https://www.ssi.gouv.fr/en/publication/anssi-views-on-the-post-quantum-cryptography-transition/

[18] X.509 Public Key Certificates, Microsoft https://learn.microsoft.com/en-us/windows/win32/seccertenroll/about-x-509-public-key-certificates

[19] BINDEL Nina, HERATH Udyani, MCKAGUE Matthew, STEBILA Douglas »Transitioning to a Quantum-Resistant Public Key Infrastructure« Post-Quantum Cryptography, Springer International Publishing, 2017, pp. 384–405, isbn: 978-3-319-59879-6

[20] STEBILA Douglas, MOSCA Michele »Post-quantum Key Exchange for the Internet and the Open Quantum Safe Project« Lecture Notes in Computer Science, 2017, DOI: 10.1007/978-3-319-69453-5_2

[21] BINDEL Nina, BRAUN Johannes, GLADIATOR Luca, STOCKERT Tobias, WIRTH Johannes »X.509-Compliant Hybrid Certificates for the Post-Quantum Transition« Journal of Open Source Software, vol. 4, no. 40, p. 1606, 2019, issn: 2475-9066. doi: 10.21105/joss.01606.

[22] GLADIATOR Luca, »Hybrid Certificates in OpenSSL«, GitHub, 2019. [Online]. Available: https://github.com/CROSSINGTUD/openssl-hybrid-certificates/blob/OQS-OpenSSL_1_1_1-stable/HybridCert_technical_documentation.pdf

[23] Python knjižnica "Cryptography": https://cryptography.io/en/

[24] Python knjižnica "liboqs-python": https://github.com/open-quantum-safe/liboqs-python/releases/tag/0.8.0

[25] "Open Quantum Safe" projekt: https://openquantumsafe.org/

# Nadgradnja obstoječega sistema za neprekinjeno integracijo in dostavo

Martin Potrč, Nejc Maleš, Matjaž Matjašec, Dušan Bratuša

Novum-RGI Germany GmbH Podružnica Maribor, Maribor, Slovenija
martin.potrc@novum-rgi.si, nejc.males@novum-rgi.si,
matjaz.matjasec@novum-rgi.si, dusan.bratusa@novum-rgi.si

V podjetju smo se odločili za nadgradnja orodja Jenkins iz različice 1 v različico 2 z namenom uvajanja metodologije zaporednega izvajanja podprocesov grajenja aplikacij, ki omogoča vzporedno delovanje in s tem povečuje učinkovitost. Ta nadgradnja je prinesla številne prednosti, med njimi boljši pregled in organizacijo opravil, ter olajšano odkrivanje in reševanje napak v programski kodi. Vizualizacija projektnih odvisnosti omogoča globlje razumevanje povezav med projekti.  Kljub prednostim so se pojavili tudi izzivi. Med njimi so se izpostavljali potreba po poenotenju procesov med različnimi projekti in prehod s tradicionalnega načina konfiguracije na uporabo programske kode za določanje procesov. Ta prehod je zahteval prilagoditev miselnosti in učenje novih pristopov v upravljanju orodja.

**Ključne besede:**
Jenkins
neprekinjena dostava
neprekinjena integracija
vzporedno izvajanje
DevOps
programska konfiguracija
cevovod
vozlišče

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.4

## 1 Uvod

Sistem za neprekinjeno integracijo in dostavo je eden izmed ključnih delov razvojnega procesa. Zagotavlja nam dodatno mero varnosti pri integraciji programske kode in obenem združuje informacije potrebne za analizo težav, ki lahko občasno nastanejo ob le tej.

Skozi zadnja leta je prišlo do sprememb pri procesu neprekinjene integracije. Uvedli so se novi postopki, ki so se bolje obnesli pri grajenju modernih aplikacij, pri katerih je za grajenje bilo potrebno izvesti več različnih faz.

V našem podjetju smo se tega lotevali z nadgradnjo procesa, ki smo ga že imeli in skušali proces grajenja aplikacij v več fazah nekako vzpostaviti z gradniki, ki so na voljo. To nam je skozi leta sicer dobro služilo ampak je postalo z uvedbo novih projektov domala nevzdržno. Za vsako novo stranko, za katero smo morali zgraditi aplikacijo, je bilo potrebno vzpostaviti proces grajenja:

- testiranje enot (angl. unit testing),
- dnevna namestitev (angl. daily build),
- test grafičnega vmesnika (z orodjem Selenium),
- izdaja posodobitev (angl. release).

To je posamezniku vzelo kar nekaj časa, saj je vzpostavitev tega procesa potekala bolj ali manj ročno.

Rešitev se je ponudila v obliki neprekinjene dostave in podpore v novejši različici Jenkins CI. Ovrednotili smo več alternativnih rešitev, ampak na koncu smo se odločili za Jenkins, saj nam je ponujal celostno rešitev našega problema. Prav tako smo že vrsto let uporabljali Hudson CI nato njegov derivat Jenkins CI in to nam je dalo dodatno zaupanje v to odločitev. Taka odločitev je seveda zelo pomembna, saj sistema za proces integracije izvorne kode v podjetjih načeloma na menjujemo prav pogosto.

Odločitev za uporabo Jenkins CI se je na koncu izkazala kot primerna. Prehod na novo različico ter na nov način grajenja je trajal kar nekaj časa in še ni popolnoma končan. Prestaviti je bilo potrebno približno 80 Jenkins opravil, med katerimi so bila opravila za izvajanje testov, dnevnih namestitev ter opravila za grajenje paketov za izdajo. Slednja so povzročala največ težav in so še zmeraj v fazi postopnega prestavljanja na nov proces.

## 2 Neprekinjena integracija, dostava in namestitev

*Neprekinjena integracija* (angl. continuous integration) je proces sprotne integracije (vsaj enkrat na dan) programske kode v sistem za upravljanje izvorne kode (SCM) in poganjanje avtomatiziranih opravil za izvajanje avtomatiziranih testov (Unit Test, Acceptance Test).

*Neprekinjena dostava* (angl. continuous deployment) je razširitev procesa neprekinjene integracije z cevovodom opravil, v katerem lahko zaporedno ali vzporedno izvajamo opravila potrebna za izgradnjo aplikacij oz. rešitev do te mere, da so pripravljene za namestitev bodisi v testna ali produkcijska okolja.

*Neprekinjena namestitev* (angl. continuous delivery) je razširitev procesa neprekinjene dostave, kjer obstoječ cevovod razširimo z avtomatizirano namestitvijo v bodisi testna ali produkcijska okolja. Uporaba tega procesa ni nujno zmeraj možna, saj lahko avtomatizirano nameščanje zahteva tudi večstopenjsko preverjanje kvalitete.

## 3 Glavne pomanjkljivosti in izzivi obstoječe rešitve

Čeprav smo obstoječo rešitev neprekinjene integracije skušali pragmatično uporabljati in z njo uvesti neko vrsto cevovodov, preko katerih bi lahko izvajali preproste cevovode (z uporabo projektnih odvisnosti), smo vendarle bili primorani preiti na modernejšo rešitev. Jenkins različica 1.0 je dosegla konec svoje življenjske dobe (angl. End of Life - EOL) že leta 2016 [1]. Takrat še tega nismo jemali tako resno, saj je sistem služil svojemu namenu v obstoječi konfiguraciji še kar nekaj let, preden je prišlo do resne diskusije o tem, da je potrebno tudi v tej smeri kaj postoriti.

EOL je sicer eden izmed glavnih razlogov za nadgradnjo oz. prehod, dejansko je pa razlogov seveda več. Četudi bi Jenkins 1.0 bil še zmeraj vzdrževan, bi bilo za dosego naših ciljev potrebno preiti na drugo rešitev.

Ključnega pomena pri razvoju modernih informacijskih rešitev je uporaba modernega sistema za neprekinjeno integracijo. Ena izmed ključnih praks v razvoju procesa neprekinjene integracije je uvedba cevovodnega povezovanja opravil, kar lahko krajše strnemo kot proces neprekinjene dostave. Jenkins 1.0 ne ponuja podpore za cevovode opravil (angl. job pipelining), kar otežuje implementacijo takega procesa. Rešitve za uvedbo neprekinjene dostave v Jenkins 1.0 so obstajale ampak niso dosegale željene kvalitete.

V obstoječi rešitvi z Jenkins 1.0 smo poskušali v opravila strniti tudi dodatne informacije o projektih, ki bi bile dostopne razvijalcem na enem mestu. Pojavila se je ideja, da bi lahko razvijalec na spletnem mestu opravila našel pomembne informacije o rezultatih Jenkins opravil, kot tudi dostopal do informacij o izvajalnih okoljih ter protokolih izvajanj. To je bilo možno izvesti v dokaj okrnjeni obliki v Jenkins 1.0 z veliko podvajanja informacij in naknadnega vzdrževanja le teh.

Vzdrževanje in uvajanje novih opravil v obstoječi rešitvi z Jenkins 1.0 smo izvajali ročno. Obstoječa rešitev ni ponujala avtomatizacije za opravila brez dodatnih razširitev. Sicer bi lahko to dosegli z alternativnimi rešitvami kot so Jenkins DSL vtičnik ali z Jenkins REST vmesnikom vendar teh rešitev nikoli nismo resnično ovrednotili. Ob poskusu uporabe te prve verzije DSL vtičnika je sama uporaba tega v našem CI okolju kazala na časovno zahtevno uvedbo. Prav tako smo se na tej točki že zavedali modernejših rešitev, kjer je integracija avtomatizacije privzeto dostopna v izvorni rešitvi (angl. out-of-the-box), zato smo uvajanje in ovrednotenje takih razširitev obstoječe Jenkins 1.0 enostavno opustili.

## 4 Evaluacija CI ponudnikov

Trenutno je na trgu ogromno rešitev za CI/CD, vendar smo pregled orodij zmanjšali na rešitve, ki so ustaljene, imajo dovolj veliko bazo uporabnikov in sprejemljivo ceno. S tem smo želeli doseči stabilnost in razširljivost orodja v prihodnosti.

Trenutno na trgu obstaja veliko rešitev, ki bi jih lahko razdelili v dve skupini. Inovativne (angl. cutting edge) in ustaljene ter razširljive. Največje razlike med tema kategorijama so novejši in lepši vmesniki z modernejšimi pristopi ali močno integracijo z drugimi orodji v nasprotju s funkcionalnostjo in razširitvami. Da bi zmanjšali nabor možnih opcij, je bila naša zahteva podpora naslednjih orodij: Git, SVN, Maven, Java, Docker. Zaradi velikosti projektov in trajanja izvajanja testov je bila dodatna zahteva gostovanje na lastni infrastrukturi. Nekatera orodja pa smo izločili, ker zahtevajo gostovanje izvorne kode v oblaku ali pa celo v odlagališču oziroma repozitoriju (angl. repository) specifičnega ponudnika, npr. BitBucket.

V enakem obdobju smo posodabljali več področij infrastrukture. Med njih spadajo tudi Git repozitorij, zato je potrebno omeniti GitLab in njihov CI. Čeprav nudi zanimiv pristop za ustvarjanje in zaganjanje cevovodov in zelo dobro integracijo s samim Git repozitorijem, so za nas nastale težave z vključitvijo SVN repozitorijev in pregledom nad testi ter okolji, ki se nameščajo.

Med trenutnimi rešitvami in na podlagi naših zahtev je bila izbira Jenkins 2.0 [2], dokaj enostavna in naravna. V primerjavi z ostalimi orodji je najbolj univerzalna rešitev in je ni potrebno združevati z dodatnimi orodji.

## 5 Cilji uvedbe nove CI rešitve

Namen nove rešitve za neprekinjeno integracijo in neprekinjeno dostavo (CI/CD) je bil doseči jasne cilje: poenostavitev postopkov z izgradnjo, testiranjem in dostavo programske opreme; učinkovita uporaba že vzpostavljenih postopkov; natančna sledljivost vseh sprememb; zmanjšanje operativnih nalog ekipe, ki skrbi za razvojne sisteme (angl. DevOps); samodejno vključevanje CI/CD procesov v skladu s spremembami v repozitorijih ter olajšanje nadaljnjega razvoja s pomočjo gradnje razširljive knjižnice.

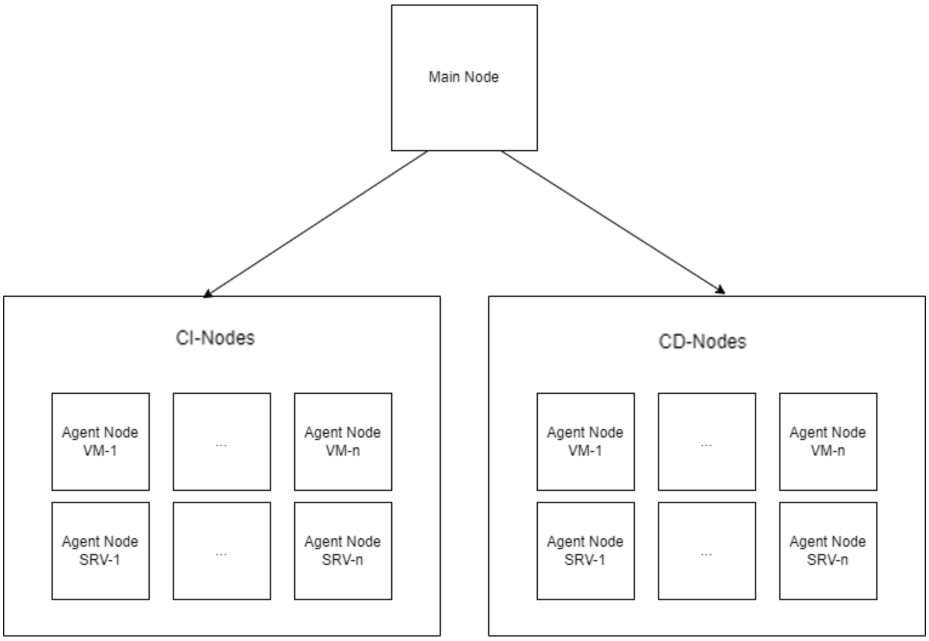
## 6 Postopek prehoda

Pri menjavi orodja smo imeli dovolj časa, virov in sodelavcev, da smo prehod dobro načrtovali in ga naredili postopoma. Postopek je bil razdeljen na različne faze: načrtovanje, vzpostavitev Jenkins 2.0 strežnika, vzpostavitev delovnih vozlišč (angl. node), implementacija cevovoda za CI za posamezen projekt, preoblikovanje in poenotenje CI cevovodov, implementacija CD cevovodov za dnevno namestitev, implementacija CD cevovodov za testiranje s Selenium orodjem, preoblikovanje in poenotenje CD cevovodov ter vpeljava skupne knjižnice, implementacija CD cevovodov za izdajo posodobitev ter nazadnje še implementacija cevovodov za administrativne naloge.

### 6.1 Infrastruktura

Po načrtovanju smo vzpostavili vzporeden strežnik za Jenkins 2.0. Da bi lahko zagotovili neodvisnost infrastrukture in dinamičnost vozlišč, smo se odločili, da bo eden izmed strežnikov deloval le kot osnovno vozlišče (angl. main node) brez izvajalcev in bo služil le kot nadzornik opravil. Vzpostavili smo tudi delovna vozlišča na virtualnih izvajalnih okoljih. Čeprav je vzpostavitev dodatnega izvajalca dokaj enostavna, smo odvisni od veliko orodij, ki jih ni mogoče vključiti preko Jenkinsa. Za te namene smo pripravili šablone za virtualna izvajalna okolja Unix in Windows, pri slednjih pa definirali še pravila preko skupinske politike (angl. group policy) objektov. Nastala je tudi skripta, ki z nekaj vhodnimi podatki avtomatizirano namesti izvajalca (agenta) in ga doda v delovno skupino.

Dodatna vozlišča smo vključevali vzporedno z migracijo. V delovne skupine vozlišč smo poleg virtualnih vključili tudi fizične računalnike.



**Slika 5: Arhitektura vozlišč**

### 6.2 Konfiguracija cevovodov

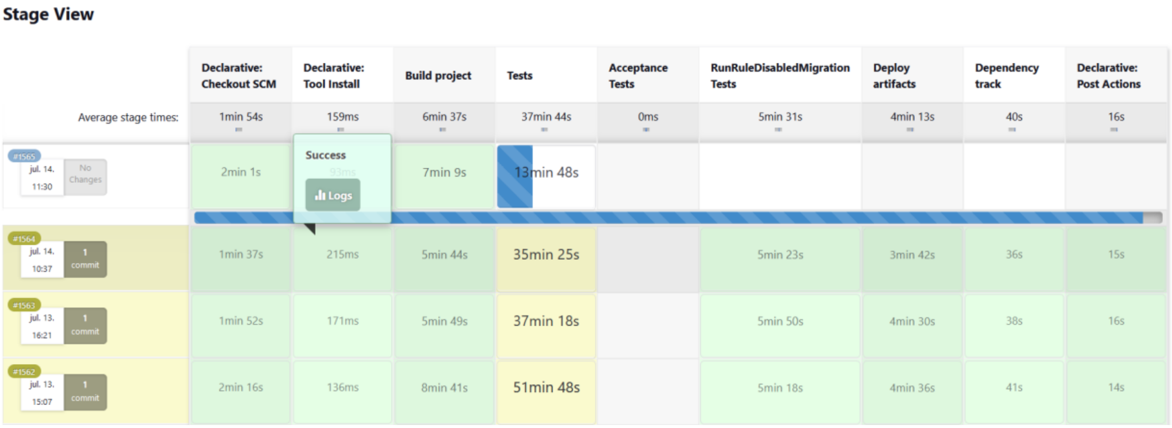
V želji, da bi poenotili in avtomatizirali čim več procesov, smo uvedli tok delovnih procesov z uporabo sledečih funkcij, ki jih Jenkins v verziji 2.0 nudi:

- Jenkins Multibranch Pipeline omogoča ustvarjanje osnovnih opravil (angl. Job), kateri prebirajo vse veje repozitorija po definiciji cevovoda.
- Definicija za izvajanje cevovoda je shranjena v samem repozitoriju kot datoteka s konfiguracijo in vsebuje izvorno kodo za cevovod spisano v jeziku Groovy. Groovy je skriptni jezik, ki se izvaja v JVM in lahko dostopa tudi do razredov in metod v Javi.
- Vpeljali smo tudi globalno knjižnico in preselili vso logiko iz cevovodov na projektih. Tako je ostala na projektih samo konfiguracija za same projekte, ki uporablja definirane cevovode v knjižnici. Na ta način lahko spremenimo cevovod za vse projekte hitro in enostavno.

### 6.3 Neprekinjena integracija – CI

Z vzpostavljeno infrastrukturo, smo lahko začeli z implementacijo cevovodov za CI, ki je razdeljen na devet faz, kar lahko vidimo na sliki Slika 6

1. Pridobitev izvorne kode (angl. checkout)
2. Deklarativna namestitev orodij (Maven, Java ipd.)
3. Prevajanje kode (angl. compile)
4. Izvajanje testov enot (angl. unit tests)
5. Izvajanje sprejemnih testov (angl. acceptance tests)
6. Opcijska faza za potrebe izvajanja testov pod določenimi specifičnimi nastavitvami (kot na primer za potrebe migracijskih projektov).
7. Nalaganje artefaktov v skupen repozitorij
8. Preverjanje paketnih odvisnosti (angl. dependency track) in varnostnih lukenj
9. Čiščenje delovnega prostora



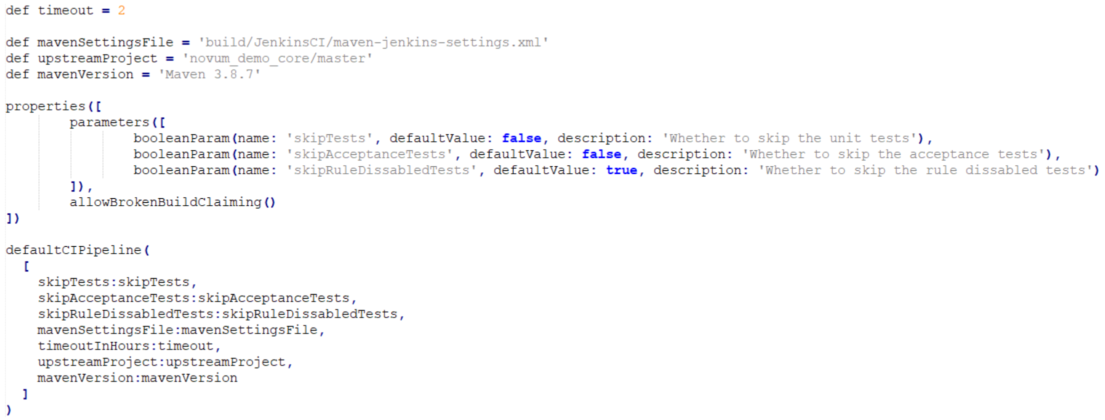
**Stage View**

|  | Declarative: Checkout SCM | Declarative: Tool Install | Build project | Tests | Acceptance Tests | RunRuleDisabledMigration Tests | Deploy artifacts | Dependency track | Declarative: Post Actions |
|---|---|---|---|---|---|---|---|---|---|
| Average stage times: | 1min 54s | 159ms | 6min 37s | 37min 44s | 0ms | 5min 31s | 4min 13s | 40s | 16s |
| #1565<br>jul. 14.<br>11:30<br>No Changes | 2min 1s | Success<br>93ms<br>Logs | 7min 9s | 13min 48s |  |  |  |  |  |
| #1564<br>jul. 14.<br>10:37<br>1 commit | 1min 37s | 215ms | 5min 44s | 35min 25s |  | 5min 23s | 3min 42s | 36s | 15s |
| #1563<br>jul. 13.<br>16:21<br>1 commit | 1min 52s | 171ms | 5min 49s | 37min 18s |  | 5min 50s | 4min 30s | 38s | 16s |
| #1562<br>jul. 13.<br>15:07<br>1 commit | 2min 16s | 136ms | 8min 41s | 51min 48s |  | 5min 18s | 4min 36s | 41s | 14s |

Slika 6: Prikaz faz za CI na posameznem projektu

Cevovod je bil razvit in pripravljen na projekte. Preden smo migrirali vse projekte, smo naredili preoblikovanje in prestavili definicijo cevovoda v globalno knjižnico. S tem smo na projektu obdržali le minimalno konfiguracijo in možnost spremembe le te, brez da vpliva na ostale projekte. Vsi projekti so konfigurirani po enakem postopku kot kaže spodnji primer Groovy kode na sliki Slika 7.

Na ta način lahko definiramo privzeto obnašanje cevovoda na projektu, obstoječi parametri pa omogočajo tudi ročne spremembe.



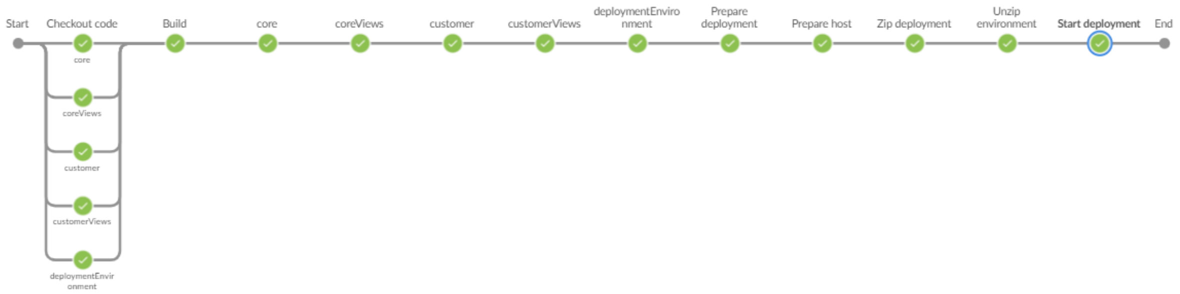
```
def timeout = 2

def mavenSettingsFile = 'build/JenkinsCI/maven-jenkins-settings.xml'
def upstreamProject = 'novum_demo_core/master'
def mavenVersion = 'Maven 3.8.7'

properties([
        parameters([
                booleanParam(name: 'skipTests', defaultValue: false, description: 'Whether to skip the unit tests'),
                booleanParam(name: 'skipAcceptanceTests', defaultValue: false, description: 'Whether to skip the acceptance tests'),
                booleanParam(name: 'skipRuleDissabledTests', defaultValue: true, description: 'Whether to skip the rule dissabled tests')
        ]),
        allowBrokenBuildClaiming()
])

defaultCIPipeline(
  [
    skipTests:skipTests,
    skipAcceptanceTests:skipAcceptanceTests,
    skipRuleDissabledTests:skipRuleDissabledTests,
    mavenSettingsFile:mavenSettingsFile,
    timeoutInHours:timeout,
    upstreamProject:upstreamProject,
    mavenVersion:mavenVersion
  ]
)
```

Slika 7: Primer konfiguracije za Jenkins cevovod

## 6.4 Neprekinjena dostava – CD

Nadaljevali smo s cevovodi za dnevne namestitve (angl. daily build). Poslužili smo se enakega pristopa za konfiguracijo in definicijo cevovoda. Ker so projekti med seboj sklopljeni, cevovodov nismo vključili v repozitorije projektov, ampak naredili namenskega, v katerem veje predstavljajo okolje, ki ga želimo namestiti. Ker so projekti v produkcijski rabi in imajo lastne razvojne poti, imamo namestitve za vse aktivne različice na projektih. Blue Ocean vmesnik na sliki Slika 8 predstavlja faze, ki so potrebne za namestitev okolja.

Pridobivanje izvorne kode iz repozitorijev se izvaja paralelno, grajenje pa zaporedno. Faze za ta del cevovoda se generirajo dinamično, glede na konfiguracijo okolja. V sklopu cevovoda še pripravimo gostiteljski sistem in preverimo ali obstaja kakšna namestitev tega projekta nato jo ugasnemo in odstranimo. Rezultat se arhivira in prenese na gostitelja, kjer se razpakira in namesti. Namestitev zajema namestitev servisov, pripravo baze, zagon okolja in zagon parametrizacije.



Slika 8: Blue Ocean pogled na Demo dnevne namestitve

### 6.4.1 *CD – Selenium*

Selenium okolja za nas predstavljajo okolja podobna okoljem dnevne namestitve, vendar so namenjena zagonu testov s pomočjo Selenium orodja, ki je namenjeno testiranju spletnih strani. V primerjavi s prejšnjo točko, ima cevovod le dodatno fazo zaganjanja testov in dodatno parametrizacijo za teste. Tako kot pri dnevnih namestitvah, se aplikacija prenese na gostitelja, kjer se sistem zažene, testi pa se izvajajo na Jenkins vozlišču. Ker se testira spletni vmesnik, se testi izvajajo počasneje. Uporabljamo malo večje časovne omejitve (na primer čakanje na obnovitev strani pri spremembi vnosnih podatkov), da smo prepričani, da vse funkcionalnosti delujejo, tudi če je v tistem trenutku na voljo manj virov.  V primeru da zaznamo večje nihanje v učinkovitosti, izvajamo teste učinkovitosti ločeno.

V sklopu Selenium testov, zajemamo tudi zaslonske slike v primeru napake ali padlega testa, ki so dostopne direktno preko Jenkinsa, kar razvijalcem malo olajša iskanje napak.

V primerjavi z Jenkins 1.0, kjer so delovni tokovi delali avtomatizirano, dostop do informacij pa je bil otežen, potrebno je bilo poiskati in se povezati na gostitelja, na katerem smo ročno iskali te datoteke.

### *6.4.2 CD – Posodobitev*

Tudi posodabljanje ima veliko korakov skupnih s CD cevovodi, saj je tudi tam potrebno pripraviti paket aplikacije za gostiteljski sistem. Razlikuje pa se v tem, da se ne proizvede arhiv aplikacije na podlagi zadnjega stanja kode, ampak na podlagi oznak verzije (angl. tag). Na tej točki smo se namenoma odločili, da se oznake verzij pripravijo vnaprej in ne v sklopu cevovoda za posodobitev. Obnašanje cevovoda se razlikuje glede na projekt, odvisno od tega, ali našo aplikacijo izdajamo le kot namestitveni paket ali pa kot paket v kombinaciji z aplikacijskim strežnikom, na primer v obliki celotnega vsebnika ali "Docker Image". Pripravimo lahko tudi različne arhive. Vse to je nastavljivo preko parametrov na grafičnem vmesniku, privzete vrednosti pa so definirane v konfiguraciji za posamezno stranko.

## 6.5 Administracija

Na Jenkinsu 1.0 smo imeli veliko več administrativnih oz. nadzornih opravil, ki pa smo jih prestavili na druga namenska orodja, kot so Zabbix, Grafana ipd. Največ opravil je bilo za preverjanje stanja namestitev in njihove dosegljivosti, tudi za produkcijska okolja, ter varnostno kopiranje internih sistemov, nastavitev in podatkov.

Trenutno so administrativni cevovodi namenjeni lažjemu vzdrževanju infrastrukture. Za potrebe Selenium testov, smo vezani na uporabo Firefox brskalnika in Gecko gonilnika. Zaradi omejitev iz strani teh dveh aplikacij, smo se odločili, da te sisteme posodabljamo kar preko Jenkinsa. S tem zagotovimo, da se vsa testna okolja in izvajanje testov obnaša enako. Ne pride do težav, da se na kakšnem strežnik/vozlišču pozabi ali spregleda. Na ta način lahko verzijo tudi fiksiramo, v primeru, da so najnovejše različice nekompatibilne.

## 6.6 Izzivi in težave pri razvoju

Po naši evaluaciji se je orodje Jenkins izkazalo kot najustreznejše. Prepričala nas je univerzalnost orodja za namene CI in CD, predvsem zaradi odprte kode in razširljivosti s pomočjo vtičnikov. Deloma so ti vtičniki razviti iz strani skupnosti (angl. community), za katere pa ni nujno, da se držijo standardov, da so dobro testirani ali dokumentirani. Ena izmed večjih težav je bila pri uporabi različnih vtičnikov in njihovim obnašanjem v sklopu naših cevovodov. Neobstoječa ali zelo površna dokumentacija nas je privedla do veliko tako imenovanih "poskus in napaka" pristopov (angl. trial and error) pri razvoju cevovodov.

Ko smo začeli ustvarjati cevovode, smo se odločili za definicijo cevovodov kot "konfiguracija v kodi" (angl. configuration by code). Za te namene sta na voljo skriptni in deklarativni cevovod. Oba uporabljata datoteko konfiguracije "jenkinsfile", ki je spisana v Groovy jeziku.

Skriptni cevovodi so najbolj zmogljivi, saj omogočajo direkten dostop in konfiguracijo do vseh modulov, v nasprotju z deklarativnimi cevovodi, ki so enostavnejši in bolj pregledni, kar je bil razlog, da smo se odločili za deklarativni pristop. Hitro smo ugotovili, da samo z deklarativnim pristopom ne bomo uspeli rešiti vseh zastavljenih ciljev in delovnih tokov. Odločili smo se, da vse gradnike poskušamo spisati v deklarativnem načinu in uporabimo skriptni pristop, kjer je potrebno. Kot smo ugotovili kasneje, je to za nas obvezno. Določeni vtičniki ponujajo uporabo samo preko Jenkins vmesnika ali pa preko skriptnega pristopa. Izogniti smo se morali tudi različnim vtičnikom, ki so namenjeni predvsem uporabi preko spletnega vmesnika in jih je skoraj nemogoče konfigurirati preko kode.

# 7 Zaključek

Splošno zadovoljstvo s prehodom na Jenkins 2.0 je bilo s strani naših razvijalcev izboljšano v primerjavi z različico 1.0. Sam razvoj in vzdrževanje Jenkins cevovodov je sicer še nekoliko težaven, vendar počasi pridobivamo izkušnje in sproti rešujemo različne izzive ter težave. V načrtu je tudi posodobitev samih Jenkins vozlišč in Jave, s katero se poganjajo. Zanimivo pri tem je, da samih vozlišč ni mogoče posodobiti preko Jenkins orodja. Edini način je, da na glavnem vozlišču sprožamo skripte, ki ta vozlišča obvestijo, da naj posodobijo določena orodja, kot so Gecko gonilnik, Firefox brskalnik za potrebe testiranja v Selenium ogrodju ipd.

V ekipi DevOps se že opaža manj ročnega posredovanja pri uvajanju novih cevovodov ali spreminjanju obstoječih. Razvijalci so namreč že navajeni, da smejo take spremembe urediti sami glede na zahteve, ki bi sicer prešle od njih samih na DevOps ekipo.

# Literatura

[1] https://www.versio.io/product-release-end-of-life-eol-Jenkins-Jenkins.html, obiskano 17. 6. 2023

[2] https://www.jenkins.io/doc, obiskano 2. 6. 2023

# Vpeljava zabojnikov v oblačno zasnovana zasebna omrežja 5G

Urban Zaletel, Kristjan Voje, Benjamin Burgar, Uroš Brovč

Kontron d.o.o, Kranj, Slovenija
urban.zaletel@kontron.si, kristjan.voje@kontron.si,
benjamin.burgar@kontron.si, uros.brovc@kontron.si

Današnja sodobna podjetja imajo vse večje zahteve na področju mobilne povezljivosti. Nastajajoče aplikacije in industrijska okolja zahtevajo večjo pasovno širino, predvsem pa izjemno nizko zakasnitev, več spektra in večjo zanesljivost. Z dodeljevanjem namenskega spektra za ta okolja s strani regulatorjev (Nemčija, Norveška, Združeno kraljestvo in druge napredne države) so v podjetjih nastali pogoji za uvajanje zasebnih mobilnih omrežij, ki so samostojna in razširjajo njihova obstoječa brezžična omrežja. Oblačno zasnovana omrežja podjetjem omogočajo učinkovito upravljanje in pomagajo pri sprejemanju hitrejših in boljših poslovnih odločitev. V prispevku smo naslovili motivacijo prehoda na oblačno arhitekturo in pridobljene izkušnje, ki smo jih pridobili ob razvoju in uvajanju oblačno zasnovanih rešitev na Kubernetesu. Razvoj 5G je zahteval usklajenost med ekipami, ki so razvijali omrežne elemente v različnih jezikih (Java, Golang, C) in enoten pristop do modernih DevSecOps praks (infrastruktura-kot-koda, avtomatizacija, varnostno skeniranje, Helm). Kubernetes in mikroservisi so nov koncept za telekomunikacijske aplikacije, ki se počasi selijo iz paradigme VNF (virtual network functions) proti paradigmi CNF (cloud-native network functions)[6]. V prispevku so izpostavljeni izzivi, s katerimi smo se soočali pri implementacij oblačnih omrežnih funkcij. Nanizane so tudi številne pridobljene kompetence, ki smo jih inženirji pridobili tekom trajanja projekta.

**Ključne besede:**
5G
privatna omrežja
CNF
DevOps
DevSecOps
Kubernetes
Helm

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.5

# 1 Uvod

5G privatna omrežja predstavljajo razširitev 5G tehnologije na področje podjetij in organizacij, ki želijo vzpostaviti lastno zasebno omrežje z visoko zmogljivostjo in nizko zakasnitvijo. Ta omrežja omogočajo podjetjem večji nadzor, varnost in prilagodljivost pri povezovanju njihovih naprav in sistemov.

Privatna 5G omrežja se gradijo na enakih temeljih kot javna 5G omrežja, vendar se upravljajo in nadzirajo znotraj organizacije. To pomeni, da podjetja dobijo lastno omrežno infrastrukturo, vključno s baznimi postajami in strežniškimi zmogljivostmi, kar jim omogoča, da prilagodijo omrežje svojim specifičnim potrebam.

Eden glavnih razlogov za uvedbo privatnih 5G omrežij je izboljšanje zanesljivosti in varnosti povezave. Podjetja lahko nadzorujejo in omejijo dostop do svojega omrežja, kar zmanjšuje tveganje za morebitne vdore ali izpade povezave. Poleg tega imajo privatna omrežja višjo prioriteto in večjo pasovno širino, kar zagotavlja stabilno povezavo za kritične aplikacije.

Privatna 5G omrežja omogočajo tudi večjo fleksibilnost pri upravljanju in integraciji naprav. Organizacije lahko povežejo različne naprave in senzorje v svoje omrežje ter izkoristijo prednosti interneta stvari. To odpira vrata za inovacije na področjih kot so industrija 4.0, pametne tovarne, avtonomna vozila, zdravstvena oskrba in še veliko več.

Kljub temu pa obstajajo tudi nekateri izzivi pri uvedbi privatnih 5G omrežij, vključno z visokimi stroški vzpostavitve in vzdrževanja infrastrukture ter potrebo po posebnem znanju in usposabljanju za upravljanje omrežja.

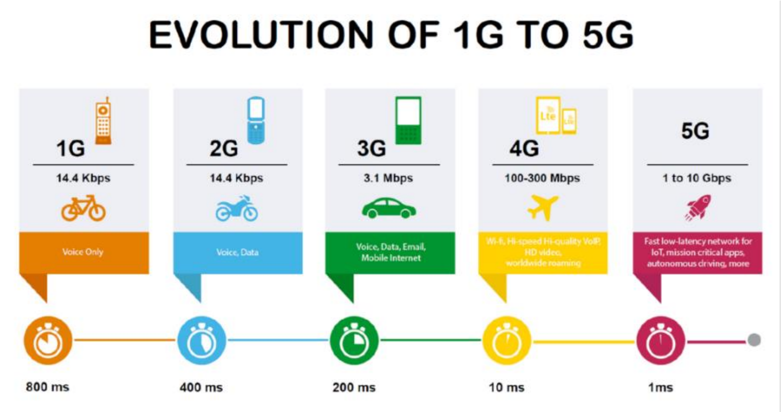
Optimizacija stroškov postavitve in vzdrževanja 5G privatnih omrežij se lahko doseže s pomočjo uporabe oblačnih postavitev. Oblačno zasnovana omrežja omogočajo organizacijam koriščenje infrastrukture zunanjih ponudnikov oblakov, kar je v nekaterih primerih veliko bolj učinkovito od postavljanja in vzdrževanja svoje infrastrukture.

# 2 5G omrežje

## 2.1 Evolucija mobilnih omrežij

Evolucija mobilnih omrežij po generacijah je temeljila na potrebi po zadovoljevanju naraščajoče potrebe po prenosu podatkov, izboljšanju učinkovitosti omrežja in omogočanju novih storitev in aplikacij. Vsaka generacija je nadgradila prejšnjo, uvedla nove tehnologije in zmogljivosti, ki so oblikovale način komunikacije in interakcij v digitalni dobi.

Na sliki 1 so podane ključne značilnosti posamezne generacije mobilnih omrežij.



**Slika 1: Evolucija mobilnih omrežij**

Vir: [3].

## 2.2 Kaj nam omogoča 5G

5G omrežje omogoča izjemno hitrost prenosa podatkov, nizko zakasnitev ter večjo zmogljivost in povezljivost, kar omogoča napredne aplikacije, inovacije ter nove poslovne modele v različnih panogah.

Pregled ključnih prednosti:

- Visoka hitrost prenosa podatkov: izjemno visoke hitrosti prenosa podatkov, ki presegajo zmogljivosti prejšnjih generacij omrežij. To vključuje podporo za prenos velikih količin podatkov, kot so visokokakovostni video posnetki in pretočne storitve v realnem času.
- Nizka zakasnitev (latenca): izjemno nizka zakasnitev pri prenosu podatkov, kar je ključno za aplikacije, ki zahtevajo takojšen odziv. To vključuje avtonomna vozila, virtualno resničnost, obogateno resničnost in druge aplikacije, ki zahtevajo hitro in brezhibno delovanje v realnem času.
- Večja zmogljivost omrežja: zasnova za večjo zmogljivost, kar pomeni, da mora omogočati hkratno povezavo večjega števila naprav na istem območju. To je ključno za podporo različnim napravam v okviru interneta stvari (IoT), kot tudi za zagotavljanje visoke kakovosti storitev v gostih urbanih območjih.
- Podpora za različne uporabniške scenarije: prilagodljivost in sposobnost podpirati različne uporabniške scenarije, vključno z mobilnimi napravami, IoT napravami, industrijsko avtomatizacijo, pametnimi mesti in drugimi naprednimi aplikacijami.
- Energijska učinkovitost: zmanjšuje porabo energije v omrežju in podaljšuje življenjsko dobo baterij pri napravah.
- Visoka zanesljivost in varnost: zagotavlja visoko zanesljivost in varnost komunikacije, še posebej pri kritičnih aplikacijah, kot so kritična infrastruktura, pametni sistemi v prometu in oddaljena medicinska oskrba. Omrežje mora imeti tudi ustrezne varnostne mehanizme za zaščito uporabniških podatkov in preprečevanje napadov.

## 2.3 Arhitektura

Arhitektura 5G omrežja je bila sestavljena tako, da zagotavlja zahteve, kot so zapisane v poglavju 2.2. Ključne komponente 5G arhitekture vključujejo (slika 2):

- Uporabniška oprema[1] (UE): Naprava, ki jo uporabnik uporablja za dostop do 5G omrežja, na primer pametni telefon, tablica ali druga naprava z vgrajeno 5G tehnologijo.
- Radijsko dostopno omrežje[2] (RAN): Brezžično omrežje, ki omogoča komunikacijo med UE in omrežno infrastrukturo. RAN v 5G sistemu je znan kot NG-RAN in uporablja tehnologije, kot so Massive MIMO[3] in oblikovanje snopa[4] [7] za izboljšanje zmogljivosti in kakovosti povezave.
- 5G jedro[5] (5GC): Osrednji del arhitekture omrežja 5G, ki obvladuje in upravlja vse ključne funkcije omrežja. Deluje kot centralni vozliščni sistem, ki zagotavlja povezljivost, komunikacijo in upravljanje med uporabniki, napravami in storitvami v omrežju 5G.

---

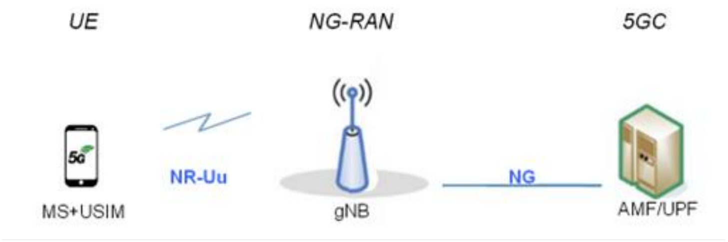
[1] uporabniška oprema – izvorno "user equipment", okrajšava "UE"

[2] radijsko dostopno omrežje - izvorno "radio access network", okrajšava "RAN"

[3] mMIMO – Massive Multiple Input Multiple Output

[4] oblikovanje snopa – izvorno "beamforming"

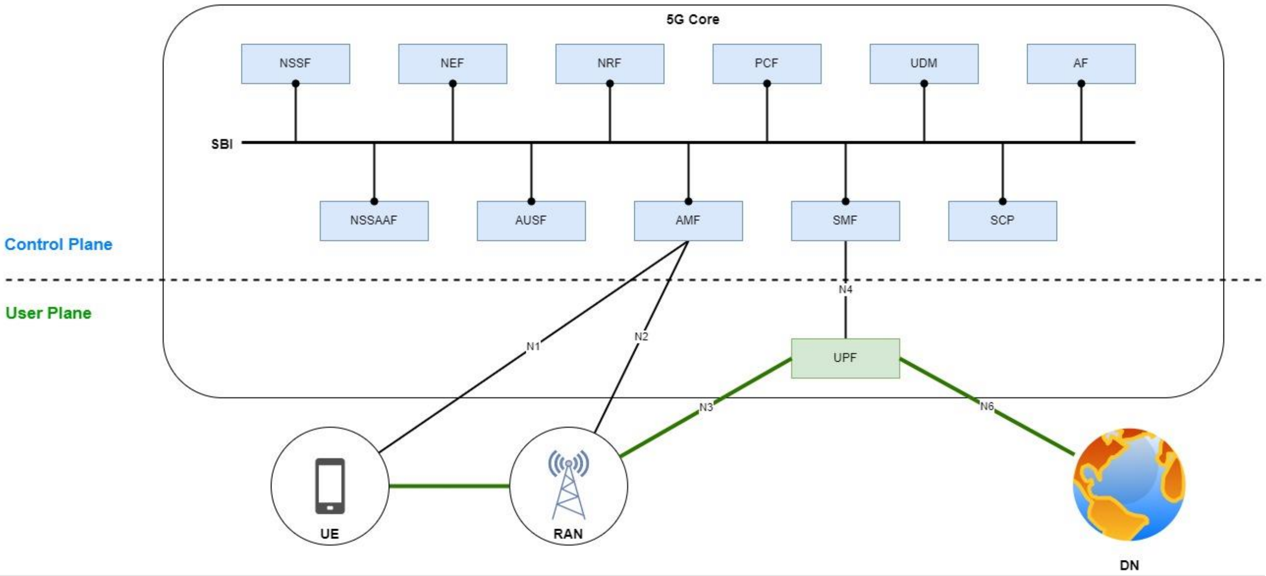
[5] 5G jedro – izvorno "5G core", okrajšava "5GC"



**Slika 2: Pregled 5G sistema**
Vir: [1].

## 2.4 Omrežne funkcije in vmesniki

Podrobnejša arhitektura, z osnovnimi omrežnimi funkcijami[6] jedra (NF), vmesniki med omrežnimi funkcijami, radijskim delom (RAN in UE) ter podatkovnim omrežjem[7] (DN), je prikazana na sliki 3.



**Slika 3: Arhitektura 5G sistema.**

Arhitektura 5G omrežja je specificirana v standardu 3GPP[8]. Jedro 5G omrežja oz. omrežne funkcije lahko v osnovi delimo na kontrolno ravnino[9] in uporabniško ravnino[10].

Kontrolna ravnina je del omrežja, ki upravlja in nadzira pretok podatkov med napravami in storitvami. Nekatere od njenih nalog so:

- **Alokacija virov**: kontrolna ravnina dodeljuje frekvenčne pasove, kanale in časovne reže za prenos podatkov med napravami in baznimi postajami.
- **Upravljanje mobilnosti**: kontrolna ravnina skrbi za nemoteno preklapljanje med različnimi celicami in omrežji, ko se naprave premikajo po prostoru.
- **Upravljanje povezljivosti**: kontrolna ravnina vzpostavlja, vzdržuje in prekinja povezave med napravami in storitvami, glede na njihove potrebe in zahteve.

---

[6] omrežna funkcija – izvorno "Network Function", okrajšava "NF"
[7] podatkovno omrežje - izvorno "data network", okrajšava "DN"
[8] 3GPP - 3rd Generation Partnership Project [2]
[9] kontrolna ravnina – izvorno "control plane"
[10] uporabniška ravnina – izvorno "user plane"

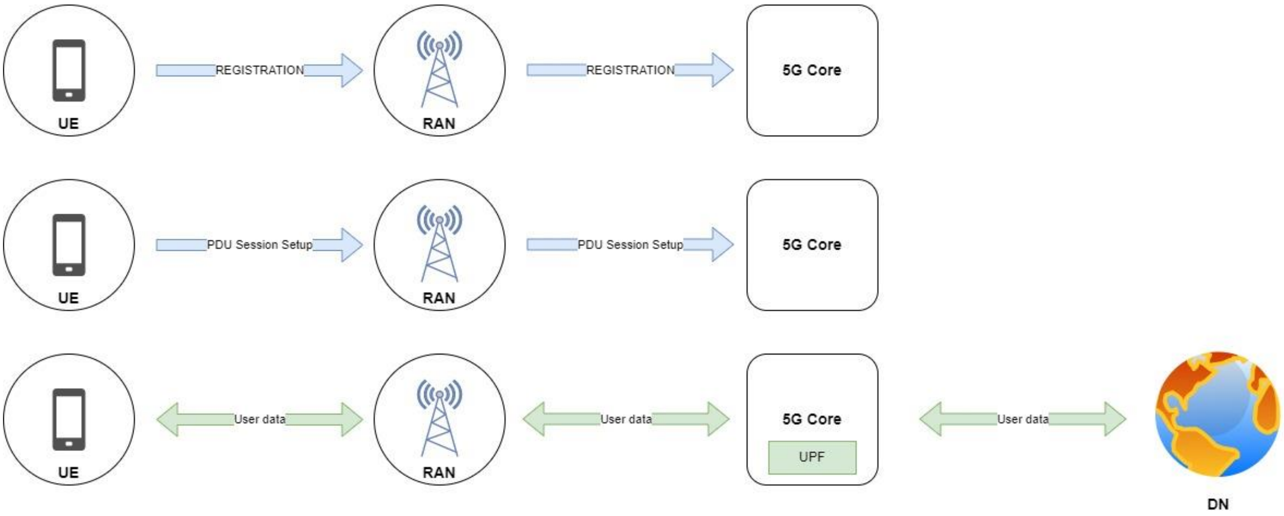
- **Upravljanje identitete**: kontrolna ravnina preverja avtentičnost in avtorizacijo naprav in uporabnikov, ki dostopajo do omrežja in storitev.
- **Upravljanje varnosti**: kontrolna ravnina zagotavlja šifriranje, integriteto in zaupnost podatkov, ki potekajo po omrežju.

Uporabniška ravnina je del omrežja, ki prenaša in usmerja podatke med napravami in storitvami. Nekatere od njenih nalog so:

- **Prenos podatkov:** Omogoča učinkovit prenos podatkov med uporabniškimi napravami in omrežjem. To vključuje prenos glasovnih klicev, videokonferenc, prenosa datotek, internetnega brskanja in drugih oblik podatkovne komunikacije.
- **Usmerjanje in preklapljanje**: Zagotavlja usmerjanje podatkovnega prometa in preklapljanje med omrežnimi vozlišči za optimalno pot prenosa podatkov.
- **Nadzor kakovosti storitev**[11]: Skrbi za nadzor in upravljanje kakovosti storitev. To pomeni, da se podatkovni promet razvrsti in obdela glede na različne parametre, kot so pasovna širina, zakasnitev, izgube paketov in druge metrike, da se zagotovi ustrezna raven storitev in izboljša uporabniška izkušnja.
- **Šifriranje in varnost:** Odgovorna je za šifriranje podatkov in zagotavljanje varnosti med prenosom, da se zaščiti pred nepooblaščenim dostopom.
- **Komunikacija z omrežnimi storitvami**: Sodeluje z omrežnimi funkcijami za avtentikacijo, identifikacijo, upravljanje mobilnosti in nadzor dostopa.

### 2.5 Registracija in vzpostavitev seje

5G kompatibilna naprava z ustrezno SIM kartico, ki želi dostopati do podatkovnega omrežja (DN) preko 5G omrežja, se mora najprej prijaviti v 5G omrežje. Uspešna prijava naprave je predpogoj za vzpostavitev PDU seje[12] do 5G jedra. PDU seja je logična povezava med uporabniško napravo (UE) in 5G jedrom (natančneje med UE in UPF[13]), ki uporabniku omogoča povezavo z podatkovnim omrežjem (DN). Preko PDU seje se omogoči pretok uporabniškega prometa med 5G napravo (UE) in ciljno storitvijo (npr. brskanje po internetu ali pošiljanje senzorskih meritev na IoT platformo). Osnovni koraki vzpostavitve PDU seje so prikazani na sliki 4.



Slika 4: Vzpostavitev PDU seje.

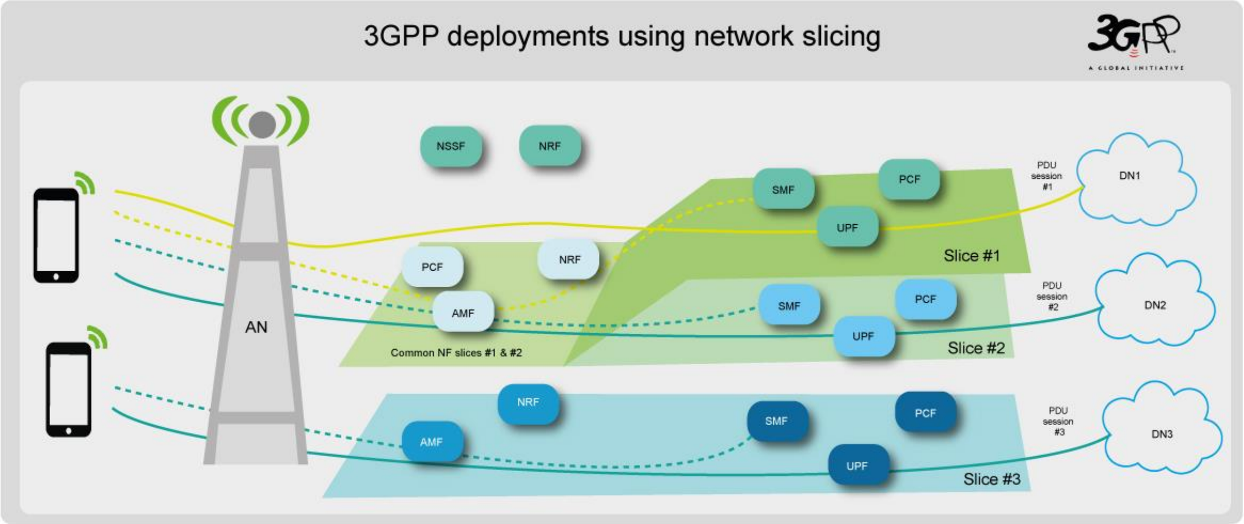
---

[11] kakovost storitve – izvorno "Quality of Service", okrajšava "QoS"
[12] PDU seja – izvorno "Protocol Data Unit Session", okrajšava "PDU session"
[13] UPF – izvorno "User Plane Function"

### 2.6 Segmentacija 5G omrežja in kakovost storitev

Med glavne in kompleksnejše funkcionalnosti 5G omrežja štejemo segmentacijo in upravljanje s kakovostjo storitev.

Segmentacija 5G omrežja[14] je koncept, ki omogoča segmentacijo omrežja 5G na več virtualnih omrežij, imenovanih "rezine"[15]. Vsaka rezina je samostojno virtualno omrežje z lastnimi zmogljivostmi in parametri kakovosti storitve[16] (QoS). Rezine omogočajo prilagojeno in optimizirano povezljivost za različne vrste uporabnikov, aplikacij in industrijskih vertikal. S segmentacijo 5G omrežja se omogoča izpolnjevanje specifičnih zahtev glede zmogljivosti, odzivnega časa, zanesljivosti in varnosti, kar omogoča učinkovito upravljanje različnih uporabniških scenarijev v okviru enega samega omrežja. Rezine se lahko dinamično konfigurira in prilagaja, kar omogoča prilagajanje omrežnih virov glede na spreminjajoče se zahteve in potrebe uporabnikov. Na sliki 5 je primer omrežja s segmentacijo.

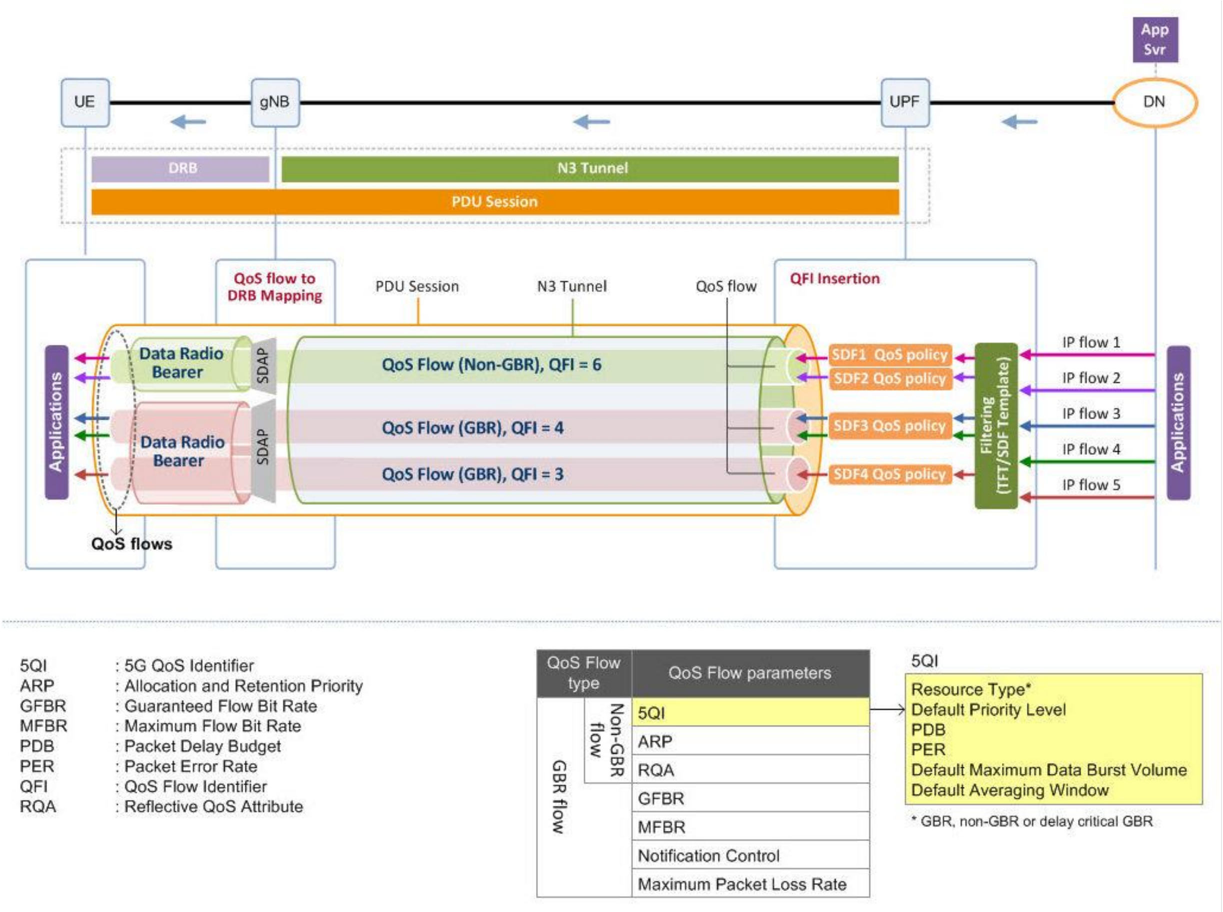


**Slika 5: Segmentacija 5G omrežja**
Vir: [4].

Kakovost storitev (QoS) je koncept, ki se nanaša na sposobnost omrežja 5G, da zagotavlja različne ravni storitev in zadovolji določene zahteve glede kakovosti in učinkovitosti prenosa podatkov (Slika 6). S pomočjo QoS lahko omrežje prilagodi svoje delovanje glede na specifične potrebe aplikacij, uporabnikov in storitev. To omogoča, da se različni vrsti prometa dodelijo prednostne ravni, uporabljajo se mehanizmi za prioritetno obdelavo prometa ter zagotavlja se kakovostno in zanesljivo izkušnjo uporabnikom.

---

[14] segmentacija 5G omrežja - izvorno "5G slicing"
[15] rezina – izvorno "slice"
[16] kakovost storitve – izvorno "Quality of Service", okrajšava "QoS"



5QI : 5G QoS Identifier
ARP : Allocation and Retention Priority
GFBR : Guaranteed Flow Bit Rate
MFBR : Maximum Flow Bit Rate
PDB : Packet Delay Budget
PER : Packet Error Rate
QFI : QoS Flow Identifier
RQA : Reflective QoS Attribute

| QoS Flow type | | QoS Flow parameters |
|---|---|---|
| GBR flow | Non-GBR flow | 5QI |
| | | ARP |
| | | RQA |
| | | GFBR |
| | | MFBR |
| | | Notification Control |
| | | Maximum Packet Loss Rate |

5QI

Resource Type*
Default Priority Level
PDB
PER
Default Maximum Data Burst Volume
Default Averaging Window

* GBR, non-GBR or delay critical GBR

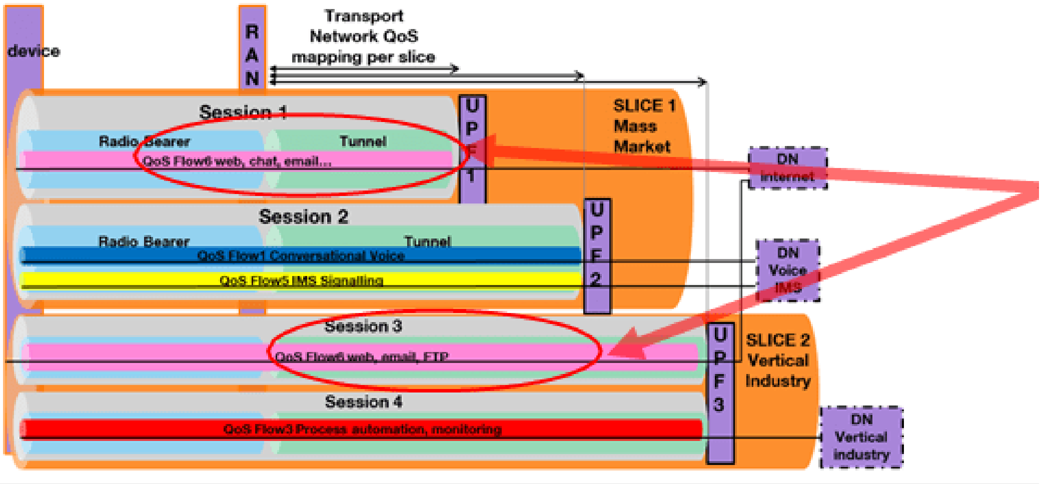
**Slika 6: QoS 5G omrežja**

Vir: [5].

QoS lahko razlikuje med različnimi vrstami prometa, vendar ne more razlikovati in različno obravnavati iste vrste prometa, ki prihaja iz različnih virov. QoS nima sposobnosti za izvedbo izolacije prometa od konca do konca.

Po drugi strani 5G segmentacija skupaj s kvaliteto storitev (QoS), lahko razlikuje med istimi vrstami prometa, ki prihajajo od različnih najemnikov/rezin. Segmentacija omrežja omogoča delovanje več logičnih omrežij na osnovi skupne fizične omrežne infrastrukture, kar zagotavlja izolacijo med njimi.

Na sliki 7 je primer združevanja koncepta segmentacije in kvalitete storitev.

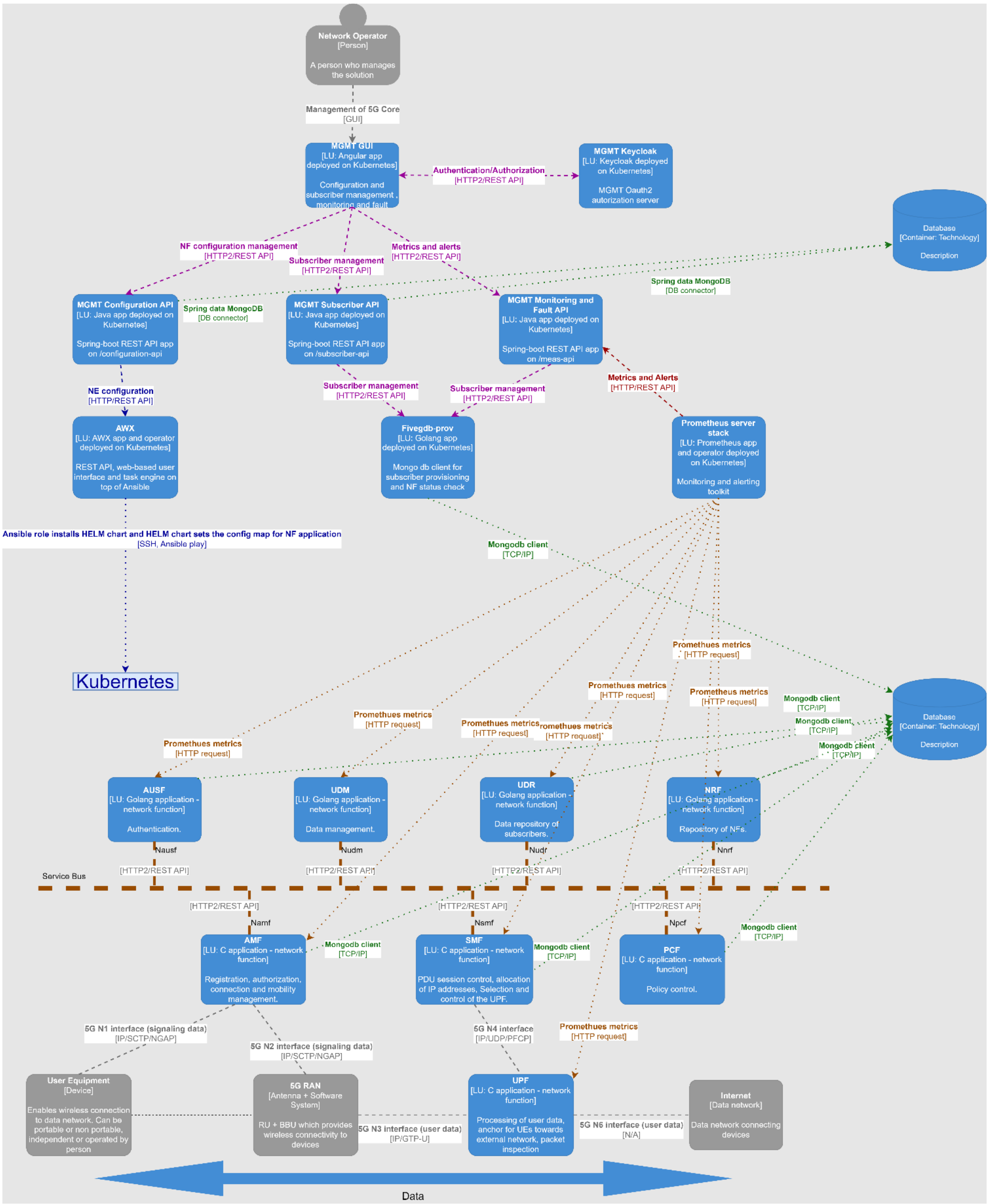


**Slika 7: Združevanje segmentacije in QoS 5G omrežja**

Vir: [5].

## 3 Kontron 5G rešitev

Kontron 5G rešitev je v prvi vrsti predvidena za mobilna privatna omrežja. Glavna naloga naše ekipe je razvoj 5G jedra, katerega postopoma razvijamo in nadgrajujemo (dopolnitve omrežnih funkcij in vpeljava novih) glede na primere uporabe in zahteve s strani končnega kupca. Od zunanjih partnerjev pridobivamo naprave RAN in UE ter medsebojno testiramo delovanje 5G jedra ter delovanje in povezljivost naprav.



**Slika 8: Storitve Kontron 5G rešitve.**

Kontron rešitev je sestavljena iz dveh komponent: 5G jedro ter upravitelj elementov 5G jedra. Vsaka komponenta je sestavljen iz več samostojnih storitev (slika 8).

## 3.1 Upravitelj elementov 5G jedra

Glavne naloge upravitelja elementov 5G jedra so:

- konfiguracija in namestitev omrežnih funkcij jedra,
- posodobitev omrežnih funkcij jedra,
- vpis in administracija naročnikov,
- prikaz stanja 5G omrežja.

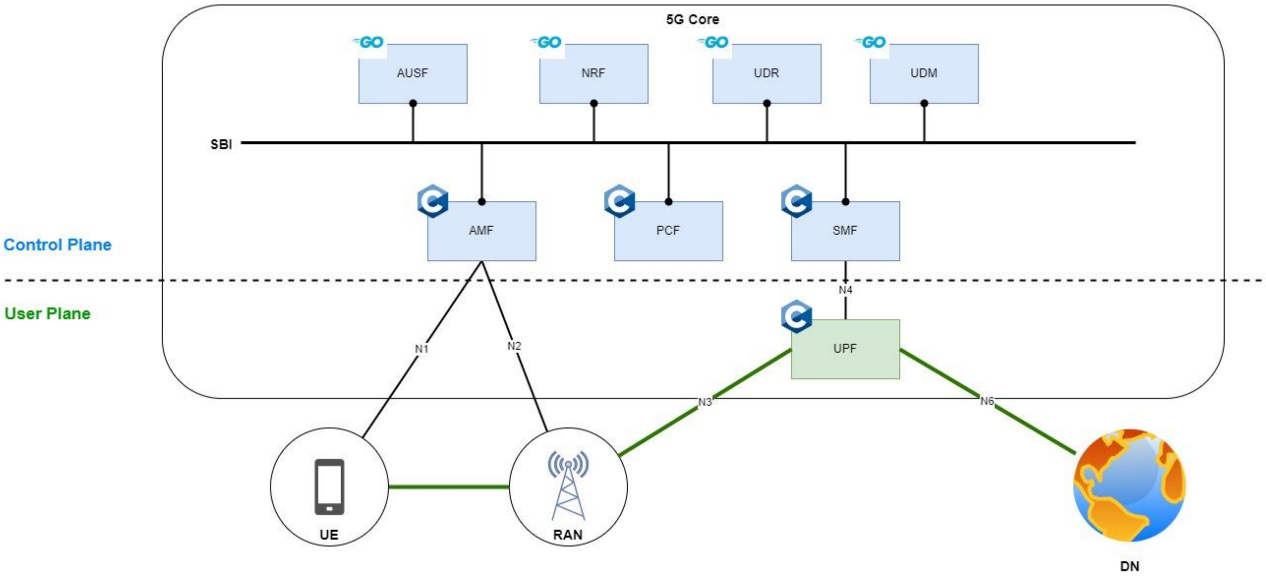
Upravitelj elementov je sestavljen iz zalednih storitev, ki so napisane v programskih jezikih Java (Spring Boot[17]) in Golang, medtem ko je za čelni del sistema[18] uporabljeno JavaScript ogrodje Angular[19].

Za upravljanje identitet in dostopov do upravitelja elementov se uporablja Keycloak[20].

Aplikacije se konfigurirajo in nameščajo s pomočjo orodja Helm[21].

## 3.2 5G jedro

Na sliki 9 so predstavljene omrežne funkcije Kontron 5G jedra. Delimo jih na kontrolno raven, ki skrbi za signalizacijo ter uporabniško raven, preko katere se pretaka podatkovni promet.



**Slika 9: Omrežne funkcije Kontron 5G jedra.**

Standard 3GPP[22] specificira omrežne funkcije kot strežnike z dobro definiranimi REST vmesniki. To nam omogoča implementacijo aplikacij v različnih programskih jezikih. Uporabljamo jezika Golang in C.

Golang omogoča hiter razvoj programske opreme z dobrimi zagotovili za dolgotrajno delovanje aplikacij. Jezik omogoča hitro prevajanje in dober nadzor nad knjižnicami in odvisnostmi. Golang ponuja številna orodja za formatiranje in testiranje kode, kar ekipi omogoča hiter razvoj aplikacij. Rezultat prevajanja so relativno majhni in učinkoviti programi, dovolj zmogljivi za večino primerov uporabe v 5G jedru.

---

[17] https://spring.io/projects/spring-boot

[18] čelni del sistema – izvorno "frontend", okrajšava "FE"

[19] https://angular.io/

[20] https://www.keycloak.org/

[21] https://helm.sh/

[22] https://www.3gpp.org/

Komponente 5G jedra, ki zahtevajo dodatno zmogljivost so napisane v programskem jeziku C. Primer je UPF, ki za hitro procesiranje paketov uporablja knjižnice DPDK[23].

## 3.3 Motivacija za prehod na oblačno zasnovano omrežje

3GPP je standardizacijska organizacija, ki piše standarde za mobilna omrežja. Pri definiciji standardov za 5G omrežje so veliko poudarka namenili vpeljavi storitveno orientirane arhitekture[24] (SBA). Ker želimo narediti 5G rešitev kompatibilno s standardom, je bila že takoj sprejeta odločitev za vpeljavo storitev z vmesniki, kot je predlagano s strani 3GPP.

Jedrne omrežne funkcije med sabo komunicirajo preko REST vmesnikov preko protokola http/2. Najdemo tudi manj razširjene omrežne protokole kot so NGAP, PFCP ter GTP-U. Protokola NGAP in GTP-U se uporabljata za komunikacijo jedra z zunanjimi elementi omrežja (UE, RAN, DN).

Jedro primarno razvijamo za privatna mobilna omrežja, zato velik poudarek namenjamo postopku namestitve in posodobitve jedra na različnih okoljih. Omenjeni postopki morajo biti čim preprostejši za končno stranko. Zagotavljati je potrebno tudi ustrezno spremljanje stanja rešitve in alarmiranje v primeru kritičnih napak.

Dodatno se od 5G jedra zahteva, da bo zagotavljalo visoko razpoložljivost omrežja, prilagodljivost in razširljivost, učinkovito porabo virov in omogočalo hitro dostavo novih rešitev. Na primer v primeru vpeljave segmentacije omrežja (poglavje 2.6), je potrebno zagotavljati dinamično spremembo topologije jedra. Dodajanje, odvzemanje ali spremembo rezin omrežja je potrebno zagotoviti med delovanjem jedra. To pomeni, da je potrebno zagotoviti namestitev ali odstranitev omrežnih funkcij in spremembo konfiguracij omrežnih funkcij brez izpada omrežja.

Glede na raznolikost rešitve in omenjene zahteve, smo se odločili za oblačno zasnovo omrežja. Storitve so pripravljene za zabojnike, ki jih orkestrira Kubernetes[25]. Kubernetes nam zagotavlja orkestracijo, razširljivost, visoko razpoložljivost, prenosljivost in lažje upravljanje konfiguracij. Poleg tega se z vpeljavo zabojnikov poenostavi namestitev rešitve na različna okolja (na strežnike pri stranki, na oblak različnih ponudnikov). Za namestitev in konfiguracijo samih komponent se uporablja orodje Helm.

Avtomatizacija z uporabo Kubernetesa nam omogoča tudi hitrejšo in lažjo vpeljavo različnih DevSecOps cevovodov za gradnjo produkta, nočno testiranje in varnostno skeniranje.

## 3.4 Sistemske komponente

Rešitev smo poskusili približati čim bolj generični namestitvi aplikacij na Kubernetes gručo. Aplikacije so v veliki večini spletni strežniki z REST vmesnikom, kar jih naredi optimalne uporabnike tovrstne platforme.

Nekatere aplikacije 5G jedra potrebujejo dostop do omrežnih vmesnikov, ki jih splošna Kubernetes gruča ne ponuja. Uporabo poljubnih omrežnih vmesnikov omogočimo z uporabo vtičnika Multus.

Kot podatkovno bazo uporabljamo MongoDB. Za postavitev visoko-razpoložljive baze uporabljamo Kubernetes s posebnim vtičnikom za koriščenje shrambe na gostiteljskih sistemih.

### *3.4.1 MongoDB*

Aplikacije 5G jedra shranjujejo svoje podatke v dokumentni bazi MongoDB. Tovrstna baza optimalno teče na namenskih strežnikih izven Kubernetes gruče. Postavitev 5G jedra smo približali minimalnemu številu strežnikov, s katerim lahko zagotavljamo visoko razpoložljivost. MongoDB tako teče znotraj Kubernetes gruče na treh strežnikih in si deli vire z ostalimi aplikacijami. Z uporabo Kubernetes nastavitev "requests" in "limits" strogo nadzorujemo porabo resursov vsake aplikacije. Podatki so shranjeni na gostiteljevem podatkovnem sistemu.

---

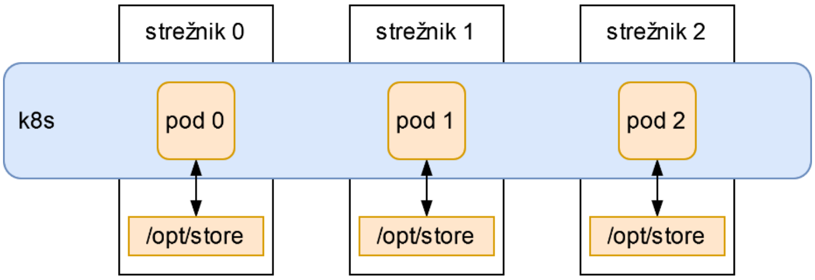
[23] https://www.dpdk.org/
[24] izvorno "Service Based Architecture", okrajšava. "SBA"
[25] https://kubernetes.io/

### *3.4.2 Vtičnik za shrambo*

Nekatere aplikacije v Kubernetes gruči za delovanje potrebujejo obstojno shrambo ("persistent storage"). Najbolj zahtevna izmed teh aplikacij je baza (MongoDB), za katero smo potrebovali hitro in zanesljivo shrambo. Izbrali smo lokalno shrambo na gostiteljskem sistemu, ki jo implementira vtičnik local-path-provisioner[26]. Pri uporabi tovrstne shrambe moramo biti pozorni na njene omejitve:

- V privatnih omrežjih lahko predvidimo zgornjo mejo naročnikov (naprav), kar nam omogoča načrtovanje zgornje meje velikosti shrambe. Zavedamo se, da je povečava shrambe zahteven postopek, ki zahteva zaustavitev sistema.
- Gostiteljski operacijski sistem mora imeti implementirane mehanizme, s katerimi loči podatkovno shrambo od shrambe operacijskega sistema – to smo dosegli z logično particijo (LVM[27]).
- Implementiran mora biti nadzor porabe diska ter primerni alarmi.
- Aplikacije, ki zahtevajo obstojno shrambo so vezane na strežnik, na katerem je bila shramba kreirana.

Vtičnik za shrambo local-path-provisioner Kubernetes gruči ponudi t.i. shrambni razred[28] (SC). Ko aplikacija od shrambnega razreda zahteva obstojni disk[29] (PV), vtičnik na strežnikovem podatkovnem sistemu ustvari nov direktorij ter ga vpne v zabojnik.



**Slika 10: vtičnik za shrambo local-path-provisioner.**

### *3.4.3 Multus*

Omrežni vtičnik Multus[30] omogoča uporabo več različnih omrežij za Kubernetes zabojnike. Multus je meta-vtičnik - to je CNI vtičnik preko katerega lahko uporabljamo več različnih CNI vtičnikov. CNI (Container Network Interface[31]) je standard, ki predpisuje API preko katerega vtičnik Kubernetes gruči ponuja omrežje za zabojnike.

Kot privzeto omrežje za zabojnike uporabljamo Cilium[32]. Poleg privzetega omrežja uporabljamo vtičnik za omrežni most ("bridge") ter vtičnik za gostiteljsko napravo ("host device").

Vtičnik "bridge" zabojniku omrežne funkcije AMF[33] pripne gostiteljev "bridge". Aplikacija AMF na ta način komunicira z zunanjim svetom preko protokola SCTP. Omeniti je treba, da nekateri Kubernetes omrežni vtičniki že podpirajo protokol SCTP, kljub temu smo se zaradi večje stabilnosti in nadzora nad prometom odločili za implementacijo preko vtičnika Multus.

---

[26] https://github.com/rancher/local-path-provisioner
[27] https://www.man7.org/linux/man-pages/man8/lvm.8.html
[28] Shrambni razred – izvorno "StorageClass", okrajšava "SC"
[29] Obstojni disk – izvorno "PersistentVolume", okrajšava "PV"
[30] https://github.com/k8snetworkplumbingwg/multus-cni
[31] https://www.cni.dev/
[32] https://cilium.io/
[33] AMF – Access and Mobility Function

Vtičnik "host-device" zabojniku omrežne funkcije UPF pripne vmesnika za povezavi N3 in N6, preko katerih se pretaka podatkovni promet. To omogoča aplikaciji v UPF zabojniku direkten dostop do fizičnih omrežnih vmesnikov ter optimalni izkoristek zmogljivosti omrežne kartice. Z uporabo vtičnika Multus se izognemo dodatnim obremenitvam omrežnega prometa preko Kubernetes omrežja.

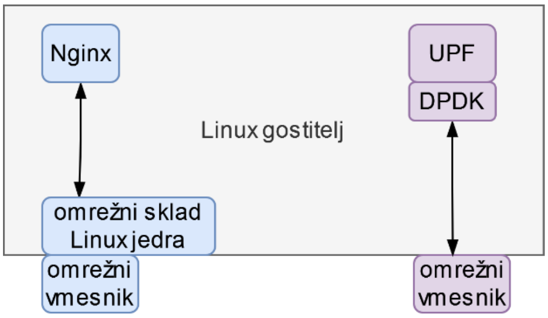
### *3.4.4 DPDK*

Multus nam omogoča direktno uporabo omrežnih vmesnikov brez dodatne obremenitve omrežnega prometa preko Kubernetes omrežja. Omrežno kartico lahko izkoristimo še bolje.

Neposredna uporaba omrežnega vmesnika, ki ga ponuja Linux gostiteljski sistem še vedno predstavlja dodatno obremenitev in sicer procesiranje omrežnih paketov na nivoju Linux jedra. Tej dodatni obremenitvi se lahko izognemo z uporabo knjižnic DPDK.

DPDK[34] (Data Plane Development Kit ali Komplet za razvoj podatkovne ravnine) je skupek knjižnic za hitro procesiranje omrežnih paketov z ozirom na optimalno uporabo razpoložljive strojne opreme. DPDK Linux jedru odvzame nadzor nad omrežno kartico ter ga izroči aplikaciji v uporabniškem prostoru. Uporabniku omogoči razvoj lastnega omrežnega sklada, optimiziranega za specifičen primer uporabe.

V primeru 5G jedra, omrežna funkcija UPF opravlja vlogo omrežnega usmerjevalnika. UPF prejema omrežne pakete s povezav N3 ter jih glede na pravila pridobljena iz nadzorne ravni usmerja na podomrežja preko povezave N6.



**Slika 11: Primerjava omrežnega sklada Linux jedra ter DPDK.**

## 3.5 5G jedro na Kubernetes gruči

Z uporabo Kubernetes gruče smo dosegli poenoten način namestitve, posodobitev ter konfiguracije aplikacij. 5G jedro se namesti z uporabo dveh Helm chartov[35]: Helm chart za namestitev jedrnih funkcij ter Helm chart za namestitev upravitelja elementov. Namestitev predvideva Kubernetes gručo z nekaterimi vnaprej nameščenimi sistemskimi aplikacijami ter s posebnimi delovnimi vozlišči[36] za povezave N2, N3 in N6.

### *3.5.1 Omrežna arhitektura 5G jedra na Kubernetes gruči*

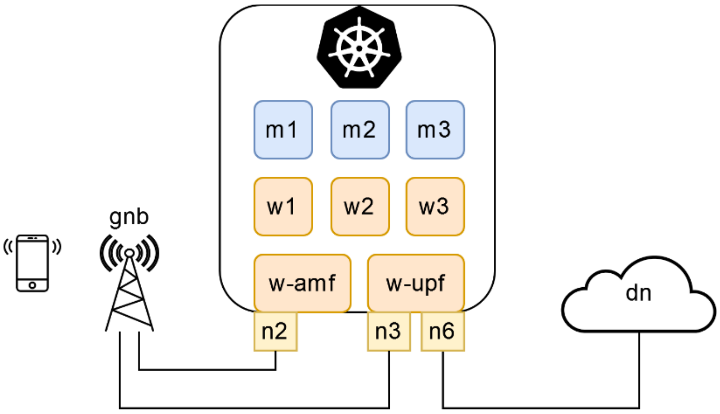
Večina aplikacij 5G rešitve lahko teče na splošnih delovnih vozliščih Kubernetes gruče. Posebni sta omrežni aplikaciji AMF in UPF, ki zahtevata dodatne omrežne vmesnike, ter dodatno konfiguracijo omrežja izven Kubernetes gruče.

Slika 12 prikazuje primer postavitve 5G jedra na Kubernetes gruči s posebnimi delovnimi vozlišči. Kubernetes gruča je zgrajena iz treh vozlišč Kubernetes kontrolne ravni[37] (m1, m2, m3), treh splošnih delovnih vozlišč (w1, w2, w3) ter dveh posebnih delovnih vozlišč (w-amf, w-upf). Aplikacije upravitelja elementov, infrastrukturne

---

[34] https://www.dpdk.org/
[35] https://helm.sh/docs/topics/charts/
[36] delovno vozlišče – izvorno "worker node"
[37] Kubernetes kontrolna raven – izvorno "Kubernetes control plane"

aplikacije ter jedrne aplikacije tečejo na splošnih delovnih vozliščih. Izjema sta aplikaciji AMF in UPF, ki potrebujeta vozlišča s posebnimi omrežnimi vmesniki za komunikacijo z zunanjim svetom.



**Slika 12: Omrežje 5G jedra na Kubernetes gruči.**

Omrežna funkcija AMF je vezana na vozlišče w-amf, ki vsebuje omrežni vmesnik N2. Preko omrežnega vmesnika N2 teče kontrolna signalizacija med AMF in bazno postajo (gNB[38]).

Omrežna funkcija UPF teče na vozlišču w-upf, ki vsebuje omrežna vmesnika N3 ter N6. Omrežni vmesnik N3 predstavlja povezavo za podatkovni promet med UPF in bazno postajo. Omrežni vmesnik N6 predstavlja povezavo za podatkovni promet med UPF in podatkovnim omrežjem (DN).

### 3.5.2 Infrastrukturne aplikacije

Namestitev 5G jedra na Kubernetes gručo zahteva nekatere vnaprej nameščene infrastrukturne aplikacije:

- Nadzorni sklad Prometheus[39]
- Podatkovna baza MongoDB
- Pošiljatelj dnevnikov FluentD[40] ter zbiratelj dnevnikov Loki[41].

Kubernetes gruča mora ponujati naslednje komponente:

- Ingress controller[42]
- Shrambni razred (StorageClass)
- Omrežni vtičnik Multus

Zgoraj naštete komponente najdemo v večini produkcijskih Kubernetes gruč. Največja posebnost sta vtičnik Multus ter dodatni omrežni vmesniki za povezave N2, N3, N6, ki morajo biti na voljo preko posebnih delovnih vozlišč. Zaradi posebnih omrežnih povezav potrebujemo vsaj eno delovno vozlišče, ki nam omogoča dodajanje poljubnih omrežnih vmesnikov. Pomembno se je zavedati, da nekatere Kubernetes distribucije ne predvidevajo posegov v operacijske sisteme vozlišč. Primer je OpenShift, ki za svoja vozlišča uporablja nepremičen ("immutable") operacijski sistem CoreOS.

Vozlišča s posebnimi omrežnimi vmesniki označimo s pomočjo Kubernetes label. Te labele so vhodni podatek za Helm charte pri namestitvi jedrnih aplikacij.

---

[38] https://www.5g-networks.net/5g-technology/5g-terminology-the-gnb/
[39] nadzorni sklad Prometheus, izvorno "Prometheus monitoring stack", https://prometheus.io/
[40] https://www.fluentd.org/
[41] https://grafana.com/oss/loki/
[42] https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/

### *3.5.3 Nadzor aplikacij 5G jedra*

Upravitelj elementov ponuja uporabniku pregledno nadzorno ploščo, ki prikazuje stanje 5G jedra. Nadzorna plošča omogoča namestitve in posodobitve aplikacij 5G jedra. Omogoča uporabniku prijazno konfiguracijo parametrov 5G omrežja, prav tako omogoča upravljanje z uporabniki 5G omrežja.

Upravitelj elementov spremlja stanje 5G aplikacij preko metrik nadzornega sklada Prometheus. Uporabniku ponuja grafični prikaz stanja 5G aplikacij ter stanja resursov platforme. Nadzorna plošča uporabniku pomaga pravilno konfigurirati aplikacije 5G jedra, prav tako ga opozarja v primeru nepravilnega delovanja kakšne od 5G komponent. Podrobnejši pregled metrik je možen na nadzorni plošči Grafana.

## 3.6 DevSecOps postopki

Za interni razvoj 5G aplikacij uporabljamo številna DevSecOps orodja, za grajenje komponent, izvajanje funkcionalnih testov ter, varnostno skeniranje izdanih aplikacij.

GitLab CI/CD, Grype, Trivy, Syft, DefectDojo, Dependency Track, IXIA in UERANSIM nam pomagajo pri izvajanju varnostnih preverjanj in zagotavljanju kakovosti kode. Eno od ključnih načel DevSecOps je vključevanje varnostnih preverjanj v celotni proces CI/CD.

Varnostne smernice je treba upoštevati že v zgodnjih fazah razvojnega cikla. To pomeni, da je potrebno izvajati varnostne preglede zabojnikov, slike in kode že v fazi razvoja kode. To zahteva sodelovanje med varnostnimi strokovnjaki in razvijalci ter vključitev varnostnih orodij v avtomatizirane procese razvoja.

Cevovodi nam omogočajo izvajanje testov ranljivosti na zabojnikih, kateri nam podajo najnovejše varnostne luknje po CVE[43]. Z namenom zagotavljanja celovite varnosti upravljamo z vsemi najdenimi ranljivostmi ter jih spremljamo, dokumentiramo in odpravljamo. Za upravljanje varnostnih pomanjkljivosti in ranljivosti programske opreme uporabljamo orodje Defect Dojo[44].

S pomočjo specializiranih orodij izvajamo statično analizo kode. Cevovod vključuje gradnjo SBOM[45], ki omogoča natančno sledenje in upravljanje vseh komponent našega sistema. S tem se zagotavlja transparentnost in sledljivost vseh elementov.

Za preverjanje delovanja in ustreznosti rešitve, nočni cevovodi redno izvajajo nočne regresijske teste. Poln nabor testov se izvede tudi pred vsako objavo novega programskega paketa. S tem zagotavljamo, da je vsak korak preizkušen, preden se nadaljuje z objavo programskega paketa in s postopki verificiranja. Napake ugotovljene med avtomatičnimi testi se rešuje prioritetno.

CI/CD cevovod izvaja avtomatsko namestitev 5G rešitve z uporabo orodja Helm. To orodje nam zagotavlja ponovljivost namestitve ter sledenje celotni namestitveni konfiguraciji v sistemu za verzioniranje.

Vsebina programskega paketa je definirana v konfiguracijski datoteki, kjer so shranjene verzije posameznih komponent (verzije Helm chartov ter verzije Docker slik). V datoteki se natančno opredeli vse potrebne komponente, njihove verzije ter druge pomembne informacije, ki so bistvene za postopek gradnje paketa. S tem se zagotovi doslednost in pravilno usklajenost med različnimi cevovodi ter omogoča učinkovito izvajanje nadaljnjih korakov. Pred objavo paketa se izvede vsa prej omenjena preverjanja (preverjanje ranljivosti, statična analiza kode, testi).

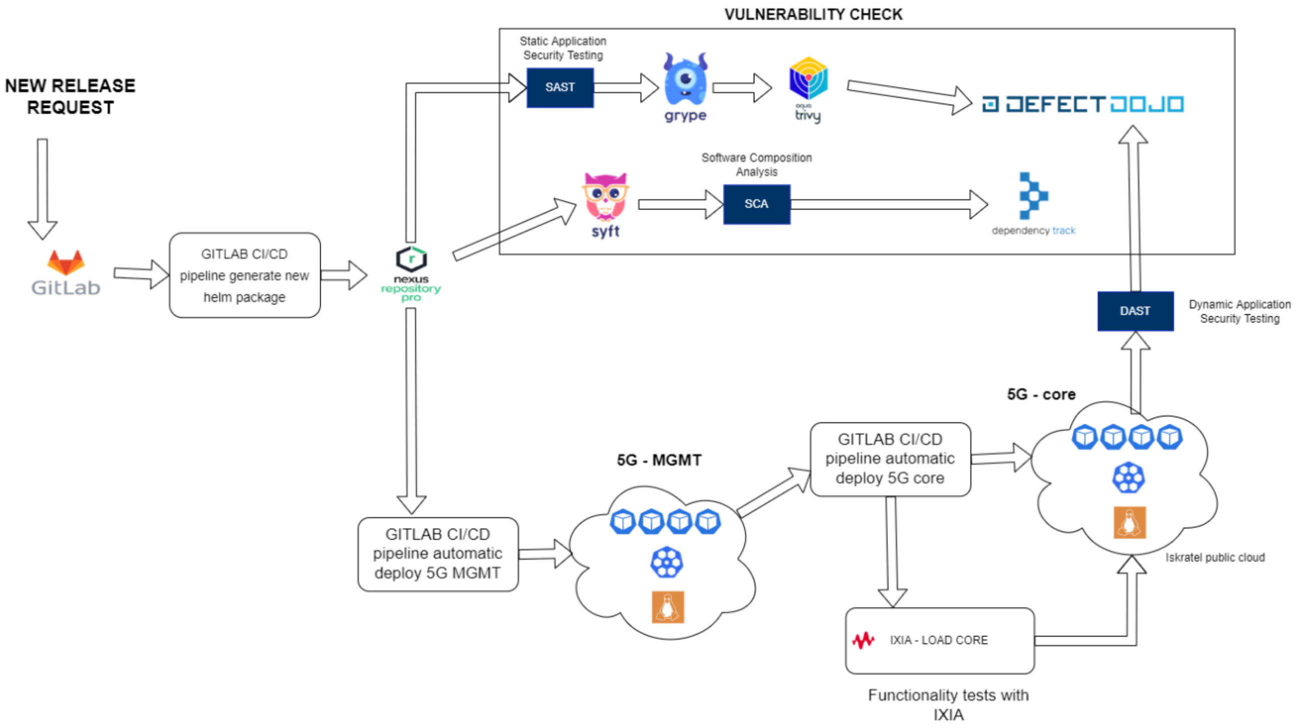
Cevovode se še vedno nadgrajuje in dopolnjuje z namenom avtomatizacije in optimizacije vseh omenjenih korakov ter vpeljavo novih.

---

[43] CVE – izvorno "Common Vulnerabilities and Exposures", https://www.cve.org/
[44] https://www.defectdojo.org/
[45] seznam sestavnih delov – izvorno "Software Bill of Materials", okrajšava "SBOM"

### 3.6.1 Cevovod za gradnjo novega paketa

Na sliki 13 je prikazan cevovod, ki se sproži ob zahtevi za gradnjo nove paketne izdaje 5G rešitve. Osnovni cevovod vključuje tudi klice dodatnih cevovodov za specifične naloge.



**Slika 13: Prikaz cevovoda ob gradnji nove paketne izdaje.**

Glavni koraki cevovoda, ki se izvedejo ob zahtevi za nov paket 5G rešitve:

- Gradnja Docker slik in Helm chartov, shranjevanje rezultatov gradnje v Nexus[46],
- Cevovod za avtomatsko postavitev upravitelja jedra,
- Cevovod za avtomatsko postavitev 5G jedra,
- Avtomatski testi za 5G jedro z orodjem IXIA[47],
- Dinamična analiza kode ter odlaganje poročil na Defect Dojo,
- Generiranje SBOM na vseh zabojnikih in odlaganje poročil na Dependency Track[48],
- Statična analiza kode, ranljivostni testi zabojnikov z orodji Grype in Trivy, odlaganje poročil na Defect Dojo.

---

[46] Repozitorij, https://www.sonatype.com/products/sonatype-nexus-repository

[47] https://support.ixiacom.com/

# 4 Zaključek

5G zagotavlja številne prednosti in izboljšave v primerjavi s prejšnjimi generacijami mobilnih omrežij. Za zagotavljanje večjih hitrosti prenosa, nižjih zakasnitev, večje omrežne zmogljivosti in večje varnosti, igra ključno vlogo jedro 5G omrežja.

V članku je predstavljeno jedro 5G omrežja kot kompleksna rešitev, ki mora omogočati razširljivost, zanesljivost in prilagodljivost za namestitev pri različnih strankah. Omenjene zahteve se rešuje s postavitvijo zabojnikov 5G rešitve v oblačno zasnovano omrežje. Za zagotavljanje večjih hitrosti in manjših zakasnitev uporabniškega prometa se uporabi specifične sistemske komponente (Multus, DPDK).

Zaradi uporabe različnih tehnologij in programskih jezikov je potrebno veliko pozornosti in časa nameniti DevSecOps postopkom. Začeten vložek v gradnjo cevovodov se kmalu povrne v obliki stabilne in zanesljive rešitve. Poleg tega je potrebno nameniti velik poudarek tudi varnostnemu preverjanju rešitve že od zgodnje faze razvoja kode in vse do končne postavitve rešitve.

Ob razvoju 5G rešitve kot oblačno zasnovanega omrežja smo prišli do ugotovitve, da je potrebno za samo obvladovanje namestitev in posodobitve rešitve ter kvalitetnih DevSecOps postopkov nameniti veliko časa in vložka. Poleg tega je ključno tesno sodelovanje med razvojno in DevSecOps ekipo.

# Literatura

[1] https://www.3gpp.org/technologies/5g-system-overview, 3rd Generation Partnership Project (3GPP), obiskano 28.6.2023

[2] 3rd Generation Partnership Project; Technical Specification Group Services and System Aspects; System architecture for the 5G System (5GS); Stage 2 (Release 16)

[3] https://blog.se.com/telecommunications/2022/07/28/the-evolution-of-5g-and-the-backup-power-it-requires/, obiskano 4.7.2023

[4] https://www.3gpp.org/news-events/3gpp-news/sys-architecture, obiskano 10.7.2023

[5] https://devopedia.org/5g-quality-of-service, obiskano 10.7.2023

[6] https://www.redhat.com/en/topics/cloud-native-apps/vnf-and-cnf-whats-the-difference, obiskano 13.7.2023

[7] https://www.analog.com/en/analog-dialogue/articles/massive-mimo-and-beamforming-the-signal-processing-behind-the-5g-buzzwords.html, obiskano 4.7.2023

# Izkušnje pri skaliranju večje rešitve z zabojniki

Andrej Krajnc, Vojko Ambrožič, Bojan Štok

IZUM – Institut informacijskih znanosti, Maribor, Slovenija
andrej.krajnc@izum.si, vojko.ambrozic@izum.si, bojan.stok@izum.si

Sistem COBISS sestavlja več aplikacij in mikrostoritev, javnost pozna predvsem spletno aplikacijo COBISS+. Pred tremi leti smo začeli prenovo aplikacije COBISS3, ki jo knjižničarji uporabljajo za poslovanje knjižnice (zaloga, nabava, izposoja, serijske publikacije, elektronski viri, izpisi). Obstoječa verzija aplikacije je večslojna in uporablja grafični uporabniški vmesnik Java Swing. Nova generacija aplikacije COBISS4 ima spletni uporabniški vmesnik, podpira vse funkcionalnosti, ki jih ima obstoječa aplikacija, pri čemer gre za kompleksno aplikacijo z nekaj 1000 poslovnimi entitetami. Obe aplikaciji uporabljata isti zaledni strežnik, ki teče za vsako od več kot 1400 knjižnic, pri čemer za večje knjižnice teče več instanc aplikacije. Nameščanje in nadgradnja več kot 2000 instanc je v virtualnem okolju zelo zahtevna, zato smo se odločili za uporabo zabojnikov. Analizirali smo dva različna orkestratorja za zabojniško okolje. Odločili smo se za uporabo Docker Swarm, ker je namestitev in vzdrževanje precej bolj enostavno kot pri orkestratorju Kubernetes. Celotni postopek smo avtomatizirali z orodjem Ansible, delovanje aplikacij in mikrostoritev pa nadziramo in spremljamo z orodjem Prometheus. Za izenačevalnik obremenitve smo uporabili Traefik. Izkušnje pri delovanju gruče so pozitivne, prav tako je sistem zelo stabilen in obvladljiv. Načrtujemo, da bomo v prihodnje uporabili zabojnike še pri razvoju drugih aplikacij v sistemu COBISS.

**Ključne besede:**
COBISS
COBISS Lib
zabojniki
Docker Swarm
IT arhitekture

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.6

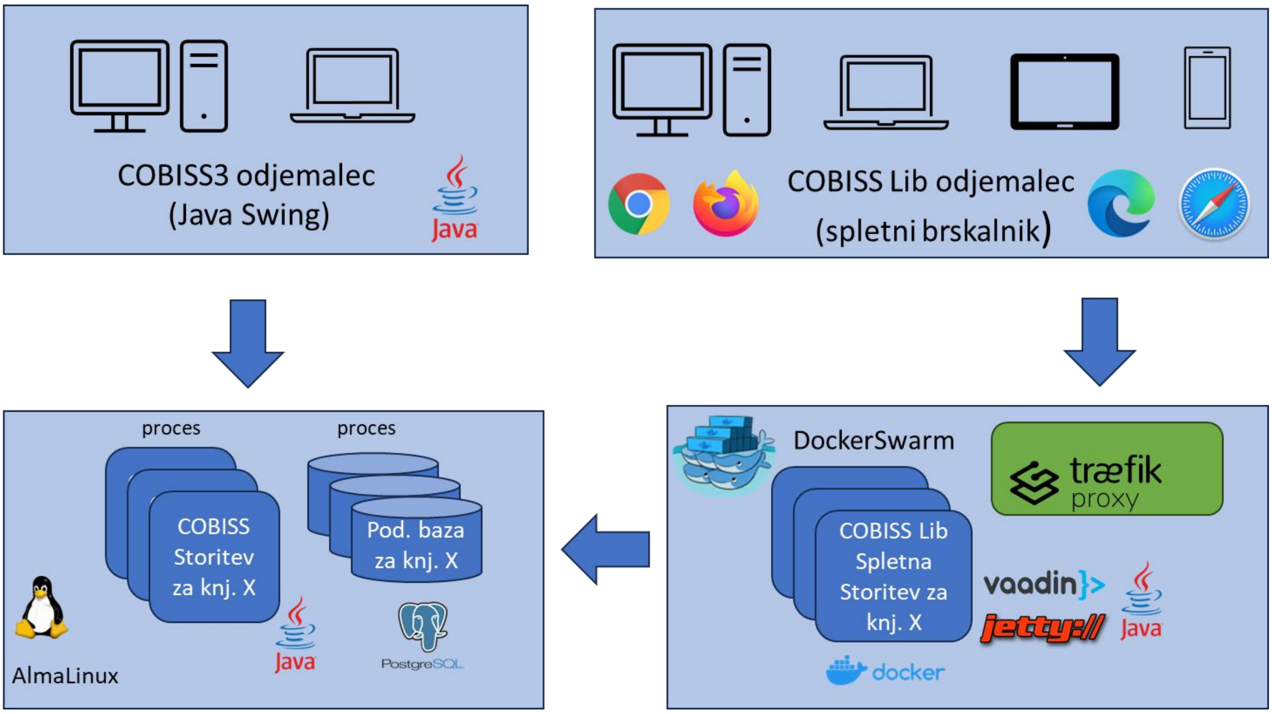
## 1 Uvod

Institut informacijskih znanosti (IZUM) razvija informacijski servis slovenske znanosti, kulture in izobraževanja. Javnosti najbolj poznan je sistem COBISS, ki predstavlja temelj knjižničnega informacijskega sistema Slovenije in nekaterih drugih držav, ki so povezane v mrežo COBISS.net. V mrežo COBISS.net so poleg Slovenije povezane še Srbija, Severna Makedonija, Bosna in Hercegovina, Črna gora, Albanija in Bolgarija.

Sistem COBISS ni le ena sama aplikacija, temveč zaobjema več aplikacij in mikrostoritev. Splošni javnosti sta najbolj znani spletna aplikacija COBISS+ in mobilna aplikacija mCOBISS. Ti aplikaciji uporabnikom knjižnic, ki iščejo relevantne informacije, knjižnično gradivo ali e-vire, omogočata iskanje, rezervacije, podaljšanje roka izposoje itd. Za same knjižnice je najpomembnejša knjižnična aplikacija, ki jo IZUM razvija posebej za knjižnice. COBISS3 je tretja generacija te programske opreme, ki pokrivajo posamezna ključna področja avtomatizacije knjižničnih postopkov. COBISS3 je namizna aplikacija, ki jo knjižničarji uporabljajo za poslovanje knjižnice (zaloga, nabava, izposoja, serijske publikacije, elektronski viri, izpisi). Aplikacija COBISS3 je večslojna, napisana je v programskem jeziku Java oz. s pomočjo nabora orodij za gradnike grafičnega vmesnika za Javo, torej s pomočjo knjižnice Swing.

Nova generacija knjižničnih aplikacij se imenuje COBISS Lib in je na voljo kot spletna aplikacija. Grafični vmesnik v aplikaciji COBISS Lib je narejen s pomočjo ogrodja Vaadin, ki omogoča razvoj spletnih aplikacij z uporabo spletnih komponent [1]. Aplikacija COBISS Lib podpira vse funkcionalnosti, ki jih ima obstoječa aplikacija, pri čemer gre za kompleksno aplikacijo z nekaj 1000 poslovnimi entitetami.

Obe aplikaciji (COBISS3 in COBISS Lib) uporabljata isti zaledni strežnik, ki teče za vsako od več kot 1400 knjižnic, pri čemer za večje knjižnice teče več instanc aplikacije. Nameščanje in nadgradnja več kot 2000 instanc je v virtualnem okolju zelo zahtevna, zato smo se pri COBISS Lib odločili za uporabo zabojnikov.



**Slika 1: Arhitektura knjižničnih aplikacij COBISS3 in COBISS Lib.**

## 2 Uporaba orkestratorjev za zabojniško okolje

### 2.1 Primerjava orkestratorjev Kubernetes in Docker Swarm

Pri uporabi zabojnikov je pomembna izbira orkestratorja za zabojniško okolje. S pomočjo orkestratorja dosežemo visoko razpoložljivost (gruče) in visoko skalabilnost storitev. Analizirali smo dva različna orkestratorja za zabojniško okolje: Kubernetes in Docker Swarm. Odločili smo se za uporabo Docker Swarm, ker je namestitev in vzdrževanje precej bolj enostavno kot pri orkestratorju Kubernetes.

V nadaljevanju je na kratko opisana primerjava orodij Docker Swarm in Kubernetes.

Swarm in Kubernetes sta orodji za orkestracijo zabojnikov, ki pomagata pri upravljanju in razporejanju storitev v oblačnih okoljih. Kubernetes je naprednejši in bolj uveljavljen, toda mi smo se odločili za Swarm, ker je brezplačen, zelo enostaven za vzpostavitev in vzdrževanje. Prav tako ni treba izvesti dodatne namestitve, saj je vključen v samo namestitev Docker. Dodatno še velja omeniti, da je dokumentacija za Swarm odlična. Ocenili smo, da kljub nekaterim pomanjkljivostim še vedno dovolj dobro ustreza našim potrebam.

Kubernetes (tudi k8s) je bolj zrel projekt z večjim in bolj razširjenim ekosistemom. Ker je del odprtokodne platforme (Cloud Native Computing Foundation), ima veliko skupnost razvijalcev in podjetij, ki prispevajo k njenemu razvoju, medtem ko je Swarm samo del ekosistema Docker.

Kubernetes ima bolj kompleksno arhitekturo. Ta temelji na skupini vozlišč, ki tvorijo gručo. Vsako vozlišče ima glavni krmilnik (ang. control plane), ki upravlja in nadzoruje delovanje gruče. Swarm ima bolj preprosto arhitekturo za upravljanje.

Kubernetes ima zmogljivejše orodje za samodejno skaliranje, ki omogoča samodejno prilagajanje števila delujočih instanc aplikacije glede na obremenitev. Swarm prav tako omogoča skaliranje, vendar nima funkcionalnosti za samodejno prilagajanje števila instanc.

Oba sistema uporabljata ukazno vrstico za upravljanje in konfiguracijo storitev, kakor tudi deklarativno konfiguracijo z uporabo datotek manifest oz. stack YAML. Te opisujejo želeno stanje sistema, ki ga sistem skuša zadostiti.

Kubernetes ima širok nabor funkcionalnosti in dodatkov, ki podpirajo bolj napredne funkcionalnosti, kot so podpora za storitve s stanjem, naprednejše upravljanje z diskovnimi kapacitetami itd.

Kubernetes je zgrajen tako, da lahko teče na kateri koli oblačni platformi, medtem ko Docker Swarm deluje samo znotraj izvajalnega okolja Docker.

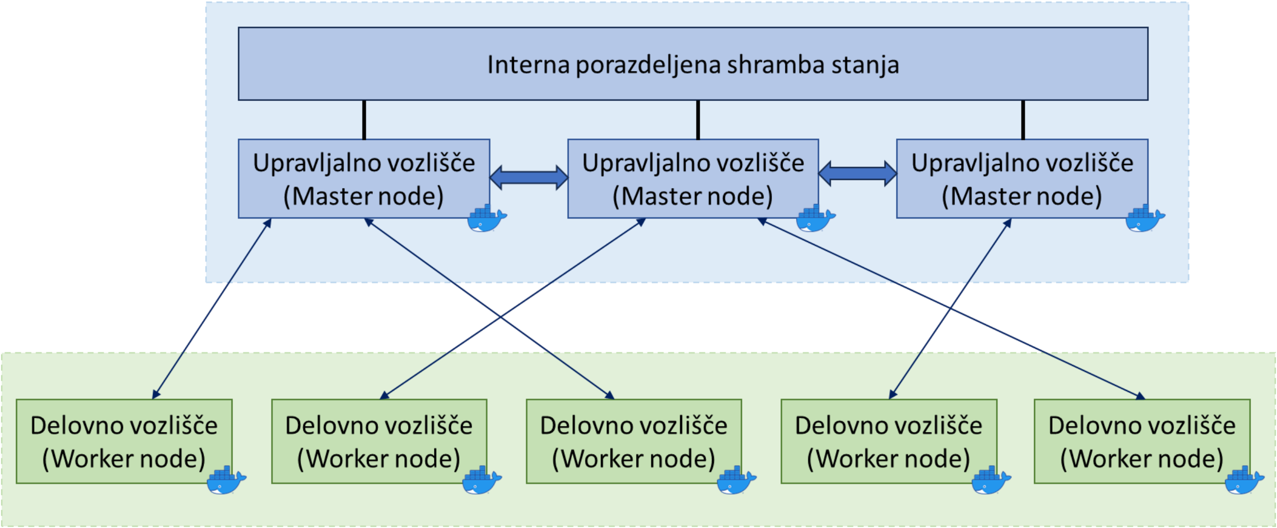
V prihodnosti bomo preverili delovanje naših storitev tudi v okolju Kubernetes. Ker pri nas uporabljamo virtualizacijo vSphere (VMware), bi bila lahko za nas zanimiva Kubernetesova rešitev "vSphere with Tanzu". To je integrirana rešitev, ki združuje virtualizacijsko platformo vSphere s Kubernetesovo platformo Tanzu. Namenjena je zagotavljanju celovitega orodja za upravljanje in delovanje zabojnikov v virtualiziranih okoljih.

### 2.2 Arhitektura Docker Swarm

Docker Swarm temelji na uporabi **rojev** (ang. swarm). En roj je sestavljen iz več Dockerjevih gostiteljev oz. vozlišč, ki tečejo v načinu rojev (ang. swarm mode). Ločimo dve vrsti vozlišč: upravljalna in delovna vozlišča.

**Upravljalna vozlišča** (ang. manager nodes) skrbijo za upravljanje roja (upravljanje Dockerjevih instanc), opravljajo funkcije orkestracije in upravljanja gruče itd. Eno izmed upravljalnih vozlišč je vodja (ang. leader) za opravljanje orkestracijskih nalog. Konsistenca stanja upravljalnih vozlišč se zagotavlja z algoritmom soglasja Raft, kjer se med kandidati izvedejo volitve in se med vozlišči določi vodja, druga vozlišča pa postanejo sledilci (ang. followers).

**Delovna vozlišča** (ang. worker nodes) dobivajo in izvajajo naloge (ang. task), posredovane od upravljalnih vozlišč. Privzeto tudi upravljalna vozlišča opravljajo funkcijo delovnih vozlišč, lahko pa so zgolj upravljalna vozlišča. Na vsakem delovnem vozlišču je agent, ki upravljalnim vozliščem poroča o opravilih, ki so bila dodeljena delovnemu vozlišču. Tako lahko upravljalno vozlišče vzdržuje zaželeno stanje posameznega delovnega vozlišča.

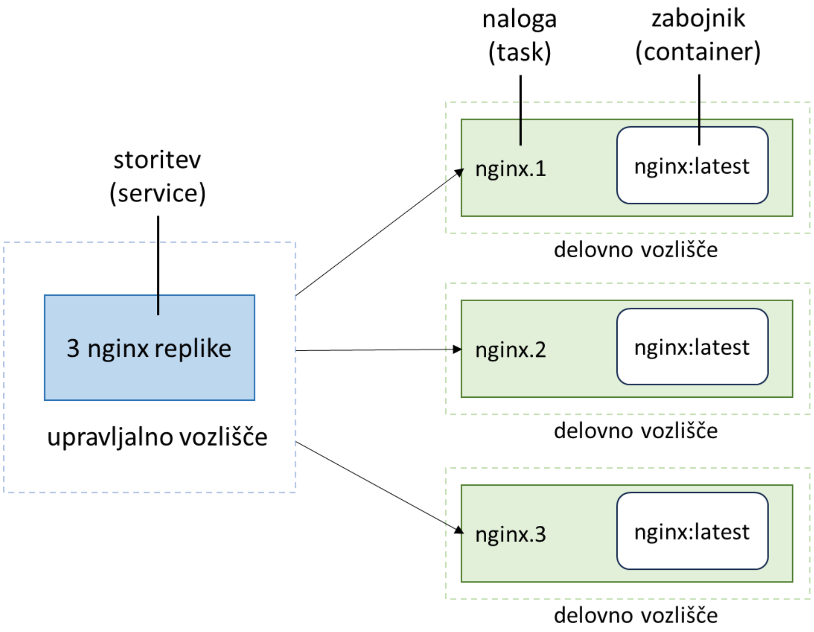


**Slika 2: Docker Swarm vozlišča.**

**Storitev** (ang. service) je definicija nalog, ki naj se izvedejo na upravljalnem ali delovnem vozlišču. Ko se storitev ustvari, se določi, katera zabojniška slika (ang. container image) naj se uporabi in kateri ukazi naj se izvedejo znotraj delujočih zabojnikov.

V primeru repliciranih storitev upravljalno vozlišče distribuira določeno število repliciranih nalog med delovna vozlišča.

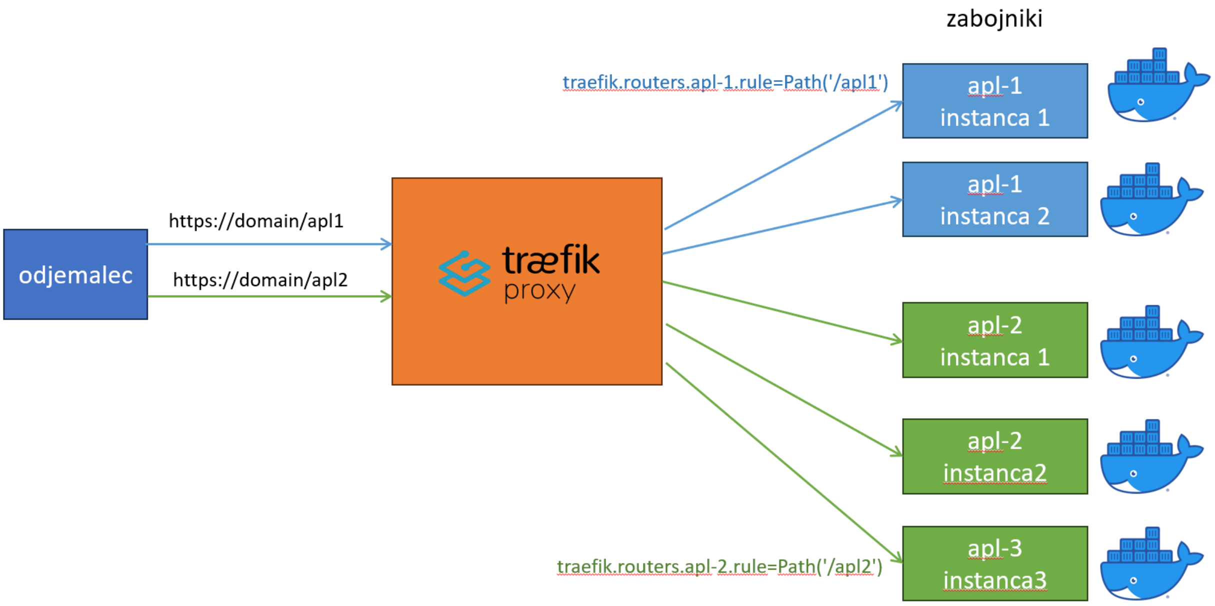
**Naloga** (ang. task) zaobjema Dockerjev zabojnik in vse ukaze, ki naj se izvedejo znotraj zabojnika. Naloga je atomarna enota v roju. Upravljalna vozlišča dodeljujejo naloge delovnim vozliščem glede na število replik, ki naj bi jih neka storitev imela. Ko je naloga dodeljena nekemu vozlišču, je ne moremo več prestaviti v drugo vozlišče. Naloga se na dodeljenem vozlišču izvede ali pa pade.



**Slika 3: Storitve in naloge Docker Swarm.**

## 3 Traefik

Za posrednika med zunanjimi odjemalci in našimi storitvami (dinamični reverse proxy) smo uporabili odprtokodni program Traefik, ki smo ga namestili na eno izmed upravljalnih vozlišč gruče. Traefik nadzira registracijo zabojnikov v mrežo ter nato na osnovi URL-naslova preusmeri zahteve na ustrezno instanco zabojnika naše storitve.



**Slika 4: Traefik v okolju Docker.**

Ker storitev hrani kontekst seje, mora Traefik preusmeriti odjemalčevo zahtevo vedno na isto instanco zabojnika (ang. sticky mode). Navodila za preslikavo zahtev dobi Traefik preko znač (ang. label) v opisu storitve (stack-datoteka).

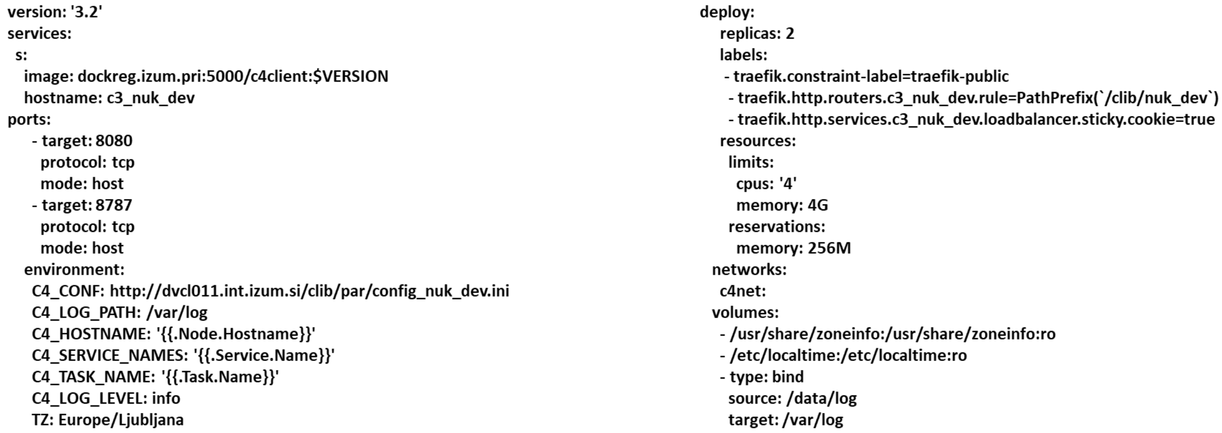
Za visoko razpoložljivost sistema skrbi zunanji izenačevalnik obremenitve (load balancer), ki zahteve na zunanji mreži (DMZ) preusmeri na storitev Traefik na enem izmed upravljalnih vozlišč v notranjem omrežju (intranet).

## 4 Izgradnja in upravljanje Docker okolja

Za inicializacijo roja Swarm, dodajanje oz. odvzemanje upravljalnih in delovnih vozlišč, nameščanje Dockerjevih strežnikov in samih virtualnih strežnikov smo uporabili odprtokodno orodje Ansible, ki je namenjeno upravljanju konfiguracij, namestitvam in avtomatizaciji nalog na večjem številu strežnikov. Naloge opišemo v tako imenovanih Ansible-skriptah (Ansible Playbook).

Za upravljanje in nadzor nad Ansible-skriptami smo uporabili odprtokodno spletno platformo AWX, ki je brezplačna različica komercialne platforme Tower podjetja Red Hat. Na osnovi teh orodij je izgradnja roja Swarm rešljiva znotraj nekaj minut.

Storitve v roju Swarm so opisane v tako imenovanih stack-datotekah (format YAML). V opisu so navedeni računalniški viri, ki jih storitev potrebuje. Navedeni so parametri in razne nastavitve za storitev, vrata TCP/IP, ki jih storitev izpostavlja in marsikaj drugega. V IZUM-u imamo razvito orodje »Monitor Manager« za izgradnjo meta modela infrastrukture, strežnikov, aplikacijskih strežnikov, baz podatkov, mikrostoritev in modela knjižnic. Na osnovi tega modela se samodejno kreirajo stack-datoteke za storitve in se hkrati namestijo na upravljalna vozlišča roja. Vsaka sprememba na knjižnicah oz. infrastrukturi je takoj zaznana v storitvah roja Swarm.

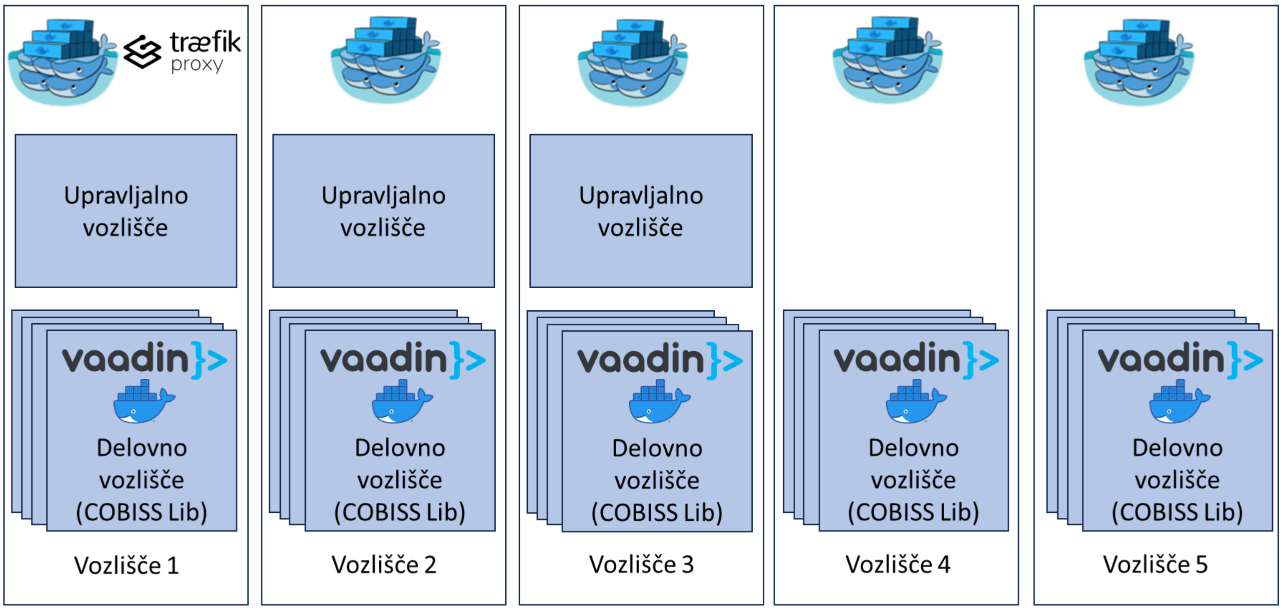


```
version: '3.2'
services:
 s:
   image: dockreg.izum.pri:5000/c4client:$VERSION
   hostname: c3_nuk_dev
ports:
    - target: 8080
      protocol: tcp
      mode: host
    - target: 8787
      protocol: tcp
      mode: host
   environment:
    C4_CONF: http://dvcl011.int.izum.si/clib/par/config_nuk_dev.ini
    C4_LOG_PATH: /var/log
    C4_HOSTNAME: '{{.Node.Hostname}}'
    C4_SERVICE_NAMES: '{{.Service.Name}}'
    C4_TASK_NAME: '{{.Task.Name}}'
    C4_LOG_LEVEL: info
    TZ: Europe/Ljubljana
```

```
deploy:
    replicas: 2
    labels:
     - traefik.constraint-label=traefik-public
      - traefik.http.routers.c3_nuk_dev.rule=PathPrefix(`/clib/nuk_dev`)
      - traefik.http.services.c3_nuk_dev.loadbalancer.sticky.cookie=true
    resources:
      limits:
        cpus: '4'
        memory: 4G
      reservations:
        memory: 256M
   networks:
    c4net:
   volumes:
    - /usr/share/zoneinfo:/usr/share/zoneinfo:ro
    - /etc/localtime:/etc/localtime:ro
    - type: bind
      source: /data/log
      target: /var/log
```

**Slika 5: Izsek opisa Dockerjeve storitve za COBISS Lib.**

Pri razvoju programske opreme uporabljamo tudi odprtokodno orodje Jenkins za neprekinjeno integracijo (Continuous Integration – CI) in neprekinjeno dostavo (Continuous Delivery – CD). Uporablja se za avtomatizacijo gradnje, testiranja in dostave programske opreme, kar omogoča hitro in zanesljivo uvajanje sprememb v aplikaciji. V razvojnem okolju nam orodje Jenkins samodejno dvakrat dnevno generira Dockerjeve posnetke (ang. image) ter jih namesti na naš zasebni Dockerjev repozitorij in nato izvede nadgradnjo naših storitev v roju Swarm. Nadgradnja storitev v produkcijskem okolju ni samodejna in se izvede le na zahtevo.

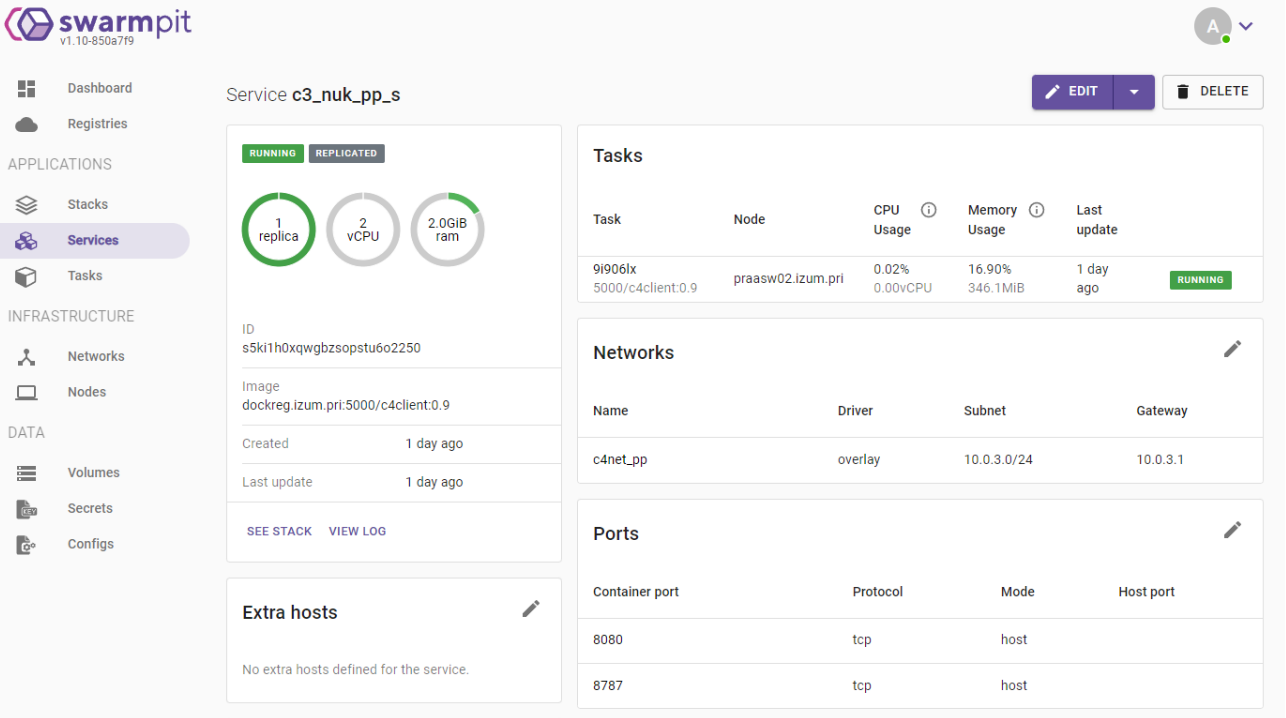


**Slika 6: Okolje COBISS Lib Docker.**

# 5 Podporna orodja v Dockerjevem okolju

## 5.1 Nadzor Dockerjevih storitev

Za nadzor nad Dockerjevimi storitvami lahko uporabljamo Dockerjevo konzolo in odprtokodno platformo za upravljanje roja Swarmpit, ki je nameščena na enem izmed upravljalnih vozlišč roja.



Service **c3_nuk_pp_s**

ID
s5ki1h0xqwgbzsopstu6o2250

Image
dockreg.izum.pri:5000/c4client:0.9

Created 1 day ago

Last update 1 day ago

Extra hosts

No extra hosts defined for the service.

Tasks

| Task | Node | CPU Usage | Memory Usage | Last update | |
|---|---|---|---|---|---|
| 9i906lx<br>5000/c4client:0.9 | praasw02.izum.pri | 0.02%<br>0.00vCPU | 16.90%<br>346.1MiB | 1 day ago | RUNNING |

Networks

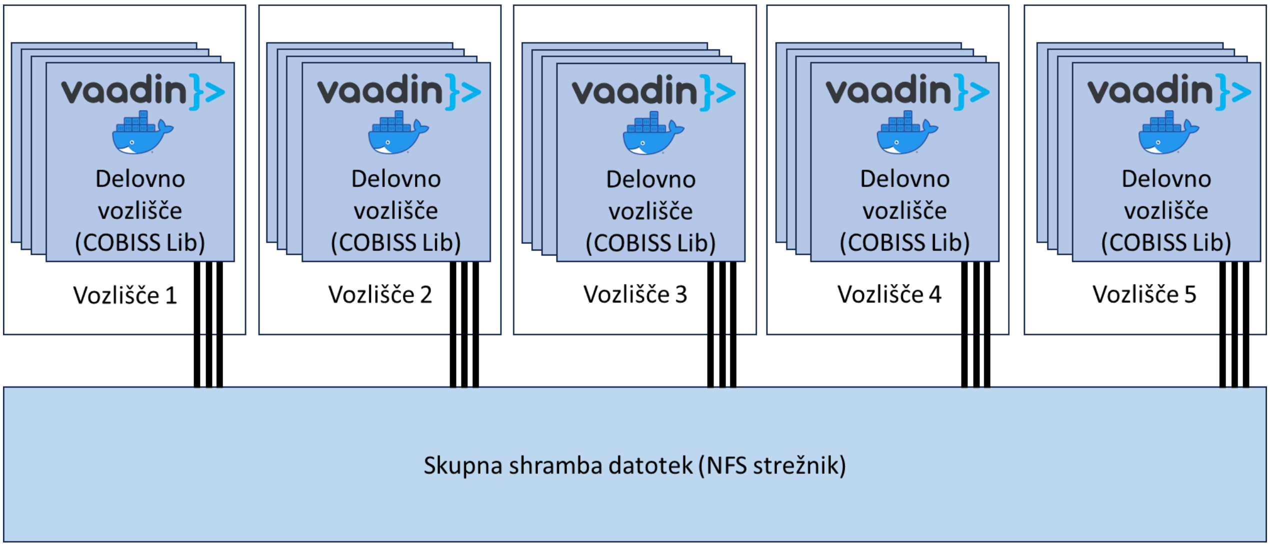
| Name | Driver | Subnet | Gateway |
|---|---|---|---|
| c4net_pp | overlay | 10.0.3.0/24 | 10.0.3.1 |

Ports

| Container port | Protocol | Mode | Host port |
|---|---|---|---|
| 8080 | tcp | host | |
| 8787 | tcp | host | |

**Slika 7: Primer zaslonske slike orodja Swarmpit.**

Dnevniške (ang. log) datoteke spremljamo prek konzole, zbiramo jih pa tudi na skupnem NFS-strežniku, kjer izvajamo analize teh podatkov. V prihodnosti načrtujemo namestitev vtičnika za odprtokodno orodje Elasticsearch, ki je namenjeno indeksiranju, shranjevanju in iskanju velikih količin strukturiranih in nestrukturiranih podatkov v realnem času.



**Slika 8: Skupni NFS-strežnik.**

Virtualne strežnike nadzorujemo tudi z odprtokodnim orodjem Icinga, ki omogoča odkrivanje težav, generiranje opozoril in spremljanje zmogljivosti.

### 5.2 Zbiranje in spremljanje metrik v Dockerjevem okolju

Za zbiranje in spremljanje metrik našega okolja uporabljamo odprtokodne storitve Swarmprom, ki združujejo orodja odprtokodnih projektov Prometheus in Grafana. Prometheus je sistem za spremljanje in zbiranje metrik, ki se osredotoča na zbiranje časovno vrstičnih podatkov. Spremlja zdravstveno stanje, zmogljivost in druge metrike iz različnih virov v realnem času. Metrike se shranjujejo lokalno in omogočajo analizo, vizualizacijo in generiranje opozoril prek orodja Alert manager. Grafana je platforma za vizualizacijo metrik in analizo podatkov. Omogoča ustvarjanje prilagojenih grafov, nadzornih plošč in poročil iz metrik, ki jih zbere Prometheus.

### 5.3 Diagnostična orodja v Dockerjevem okolju

V primeru težav z našimi storitvami lahko uporabimo bolj ali manj vsa diagnostična orodja, ki smo jih vajeni uporabljati v klasičnih okoljih. Med drugim si pomagamo z različnimi javanskimi orodji, kot so jmap, jstack itd.

## 6 Zaključek

Praktične izkušnje pri delovanju roja Docker Swarm so zelo pozitivne, prav tako je sam sistem zelo stabilen in obvladljiv. Čeprav je Docker Swarm manj zmogljiv orkestrator kot Kubernetes, se je izkazal kot primerna izbira, saj vsebuje vse, kar potrebujemo pri aplikaciji COBISS Lib.

Če bi se slučajno pojavile težave z obstoječo gručo, lahko zelo hitro (v nekaj minutah) namestimo novo gručo Docker Swarm, saj je sam postopek namestitve avtomatiziran.

Načrtujemo, da bomo v prihodnje uporabili zabojnike še pri razvoju novih aplikacij v sistemu COBISS in da bomo obstoječe spletne aplikacije in mikrostoritve, ki zdaj tečejo v navideznih strežnikih, preselili v zabojnike. Zabojnikov ta trenutek ne bomo uporabili za shrambo podatkovnih baz, kot so PostgreSQL, Apache Solr, ElasticSearch itd. Za te storitve bomo še naprej uporabljali virtualne strežnike.

V prihodnosti bomo sistem NFS, ki ga zdaj uporabljamo za dnevniške datoteke, nadomestili s porazdeljenimi diski Ceph in vpeljavo shrambe S3.

## Literatura

[1] KRAJNC Andrej, PULKO Jani, KOROŠEC Andrej, ŠTOK Bojan, "Razvoj spletnih aplikacij z uporabo spletnih komponent", Sodobne informacijske tehnologije in storitve OTS 2022: Zbornik petindvajsete konference, Maribor, Fakulteta za elektrotehniko, računalništvo in informatiko, Inštitut za informatiko, 42-54.

[2] Kubernetes, https://kubernetes.io/, obiskano 13.7.2023

[3] Docker Swarm, https://docs.docker.com/engine/swarm/, obiskano 13.7.2023

[4] Traefik, https://traefik.io/traefik/, obiskano 13.7.2023

[5] Ansible, https://www.ansible.com/, obiskano 13.7.2023

# Monolitne rešitve za mikrostoritvene izzive

Miroslav Beranič

Bintegra d.o.o., Maribor, Slovenija
miroslav@beranic.si

V članku bom raziskal Quarkus, ogrodje za razvoj Java aplikacij. Z optimiziranimi lastnostmi delovanja ogrodje primarno cilja na razvoj Java aplikacij v oblaku. Quarkus se osredotoča na hitrost, majhen odtis pomnilnika in nizko latenco; sebe opisuje kot "supersonic subatomic Java". Ena izmed ključnih sestavin Quarkusa je GraalVM, ki omogoča hitrejši čas zagona aplikacij in učinkovito porabo virov s pretvorbo Java kode v native obliko, brez vmesnega Java VM. To je še posebej koristno za mikrostoritvene arhitekture. V članku bom pregledal tudi konkurenčne rešitve ter primerjal prednosti Quarkusa v primerjavi z drugimi razvojnimi ogrodji za oblak. Predstavil bom preprost primer uporabe Quarkusa, da bo bralec dobil vpogled v način razvoja in delovanje. Kot praktičen primer bo opisan tudi način objave aplikacije v sistemu za upravljanje z mikrostoritvam. S celovito analizo Quarkusa in tehnologije GraalVM bo članek bralcu omogočil boljše razumevanje in uporabno ter izkoristek teh inovativnih rešitev za razvoj Java aplikacij v oblaku. Članek se lahko uporabi kot vodnik: kako lahko Quarkus izboljša izkušnje pri razvoju sodobnih oblačnih storitev in aplikacij.

**Ključne besede:**

Quarkus

Java

GraalVM

mikrostoritve

Kubernetes

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.7

# 1 Uvod

Dobrodošel v svetu Quarkusa - inovativnega ogrodja za razvoj Java aplikacij v oblaku [1]. Quarkus je bil zasnovan, da odpravi izzive tradicionalnih pristopov razvoja in zagona aplikacij in ponudi razvijalcem visoko zmogljive rešitve. S svojo osredotočenostjo na hitrost, majhen odtis pomnilnika in nizko latentnost, Quarkus omogoča razvoj izjemno odzivnih aplikacij.

V tem članku bom raziskal bistvo Quarkusa, kako deluje in zakaj je postal priljubljen med razvijalci. Osvetlil bom tehnologijo GraalVM, ki prispeva k izjemno hitremu zagonu aplikacij. Poleg tega bom analiziral konkurenčne rešitve na trgu, da boš razumel, kako se Quarkus primerja z njimi in kakšne so prednosti ter slabosti. Pogledal bom tudi, kaj je novega v verziji 3 in v čem se razlikuje s predhodno iteracijo.

Za lažje razumevanje bom predstavil praktičen primer uporabe, ki bo razkril, kako Quarkus izboljša učinkovitost in odzivnost aplikacij. S tem boš dobil vpogled v to, kako enostavno je razvijati aplikacije z Quarkusom in kako ta tehnologija prinaša konkurenčno prednost. Predstavljeno aplikacijo bom zagnal preko aplikacije za upravljanje z mikrostoritvami.

Brez komercialnih buzzwordov in v tehničnem kontekstu bom raziskal vse vidike Quarkusa, kar ti bo pomagalo bolje razumeti njegove koristi in možnosti za razvoj Java aplikacij v oblaku.

# 2 Quarkus

## 2.1 Zgodovina

Ogrodje Quarkus ima objavljeno prvo verzijo, 0.0.1, dne 12.12.2018 [3], prvi zapis v zgodovini izvorne kode je bil narejen dne 22.6.2018 [4]. Časovno to pomeni, da je ogrodje relativno mlado in hkrati hitro rastoče in razvijajoče, če vemo, da je trenutna verzija 3.3.2 z dne 21.7.2023 [5]. Verzijo 1.0 ( 1.0.0.Final ) je doseglo dne 25.11.2019, verzijo 2.0 ( 2.0.0.Final ) dne 30.6.2021 in verzijo 3.0 ( 3.0.0.Final ) dne 26.4.2023.

**Tabela 1: Krajši pregled zgodovine verzij.**

| Verzija | Datum | Glavne lastnosti |
|---|---|---|
| 0.0.1 | 12.12.2018 | Prva verzija, praktično skelet projekta in osnovni koncepti. |
| 0.2.0 | 19.12.2018 | Izboljšana podpora za *Apache Maven*, uporablja se *fedora-minimal* za zabojnik, podpora za referenciranje Java razredov preko imen. |
| 0.18.0 | 27.6.2019 | GraalVM 19, native image je ločen paket izven GraalVM |
| 0.20.0 | 31.7.2019 | Dodan *quarkus-jackson*, REST client privzeto uporablj *RESTEasy*, premik razširitev za *Apache Camel* v ločen projekt. |
| 1.0.0 | 25.11.2019 | Stabilizacija in uporaba GraalVM 19.2.1, uporaba Quarkus Security 1.0.0. |
| 1.1.0 | 17.12.2019 | Ločen build za native-image, uvedba YAML za nastavitve, uporaba GraalVM 19.3, zagon samostojne RESTeasy aplikacije v glavni niti. |
| 1.2.0 | 23.1.2020 | Razširitev za zalogo ( cache ), Hibernate ORM |
| 1.3.0 | 13.3.2020 | Nov classloader in podpora za GraalVM 20.0.0, ena Vert.x instanca, dopolnjena podpora za deploy v OpenShift implementacija Eclipse MicroProfile specifikacije 3.3. |
| 2.0.0 | 30.6.2021 | Minimalno Java 11, uporaba GraalVM 21.1, Vert.x 4 ter implementacija MicroProfile 4, Hibernate 5.5, MongoDB in Panache s podporo za transakcije. |
| 2.1.0 | 29.7.2021 | Podpora Dev Services za Keycloak, Reactive SQL Server, Kotlin 1.5 |
| 2.2.1 | 31.8.2021 | Podpora za GraalVM 21.2 |
| 2.3.0 | 6.10.2021 | Podpora za Jakarta EL 3 |

| Verzija | Datum | Glavne lastnosti |
|---|---|---|
| 3.0.0 | 26.4.2023 | Podpora za Hibernate ORM 6, Hibernate Reactive 2, izboljšan Dev UI, Jakarta EE 10, uporaba OpenTelemetry |
| 3.1.0 | 31.5.2023 | Podpora za GraalVM 22.3.2, Kotlin 1.8.21 |
| 3.2.0 | 5.7.2023 | Dodana analitika v korake izgradnje. |

Vir: [2].

### 2.1.1 GraalVM

GraalVM je visoko zmogljiva in univerzalna virtualna strojna oprema ( VM ) ter razvojno orodje, ki ga je razvilo podjetje Oracle [5]. Omogoča izvajanje različnih programskih jezikov, kot so Java, JavaScript, Python, Ruby, R, in drugi, na istem virtualnem stroju. To je edinstvena značilnost GraalVM, saj večina drugih VM-jev podpira izvajanje samo enega programskega jezika.

GraalVM se ponaša z izjemno visoko zmogljivostjo in učinkovitostjo, kar je doseženo s tehnologijo Just-In-Time (JIT) prevajanja, ki pretvori visokonivojski kode v izvorno kodo, ki jo lahko izvaja procesor. Ta tehnologija omogoča hitrejše zaganjanje in izvajanje aplikacij, saj se izvorna koda prilagaja konkretnemu okolju in arhitekturi sistema.

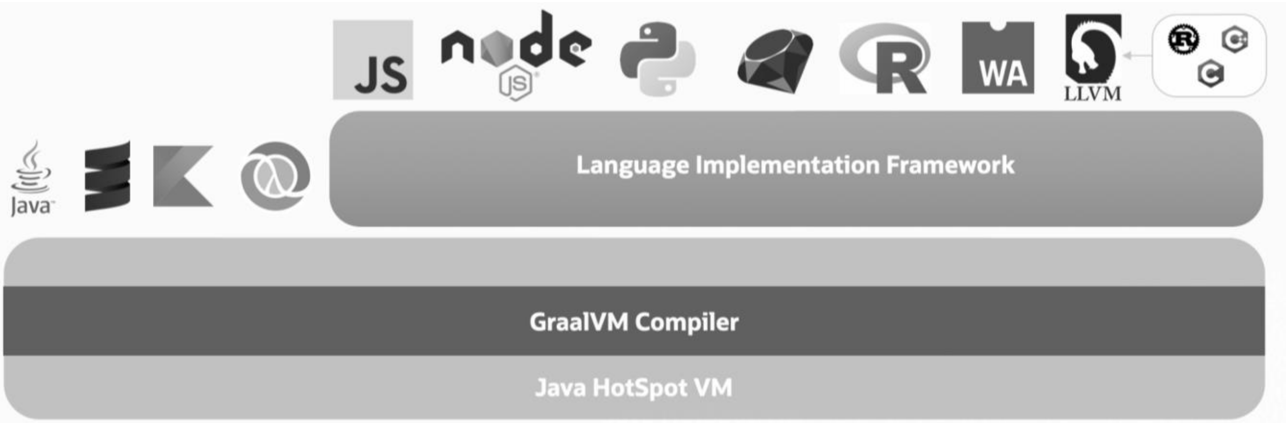
Ena izmed posebnosti GraalVM je tudi možnost prevajanja Java bytecode-a v native kodo. To omogoča ustvarjanje native izvedljivih aplikacij, ki ne potrebujejo ločenega nameščanja virtualnega stroja za njihovo delovanje. S tem se doseže še večja učinkovitost in zmanjša odtis pomnilnika, kar je ključnega pomena za izvajanje aplikacij v oblaku in v mikrostoritvenih arhitekturah.

GraalVM se uporablja v različnih scenarijih, kot so optimizacija izvajanja Java aplikacij, izvajanje več programskih jezikov na istem VM-ju, izdelava native izvedljivih aplikacij in celo kot razvojno orodje za ustvarjanje novih programskih jezikov in prevajalnikov programske kode.

Zaradi svoje zmogljivosti, vsestranskosti in inovativnih trikov je GraalVM postala priljubljena izbira za razvijalce, ki želijo izboljšati odzivnost svojih aplikacij in optimizirati njihovo delovanje v različnih okoljih [6].

Glavne razlike GraalVM v primerjavi z klasičnim JDK so:

1. GraalVM Compiler, JIT prevajalnik za Java
2. GraalVM Native Image, ki omogoča prevajanje Java aplikacij v izvorno kodo pred časom (AOT - ahead-of-time)
3. Ogrodje za implementacijo jezikov Truffle in GraalVM SDK, za dodajanje izvajanja programskih jezikov
4. LLVM Runtime in JavaScript Runtime



**Slika 1: Shema arhitekture GraalVM**

Vir: [7].

### 2.1.2 Mandrel

Mandrel je različica GraalVM skupnostne izdaje - GraalVM CE. Glavni cilj Mandrela je zagotoviti izdelave *native-image*-a, posebej prilagojenega za podporo Quarkus-a. Namen je uskladiti podporo za *native-image* med GraalVM z OpenJDK in knjižnicami Red Hat Enterprise Linux, da se izboljša podpora za native Quarkus aplikacije. Mandrel se lahko najbolje opiše kot različica običajnega OpenJDK s posebej zapakiranim graditeljem GraalVM Native Image (native-image).

Mandrel izdaje so zgrajene iz kode, ki izhaja iz izvorne kode GraalVM, pri čemer so narejene le manjše spremembe, vendar nekatere pomembne izključitve. Polna distribucija GraalVM presega *native-image*: vključuje podporo za več programskih jezikov (polyglot), ogrodje Truffle, ki omogoča učinkovito izvajanje interpreterjev, prevajalnik LLVM za native image, JIT prevajalnik libgraal kot zamenjavo za HotSpot-ov C2 strežni prevajalnik in še veliko več. Mandrel predstavlja majhen podsklop funkcionalnosti, ki se potrebujejo za uporabo z native-image [8].

Mandrel-ova izdaja *native-image* tudi ne vključuje naslednjih funkcij:

- Eksperimentalni strežnik za gradnjo slik, tj. možnost *--experimental-build-server*
- Prevajalnik LLVM, tj. možnost *-H:CompilerBackend=llvm*
- Implementacija musl libc, tj. možnost *--libc=musl*
- Podpora za ustvarjanje statičnih native slik, tj. možnost *--static*
- Podpora za jezike, ki niso osnovani na JVM, in polyglot, tj. možnost *--language:<languageId>*

Mandrel je zgrajen nekoliko drugače od GraalVM, ki uporablja standardno izdajo projekta OpenJDK verzije JDK17. To pomeni, da ne izkoristi nekaj majhnih izboljšav, ki jih je Oracle dodal v različico OpenJDK, ki se uporablja za gradnjo njihovih lastnih GraalVM distribucij. Večina teh izboljšav je v modulu JVMCI, ki omogoča zagon Graal prevajalnika znotraj OpenJDK. Ostale so manjše kozmetične spremembe v delovanju. Te izboljšave lahko v nekaterih primerih povzročijo manjše razlike v postopku generiranja native slik. Sama izvedba slik naj ne bi povzročila opaznih razlik v načinu izvajanja.

### 2.1.3 Microprofile

Eclipse MicroProfile je odprtokodni projekt, ki je namenjen razvoju mikrostoritvenih aplikacij na platformi Java. Projekt je bil ustanovljen leta 2016 kot skupna pobuda več tehnoloških podjetij in organizacij, vključno z Eclipse Foundation, Red Hat, IBM, Payara, Tomitribe in drugimi [9].

Glavni cilj Eclipse MicroProfile je poenostaviti razvoj mikrostoritvenih aplikacij in zagotoviti standardizirano platformo za izgradnjo in upravljanje mikrostoritvenih arhitektur v Java okolju. MicroProfile temelji na obstoječih Java EE (Java Platform, Enterprise Edition) tehnologijah in specifikacijah, ter dodaja nove funkcionalnosti in lastnosti, ki so posebej zasnovane za potrebe mikrostoritvenih aplikacij.

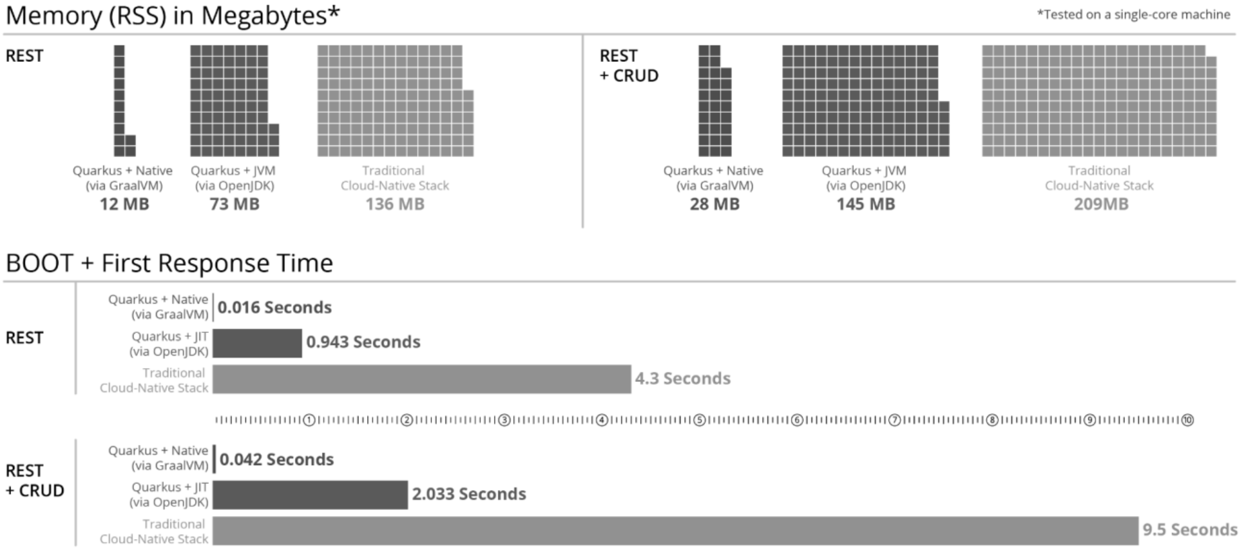
Med glavnimi lastnostmi in specifikacijami, ki jih prinaša Eclipse MicroProfile, so podpora za zaganjanje mikrostoritvenih aplikacij v oblaku, upravljanje konfiguracije, sledenje (tracing), upravljanje z napakami, varnost in še več. Vse te lastnosti so zasnovane z mislijo na prilagodljivost, lahkotnost in učinkovitost, ki so ključnega pomena za razvoj in izvajanje mikrostoritvenih aplikacij.

Eclipse MicroProfile je postal zelo priljubljen med razvijalci, ki se ukvarjajo z razvojem oblačnih in mikrostoritvenih aplikacij. Projekt je v aktivnem razvoju in se hitro razvija, saj skupnost neprestano prispeva z novimi funkcionalnostmi in izboljšavami. Standardizacija, ki jo prinaša Eclipse MicroProfile, omogoča razvijalcem, da svoje mikrostoritvene aplikacije izdelujejo na enoten način, kar olajša prenosljivost aplikacij med različnimi platformami in omogoča boljšo vzdrževanost ter skalabilnost.

### *2.1.4 Quarkus*

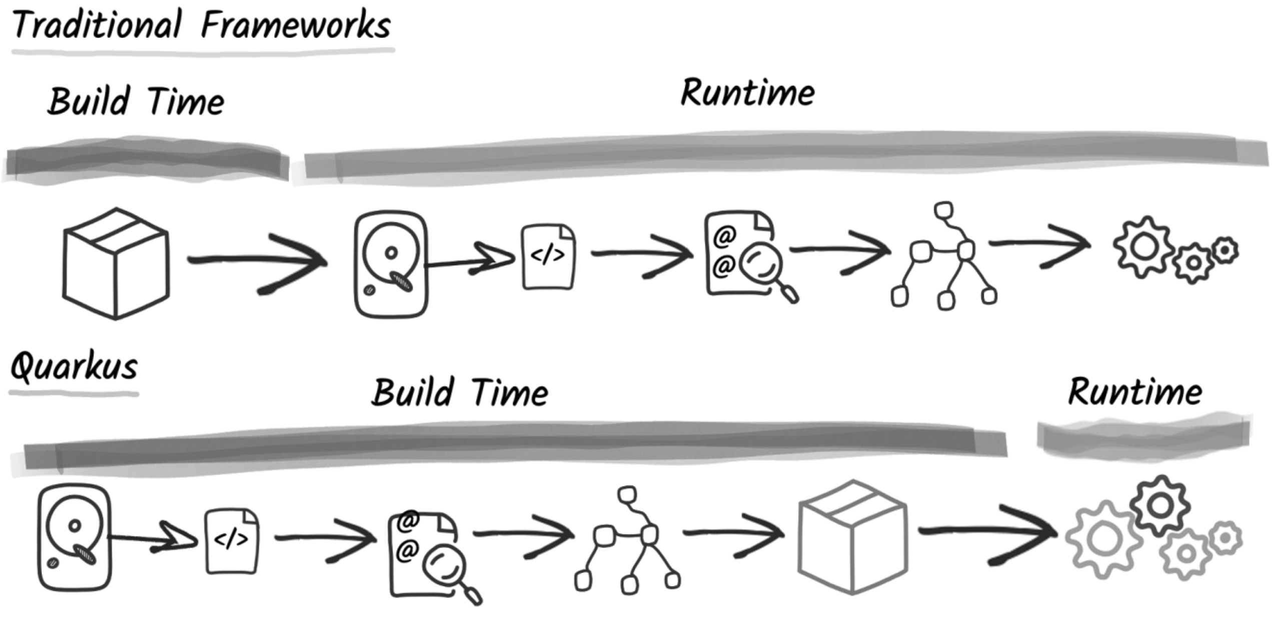
Quarkus je ogrodje, ki temelji na številnih povezanih in ločenih projektih ter jih povezuje v uporabno celoto. Primaren fokus ogrodja je razvoj oblačnih rešitev, a to ne pomeni, da ni mogoče razvijati "klasičnih" aplikacij. Zadnja večja iteracija ogrodja je verzija 3, ki prinaša posodobitve z zunanjimi knjižnicami in ogrodji ter prvič dolgoročno podprto verzijo, ki zagotavlja stabilizacijo in podporo za daljše časovno obdobje.

Quarkus omogoča izgradnjo aplikacije, ki je v primerjavi z klasičnim načinom razvoja hitrejša v času izvajanja.



**Slika 2: Razlike v načinu zagona aplikacij**
Vir: [10].

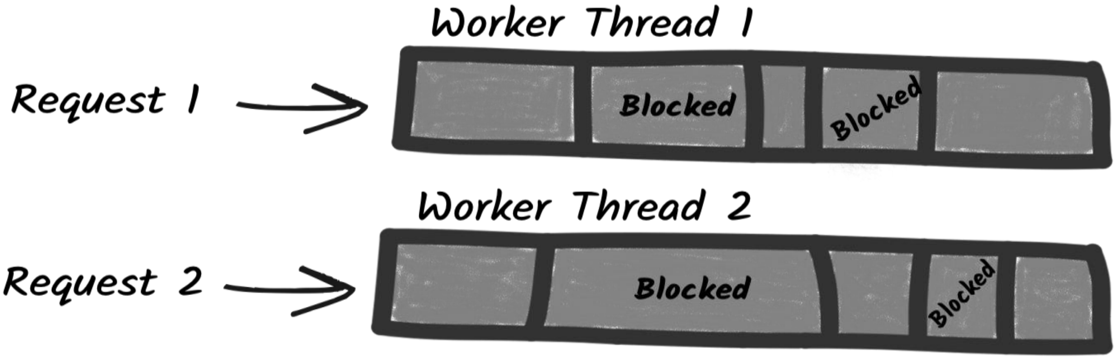
Praviloma velja, da se prihranek na eni strani pozna na drugi strani. Pri Quarkusu to pomeni, da potrebuje več časa za prevajanje in pripravo zagonske slike aplikacije. Daljše prevanje pa je posledica, ker Quarkus prenese proces razrešitve odvisnosti in preračun izvajanja programske kode v čas prevajanja.



**Slika 3: Zagon običajnih Java aplikacij in Quarkus Java aplikacije**
Vir: [11].

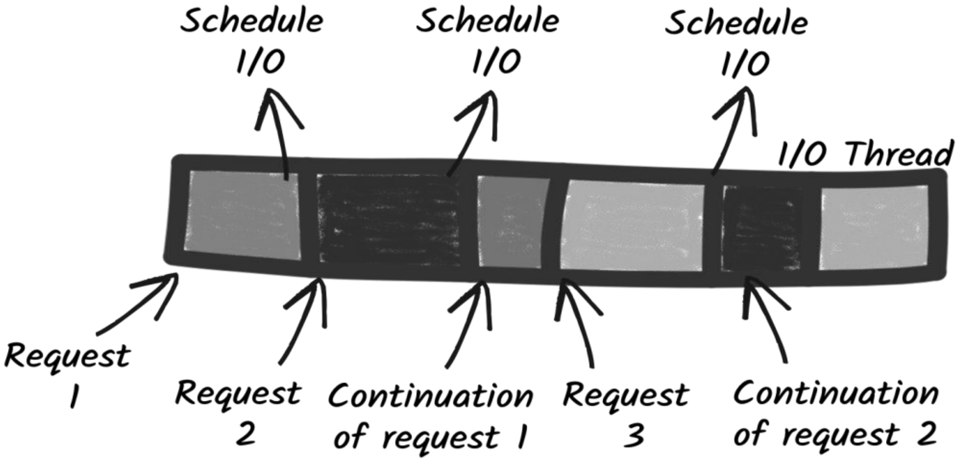
Tak način priprave zagonske aplikacije pomeni, da se pred zagonom pripravi praktično vse, kar se v klasični Java aplikaciji zgodi ob zagonu. Prenos teh korakov v čas gradnje in uporaba t.i. reaktivnih načinov izvajanja programske kode, so glavni stebri razlik in prednosti proti trenutnim klasičnim načinom razvoja.

V tradicionalnem oz. imperativnem pristopu ogrodja dodelijo nit za obdelavo zahtevka. Tako celoten postopek zahtevka poteka na tej delovni niti. Ta model se ne skalira zelo dobro. Dejansko za obdelavo več sočasnih zahtevkov potrebujete več niti; in tako je vzporednost vaše aplikacije omejena s številom niti. Poleg tega so te niti blokirane, takoj ko vaša koda komunicira z oddaljenimi storitvami. To vodi v neučinkovito uporabo virov, saj bo morda potreboval več niti, in vsaka nit, saj so preslikane na niti operacijskega sistema, ima strošek v smislu pomnilnika in procesorske moči.



**Slika 4: Zagon običajnih Java aplikacij in Quarkus Java aplikacije**
Vir: [12].

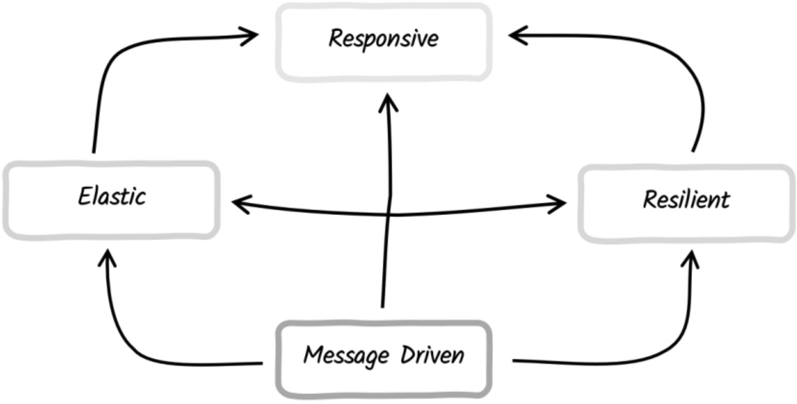
Na drugi strani pa reaktivni model temelji na neblokirajočih I/O operacijah in drugačnem modelu izvajanja. Neblokirajoči I/O zagotavlja učinkovit način za ravnanje s sočasnim I/O. Minimalna količina niti, imenovanih I/O niti, lahko obdela veliko sočasnih I/O operacij. S takšnim modelom se obdelava zahtevka ne prenaša na delovno nit, ampak uporablja te I/O niti neposredno. S tem se prihrani pomnilnik in procesorska moč, saj ni potrebe po ustvarjanju delovnih niti za obdelavo zahtevkov. Prav tako izboljša vzporednost, saj odpravlja omejitev glede števila niti. Nazadnje, izboljša tudi čas odziva, saj zmanjša število preklopov niti.



**Slika 5: Zagon običajnih Java aplikacij in Quarkus Java aplikacije**
Vir: [13].

Quarkus reaktivnost sledi “Reactive Manifesto”, ki opisuje Reaktiven sistem kot distribuiran sistem, ki ima štiri lastnosti:

1. Odzivni (Responsive) - morajo se odzivati pravočasno.
2. Elastični (Elastic) - prilagajajo se nihajoči obremenitvi.
3. Vzdržljivi (Resilient) - omogočajo grajen prehod ob napakah.
4. Asinhrono posredovanje sporočil (Asynchronous message passing) - komponente reaktivnega sistema medsebojno delujejo preko sporočil.



**Slika 6: Zagon običajnih Java aplikacij in Quarkus Java aplikacije**
Vir: [14].

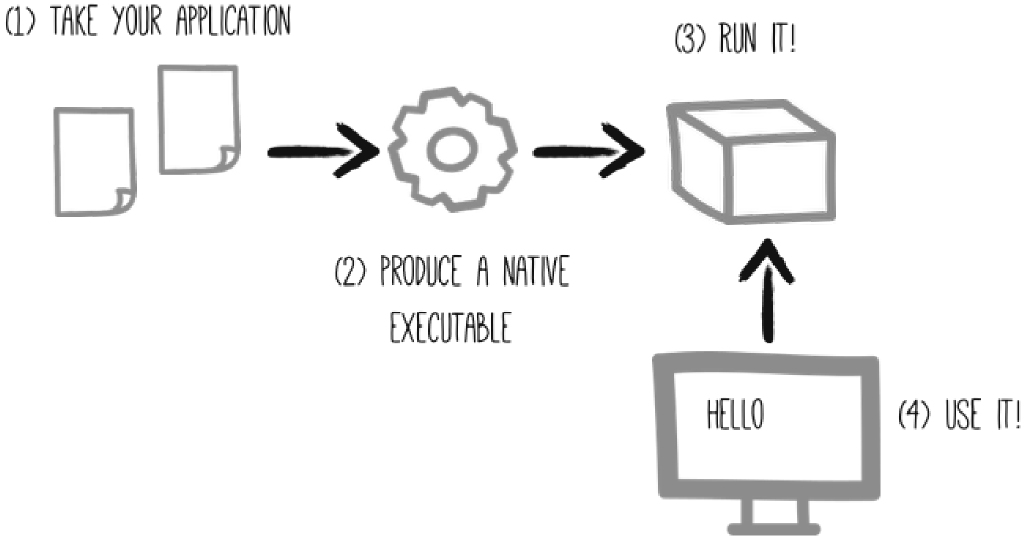
Poleg tega dokument o reaktivnih načelih (Reactive Principles white paper) našteva niz pravil in vzorcev, ki pomagajo pri gradnji reaktivnih sistemov.

Reaktivna načela se nahajajo na spletni strani https://www.reactiveprinciples.org/

Quarkus podpora zagon aplikacije v dveh načinih:

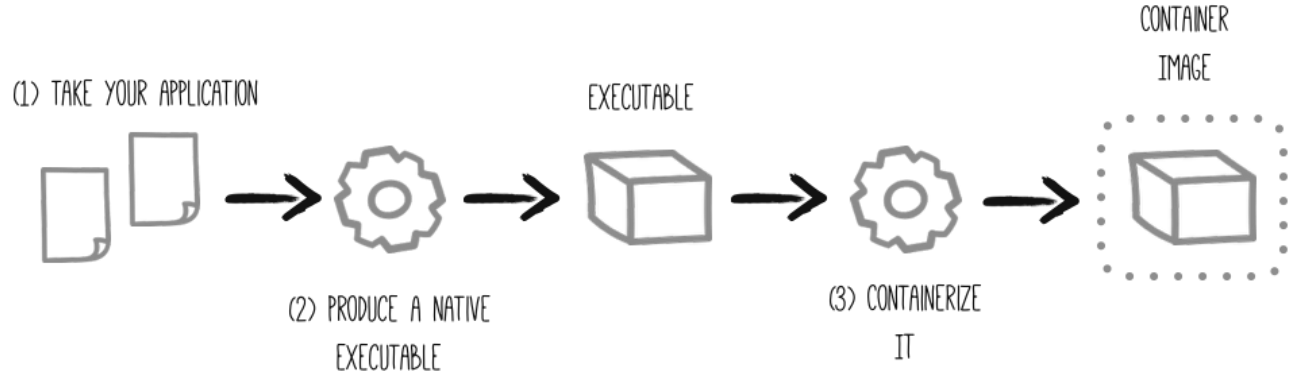
1. tradicionalen preko uporabe Java JDK
2. izdelavo binarne oblike za gostiteljevo arhitekturo
    a. specializirana oblika je “native zabojnik”, ki naredi linux x64 native image ( tudi na ne-linux x64 sistemih )

Native zagonska datoteka zajema celotno aplikacijo in vse odvisnosti. Nabor odvisnosti se je izdelal v času gradnje.



**Slika 7: Zagon običajnih Java aplikacij in Quarkus Java aplikacije**
Vir: [15].

Zabojnik je specializirana oblika, kjer se vse potrebne datoteke za zagon zapakirajo v sliko zabojnika.



**Slika 8: Zagon običajnih Java aplikacij in Quarkus Java aplikacije**
Vir: [16].

Zagon preko načina native-image ima specifične lastnosti, ki lahko določenih scenarijih uporabe privede do neželenih stranskih učinkov, zato tak način zagona ni najboljša rešitev za vse. Glede na lastnosti in potrebe delovanja aplikacije se izvede odločitev, kdaj uporabiti kateri način zagona.

JIT - OpenJDK HotSpot

- Učinkovita poraba CPU-ja.
- Najboljši Garbage Collector.
- Večja poraba internega spomina.
- Obstoječa monitoring in instrumentalna orodja in ogrodja.
- Knjižnice podpirajo samo klasičen JDK.

AOT - GraalVM native image

- Zahteva po veliki količini spomina in obdelava velike količine podatkov - recimo obdelava slik ali velikih datotek.
- Veliko število zahtev - zahtev/sekundo.
- Hitrejši prvi zagon.

## 3 Alternativne rešitve

V zadnjem času so bile ustvarjene alternativne rešitve, ki konkurirajo direktno ali posredno. Pri tem je potrebno imeti v mislih, da je Quarkus, čeprav je odprtokodno ogrodje, primarno otrok Red Hat-a. Že iz tega je pričakovati, da se pojavijo alternativne rešitve. Glavne alternativne rešitve, ki obstajajo so:

- Micronaut,
- Spring Boot in
- OSGi.

*OSGi* je Java specifikacija in ogrodje, ki ima jedro v dinamičnem načinu nalaganja storitev in prilagajanja na spremembe v okolju brez vnovičnih zagonov. Vsa ta dinamičnost je v Quarkus-u praktično nemogoča, saj se vse dinamične variacije razdelajo v času gradnje; kar predstavlja konceptualno drugačnost. OSGi nima nekega lastnika, čeprav nekako spada pod Eclipse fundacijo, največ doprinosa izvorne kode pride iz krogov Red Hat-a.

*Spring Boot* je najstarejše ogrodje in knjižnica, od vseh tukaj opisanih. Razvoj se je začel leta 2012, nato se je v letu 2013 stabiliziralo in uredilo izvorno kodo iz različnih virov ter nato leta 2014 predstavljena verzija 1.0. Spring Boot gradi nad obstoječim in dobro razvitim ter uporabnim ogrodjem Spring Framework. V zadnjem času - letu, tudi Spring Boot podpira in omogoča izdelavo native-image preko GraalVM, podobno kot Quarkus. Spring Boot je preko lastništva pod varstvom VMware-a, ki je del skupine Dell.

Micronaut je praktično edina direktna konkurenca. Glede načina razvoja aplikacije in števila ter verzij integriranih zunanjih knjižnic je razlik zelo malo - obe ogrodji sta v aktivnem razvoju in spremembe se dogajajo iz meseca v mesec; če je še pred meseci veljalo da eno ogrodje ne podpira neke knjižnice, se je naslednji mesec dejstvo obrnilo in tako se ponavlja.

Glavna razlika, kjer je vidna razlika je v celovitosti celotne vertikale, kjer mislim na podporo in sodelovanje razvojnega okolja, Java API-ja ( vmesniki, anotacije, analiza in predlogi ), dokumentacije - učni material s primeri in na koncu zagon aplikacije - v oblačnem okolju ( aka OpenShift ), je zaenkrat prednost na Quarkus, a to se lahko hitro spremeni. Koncepti so praktično isti - oboje temelji na GraalVM ( tudi Spring Boot, če se primerja po tem kriteriju ). Glavna sestavina je GraalVM in pa vzorci ter knjižnice, ki podpirajo vzorce - recimo Vert.x in OpenTelemetry ter podobno.

Micronaut trenutno nima tako podrobno razvitega prevajalnika, ki bi bil sposoben tako podrobno in učinkoviti prevesti in prenesti izvorno kodo v izvajanje. Dodatno k temu Micronaut nima primerljive rešitve za Quarkus Dev Services in pa tako dobre podpore za razvojna okolja - IDE-je. Micronaut je lastniško pod Oracle.

# 4 Praktični primer

Prikazan je praktičen primer razvoja aplikacije z uporabo ogrodja Quarkus 3. Začne se z kreiranjem Java projekta z uporabo Quarkus CLI. Ustvari se PostgreSQL baza s tabelo za hrambo podatkov. Doda se razred za modeliranje podatkovnega modela in za dostop do podatkov v SQL bazi. Na koncu se doda reaktivna REST storitev, ki omogoča osnovne CRUD operacije. Izvorna koda se prevede in naredi paket za zagon. Zagonska datoteka se izvede in z uporabo programa curl se pokliče REST spletni vmesnik.

Za kreiranje in delo z Quarkus projekti sta na voljo dva načina:

1. Maven CLI - uporablja se Apache Maven in se preko razširitev kreira in upravlja s projektom
2. Quarkus CLI - uporablja se namenski CLI, ki ima jasno sintakso in omogoča “samodejno dopolnjevanje ukazov” (auto-complete). Uporabljam oboje, čeprav se pogosteje obrnem na Quarkus CLI, ki se bo uporabljal tudi v nadaljevanju.

Najlažji način namestitve Quarkus CLI-ja je z uporabo aplikacije SDKMAN! [17]:

```
$ sdk install quarkus
```

Ko imam nameščen Quarkus CLI, lahko ustvarim projekt z ukazom:

```
$ quarkus create app si.beranic:rest-json-quickstart \
    --extension='resteasy-reactive-jackson' \
    --no-code
$ cd rest-json-quickstart
```

Argument --extension='resteasy-reactive-jackson' doda Maven odvisnost v datoteko pom.xml:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-resteasy-reactive-jackson</artifactId>
</dependency>
```

Začnem z kreiranjem podatkovnega modela. Za komunikacijo z SQL bazo se uporablja knjižnica Hibernate, za oblikovanje programskega vmesnika pa knjižnica Panache. Podatkovni model SQL tabele:

```
@Entity
@Cacheable
public class Fruit extends PanacheEntity {

       @Column(length = 40, unique = true)
       public String name;

}
```

V nastavitveni datoteki src/main/resources/application.properties vpišem parametre za dostop do PostgreSQL baze:

```
quarkus.datasource.db-kind=postgresql
quarkus.hibernate-orm.database.generation=drop-and-create
```

Ti parametri so za Quarkus zadostni, da kreira vse potrebno za uspešno komunikacijo in izmenjavo podatkov. V datoteko import.sql vpišem začetne SQL stavke, ki napolnijo SQL tabelo:

```
INSERT INTO fruit(id, name) VALUES (1, 'Cherry');
INSERT INTO fruit(id, name) VALUES (2, 'Apple');
INSERT INTO fruit(id, name) VALUES (3, 'Banana');
ALTER SEQUENCE fruit_seq RESTART WITH 4;
```

Aplikacijo lahko poženemo v t.i. dev mode načinu:

```
$ quarkus dev
```

Quarkus ustvari SQL strežnik z bazo in tabelo primerno za nadaljnji razvoj. Dodam REST API storitev:

```
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.ws.rs.Path;

@Path("/fruits")
@ApplicationScoped
public class FruitResource {

  @GET
  public Uni<List<Fruit>> get() {
    return Fruit.listAll(Sort.by("name"));
  }

  @GET
  @Path("/{id}")
  public Uni<Fruit> getSingle(Long id) {
    return Fruit.findById(id);
  }

  @POST
  public Uni<RestResponse<Fruit>> create(Fruit fruit) {
```

```
    return
Panache.withTransaction(fruit::persist).replaceWith(RestResponse.status(CREATED,
fruit));
  }

}
```

S tem je aplikacija primerna za testiranje. Z uporabo aplikacije curl preverimo zapis nove vrednosti, preko REST vmesnika v SQL tabelo.

```
$ curl --header "Content-Type: application/json" \
  --request POST \
  --data '{"name":"Peach"}' \
  http://localhost:8080/fruits
```

Aplikacija je pripravljena na prenos v ciljno okolje. Najprej ga prevedem, to lahko naredim na dva različna načina: a.) klasičen "java" način ali b.) "native-image" način. Za klasične način izvedem:

```
$ quarkus build
```

Za "native-image" način izvedem:

```
$ quarkus build --native
```

Na ciljnem okolju se ustvari SQL strežnik, preko zabojnika:

```
$ podman run -it --rm=true \
    --name postgres-quarkus -e POSTGRES_USER=quarkus \
    -e POSTGRES_PASSWORD=quarkus -e POSTGRES_DB=fruits \
    -p 5432:5432 postgres:15.3
```

Če sem aplikacijo prevedel v klasičen način se izvede:

```
$ java \
   -Dquarkus.datasource.reactive.url=postgresql://localhost/fruits \
   -Dquarkus.datasource.username=quarkus \
   -Dquarkus.datasource.password=quarkus \
   -jar target/quarkus-app/quarkus-run.jar
```

Če pa sem program prevedel v "native-image" način, pa izvedem sledeče:

```
$ ./target/getting-started-with-reactive-runner \
  -Dquarkus.datasource.reactive.url=postgresql://localhost/fruits \
  -Dquarkus.datasource.username=quarkus \
  -Dquarkus.datasource.password=quarkus
```

## 5 Zaključek

V prispevku smo pogledali začetke in namen ogrodja Quarkus ter načine uporabe. Pogledali smo alternativne rešitve in kako se Quarkus primerja z njimi. Prikazan je način izdelave splošno uporabne native aplikacije, ki se lahko izvaja ročno ali preko uporabe zabojnikov. Predstavitev služi kot priročnik začetniku pri prvih korakih uporabe ogrodja Quarkus od razvoja do zagona.

## Literatura

[1] https://quarkus.io/, Quarkus - Supersonic Subatomic Java, dostopano 1.7.2023

[2] https://github.com/quarkusio/quarkus/releases, Releases · quarkusio/quarkus, dostopano 1.7.2023

[3] https://github.com/quarkusio/quarkus/releases/tag/0.0.1, Release 0.0.1 · quarkusio/quarkus, dostopano 1.7.2023

[4] https://github.com/quarkusio/quarkus/commit/161cfa303b4ea366dbd07e54bf4fe5a67ddec497, Initial · quarkusio/quarkus@161cfa3, dostopano 1.7.2023

[5] https://github.com/quarkusio/quarkus/releases/tag/3.2.2.Final, Release 3.2.2.Final · quarkusio/quarkus, dostopano 23.7.2023

[6] https://en.wikipedia.org/wiki/GraalVM, GraalVM - Wikipedia, dostopano 1.7.2023

[7] https://upload.wikimedia.org/wikipedia/commons/1/13/GraalVM_CE_architecture.png, GraalVM_CE_architecture.png, dostopano 1.7.2023

[8] https://github.com/graalvm/mandrel, graalvm/mandrel, dostopano 1.7.2023

[9] https://microprofile.io/, Home - MicroProfile, dostopano 1.7.2023

[10] https://microprofile.io/workinggroup/#mptab-members, Working Group - MicroProfile, dostopano 1.7.2023

[11] https://quarkus.io/assets/images/quarkus_metrics_graphic_bootmem_wide.png, quarkus_metrics_graphic_bootmem_wide.png, dostopano 1.7.2023

[12] https://quarkus.io/guides/images/build-time-principle.png, build-time-principle.png, dostopano 1.7.2023

[13] https://quarkus.io/guides/images/blocking-threads.png, blocking-threads.png, dostopano 1.7.2023

[14] https://quarkus.io/guides/images/reactive-thread.png, reactive-thread.png, dostopano 1.7.2023

[15] https://quarkus.io/guides/images/reactive-systems.png, reactive-systems.png, dostopano 1.7.2023

[16] https://quarkus.io/guides/images/native-executable-process.png, native-executable-process.png, dostopano 1.7.2023

[17] https://quarkus.io/guides/images/containerization-process.png, containerization-process.png, dostopano 1.7.2023

[18] https://sdkman.io/install, Installation - SDKMAN!, dostopano 1.7.2023

# Remix: zmogljivo meta-ogrodje za razvoj spletnih aplikacij

Gregor Jošt,[1] Miha Kočevar[2]

[1] 3fs Nordic SI, d.o.o., Maribor, Slovenija
gregor.jost@3fs.si

[2] AGILEDROP d.o.o., Celje, Slovenija
miha.kocevar@agiledrop.si

V hitro spreminjajočem se svetu spletnega razvoja imajo ogrodja pomembno vlogo pri poenostavljanju in pospeševanju razvoja zmogljivih in uporabniku prijaznih spletnih aplikacij. Med številnimi ogrodji za razvoj odjemalskega dela spletnih aplikacij se vse bolj uveljavlja Remix, zmogljivo meta-ogrodje, ki temelji na knjižnici React. Medtem ko se knjižnica React odlično obnese pri prikazu elementov uporabniškega vmesnika in njihovih posodobitev, Remix kot ogrodje ponuja celovit pristop razvoja spletnih aplikacij. To vključuje usmerjanje (angl. routing), pridobivanje podatkov, optimizacijo delovanja in boljšo izkušnjo za razvijalce. Med pomembnejše lastnosti ogrodja Remix štejemo generiranje spletne strani na strežniku (angl. server-side rendering), kar izboljšuje učinkovitost, omogoča optimizacijo SEO (angl. search engine optimization) in zagotavlja hitro in učinkovito nalaganje začetne strani. V prispevku se bomo osredotočili na več vidikov ogrodja Remix. Najprej bomo podrobneje predstavili samo ogrodje, raziskali njegovo ozadje in poudarili glavne lastnosti. Poleg tega se bomo posvetili tudi primerjavi ogrodja Remix z nekaterimi obstoječimi meta-ogrodji, narejenimi na podlagi knjižnice React. Sledila bo implementacija majhne aplikacije, ki bo omogočila praktičen prikaz določenih prednosti, ki jih ogrodje prinaša. Skozi ta proces bomo raziskali, kako ogrodje omogoča dinamičnost, pridobivanje podatkov z uporabo gnezdenih poti in kako v splošnem izboljša delovanje spletnih aplikacij.

**Ključne besede:**
Remix
React
meta-ogrodje
uporabniški vmesniki
spletne aplikacije

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.8

# 1 Uvod

Pri razvoju spletnih aplikacij imajo ogrodja in knjižnice pomembno vlogo, saj zagotavljajo razvijalcem že vnaprej pripravljena orodja in abstrakcije, ki poenostavljajo postopek razvoja. S tega vidika lahko ogrodje definiramo kot celovit nabor orodij, smernic in konvencij, ki omogočajo strukturiran pristop k razvoju aplikacij, medtem ko so knjižnice zbirke modulov kode, ki jih razvijalci selektivno uporabljajo za lažji razvoj. Knjižnice so torej osredotočene na zagotavljanje specifičnih funkcionalnosti, ogrodja pa ponujajo širšo strukturo in bolj strog pristop k razvoju [1].

Ogrodja za razvoj spletnih aplikacij razvijalcem ponujajo zbirko ponovno uporabljivih komponent, orodij in funkcionalnosti, ki poenostavljajo pogoste naloge in zagotavljajo dosledno strukturo za gradnjo aplikacij. Nekatera najbolj znana in pogosto uporabljena ogrodja za razvoj spletnih aplikacij vključujejo Angular, Vue.js ali Svelte [2]. Ponujajo funkcionalnosti, kot so arhitektura na osnovi komponent, vezava podatkov (angl. data binding), upravljanje stanja in učinkovito izrisovanje (angl. rendering), kar omogoča razvijalcem gradnjo razširljivih in odzivnih spletnih aplikacij.

Po drugi strani pa je knjižnica React v zadnjih letih postala eno izmed najbolj priljubljenih orodij za razvoj spletnih vmesnikov [2], saj zagotavlja deklarativen, komponentno-orientiran pristopu k razvoj. Kljub temu pa se z naraščajočo kompleksnostjo spletnih aplikacij pojavlja potreba po dodatnih orodjih in konvencijah. To je privedlo do pojava meta-ogrodij za React, ki temeljijo na omenjeni knjižnici in zagotavljajo celovito izkušnjo razvoja. V tem kontekstu so meta-ogrodja ogrodja, ki združujejo obstoječa ogrodja ali knjižnice z dodatnimi funkcionalnostmi in orodji ter ponujajo celovito rešitev za razvoj spletnih aplikacij. Pogosto vključujejo funkcionalnosti, kot so usmerjanje, izvajanje na strežniku (SSR), upravljanje stanja in optimizacijo izgradnje [3]. Ta meta-ogrodja, kot je na primer Next.js, so postala priljubljena zaradi svoje sposobnosti poenostavljanja in izboljšanja učinkovitosti razvoja.

V tem prispevku se bomo osredotočili na (meta-)ogrodje Remix, ki je vse bolj priljubljeno ogrodje za razvoj spletnih aplikacij. Ogrodje Remix temelji na knjižnici React in zagotavlja celovit pristop k razvoju spletnih aplikacij, vključno z usmerjanjem, pridobivanjem podatkov, optimizacijo delovanja in izboljšano izkušnjo za razvijalce. S poudarkom na generiranju spletne strani na strežniku (SSR) Remix izboljšuje učinkovitost, omogoča optimizacijo SEO in zagotavlja hitro in učinkovito nalaganje začetne strani [4].

# 2 Ogrodje Remix

## 2.1 Pregled glavnih lastnosti in funkcionalnosti

Ogrodje Remix temelji na filozofiji *postopnega izboljšanja* (angl. Progressive Enhancement) [4]. V osnovi postopno izboljšanje pomeni, da zagotavljamo osnovno vsebino in funkcionalnosti za čim več uporabnikov, hkrati pa omogočamo najboljšo možno izkušnjo uporabnikom novejših brskalnikov, ki lahko izvajajo vse sodobne funkcionalnosti spletnih aplikacij. Podrobneje, pristop se osredotoča na gradnjo spletne strani ali aplikacije tako, da najprej zagotovi osnovno funkcionalnost in vsebino, ki deluje na vseh vrstah brskalnikov in naprav, vključno s starejšimi brskalniki ali tistimi z omejenimi zmogljivostmi. Ta osnovna različica mora biti uporabna in zagotoviti osnovne informacije in funkcionalnosti. Nato se postopno izboljšuje izkušnja tako, da se uporabnikom z novejšimi in zmogljivejšimi brskalniki ponudi izboljšan uporabniški vmesnik, animacije, interaktivne elemente in druge dodatne napredne funkcionalnosti [5]. V skladu s tem Remix omogoča razvijalcem, da začnejo zgraditi preprost HTML na strežniku in nato postopoma nadgrajujejo funkcionalnost z odjemalskim JavaScriptom in interaktivnimi elementi. To omogoča hitro začetno fazo razvoja in nato postopno dodajanje naprednih funkcionalnosti, ko jih dejansko potrebujemo.

Remix prav tako podpira koncept *predpomnjenja*. S pomočjo te funkcionalnosti se vse potrebne datoteke, kot so moduli JavaScript in CSS, predpomnijo v brskalniku, kar omogoča, da se strani hitreje naložijo, zlasti v primeru počasnejšega omrežja. To izboljšuje uporabniško izkušnjo in zmanjšuje čas nalaganja strani.

Ker Remix temelji na knjižnici *React Router*, vključuje prilagodljiv in zmogljiv sistem za *usmerjanje* (angl. routing), ki zagotavlja gradnjo kompleksnih spletnih aplikacij z več (pod)stranmi. To omogoča, da se aplikacija razdeli na več delov, pri čemer ima vsaka svojo URL pot in ločeno logiko. S tem se zagotovi boljša organizacija aplikacije in lažje upravljanje navigacije med stranmi. Takšna optimizacija navigacije na strani odjemalca omogoča, da ogrodje ugotovi, kateri deli strani se oz. se ne spreminjajo med prehodi med URL-ji. Posledično aplikacija pridobiva podatke samo za tiste poglede, ki vključujejo spremembe, kar zmanjšuje nepotrebne zahtevke in izboljšuje odzivnost aplikacije. Takšne *gnezdene poti* (angl. nested routes) olajšajo gradnjo kompleksnih spletnih aplikacij, ki vsebujejo več nivojev poti.

Remix vsebuje tudi vse potrebno za enostavno *pridobivanje podatkov* iz oddaljenih virov. Z vgrajeno podporo za pridobivanje podatkov lahko spletno aplikacijo opremimo z vsemi potrebnimi informacijami za prikazovanje vsebine. To velja tudi za upravljanje z obrazci (angl. forms) spletne strani. Remix v ta namen zagotavlja dve funkciji, prejemnik (loader) in akcijo (action), ki omogočata operacije na strežniški strani z enostavnim dostopom do podatkov. Posledično ni potrebe, da bi na strani odjemalca imeli posebej logiko JavaScript za upravljanje z obrazci. Za razliko od drugih ogrodij, kjer bi morali vključiti JavaScript za izvedbo klicev preko fetch ali Axios, v Remixu to ni potrebno, saj delo z obrazci temelji na njihovem osnovnem delovanju.

Ker gre za ogrodje, ki temelji na knjižnici React, vključuje tudi funkcionalnosti, potrebne za preprosto *upravljanje stanja* (angl. state management). Poleg že poznanih konceptov Reacta, Remix vključuje tudi svoje dodatne funkcionalnosti za upravljanje stanja s podatki, pridobljenimi na strani strežnika. Na ta način lahko učinkovito obvladujemo stanje aplikacije, kar omogoča boljšo uporabniško izkušnjo in lažje upravljanje s podatki aplikacije.

*Prilagodljiva arhitektura* je še ena prednost ogrodja. Z njeno pomočjo lahko razvijalci enostavno organizirajo kodo in ločijo funkcionalnosti, kar olajša vzdrževanje in nadgradnjo aplikacij. Ogrodje je zasnovano tako, da podpira enostavno povezovanje med strežnikom in odjemalcem, kar omogoča dodajanje odjemalske logike brez obsežnih sprememb v obstoječi kodi na strežniku.

V grobem so razvijalci Remixa strnili filozofijo ogrodja v štiri točke [4]:

1. Sprejmi model strežnik/odjemalec, vključno z ločitvijo izvorne kode od vsebine/podatkov.
2. Delaj s temelji spleta, ne proti njim: izkoristi moč in funkcionalnosti, ki jih že ponujajo osnovni gradniki spleta, kot sta protokol HTTP in jezik HTML.
3. Uporabljaj JavaScript za nadgradnjo uporabniške izkušnje z oponašanjem vedenja brskalnika: pri uporabi JavaScripta v spletnih aplikacijah je smiselno dodajati nove funkcionalnosti, ki izboljšajo uporabniško izkušnjo. Vendar pa se ne bi smeli truditi ponovno implementirati že obstoječe funkcionalnosti, ki jo brskalniki že sami ponujajo.
4. Ne pretiravaj z abstrakcijo osnovnih tehnologij.

## 2.2 Podroben pregled ogrodja

V osnovi je ogrodje Remix sestavljeno iz štirih komponent [4]:

1. Prevajalnik (angl. Compiler)
2. Upravljavec strežniških HTTP zahtevkov (angl. server-side HTTP handler)
3. Strežniško ogrodje (angl. server framework)
4. Ogrodje za brskalnik (angl. browser framework)

Za razvoj aplikacij v ogrodju Remix je ključnega pomena uporaba *prevajalnika*. Prevajalnik, ki temelji na **esbuild** orodju, omogoča ustvarjanje različnih delov aplikacije. Med temi deli je tudi strežniški upravljavec HTTP zahtevkov, ki vključuje vse podprte poti spletne strani in ustrezne module. Slednje omogoča izrisovanje na strežniku in obdelavo zahtevkov za različne vire. Poleg tega prevajalnik ustvari tudi vse, kar je potrebno za izvajanje aplikacije v brskalniku. To vključuje samodejno deljenje programske kode glede na pot aplikacije in uvoz virov, na primer stilske datoteke CSS ali slike. Prevajalnik vzpostavi tudi manifest virov (angl. asset manifest), ki ima ključno vlogo pri pred-nalaganju virov med prvotnim izrisom na strežniku in pred-pridobivanju virov za prehode med stranmi na strani odjemalca. Vse tri gradbene artefakte lahko nato naložimo na katerikoli strežnik, ki podpira izvajanje programskega jezika JavaScript.

Naslednja komponenta, ki je sestavni del ogrodja Remix, je *Upravljavec strežniških HTTP zahtevkov*. Čeprav se Remix izvaja na strežniku, dejansko ne predstavlja strežnika. Je le upravitelj (angl. handler), ki se uporablja znotraj dejanskega strežnika JavaScript. Ogrodje je namreč zgrajeno na osnovi spletnega Fetch API-ja. Slednje omogoča, da Remix deluje na različnih strežnikih, kot so Vercel, Netlify, Architect, Cloudflare Workers ali Deno Deploy.

Programski kod spodaj prikazuje, kako je videti Remix, ko se izvaja v ogrodju Express.

```
const remix = require("@remix-run/express");
const express = require("express");

const app = express();

app.all("*", remix.createRequestHandler({ build: require("./build") }));
```

Express (oz. Node.js) je dejanski strežnik, medtem ko je Remix zgolj upravitelj (angl. handler) na tem strežniku. Paket **@remix-run/express**, ki se nahaja v prvi vrstici programskega koda zgoraj, imenujemo adapter. Ker so upravitelji neodvisni od strežnikov, za delovanje potrebujemo takšne adapterje.

V osnovi adapterji pretvorijo vmesnik zahtev in odgovorov določenega strežnika v standardizirani Fetch API na vhodu, ko zahteva pride do Remixa. Nato pa prilagodijo Fetch odgovor, ki ga vrne Remix, nazaj v vmesnik odgovora strežnika. S tem pristopom Remix postane prilagodljiv in sposoben delovati na različnih strežnikih, saj adapterji poskrbijo za medsebojno razumevanje med Remixom in posameznimi strežniki. Adapterji omogočajo, da Remix izkoristi prednosti in funkcionalnosti različnih strežnikov, hkrati pa ohranja enoten in dosleden način delovanja.

Komponenta *Strežniško ogrodje* predstavlja poglavitni del ogrodja. Remix je v osnovi strežniško ogrodje, ki sledi arhitekturi MVC (Model-View-Controller). V primerjavi s tradicionalnimi ogrodji, kot je npr. ASP.NET Core MVC, se v Remixu vlogi pogleda (View) in krmilnika (Controller) združita, medtem ko je model v celoti prepuščen razvijalcem. Namesto da bi bili omejeni na določene tehnologije, lahko izkoristimo celoten ekosistem JavaScripta, ki ponuja številne podatkovne baze, ORM (angl. Object-Relational Mapping) in podobna orodja. Poleg tega Remix vključuje funkcionalnosti za upravljanje piškotkov (angl. cookies) in sej (angl. sessions), kar olajša obvladovanje stanja med zahtevami.

Za razliko od običajnih strežniških ogrodij je Remix osredotočen na uporabniški vmesnik in ne na modele. V Remixu imajo poti (angl. routes) možnost obdelave celotnega spletnega naslova (URL) ali samo njegovega dela. Tako vsaka postavitev (pogled) postane svoj krmilnik, pri čemer ogrodje združuje podatke in komponente za izgradnjo uporabniškega vmesnika. Ena izmed ključnih prednosti tega pristopa je, da ne potrebujemo ločenih krmilnikov za vsak URL. Namesto tega lahko vsaka postavitev (pogled) vsebuje svojo logiko za obdelavo zahtev, kar omogoča, da so strani, ustvarjene z Remixom, bolj dinamične in interaktivne.

Pogosto poti Remixa vključujejo tako uporabniški vmesnik kot interakcije z modeli kar v isti datoteki. Datoteke imajo tri glavne izvoze: prejemnik (loader), akcijo (action) in privzet izvoz (dejanska komponenta). Primer je podan v programskem kodu spodaj.

```
// Prejemniki (loaders) se izvajajo le na strežniku in
// zagotavljajo podatke komponentam med zahtevami GET
export async function loader() {
  return json(await db.projects.findAll());
}

// Akcije (actions) se izvajajo le na strežniku in obdelujejo zahteve POST, PUT, PATCH in DELETE.
// Prav tako lahko zagotovijo podatke komponentam
export async function action({ request }: ActionArgs) {
  const form = await request.formData();

  await createProject({ title: form.get("title") });

  return json({ ok: true });
}

// Izvoz privzete vrednosti je komponenta, ki bo prikazana, ko pot ustreza URL-ju.
// To deluje tako na strežniku kot na odjemalcu.
export default function Projects() {
  const projects = useLoaderData<typeof loader>();
  const actionData = useActionData<typeof action>();

  return (
    <div>
      {projects.map((project) => (
        <Link key={project.slug} to={project.slug}>
          {project.title}
        </Link>
      ))}

      <Form method="post">
        <input name="title" />
        <button type="submit">Create New Project</button>
      </Form>
      {actionData?.errors ? <ErrorMessages errors={actionData.errors} /> : null}

      <Outlet />
    </div>
  );
}
```

Remix lahko uporabljamo tudi izključno kot strežniško ogrodje, brez kakršne koli uporabe JavaScripta na strani odjemalca. Za nalaganje podatkov se lahko uporablja funkcija **loader**, spremembe lahko upravljamo s funkcijo **action**, medtem ko obrazci HTML in komponente zagotavljajo osnovne funkcionalnosti spletne aplikacije.

Zadnja komponenta, *Ogrodje za brskalnik*, pride do izraza, ko Remix brskalniku posreduje dokument HTML. Takrat ogrodje poskrbi, da se stran "hidrira" (angl. hydrate). V kontekstu razvoja spletnih strani hidracija označuje postopek pretvorbe statične vsebine HTML-ja v interaktivno aplikacijo. Ko se spletna stran sprva izriše na strežniku in pošlje odjemalcu, vsebuje statično vsebino in nima dinamičnih funkcionalnosti. Hidracija je nadaljnji korak, ko brskalnik naloži JavaScript kodo in doda upravljavce dogodkov (angl. event handlers), kar naredi stran interaktivno [4].

Remix vključuje tudi nekaj optimizacij za navigacijo med stranmi na strani odjemalca. Ogrodje ve, katere postavitve (angl. layout) se bodo ohranile med dvema URL-jema, zato pridobiva podatke le za tiste, ki se spreminjajo. Ko uporabnik klikne povezavo, namesto da bi na strežnik pošiljali zahtevo za celoten dokument in vse vire, Remix pridobi podatke za prikaz želene strani in posodobi uporabniški vmesnik. Slednje izboljša zmogljivosti v primerjavi z zahtevo za celoten dokument.

Pomembna lastnost ogrodja je tudi *predpomnjenje* (angl. Caching), kar je podprto na naslednji način. Ko uporabnik z miško prehaja preko povezave, Remix že vnaprej predpomni vse potrebne vire za naslednjo stran. To pomeni, da se podatki, moduli JavaScript in celo datoteke CSS za novo stran že naložijo, še preden uporabnik dejansko klikne na povezavo. To stori tako, da brskalniku pošlje manifest sredstev, ki vsebuje seznam vseh virov, ki jih potrebuje stran. Brskalnik nato te vire predpomni v ozadju, preden jih potrebuje, kar zmanjša čas nalaganja strani.

Prednalaganje virov v Remixu poteka na naslednji način:

1. Remix ustvari manifest sredstev, ki vsebuje seznam vseh virov, ki jih potrebuje stran.
2. Remix pošlje manifest sredstev brskalniku.
3. Brskalnik lahko nato te vire predpomni v ozadju.
4. Ko uporabnik klikne povezavo, bo stran že naložena in pripravljena za uporabo.

Ta pristop k predpomnjenju virov je izredno učinkovit, še posebej v primerih počasne spletne povezave. Ker se večina virov že nahaja v predpomnilniku brskalnika, se stran naloži hitro in brez čakanja, kar uporabnikom omogoča odzivno izkušnjo.

### 2.3 Generiranje spletne strani na strežniku (SSR) v ogrodju Remix

*Client-Side Rendering* (CSR) je pristop, ki se pogosto uporablja pri izdelavi spletnih aplikacij. Pri tem pristopu se HTML oz. uporabniški vmesniki oblikuje na strani odjemalca, s pomočjo JavaScripta. Ko uporabnik zahteva določeno spletno stran, se na strani odjemalca najprej naloži osnovni HTML skupaj z datotekami JavaScript. Nato se na strani odjemalca s pomočjo JavaScripta pridobijo podatki iz strežnika in dinamično oblikuje vsebina HTML [6]. CSR pristop omogoča bolj interaktivno uporabniško izkušnjo, saj se vsebina in vmesnik prilagajata uporabnikovim interakcijam brez potrebe po osveževanju celotne strani. Vendar pa CSR prinaša tudi nekatere izzive, kot so počasnejši prvi prikaz strani, saj je potrebno najprej naložiti osnovno strukturo in nato pridobiti in oblikovati podatke z JavaScriptom. Poleg tega je zaradi obsežne uporabe JavaScripta obremenitev na strani odjemalca večja, kar lahko vpliva na učinkovitost in odzivnost spletnih aplikacij.

Kot alternativo imamo že nekaj časa na voljo pristop SSR (*Server-Side Rendering*). Za razliko od CSR se strani HTML ustvarijo in oblikujejo na strežniku, preden se pošljejo na naprave odjemalca. Ko uporabnik zahteva določeno spletno stran, spletni strežnik sestavi potrebno HTML vsebino, vključno s podatki iz baze ali drugih virov, nato pa jo pošlje odjemalcu. Ta pristop omogoča, da se uporabniku prikaže popolnoma oblikovana spletna stran že ob prvem naloženem odgovoru strežnika, saj ni potrebno, da bi se spletna stran generirala šele na strani odjemalca s pomočjo JavaScripta [6]. Ko se spletna stran naloži, pa preko že prej omenjenega postopka hidracije za vso nadaljnjo izvajanje poskrbi koda JavaScript. SSR prinaša nekatere prednosti, kot so boljša uporabniška izkušnja zaradi hitrejšega prikaza začetne strani, boljša optimizacija za iskalnike (SEO) in boljša dostopnost.

Remix privzeto uporablja SSR kot osrednji mehanizem izrisovanja (angl. rendering), vendar vključuje tudi CSR za usmerjanje na strani odjemalca in dinamično posodabljanje strani po prvotnem nalaganju. Ta hibridni pristop združuje prednosti SSR za prvotno izrisovanje z ugodnostmi CSR za boljše interakcije na strani odjemalca.

### 2.4 Optimizacija SEO in hitro nalaganje začetne strani

SEO (ang. Search Engine Optimization) pomeni optimizacija spletnih strani za iskalnike. Gre za prakso optimizacije spletne strani ali spletišča za izboljšanje njene vidnosti in uvrstitve na straneh iskalnikov (ang. SERP-ji oz. Search Engine Result Pages). Glavni cilj SEO-ja je privabiti več organskega (neplačanega) prometa iz iskalnikov, kot so Google, Bing, Yandex ali Yahoo [7].

Remix dosega boljšo optimizacijo za iskalnike z naslednjimi mehanizmi [4].

Kot že omenjeno, se Remix razlikuje od običajnih enostranskih aplikacij (Single-page application), saj privzeto podpira SSR. Ko obiskovalec zahteva spletno stran, se s tem pristopom na strežniku generira začetni HTML s predhodno vključeno vsebino. To zagotavlja, da so spletni iskalniki sposobni takoj dostopati do celotne vsebine, kar olajša njihovo indeksiranje.

Ena izmed prednosti Remixa je tudi nadzor nad metapodatki za SEO. Ponuja orodja in tehnike za prilagajanje naslovov strani, meta-opisov, kanoničnih URL-jev in strukturiranih podatkov. Pravilno upravljanje metapodatkov pripomore k boljšemu razumevanju vsebine s strani spletnih iskalnikov, kar izboljšuje možnosti za višje uvrščanje v iskalnih rezultatih.

Pomembno je tudi izpostaviti, da Remix skrbi za čiste URL-je. To obenem olajša navigacijo obiskovalcem in hkrati pripomore k boljšemu indeksiranju vsebine s strani iskalnih botov.

Nenazadnje pa ogrodje tudi stremi k manjši odvisnosti od JavaScripta na strani odjemalca, kar pripomore k boljšemu razumevanju in sledenju vsebine s strani spletnih iskalnikov. Z uporabo SSR in postopnega izboljševanja, Remix omogoča enostavno dostopanje do vsebine, kar je ključnega pomena za SEO.

Kombinacija vseh omenjenih pristopov, torej SSR, postopnega izboljševanja, nadzora metapodatkov in drugih funkcij, usmerjenih v SEO, naredi Remix skladen z najboljšimi praksami za SEO.

## 3 Primerjava z obstoječimi meta-ogrodji

Remix ni edino meta-ogrodje, ki temelji na knjižnici React. Med alternativami se pogosto omenjata tudi Next.js in Gatsby. Čeprav imajo nekaj podobnosti, imajo tudi različne značilnosti in primere uporabe, kar bomo predstavili v nadaljevanju.

### 3.1 Next.js

Next.js je priljubljeno odprtokodno ogrodje za izdelavo naprednih spletnih aplikacij s pomočjo jezika JavaScript. Temelji na knjižnici React in ponuja napredne funkcionalnosti, ki olajšajo razvoj in izboljšajo uporabniško izkušnjo. Glavna značilnost Next.js je njegova sposobnost podpiranja tako SSR kot tudi CSR [8].

Next.js ponuja preprost pristop k usmerjanju, ki temelji na sistemu datotek. To pomeni, da lahko razvijalci organizirajo in upravljajo poti v svoji aplikaciji preprosto z uporabo imenikov in datotek, kar olajša vzdrževanje in razširljivost.

Poleg tega Next.js vključuje tudi podporo za pridobivanje podatkov. Na voljo imamo različne metode, kot so pridobivanje podatkov na strani strežnika, statična generacija spletnih strani ali pridobivanje podatkov na strani odjemalca.

### 3.2 Gatsby

Gatsby je še eno priljubljeno odprtokodno ogrodje za izdelavo spletnih aplikacij, ki temelji na Reactu. Ogrodje je namenjeno predvsem izdelavi statičnih spletnih strani.

Glavna značilnost Gatsbyja je njegova sposobnost pred-generiranja statičnih HTML strani med izdelavo aplikacije (angl. build time). To pomeni, da se vsebine in podatki pridobijo med gradnjo aplikacije, nato pa se izdelajo pred-izrisane (angl. pre-rendered) strani HTML. Te strani se nato posredujejo odjemalcu kot običajne statične datoteke, kar omogoča izjemno hitro nalaganje in odzivnost spletnega mesta [9].

Gatsby omogoča razvijalcem enostavno upravljanje vsebine s pomočjo datotek Markdown ali drugih formatov. Poleg tega Gatsby uporablja GraphQL za pridobivanje podatkov, kar olajša integracijo s CMS platformami in drugimi zunanjimi viri podatkov. Podatki se torej pridobivajo in uporabljajo na učinkovit način, kar prispeva k hitrosti in zmogljivosti spletne strani. Gatsby ponuja tudi bogat ekosistem vtičnikov, ki omogočajo razvijalcem dodajanje funkcionalnosti, kot so optimizacija zmogljivosti, analitika, SEO in še več.

Gatsby se predvsem osredotoča na statično generacijo spletnih strani (SSG) in je primeren za izdelavo spletnih strani, osredotočenih na vsebino, kot so blogi.

### 3.3 Rezultati primerjave

Tabela 1 prikazuje rezultate primerjave ogrodij Remix, Next.js in Gatsby. Pri primerjavi smo se osredotočili predvsem na načine izrisa, SEO, pridobivanja podatkov, usmerjanja in podpori obrazcev.

**Tabela 1: Primerjava meta-ogrodij React.**

| | Remix | Next.js | Gatsby |
|---|---|---|---|
| Izris | Server-side rendering (SSR) | Server-side rendering (SSR), Static Site Generation (SSG) | Static Site Generation (SSG) |
| SEO | Zelo dobra podpora zaradi SSR | Zelo dobra podpora zaradi SSR | Odlična podpora zaradi SSG |
| Skupnost | Narašča, vendar ni primerljiva z Next.js in Gatsby | Obsežna, dejavna | Obsežna, dejavna |
| Pridobivanje podatkov | Uporablja React Suspense za asinhrono pridobivanje podatkov | Uporablja getStaticProps ali getServerSideProps za statično pridobivanje podatkov | Uporablja GraphQL za pridobivanje podatkov |
| Usmerjanje | Uporablja Remix Router ze upravljanje s potmi, uporablja strukturo projekta za ustvarjanje poti | Uporablja Next.js Router, uporablja strukturo projekta za ustvarjanje poti | Uporablja Gatsby Router |
| Upravljanje z obrazci | Uporablja privzeto obnašanje obrazcev na strani odjemalca za pošiljanje podatkov | Uporablja prilagojene (angl. custom) obrazce | Uporablja mutacije GraphQL |

Kot je razvidno iz tabele, so Remix, Next.js in Gatsby izvrstna meta-ogrodja za razvoj spletnih aplikacij. Vsak od njih ima svoje edinstvene prednosti in značilnosti, zato je ključno izbrati ogrodje, ki najbolje ustreza specifičnim potrebam projekta.

Iz primerjave lahko razberemo, da Remix omogoča razvoj odzivnih aplikacij in zagotavlja podporo za optimizacijo spletnega iskanja (SEO). Po drugi strani Next.js poleg SSR podpira tudi izris statičnih spletnih strani. Močno podporo SEO pa zagotavlja Gatsby, ki je primarno namenjen implementaciji statičnih spletnih strani.

## 4 Implementacija aplikacije

V tem poglavju bomo predstavili proces implementacije spletne aplikacije z uporabo ogrodja Remix. V osnovi bo aplikacija omogočala uporabnikom, da vnesejo kakršne koli ideje, drugi uporabniki bodo imeli možnost »všečkati« posamezno idejo.

### 4.1 Namestitev osnovne aplikacije

Za izdelavo aplikacije Remix smo uporabili naslednje:

- Node.js v18.16.0 (okolje za izvajanje kode JavaScript),
- npm v9.5.1 (upravitelj paketov za aplikacije Node.js) in
- VS Code (urejevalnik kode).

Skladno z navodili se razvoj začne z izvedbo ukaza **npx create-remix@latest** v ukazni vrstici. Ob sami vzpostavitvi se moramo opredeliti glede določenih aspektov aplikacije, kar bomo predstavili v nadaljevanju.

Na začetku smo morali izbrati, kateri tip aplikacije želimo razviti. Na voljo imamo dve možnosti, za voljo razvoja tega primera pa smo se odločili za preprosto aplikacijo (možnost **just the basics**).

```
? What type of app do you want to create? (Use arrow keys)
> Just the basics
A pre-configured stack ready for production
```

Remix prav tako ponuja različne ponudnike za namestitev aplikacije. V našem primeru smo izbrali **Remix App Server**.

```
? Where do you want to deploy? Choose Remix App Server if you're unsure; it's easy to change
deployment targets.
> Remix App Server
  Express Server
  Architect
  Fly.io
  Netlify
  Vercel
  Cloudflare Pages
```

Ko potrdimo vse možnosti vzpostavitve aplikacije Remix, se v izbranem imeniku na podlagi šablone (angl. Template) ustvari naslednja struktura [4]:

- app/ - Imenik, kjer se nahaja celotna aplikacija Remix.
- app/entry.client.tsx – Predstavlja kodo JavaScript, ki se izvede, ko se aplikacija naloži v brskalniku. Uporablja se za hidracijo komponent React.
- app/entry.server.tsx – Predstavlja kodo JavaScript, ki se izvede, ko zahtevek doseže strežnik. Remix obdela vse potrebne podatke, razvijalci aplikacije pa implementiramo ustrezni odgovor (angl. Response). Programski kod v tej datoteki skrbi, da se aplikacija React pretvori v niz, ki se potem pošlje odjemalcu.
- app/root.tsx – V tej datoteki se nahaja korenska komponenta aplikacije, ki izriše element <html>.
- app/routes/ - Tu se nahajajo vsi moduli poti (angl. route modules). Remix na podlagi imena datotek v tej mapi ustvari URL poti za aplikacijo.
- public/ - Tu se nahajajo statični viri (slike, pisave itd.).
- remix.config.js – Vsebuje možnosti konfiguracije aplikacije.

V nadaljevanju bomo predstavili pomembnejše vidike aplikacije in podali primer programskega koda.

### 4.2 Podatkovna baza

Za upravljanje podatkovne baze smo uporabili SQLite v povezavi s Prisma. Prisma je odprtokodni objektno-relacijski preslikovalnik (ORM) za Node.js in TypeScript ter se uporablja kot alternativa pisanju običajnih poizvedb SQL. Prisma trenutno podpira različne podatkovne baze, med drugim PostgreSQL, MySQL, SQL Server, SQLite, MongoDB in CockroachDB [10].

S pomočjo Prisma Migrate smo ustvarili tabelo v podatkovni bazi, ki bo služila shranjevanju idej. Shema tabele je prikazana v naslednjem programskem kodu.

```
model Idea {
  id          String   @id @default(uuid())
  createdAt   DateTime @default(now())
  updatedAt   DateTime @updatedAt
  title       String
  description String
  likes       Int      @default(0)
}
```

### 4.3 Korenska komponenta

Kot že rečeno, korenska komponenta aplikacije izriše element <html>, skrbi pa tudi, da se naložijo metapodatki strani (<Meta />), povezave do stilov CSS (<Links />) in prostor, kamor se glede na naslov URL-ja izpiše vsebina aplikacije (<Outlet />). Če želimo izvajati kodo JavaScript tudi na strani odjemalca, pa je potrebno vključiti še <Scripts />.

Primer korenske komponente za potrebe aplikacije je predstavljen v naslednjem izseku programskega koda.

```
import globalStylesUrl from "~/styles/global.css";

export const links: LinksFunction = () => [
  { rel: "stylesheet", href: globalStylesUrl },
];

export default function App() {
  return (
    <html lang="en">
      <head>
        <meta charSet="utf-8" />
        <meta name="viewport" content="width=device-width,initial-scale=1" />
        <Meta />
        <Links />
      </head>
      <body>
        <Outlet />
        <Scripts />
        <LiveReload />
      </body>
    </html>
  );
}
```

### 4.4 Spletne poti

Aplikacija bo nudila možnost pregleda seznama idej, pregleda posamezne ideje in dodajanje nove ideje. Kot že rečeno, usmerjanje v aplikacijah Remix temelji na osnovi datotek, kjer so poti določene na podlagi strukture. Za naše potrebe bi potrebovali naslednje poti:

- /
- /ideje
- /ideje/:idejaId
- /ideja/dodaj-novo

Da dosežemo takšne poti, moramo ustrezno poimenovati datoteke. Glede na konvencijo poimenovanja poti in datotek, zgoraj opredeljene poti postanejo datoteke znotraj imenika **app/routes**:

- _index.tsx
- ideja.tsx
- ideja.$idejaId.tsx
- ideja.dodaj-novo.tsx

### 4.4.1 Glavna pot

V datoteki **_index.tsx** se nahaja glavna pot, ki se izvede, ko uporabnik obišče osnovni URL aplikacije. Ta pot predstavlja torej začetno stanje aplikacije, ki se prikaže ob prvem obisku. Programski kod glavne poti je naslednji.

```
export const links: LinksFunction = () => [
  { rel: "stylesheet", href: stylesUrl },
];

export default function IndexRoute() {
  return (
    <div className="container">
      <div className="content">
        <h1>Aplikacija za beleženje seznama idej</h1>
        <nav>
          <ul>
            <li>
              <Link to="ideja">Preglej seznam idej</Link>
            </li>
          </ul>
        </nav>
      </div>
    </div>
  );
}
```

Kot je razvidno, je v datoteki definirana komponenta **IndexRoute**, ki predstavlja začetno stanje aplikacije. Gre za preprosto komponento React, brez kakršnih koli dodatkov ogrodja Remix.

Ko se ta pot izvede, se prikaže naslov strani, navigacijski meni z gumbom za pregled seznama idej in potrebni slogi za oblikovanje. Ta komponenta se nato izriše v <Outlet /> korenske komponente aplikacije. Kot že omenjeno, je <Outlet /> posebna oznaka, ki določa mesto, kamor se vstavi vsebina (izrišejo komponente), povezane s trenutno potjo.

### 4.4.2 Pot, ki prikaže seznam idej

Za izpis seznama idej, ki se nahaja na poti **/ideje**, potrebujemo na strežniku dostop do podatkovne baze. Obenem želimo tudi omejiti, katere podatke posredujemo odjemalcu. Vse to podpira ogrodje Remix tako, da izvozimo funkcijo **loader**. Ta funkcija omogoča pridobivanje in obdelavo podatkov preden se prikažejo odjemalcu. Vsebino funkcije predstavlja naslednji programski kod.

```
export const loader = async () => {
  const ideas = await db.idea.findMany({
    orderBy: { createdAt: "desc" },
    select: { id: true, title: true, likes: true },
    take: 5,
  });

  return json({ ideas });
};
```

Iz primera zgoraj je razvidno, da se funkcija loader uporablja za pridobivanje idej iz podatkovne baze s pomočjo funkcije **db.idea.findMany**. S to funkcijo lahko določimo različne parametre, kot je ureditev po času ustvarjanja ideje, izbira določenih polj tabele in omejitev števila idej, ki jih bomo prikazali.

Podatki so nato na voljo na strani odjemalca preko **useLoaderData** kavlja (angl. hook), ki ga ponuja ogrodje Remix. Kavelj vrne vsebino, ki jo nato s pomočjo standardne sintakse Reacta prikažemo v obliki seznama, kot je razvidno iz spodnjega izseka programskega kode.

```
export default function IdeasRoute() {
  const data = useLoaderData<typeof loader>();

  return (
    <main className="ideas-main">
      <div className="ideas-list">
        <p>Seznam idej</p>
        <ul>
          {data.ideas.map(({ id, title, likes }) => (
            <li key={id}>
              <Link to={id} prefetch="intent">
                {title}
              </Link>
              {likes}👍
            </li>
          ))}
        </ul>
        <Link to="new" className="button">
          Dodaj svojo idejo
        </Link>
      </div>
      <div className="ideas-outlet">
        <Outlet />
      </div>
    </main>
  );
}
```

### *4.4.3 Pot za dodajanje nove ideje*

Za dodajanje nove ideje bomo potrebovali obrazec, ki se bo ob potrditvi na strani odjemalca shranil v podatkovno bazo. V ta namen ogrodje Remix ponuja funkcijo **action**. Funkcija action omogoča, da opravimo obdelavo obrazcev in shranjevanje podatkov neposredno na strežniku. Primer funkcije za shranjevanje novih idej predstavlja naslednji programski kod.

```
export const action = async ({ request }: ActionArgs) => {
  const form = await request.formData();

  const title = form.get("title");
  const description = form.get("description");

  if (typeof title !== "string" || typeof description !== "string") {
    return badRequest({
      fieldErrors: null,
      fields: null,
      formError: "Form not submitted correctly.",
    });
  }

  const newIdea = await db.idea.create({
    data: { description, title },
  });

  return redirect(`/ideje/${newIdea.id}`);
};
```

Za vnos podatkov potrebujemo še obrazec, ki je implementiran v obliki komponente React. V primeru omenjene komponente ne potrebujemo specifičnih funkcionalnosti ogrodje Remix, saj gre za osnovne funkcionalnosti ogrodja React, kar je razvidno iz programskega koda spodaj.

```
export default function NewIdeaRoute() {
  return (
    <div>
      <p>Dodaj svojo idejo</p>
      <form method="post">
        <div>
          <label>
            Naziv:
            <input type="text" name="title" required />
          </label>
        </div>
        <div>
          <label>
            Opis:
            <textarea name="description" required />
          </label>
        </div>
        <div>
          <button type="submit" className="button">
            Dodaj idejo
          </button>
        </div>
      </form>
    </div>
  );
}
```

Čeprav gre za preprosto komponento React, je pomembno poudariti, da ob potrditvi obrazca ogrodje Remix prevzame nadzor nad izvedbo in samodejno kliče zgoraj definirano funkcijo action. To pomeni, da se ob oddaji obrazca ne izvede le osnovna funkcionalnost komponente React, temveč ogrodje Remix poskrbi za izvedbo dejanj, kot sta obdelava obrazca in shranjevanje podatkov. S tem pristopom je doseženo tesno usklajevanje med obrazcem na odjemalcu in obdelavo na strežniku.

### 4.4.4 Pot za pregled posamezne ideje

Pot, ki omogoča tako pregled posamezne ideje glede na njeno vrednost ID kot »všečkanje« ideje, združuje oba koncepta, predstavljana v prejšnjih podpoglavjih, torej funkciji loader in action.

Funkcija loader bo poskrbela, da se na podlagi trenutnega ID-ja v naslovu URL izpišejo ustrezni podatki pripadajoče ideje. Uporabniku se bo torej prikazal naslov, opis in število »všečkov« ideje.

Funkcija action pa bo skrbela za posodabljanje števila »všečkov« ob kliku na pripadajoči gumb.

Implementacija obeh funkcij je razvidna iz naslednjega programskega koda.

```
export const loader = async ({ params }: LoaderArgs) => {
  const idea = await db.idea.findUnique({
    where: { id: params.ideaId },
    select: { title: true, description: true, likes: true },
  });

  if (!idea) {
    throw new Error("Idea not found");
  }

  return json({ idea });
};

export const action = async ({ params }: ActionArgs) => {
  const idea = await db.idea.findUnique({
    where: { id: params.ideaId },
  });

  if (!idea) throw new Error("Idea not found");

  const updatedLikes = idea?.likes + 1;

  await db.idea.update({
    where: { id: params.ideaId },
    data: {
      likes: updatedLikes,
    },
  });

  return redirect(".");
};
```

V primeru komponente, ki prikazuje podatke o posamezni ideji, se poleg osnovnih funkcionalnosti React poslužujemo tudi dodatnih funkcionalnosti, ki jih nudi Remix. Programski kod prikazuje izpis podrobnosti posamezne ideje in gumb za »všečkanje«.

```
export default function IdeaRoute() {
  const data = useLoaderData<typeof loader>();

  return (
    <div>
      <p>Idea:</p>
      <h1>{data.idea.title}</h1>
      <p>{data.idea.description}</p>
      <p>Všečki: {data.idea.likes}</p>
      <Form method="post">
        <button className="button">👍</button>
      </Form>
    </div>
  );
}
```

Kavelj useLoaderData smo že predstavili v prejšnjem podpoglavju, v zgornjem primeru pa je potrebno izpostaviti tudi komponento **<Form>**.

V ogrodju Remix je uporaba velike začetnice pri komponenti <Form> namenjena razlikovanju med običajnimi elementi HTML in prilagojenimi komponentami, ki jih zagotavlja ogrodje Remix. Ključna razlika med komponento <Form> in običajnim elementom HTML <form> je, kako ogrodje upravlja oddajo obrazcev. Medtem ko običajen

HTML obrazec običajno sproži ponovno naložitev strani, komponenta <Form> ogrodja Remix uporablja navigacijo na strani odjemalca, kar prepreči ponovno nalaganje strani.

V zgornjem primeru smo preko implementacije prikazali, kako preprosto je implementirati interaktivne spletne aplikacije s pomočjo ogrodja Remix. Zahvaljujoč intuitivni naravi in naprednim funkcionalnostim, ogrodje omogoča razvijalcem hiter in učinkovit razvoj dinamičnih aplikacij s poudarkom na uporabniški izkušnji.

## 5 Zaključek

V prispevku smo predstavili ogrodje Remix in podrobno opisali njegove lastnosti in filozofijo. Remix se je izkazal za inovativno meta-ogrodje, ki postavlja v ospredje uporabniško izkušnjo, lažje vzdrževanje in zmogljivost spletnih aplikacij. Opravili smo tudi primerjavo z drugimi podobnimi meta-ogrodji, da bi bolje razumeli, kako se Remix uvršča v ekosistem razvoja spletnih aplikacij. Medtem ko je na trgu več rešitev, Remix izstopa po svoji edinstveni kombinaciji pristopov, ki združujejo najboljše lastnosti obstoječih ogrodij. Na koncu smo še predstavili praktičen primer uporabe ogrodja, kjer smo ustvarili spletno aplikacijo, ki omogoča uporabnikom dodajanje idej. Implementacija je pokazala, da ogrodje dobro podpira zahteve dinamičnih spletnih aplikacij.

Ogrodje Remix je nedvomno obetavno ogrodje za razvoj spletnih aplikacij, ki odpira vrata inovacijam in učinkovitemu razvoju. S svojo osredotočenostjo na razvijalsko izkušnjo in podporo spletnim standardom zagotavlja, da bo imel pomembno vlogo v prihodnosti razvoja spletnih aplikacij.

## Literatura

[1] S. Roy, „The Difference Between a Framework and a Library," 23 12 2022. [Elektronski]. Dosegljivo: https://www.baeldung.com/cs/framework-vs-library.

[2] Stack Overflow, „Stack Overflow Developer Survey 2023," 2023. [Elektronski]. Dosegljivo: https://survey.stackoverflow.co/2023/.

[3] Codemotion, „Here to Stay: All About Meta-Frameworks," 20 4 2023. [Elektronski]. Dosegljivo: https://www.codemotion.com/magazine/languages/here-to-stay-all-about-meta-frameworks/.

[4] Remix Software, Inc., „Remix Docs," 2023. [Elektronski]. Dosegljivo: https://remix.run/docs/en/main.

[5] MDN contributors, „Progressive Enhancement," 2023. [Elektronski]. Dosegljivo: https://developer.mozilla.org/en-US/docs/Glossary/Progressive_Enhancement.

[6] P. Ram, „Server Side Rendering (SSR) vs. Client Side Rendering (CSR) vs. Pre-Rendering using Static Site Generators (SSG) and client-side hydration," 19 10 2021. [Elektronski]. Dosegljivo: https://medium.com/@prashantramnyc/server-side-rendering-ssr-vs-client-side-rendering-csr-vs-pre-rendering-using-static-site-89f2d05182ef.

[7] D. Goodwin, „What Is SEO – Search Engine Optimization?," 2023. [Elektronski]. Dosegljivo: https://searchengineland.com/guide/what-is-seo.

[8] D. Niżyński, „What is next js and why should you use it in 2023?," 13 7 2023. [Elektronski]. Dosegljivo: https://pagepro.co/blog/what-is-nextjs/.

[9] L. D., „What Is Gatsby and How It Works," 13 4 2023. [Elektronski]. Dosegljivo: https://www.hostinger.com/tutorials/what-is-gatsby.

[10] Prisma Data, Inc., „Prisma," 2023. [Elektronski]. Dosegljivo: https://www.prisma.io/docs/concepts/overview/what-is-prisma.

# Vtičnik v Grafani za napredno vizualizacijo vremenskih podatkov

Miha Lenko, Robert Meolic

Operato, energetske rešitve, d.o.o., Maribor, Slovenija
miha.lenko@operato.eu, robert.meolic@operato.eu

V članku obravnavamo problem vizualizacije vremenskih podatkov v sklopu tehnologije dinamično ocenjevanje prenosne zmogljivosti (DLR). To je obetavna komponenta zelenega prehoda energetike, ki omogoča bolj učinkovito integracijo obnovljivih in razpršenih virov energije samo z upoštevanjem vremenskih dejavnikov, ki vplivajo na segrevanje in hlajenje posameznih elementov. SUMO je modularna rešitev za DLR podjetja Operato, ki za vizualizacijo vremenskih podatkov uporablja orodje Grafana. V standardni različici Grafane ni namenskega pogleda na podatke o vetru, ravno veter pa ima največji vpliv na delovanje algoritma in si ga uporabniki zato želijo podrobno analizirati. Veter je za razliko od večine drugih vremenskih veličin vektor, torej ima hitrost in smer. Dodatno je smer vetra omejena z domeno vrednosti od 0 do 360 stopinj. V meteorologiji se za analizo podatkov o vetru pogosto uporablja prav poseben prikaz imenovan *roža vetrov,* ki ga ni mogoče nadomestiti ali sestaviti z uporabo drugih grafikonov. Zato je bil izdelan nov vtičnik za Grafano. Za izvedbo naloge je bilo potrebno osnovno poznavanje jezika TypeScript, tehnologije ReactJS in formata SVG za opis vektorske grafike. Izdelana je bila kvalitetna aplikacija, ki omogoča sprotno osveževanje gradnikov in podpira uporabo različnih podatkovni virov. Vtičnik je že dosegljiv za namestitev iz uradnega kataloga.

**Ključne besede:**

Grafana

vremenski podatki

veter

roža vetrov

DTR

SUMO

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.9

# 1 Uvod

Številni tehnološki procesi so odvisni od vremena. Na primer, vreme zelo vpliva na projekte v energetiki, izpostavimo lahko delovanje hidro, sončnih in vetrnih elektrarn. Za učinkovito vodenje takšnih procesov je potrebno natančno spremljanje sprotnih vremenskih podatkov in tudi vremenskih napovedi. Običajno se vremenski podatki zbirajo v lokalni bazi in se uporabljajo med produkcijo, pa tudi pri planiranju, optimiziranju, modeliranju in raznih analizah. Pogosto je potrebna vizualizacija.

Dinamično ocenjevanje prenosne zmogljivosti (DLR, angl. dynamic line rating oz. bolj splošno DTR, angl. dynamic thermal rating) je ena od naprednih tehnologij v prenosnih in distribucijskih električnih omrežjih, ki je zelo odvisna od vremenskih podatkov. Temelji na izračunu termične obremenitve daljnovodov in transformatorjev. Še posebej zanimiva je indirektna metoda, pri kateri dobimo rezultate brez namestitve dragih senzorjev, le z upoštevanjem vremenskih dejavnikov, ki vplivajo na segrevanje in hlajenje posameznih elementov. Slovenski operater prenosnega omrežja ELES je eden prvih TSO v Evropi, ki je indirektno metodo za dinamično ocenjevanje prenosnih zmogljivosti vpeljal v produkcijsko obratovanje. Njihova rešitev, ki so jo razvili skupaj s partnerji, se imenuje Sistem za ugotavljanje meja obratovanja (SUMO) [1]. SUMO je bil v zadnjem času s pomočjo podjetja Operato nameščen tudi v nekaj pilotnih projektih v tujini. Med novejšimi uporabami SUMO lahko izpostavimo še projekt TrafoFlex, katerega nosilec je bil SODO – Sistemski operater distribucijskega omrežja z električno energijo in ki se je ukvarjal z dinamičnim ocenjevanjem zmogljivosti distribucijskih transformatorjev [2].

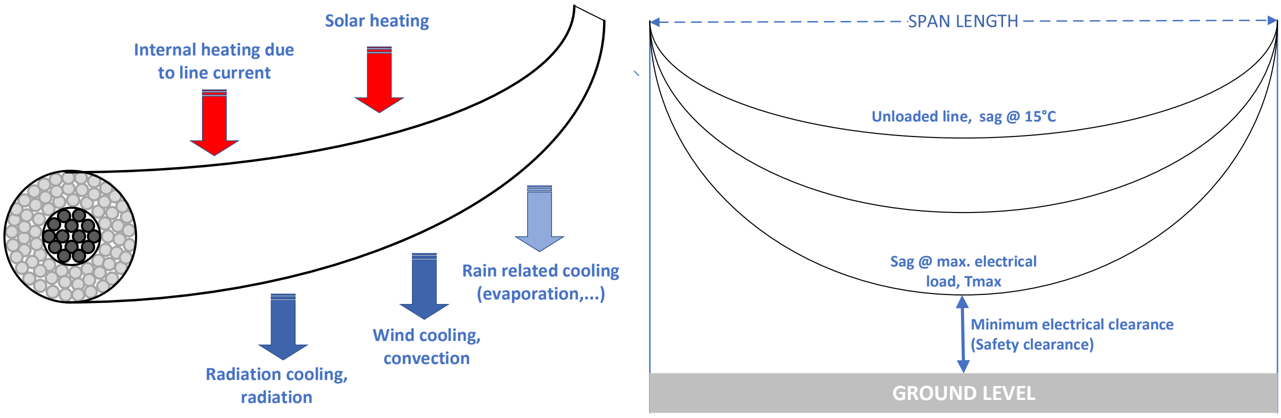
Polega izračuna DTR sta osnovni funkcionalnosti SUMO tudi vizualizacija ter analiza vhodnih podatkov in rezultatov. Analiziramo lahko tako zgodovinske podatke kot tudi napovedi za prihodnost. Trenutna različica SUMO 3 te funkcionalnosti doseže z uporabo odprtokodnega orodja Grafana (https://grafana.com/). Je zelo razširjen in dobro vzdrževan produkt. Na voljo je komercialna podpora. Najbolj primerna je za vizualizacijo časovnih serij, zmore pa prikazovati tudi tabelarične podatke. Podpira mnogo različnih vrst prikazov, ki jih lahko uporabniki dopolnijo s svojimi vtičniki.

V nadaljevanju članka je najprej podana motivacija za razvoj lastnega vtičnika v Grafani, šele nato sledi tehnični opis razvoja. Za izvedbo je bila uporabljena tehnologija ReactJS (https://react.dev/). Začetni del članka je nekoliko širši in nazorno pokaže, kako so kompleksne rešitve v informatiki odvisne od majhnih sestavnih delov kot je na primer vtičnik za vizualizacijo.

Opisan vtičnik prikazuje podatke o vetru in se uporablja v sklopu produkta SUMO. Ker pa je narejen zelo splošno, je uporaben tudi v številnih drugih projektih povezanih z vremenskimi podatki. Objavljen je v uradnem katalogu vtičnikov za Grafano.

# 2 Vremenski podatki v SUMO

Osnovna ideja tehnologije DTR je upoštevati vremenske razmere vzdolž vodnikov in ob transformatorjih tako, da omogočimo čim večji prenos energije ob še varnem obratovanju. Varnost obratovanja je pogojena s tem, da se zaradi segrevanja ne zgodi deformacija materiala, pri daljnovodu pa je pomembno tudi, da se ne poveča preveč njegov poves (slika 1). Uporaba DTR poveča varnost in pogosto tudi prepustnost prenosa energije v primerjavi s situacijo, ko za obratovanje uporabljamo statično mejo. Za vodnike je v Sloveniji statična meja izračunana ob predpostavki zunanje temperature 35 ˚C, hitrosti vetra 0,6 m/s s pravokotnim vpadnim kotom in sončnim obsevanjem 900 W/m$^2$. Statična oz. obratovalna meja pri transformatorjih je pogojena z njihovo zgradbo (npr. tip hladilnega olja) in izvedbo transformatorja (zidana izvedba, pločevinasta izvedba, izvedba v stavbi, jamborska izvedba, itd.). Ker pa so v večini časa vremenski pogoji ugodnejši od tistih upoštevanih za izračun statične meje, lahko ta dodatni pas izkoristimo za povečanje prenosne zmogljivosti, ob predpostavki seveda, da imamo dobre algoritme, zanesljiv informacijski sistem in natančne mikro-skalne vremenske podatke.

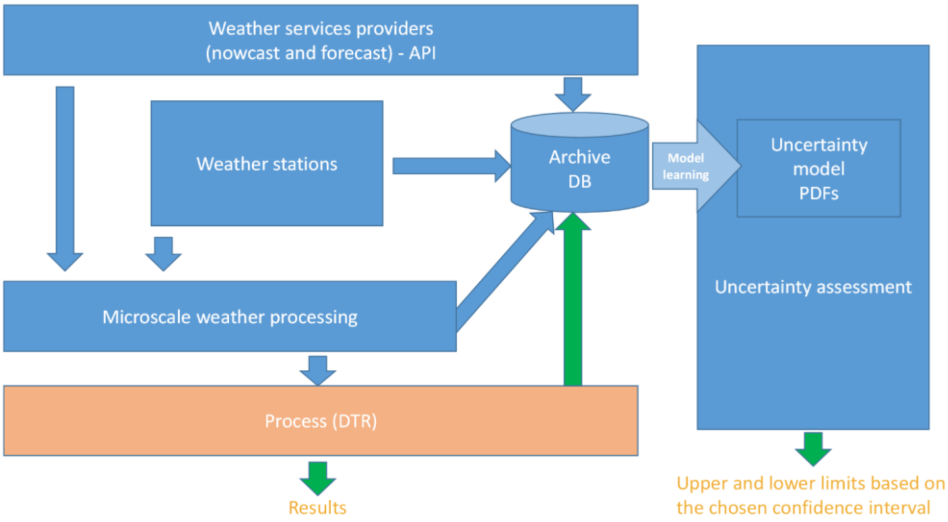


Slika 1: Vpliv vremenskih razmer na vodnike daljnovoda.

Vreme je skupek pojavov in stanj v ozračju, ki nas obkroža. V vsakodnevni rabi govorimo o vremenu tik nad površjem (tipično 2 m nad površjem), vendar pa so te vremenske razmere rezultat zelo dinamičnih in tudi kaotičnih procesov vzdolž celotne atmosfere do višine 10 km in še višje. Vremenske veličine je težko natančno meriti, kaj šele napovedovati. V sistemu SUMO potrebujemo mikro-skalne podatke (s prostorsko ločljivostjo okoli 200 m), ki jih pridobimo z izračunom na osnovi podatkov iz globalnih vremenskih modelov (tipična prostorska ločljivost 5 km) in meritev iz vremenskih postaj ob daljnovodih in transformatorjih (slika 2). V praksi imamo merjene podatke le za majhno število točk (vremenske postaje so velik strošek), vsi ostali podatki o trenutnem vremenu (angl. nowcast) so rezultat zgoraj omenjenega izračuna in zato vsebujejo napako. Napovedi vremena (angl. forecast) se izdelajo na osnovi podatkov o trenutnem stanju (ki vsebujejo napako) in omejeno zanesljivih rezultatov iz kompleksnih modelov, ki napovedujejo procese v celotni atmosferi v prihodnosti (zaradi zahtevnosti se taki izračuni tipično izvedejo le nekajkrat na dan).

SUMO uporablja algoritem DiTeR, ki so ga razvili na Inštitutu Jožef Stefan. Pri praktični uporabi se algoritem vsako minuto ločeno izvede za več tisoč lokacij. Za vsako lokacijo se z numeričnimi metodami pridobi rešitev nelinearnega sklopljenega sistema parcialnih diferencialnih enačb, ki izhajajo iz fizike termo-dinamskih sistemov. Problem ni rešljiv analitično. Uporaba matematičnega modela seveda vnese dodatne negotovosti glede rezultatov.

Ker se rezultati DTR uporabijo za upravljanje realne (in kritične!) infrastrukture elektro-energetskega sistema je pomembno, da se negotovost rezultatov ovrednoti. To izvedemo na osnovi zadostne množice zgodovinskih podatkov iz katerih izluščimo povezavo med vhodnimi podatki in napako rezultatov.

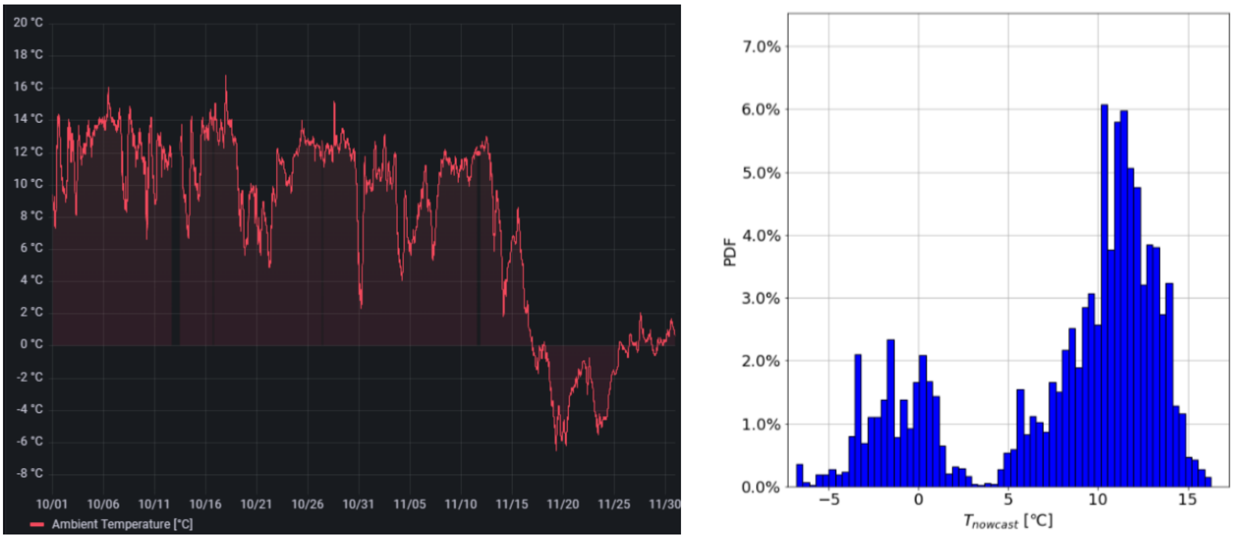


Slika 2: Uporaba vremenskih podatkov v SUMO.

Pri izvedbi DLR z indirektno metodo, kot je na primer SUMO, je za učinkovito vodenje sistema potrebno redno izvajati analize vhodnih vremenskih podatkov. Najpomembnejše primeri so:

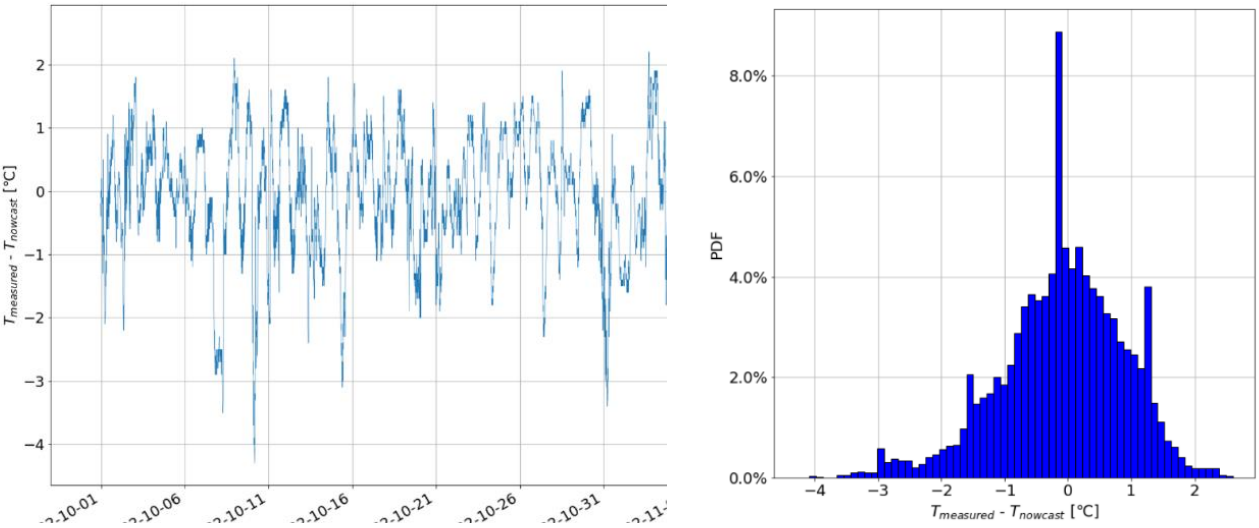
- s pomočjo analize vremenskih podatkov primerjamo različne vire (ponudnike) in se odločamo, kateri ponudnik bo najbolje zadostil potrebam izračuna DTR;
- analiza vremenskih podatkov pomaga določiti območja, za katera so podatki globalnih vremenskih modelov najmanj natančni in jih je zato smiselno opremiti z vremenskimi postajami;
- analiza vremenskih podatkov pomaga raziskovalcem pri oblikovanju boljšega modela za ocenjevanje negotovosti rezultatov.

Če se osredotočimo le na analizo nowcast podatkov (analiza napovedi ima še dodatno dimenzijo časa), sta najbolj osnovna pripomočka graf s časovnim potekom dogajanja in graf z distribucijo vrednosti (slika 3).



**Slika 3: Vizualizacija vremenskih podatkov.**

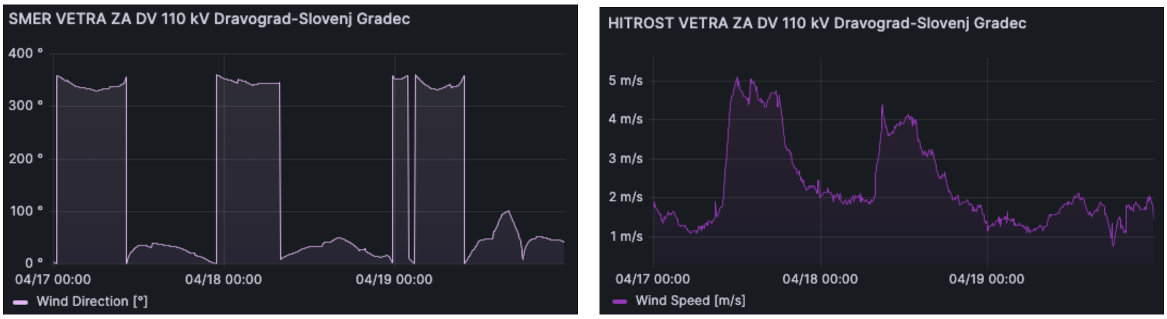
Če primerjamo dva vira podatkov, lahko uporabimo enak pristop, le da tokrat na grafih prikažemo časovni potek in distribucijo razlike obeh vrednosti (slika 4). Če je ena od vrednosti merjena oz. referenčna, potem lahko razliko vrednosti imenujemo kar napaka vhodnih oz. izhodnih podatkov.



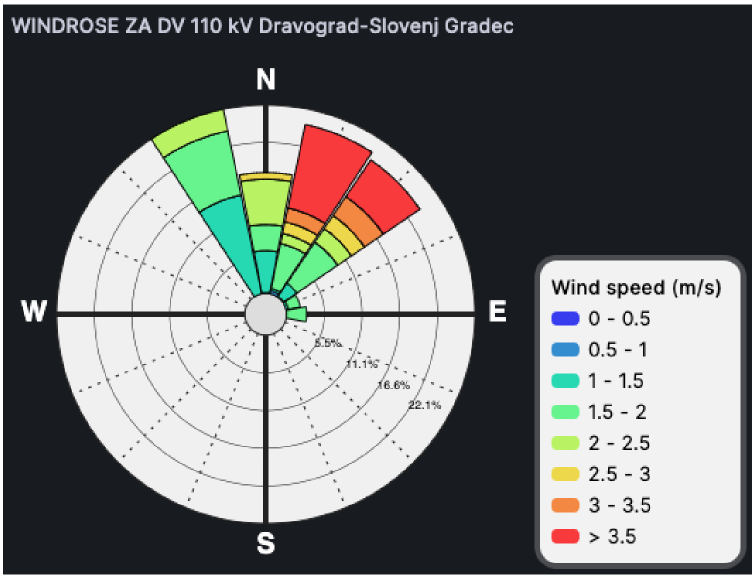
**Slika 4: Primerjava dveh serij vremenskih podatkov.**

## 3 Vizualizacija podatkov z rožo vetrov

Veter je za razliko od večine drugih vremenskih veličin vektor, torej ima hitrost in smer. Dodatno je smer vetra omejena z zalogo vrednosti od 0 do 360 stopinj. Ko vrednost za smer vetra naraste do 360 stopinj, preskoči na vrednost 0 stopinj, kar se na črtnih grafih izraža kot zelo moteča nezveznost (slika 5). Bolj pregledno je, če hitrost in smer vetra analiziramo na skupnem krožnem grafu (slika 6), ki se imenuje roža vetrov [3] (angl. Windrose [4]).



**Slika 5: Prikaz hitrosti in smeri vetra z dvema ločenima črtnima grafoma.**



**Slika 6: Prikaz hitrosti in smeri vetra z rožo vetrov.**

Kot ponazarja slika **Napaka! Vira sklicevanja ni bilo mogoče najti.**, je roža vetrov sestavljena iz več režnjev, v sak reženj pa iz več odsekov. V večini prikazov je število režnjev deljivo s številom štiri, ker s tem dobimo simetričen prikaz. Roža vetrov lahko interpretiramo kot dvo-dimenzionalni histogram, ki združuje obe komponenti vetra, hitrost in smer.

Roža vetrov najprej razdeli nabor podatkov po smeri vetra v prej omenjene režnje. Ti predstavljajo prvo dimenzijo dvodimenzionalnega histogram. Vsaka podmnožica podatkov znotraj intervala, ki ga določa režen<br>j, pa se glede na hitrost vetra nadalje razdeli v odseke (glej legendo, ki je enaka za vse odseke). Vsak odsek (z izjemo najbolj zunanjega, ki predstavlja navzdol omejen interval) predstavlja navzdol in navzgor omejen interval hitrosti vetra, ter je na grafikonu obarvan z unikatno barvo. Vsak interval hitrosti vetra ima unikatno barvo in barva v vseh režnjih predstavlja isti interval. Odseki predstavljajo drugo dimenzijo dvodimenzionalnega histograma. Da pa lahko histogram interpretiramo, se frekvenca pojavitev podatkov znotraj posameznega intervala (odseka) ponazori z relativno velikostjo odsekov. Velikost odsekov je relativna tako, da največji režen<br>j zasede celoten polmer grafikona. Na sliki **Napaka! Vira sklicevanja ni bilo mogoče najti.** je največji režen<br>j tisti, ki je levo od oznake N (ki o značuje kot 0°), torej v interval med 326,25° ter 337,5° spada največ vrednosti. Nadaljnja interpretacija nam pove, da se znotraj tega intervala največkrat pojavljajo vrednosti med 1 in 1.5 m/s.

# 4 Grafana

Grafana je močno orodje za vizualizacijo podatkov, ki omogoča analizo in spremljanje različnih virov podatkov v realnem času. S svojim uporabniku prijaznim vmesnikom omogoča ustvarjanje privlačnih in interaktivnih prikazov, ki pripomorejo k lažjemu razumevanju kompleksnih podatkovnih nizov.
Glavni gradniki Grafane so podatkovni prikazi (angl. dashboards). Ti so sestavljeni iz posameznih grafikonov (angl. panels) ali drugih prikaznih gradnikov, kot sta na primer prikaz vrednosti ter prikaz seznamov. Vse omenjene gradnike lahko vidimo na sliki 7. Na vrhu podatkovnega prikaza imamo grafikon za izris časovnih vrst, kjer vidimo izrisane tri grafe (modra, rdeča in rumena črta). Pod njim je vodoravno razvrščenih pet prikazov vrednosti (*I [A]*, *ITH [A]*, …). Na dnu pa lahko vidimo (sicer ne cele) štiri sezname (INPUT DATA, DLR RESULTS, …).

**Slika 7: Primer prikaza v Grafani.**

Grafana podpira številne vire podatkov, vključno z bazami podatkov, oblačnimi storitvami, merilniki in raznimi orodji za spremljanje stanj sistemov, kot je na primer Prometheus. Omogoča tudi različne vrste vizualizacije, na primer črtne, tabelarne in geografske predstavitve, ter mnogo drugih vrst grafikonov.

Poleg vizualizacije podatkov Grafana ponuja tudi napredne funkcije, kot so opozorila in alarmi, ki uporabnike obveščajo o kritičnih dogodkih ali mejnih vrednostih. Integracija s sistemskimi orodji za sledenje napakam in spremljanje zmogljivosti omogoča hitro odkrivanje težav in odpravljanje napak. S svojimi sprožitelji spletnih dogodkov (angl. webhooks) omogoča integracijo obveščanja v druge aplikacije kot so na primer Slack, Discord ter Microsoft Teams. Slednjega smo pri Operato uporabili za opozarjanje ob izpadih merilnih postaj v sistemu SUMO.

Sodobna različica Grafane vključuje tehnologijo ReactJS. Gre za odprtokodno JavaScript knjižnico, ki se uporablja za gradnjo uporabniških vmesnikov. Njeno vzdrževanje koordinira podjetje Meta. Zasnovana je na konceptu komponent, kar pomeni, da lahko množico različnih gradnikov združujemo v kompleksne uporabniške vmesnike. Ravno ta lastnost tehnologije ReactJS omogoča enostavno izdelavo lastnih vtičnikov za Grafano, ker je potrebno samo zasnovati nov gradnik, ki ga Grafana potem samodejno integrira v svoj uporabniški vmesnik.

# 5 Izvedba vtičnika

## 5.1 Ideja in načrtovanje vtičnika

Ideja je bila preprosta, želimo grafikon v Grafani, ki na podlagi zbranih podatkov izriše diagram roža vetrov. Omogočati mora povezovanje vsaj na podatkovno bazo PostgreSQL, ki se uporabljamo v SUMO, vendar bi bilo priporočljivo, da podpira čim širši nabor podatkovnih baz. Želimo tudi da je vtičnik odziven ter se posodablja v realnem času brez potrebe po osveževanju strani. Grafana namreč omogoča nastavitev osveževanja prikaza v izbranem časovnem intervalu. Želeli smo tudi možnost prilagajanja prikaza ter spreminjanje barvne sheme, torej dve funkcionalnosti, ki smo ju najbolj pogrešali pri obstoječih rešitvah.

Prvi korak pri razvoju kakršne koli programske rešitve je načrtovanje. Z dobrim načrtovanjem se lahko izognemo mnogim težavam, ki nas čakajo v nadaljnjem procesu razvoja. Pa vendar je vse težave težko predvideti, zato moramo vedno najti dobro razmerje med časom, ki ga porabimo za načrtovanje, ter koristjo, ki nam jo načrtovanje vrača. Zaradi slabega poznavanja tehnologije ReactJS smo fazo načrtovanja skrajšali na minimum in se dokaj hitro podali v razvoj. Načrt za vtičnik je bil zastavljen na precej abstraktnem nivoju, ker smo se s tehnologijo ReactJS srečali prvič. Vedeli smo kakšen rezultat želimo, vendar zaradi nepoznavanja tehnologije nismo mogli natančno načrtovati specifik izvedbe vtičnika.

## 5.2 Pridobivanje podatkov

Na začetku je bila ena glavnih skrbi podpora za različne tipe podatkovnih baz. A smo hitro ugotovili, kako nas modularna zasnova Grafane elegantno reši teh skrbi. V ozadju vseh funkcionalnosti Grafane so vtičniki, ki se delijo v tri kategorije. *Aplikacijske razširitve* nas tokrat niso zanimali, ker pokrivajo razširitve same platforme. Potem pa so tu še *grafikoni*, te vrste je tudi naš vtičnik roža vetrov in *podatkovni viri*. Jasno razlikovanje med tema dvema kategorijama vtičnikov pomeni, da Grafana strogo ločeno obravnava pridobivanje in prikazovanje podatkov.

Podatkovni viri so vtičniki zasnovani za povezovanje na podatkovne vire, naj bodo to podatkovne baze (PostgreSQL, MySQL, MongoDB, ...), posrednike sporočil (Apache Kafka), tabele (Google Sheets) ali celo git repozitoriji (Gitlab in Github). Ti vtičniki se zaganjajo na strežniškem delu platforme ter glede na podane zahteve uporabnika iz podatkovnega vira pridobijo podatke. Te podatke nato v posebni podatkovni strukturi, imenovani *Dataframe*, pošlje odjemalcu, ki jih izriše ali kako drugače prikaže z uporabo grafikonov. Tak pristop zagotovi varnost, ker se odjemalcu nikoli ne pošiljajo poverilnice potrebne za dostop do podatkov v podatkovni bazi.

## 5.3 Orodja in razvojno okolje

Za razvoj smo uporabili Visual Studio Code, ki je s svojim širokim naborom razširitev priljubljeno orodje za razvoj spletnih tehnologij. Potrebovali smo tudi razvojno kopijo Grafane, ki jo podobno kot v produkciji, postavimo z uporabo tehnologije Docker. Ta tehnologija je enostavna za uporabo, razlikovati moramo le med dvema osnovnima pojmoma: slika in vsebnik. Slika Docker (angl. Docker image) je posnetek stanja zelo okrnjenega navideznega računalnika z minimalno namestitvijo programov in konfiguracij potrebnih za zagon aplikacije. Vsebnik Docker (angl. Docker container) pa je dejansko izvedena in izolirana instanca, ki temelji na sliki Docker. Vsebnik Docker si lahko torej predstavljamo kot minimalističen navidezni računalnik. Razlika med vsebnikom Docker in pravim navideznim računalnikom je v njuni arhitekturi, saj si, za razliko od prave virtualizacije, vsebnik Docker deli jedro ter nekatere sistemske komponente z gostiteljevim operacijskim sistemom. Je pa zagon in upravljanje vsebnikov Docker zato dosti lažje in manj obremenjujejo gostiteljski sistem.

V ukazni lupini prenesemo željeno sliko z ukazom:

```
> docker pull grafana/grafana:latest
```

Po končanem prenosu pa zaženemo lokalno instanco Grafane z ukazom:

```
> docker run -d -p 8080:3000 -e "GF_DEFAULT_APP_MODE=development" -v
"/user/operato/development/windrose:/var/lib/grafana/plugins" grafana
```

S tem smo zagnali vsebnik Docker z vsemi potrebnimi knjižnicami in izvedljivimi datotekami potrebnimi za izvajanje Grafane, do katere lahko sedaj dostopamo na naslovu http://localhost:8080. Izbrali smo vrata 8080, vendar bi lahko izbrali poljubna vrata, le navesti jih je potrebno v ukazu z zastavico -p <VRATA>:3000. S tem preusmerimo ves promet iz vrat 8080 na gostitelju na vrata 3000 znotraj vsebnika. Vrata 3000 niso poljubna, ker Grafana privzeto posluša na vratih 3000.

Z zastavico -v smo mapo /user/operato/development/windrose na gostiteljskemu sistemu vpeli v datotečni sistem vsebnika na /var/lib/grafana/plugins. Tako bomo lahko vtičnik razvijali na gostiteljskem računalniku, obenem pa bo dostopen Grafani znotraj vsebnika.

Zastavica -e nastavi okoljsko spremenljivko `GF_DEFAULT_APP_MODE` na vrednost `development`. To sporoči Grafani, da gre za razvojno različico ter s tem omogočimo nalaganje vtičnikov, ki niso podpisani s strani razvijalca.

Preden začnemo razvijati lastne funkcionalnosti vtičnika moramo postaviti ogrodje, s pomočjo katerega bo Grafana zaznala, da gre za vtičnik in ga naložila v svoj seznam vtičnikov. V ukazni lupini smo se premaknili v mapo /user/operato/development/windrose, ki smo jo vpeli v vsebnik Docker. Tam smo izvedli sledečo serijo ukazov:

```
> npx @grafana/create-plugin@latest

> yarn install

> yarn dev
```

Prvi ukaz je ustvaril ogrodje našega novega vtičnika in od nas zahteval nekaj osnovnih podatkov o vtičniku kot so ime, organizacija, kratek opis ter tip vtičnika. Ponudil nam je tudi možnost, da nam postavi Github CI, vendar smo to možnost zavrnili, ker ne uporabljamo platforme Github. Nato z ukazom `yarn install` namestimo odvisnosti, z ukazom `yarn dev` pa zaženemo skripto, ki spremlja izvorno kodo ter ob vsaki spremembi ponovno izdela vtičnik. Po izvedbi ukazov, ki lahko, odvisno od hitrosti internetne povezave, trajajo tudi nekaj minut, je potrebno vsebnik ponovno zagnati, ker Grafana išče nove vtičnike le ob zagonu.

Na tej točki se v Grafani na seznamu vtičnikov pojavi naš nov vtičnik z imenom, ki smo ga podali ukazu za izgradnjo ogrodja vtičnika. Ime lahko pozneje, poleg mnogih drugih lastnosti vtičnika, spremenimo v datoteki `src/plugin.json`.

### 5.4 Razvoj vtičnika

Sedaj se prične postopek izdelave vtičnika, zato se premaknemo v podmapo `./src`, v kateri se mora nahajati vsa izvorna koda vtičnika. Grafana uporablja programski jezik TypeScript, ki je razširitev JavaScript, sintaksi obeh pa se med seboj nekoliko razlikujeta. Vtičnik bo zgrajen po principu modulov. Vsaka datoteka, ki predstavlja svoj modul, mora vsebovati vsaj eno vrstico `export`. Takšen modul lahko potem uvozimo kjerkoli drugje v programu.

Ukaz za izdelavo ogrodja je že pripravil osnutek izvorne kode. Vstopna točka je modul, ki ga predstavlja datoteka `module.ts`. in v kateri je funkcija `plugin`, ki vrne vtičnik kot gradnik ReactJS. Na vrhu te datoteke se najprej uvozi nekaj modulov, med drugimi tudi SimplePanel iz datoteke `components/SimplePanel.tsx` (pri uvozu lahko končnice datotek izpuščamo, zato bomo v kodi videli `./components/SimplePanel`). Vzorčna izvedba SimplePanel je primer enostavnega vtičnika, ki izriše barvni krogec.

Najprej si oglejmo del vrstice, ki izvozi naš vtičnik:

```
new PanelPlugin<SimpleOptions>(SimplePanel)
```

Gre za konstruktor, ki ustvari vtičnik. Podati mu moramo dva parametra, prvi je opis nastavitev, ki jih naš vtičnik podpira, drugi pa gradnik, ki bo izvajal vizualizacijo podatkov.

To vrstico smo spremenili v :

```
new PanelPlugin<WindroseOptions>(WindrosePanel)
```

Zato smo tudi pri uvozu iz `types.ts` dodali `WindroseOptions` ter namesto `SimplePanel` uvozili `WindrosePanel` iz `./components/WindrosePanel`. Najprej si oglejmo, kaj nam pove `WindroseOptions`. Gre za vmesnik (interface), ki opredeli nastavljive lastnosti našega vtičnika. V datoteki `module.ts` vidimo, da se poleg konstruktorja, kliče še ena funkcija:

```
.setPanelOptions((builder) => {  return builder ...
```

V tej funkciji se vrne objekt imenovan `builder`, kateremu lahko dodajamo nastavljive parametre vtičnika s klici funkcij. Med drugimi smo za naš vtičnik uporabili sledeče funkcije:

- addTextInput – za dodajanje tekstovnega polja
- addNumberInput – za dodajanje polja za vnos števila
- addBooleanSwitch – za vklopno stikalo
- addRadio – za radijski gumb (več možnosti med katerimi je lahko izbrana samo ena)
- addSelect – za izbirni seznam
- in še nekatere druge

Vsaka izmed naštetih funkcij prejme JSON objekt kot parameter v katerem so navedene lastnosti kot so opis, ime, vrednosti ter v katero polje ( v našem primeru `WindroseOptions`) se naj vnesena vrednost shrani. Te vrednosti so nato dostopne v gradniku.

Lastnosti vtičnika Operato Windrose, ki so nastavljive s pomočjo v uporabniškem vmesniku Grafane, so:

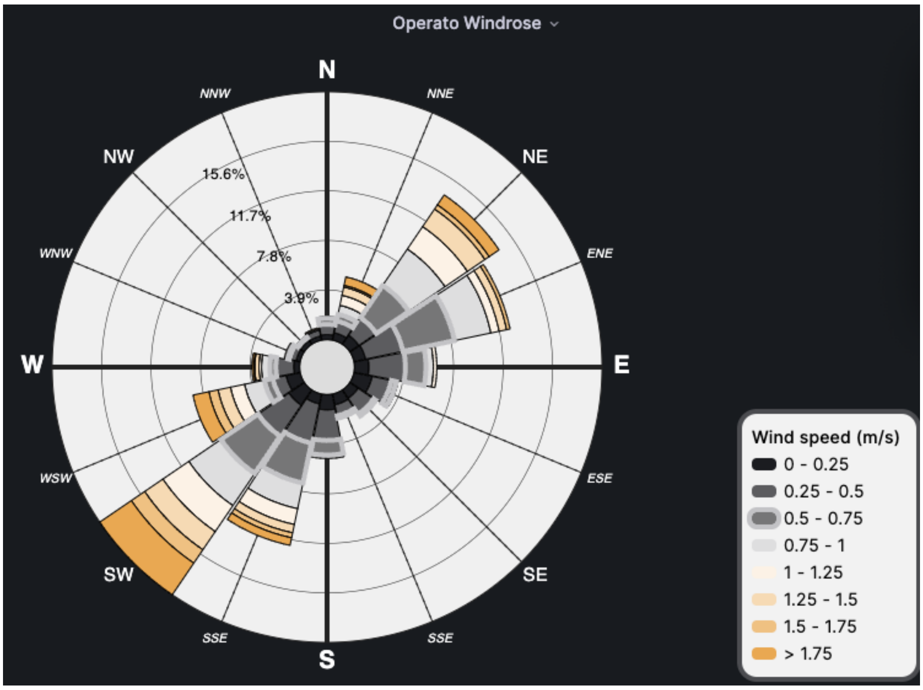
- barvna shema,
- število režnjev,
- število odsekov v posameznem režnju diagrama,
- velikost intervala hitrosti vetra v posameznem odseku režnja,
- število decimalnih mest v pojavnem oknu z opisom stanja (angl. tooltip),
- enota hitrosti vetra (km/h, m/s, mph, ...),
- izbira izpisa oznak (smeri neba ali stopinje),
- prikaz legende.

`WindrosePanel` je ReactJS gradnik, ki skrbi za izris našega diagrama roža vetrov. Zamišljen je bil kot vsebnik, ki bo skrbel za preprocesiranje podatkov ter usklajeval pretok podatkov med njegovimi sestavnimi deli. Sam je sestavljen iz dveh delov, ki sta prav tako ReactJS gradnika: Windrose ter WindroseLegend. Kot vtičnik tipa grafikon nima neposrednega dostopa do virov podatkov, zato mu Grafana preko parametrov poda vse potrebne podatke za delovanje. Prejme štiri parametre: nastavitve (options), podatke (data), širino (width) ter višino (height). V nastavitvah so shranjene vrednosti, ki so nastavljene v nastavitvah grafikona s strani končnega uporabnika in se lahko spreminjajo. Podatki do tisti, ki jih je Grafana pridobila iz podatkovnega vira. Širina in višina služita pri vizualizaciji za določanje velikosti komponent znotraj gradnika.

Kot vsi grafikoni v Grafani, je tudi naš Windrose grafikon združljiv z vsemi viri podatkov, ki jih Grafana ponuja. Edini pogoj je, da lahko viru podatkov določimo obliko podatkov, ki jih mora vračati. Grafana namreč ločeno na zalednem delu opravi pridobivanje podatkov, nato pa te podatke zloži v posebno podatkovno strukturo imenovano *Dataframe*. V bistvu gre za dvo-dimenzionalno tabelo podatkov, kjer stolpci predstavljajo vrednost posameznega atributa, vsaka vrstica pa predstavlja en vnos. Ta podatkovna struktura se pošlje k odjemalcu, ki jo posreduje vtičniku za vizualizacijo. Operato Windrose vtičnik predpostavlja, da bo prejeta podatkovna struktura sestavljena iz dveh stolpcev imenovanih *wind_direction* in *wind_speed*. Vse prejete podatke nato najprej razdeli v režnje glede na vrednost komponente *wind_direction* ter nato še v posamezne odseke znotraj režnjev na podlagi vrednosti komponente *wind_speed*. Zadnji korak predobdelave podatkov je izračun relativnih velikosti režnjev in posameznih odsekov.

WindrosePanel gradnik poskrbi tudi za barvno shemo grafikona. Uporabnik lahko v nastavitvah izbere eno izmed vnaprej pripravljenih barvnih shem, na podlagi katere se gradniki ustrezno obarvajo. Za barvne sheme uporabljamo posebno zmogljivost tehnologije ReactJS imenovano stanja (angl. state) [6]. Omogočajo enostavno proženje ponovnih izrisov gradnikov ob spremembi vrednosti stanja. V našem primeru so stanja barve posameznih odsekov ter barvnih indikatorjev v legendi. Vsem odsekom režnjev ter barvnim indikatorjem v legendi smo dodali poslušalca (angl. listener), ki čaka, da se uporabnik z miško premakne nad ta element uporabniškega vmesnika. Ob tem dogodku se v barvni shemi spremeni barva izbranega odseka, kar se odrazi v spremembi barve vseh odsekov tega vrednostnega razpona (tudi v drugih režnjih) in tudi pripadajočega indikatorju v legendi. Na sliki 8 je prikaz opisanega delovanja stanj, kjer smo šli z miško čez element v legendi, ki zaznamuje interval 0.5 – 0.75.

Izdelava samega gradnika za vizualizacijo je bila izvedena z uporabo HTML značk za vektorsko grafiko. Za krožne črte smo uporabili značko `<circle>` za režnje in odseke smo uporabili značko `<polygon>` ter za ravne črte značko `<line>`. Legenda je izdelana z uporabo HTML `<div>` značk ter z uporabo stilov CSS oblikovana in poravnana v desni spodnji kot starševskega gradnika.



**Slika 8: Primer spremembe barve z uporabo stanj.**

Uporaba vektorske grafike nam omogoča, da lahko uporabnik grafikon po želji povečuje in zmanjšuje, ob tem pa grafikon ohrani lep izgled brez zamegljenih robov, ki nastanejo pri povečavi rastrskih slik. Sprva smo z značko `<circle>` , ki smo ji nastavili belo barvo polnila, izrisali ozadje grafikona. Nato pa, prav tako z uporabo značke `<circle>`, izrisali črne koncentrične kroge. Število teh krogov je odvisno od absolutne velikosti grafikona znotraj okna brskalnika. Črte, ki kažejo smeri neba, smo izrisali z značko `<line>`, pri čemer smo število črt uskladili z izbrano nastavitvijo števila režnjev grafikona. Vsi tekstovni elementi pa so izrisani z značko `<text>`. Vsem HTML značkam za vektorsko grafiko je potrebno opredeliti, kje in kakšne oblike naj bo izrisan grafični element. Na primer, značka `<line>` zahteva štiri atribute x1, x2, y1 ter y2. Ti atributi definirajo dve točki med katerima naj se izriše črta. Podobno se uporablja značka `<circle>`, kjer moramo definirati točko, ki predstavlja središče kroga in podati polmer kroga. Še bolj enostavna je uporaba značke `<text>`, ki zahteva samo eno točko ter tekst, ki naj ga izpiše. Malo bolj zapletena pa je uporaba značke `<polygon>`. Ta zahteva tekstovni niz v katerem naštejemo sosledje točk med katerimi se naj izrišejo črte. Primer tašnega niza je: »1,1 2,2 3,2 1,1«. Takšen niz bo iz točke (1,1) narisal črto do točke (2,2), nato od (2,2) do (3,2), ter zaključil črto nazaj do (1,1). Mi smo `<polygon>` značko uporabili za izris odsekov režnjev grafikona. Z uporabo kotnih funkcij ter upoštevanjem pravil za translacijo in skaliranje, smo izračunali položaje točk. Vsak odsek je sprva trapez, ki ga definirajo štiri točke. Nato pa za lepši izgled zgornjo in spodnjo stranico najprej razdelimo na šestnajst delov, novo nastalim točka pa izračunamo položaj tako, da imajo vse enako oddaljenost od centra grafikona. Tako dobimo okrogle odseke režnjev, kot so predstavljeni na sliki **Napaka! Vira sklicevanja ni bilo mogoče najti.**.

### 5.5 Objava vtičnika

Po končanem razvoju vtičnika si želimo vtičnik prenesti iz razvojne v produkcijsko Grafano, vendar ta zavrne vsak vtičnik, ki ni podpisan. Zato je potrebno najprej zgraditi vtičnik z ukazom:

```
> yarn build
```

Za podpisovanje je potrebno ustvariti račun na www.grafana.com ter slediti navodilom za izdelavo ključa API. Tega moramo pred podpisovanjem shraniti v spremenljivko okolja GRAFANA_API_KEY (to lahko storimo z uporabo ukaza export), nato pa lahko poženemo ukaz:

```
> npx @grafana/sign-plugin@latest --rootUrls http://localhost:3000
```

S tem smo naš vtičnik podpisali z zasebnim varnostnim nivojem. To pomeni, da lahko naš vtičnik namestimo le na Grafane, ki so dostopne na naslovih, ki smo jih navedli z zastavico `--rootUrls` (navedemo jih lahko več, a morajo biti ločene z vejico).

Če pa želimo vtičnik objaviti v uradnem katalogu vtičnikov, pa moramo vtičnik dodatno pripraviti, kot je opisano v navodilih na spletni strani. To vključuje odstranjevanje vseh morebitnih občutljivih ali zaupnih podatkov v izvorni kodi, kot so na primer gesla ali API ključi. Spisati moramo navodila za uporabo vtičnika, ter dodati opis licence, pod katero želimo objaviti vtičnik. Naš vtičnik je objavljen pod licenco Apache License Version 2.0, January 2004. Preveriti moramo tudi samo delovanje vtičnika ter opraviti validacijo vtičnika z orodjem `grafana/plugin-validator-cli`. Grafana ima zelo dobro dokumentacijo, v kateri je podrobno opisan vsak omenjen korak [5].

Ko smo se prepričali, da v vtičniku ni prisotnih podatkov, za katere ne želimo, da do njih dobi dostop širša javnost, lahko na uporabniškem računu na www.grafana.com ustvarimo zahtevo za objavo vtičnika, kjer od nas zahtevajo tri povezave:

- arhiv zip z izdelanim vtičnikom,
- pripadajočo zgoščeno vrednost sha1, ter
- povezavo do izvorne kode (repozitorij git).

Po oddaji zahtevka bo recenzent na Grafani preveril skladnost našega vtičnika in nam dodelil pravico objave vtičnika z javnim varnostnim nivojem. Nato bo od nas zahteval posodobitev zahtevka. Zadnji korak je tako ponovni podpis vtičnika z ukazom:

```
> npx @grafana/sign-plugin@latest
```

Tokrat smo izpustili zastavico `--rootUrls`, kar pove skripti da gre za javni podpis (brez odobrenega varnostnega nivoja podpis ne bi uspel). Recenzent nam nato odobri objavo in v roku 24 ur je vtičnik dostopen na uradnem katalogu vtičnikov Grafane.

Za posodobitev vtičnika se izvede podoben postopek. Izpolni se vlogo za posodobitev različice vtičnika ter odda enake povezave kot pri prvi objavi vtičnika. Le da tokrat vtičnik že prvič podpišemo z javnim varnostnim nivojem.

# 6 Zaključek

Kompleksne rešitve v IT so toliko dobre, kolikor so dobri njihovi večji in manjši sestavni deli. Velja tudi za varnost, učinkovitost in za vse druge lastnosti. Uporabniki pogosto ocenjujejo celotno rešitev na osnovi nekaj najpogosteje uporabljanih funkcionalnost, ki niso nujno najbolj kompleksni deli rešitve. V primer SUMO je ena od bolj izpostavljenih funkcionalnosti vizualizacija vremenskih podatkov.

Operato nudi svojim strankam pred-pripravljeno vizualizacijo z uporabo Grafane. Grafana je zelo ustrezna komponenta naše rešitve. Je široko uporabljeno orodje v industriji, akademskih krogih in drugih sektorjih, kjer je pomembno vizualizirati in analizirati podatke v realnem času. Zaradi svoje odprtokodne narave je prilagodljiva in omogoča širok nabor razširitev ter integracij z drugimi orodji in platformami. Zaradi vseh naštetih lastnosti je Grafana odlična izbira za vse, ki si želijo prilagodljivo in zmogljivo rešitev za vizualizacijo podatkov.

Prikaz večine podatkov smo enostavno izvedli z privzetimi grafikoni v Grafani, manjkala pa nam je dobra vizualizacije smeri in hitrosti vetra. Ravno veter pa ima največji vpliv na delovanje algoritma in si ga uporabniki zato želijo podrobno analizirati. Ker v analizah podatkov, ki jih za uporabnike izvajamo s programskima jezikoma Python in R, uporabljamo diagram roža vetrov, smo želeli tako prikazati podatke tudi v Grafani. Vendar med obstoječimi vtičniki nismo našli primernega za uporabo v naši rešitvi. Nekateri so bili zastareli, drugi opuščeni sredi razvoja, nekateri pa so kar obstali v zgodnji fazi razvoja. Zato smo se odločili vložiti čas v razvoj lastnega vtičnika za izris diagrama roža vetrov, ki smo ga kasneje objavili tudi javno objavili.

Za izvedbo naloge je bilo potrebno osnovno poznavanje jezika TypeScript, tehnologije ReactJS in formata SVG za opis vektorske grafike. Vsa uporabljena orodja so prosto dostopna.

Podjetje Operato svoje rešitve gradi na odprtokodnih in prosto dostopnih tehnologijah in orodjih. Z izvedbo in javno objavo vtičnika za napredno vizualizacijo vremenskih podatkov v Grafani smo se sedaj skupnosti oddolžili s pripravo zanimivega izdelka, ki ga lahko kdorkoli vključi v svoje rešitve. Ob času pisanja tega članka, nekaj manj kot mesec dni po objavi prve različice vtičnika, je na uradnem katalogu vtičnikov Grafane zbral že več kot 500 prenosov.

# Literatura

[1] KOSEC Gregor, MAKSIĆ Miloš, DJURICA Vladimir. »Dynamic Thermal Rating of Power Lines – Model and Measurements in Rainy Conditions«. International Journal of Electrical Power & Energy Systems, 2017.

[2] MEOLIC Robert, LENKO Miha, SOUVENT Andrej. »Napredne IT rešitve za pospeševanje zelenega prehoda«. In Proc. OTS 2022, pp. 206-217, 2022.

[3] Roža vetrov. Wikipedija. https://sl.wikipedia.org/wiki/Roža_vetrov

[4] Wind rose. Wikipedia. https://en.wikipedia.org/wiki/Wind_rose

[5] Grafana Labs. Grafana plugin developer's guide. https://grafana.com/docs/grafana/latest/developers/plugins/

[6] Meta Open Source. Built-in React Hooks. https://react.dev/reference/react

# Neprekinjena integracija in postavitev MAUI mobilnih aplikacij

Alen Granda

Alcad, Zgornja Bistrica, Slovenija
alen.granda@alcad.si

S prihodom ogrodja MAUI je v svet mobilnih informacijskih rešitev razvitih v tehnologiji .NET prispela možnost avtomatizacije postopka gradnje in izdaje novih verzij programske opreme s pomočjo vsebnikov, ki se izvajajo pod operacijskim sistemom Linux. Posledično razvoj poteka bolj produktivno, hitreje zaznamo morebitne napake v programski kodi ter poskrbimo za boljšo uporabniško izkušnjo. V prispevku predstavimo poglavitne lastnosti ogrodja MAUI, opišemo primerjavo z njegovim predhodnikom Xamarin ter izpostavimo učinkovitostne izboljšave. Prikažemo način, kako avtomatizirati gradnjo in izdajo novih verzij produkcijskih mobilnih aplikacij razvitih v ogrodju MAUI s pomočjo metode neprekinjene integracije in postavitve v platformi GitLab. Metoda se v celoti izvaja v vsebniku pod operacijskim sistemom Linux. Predstavimo tudi praktični primer samodejnega posodabljanja demonstracijske mobilne aplikacije ob izdaji nove verzije z opisanim postopkom. Dodani so še koristni napotki ob implementaciji predstavljenega demonstracijskega primera.

**Ključne besede:**
MAUI
CI/CD
GitLab
mobilne aplikacije
.NET

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.10

# 1 Uvod

Razvoj nativnih mobilnih aplikacij je še danes pogost način implementacije rešitev poslovnih problemov. Z nativnostjo poskrbimo za učinkovitost, imamo večji nadzor nad napravo ter izkoristimo vse njene prednosti. Zaradi uporabe vse bolj popularnega agilnega pristopa k razvoju programske opreme se nove funkcionalnosti kot tudi popravki zelo pogosto integrirajo k obstoječi mobilni aplikaciji. Hkrati s hitrimi nadgradnjami je potrebno poskrbeti tudi za dostavo novih različic programske opreme h končnemu uporabniku. Žal je lahko v primeru razvoja nativnih mobilnih aplikacij na podlagi ogrodja Xamarin izvajanje posodobitev zamudno. Nadgradnje najpogosteje potekajo ročno, od grajenja projekta, generacije paketa aplikacije ter vse do nalaganja h končnemu uporabniku. Razvojne smernice iz sorodnih področij v ta namen koristijo metodo neprekinjene integracije in postavitve (ang. CI/CD) programske opreme, ki poskrbi za avtomatizacijo izvajanja posodobitev in razbremenitev razvijalca od tega procesa.

V prispevku predstavimo način, kako integrirati metodo neprekinjene integracije in postavitve mobilnih informacijskih rešitev na osnovi ogrodja .NET MAUI. Pripravili smo demonstracijski projekt v ogrodju .NET MAUI, ki je namenjen operacijskemu sistemu Android. Dodali smo primer konfiguracije cevovoda CI/CD na platformi GitLab. Ob izdaji nove verzije programske opreme se paket mobilne aplikacije samodejno ustvari in shrani na register paketov na repozitoriju platforme GitLab s pomočjo zaganjalnika GitLab Runner, ki zažene registrirane naloge v cevovodu CI/CD. Mobilna aplikacija ob vsakem zagonu s pomočjo poizvedb na aplikacijski programski vmesnik GitLab REST API preveri, ali je na voljo nova verzija ter v pritrdilnem primeru ponudi uporabniku avtomatsko posodobitev. Z opisanim načinom smo poskrbeli za samodejno posodabljanje mobilne aplikacije brez uporabe tretjih ponudnikov kot je npr. Google Play. Postopek je uporaben predvsem za poslovne rešitve, ki so namenjene privatnim podjetjem. Dodatno v prispevku omenimo še primerjavo z ogrodjem Xamarin, učinkovitostne primerjave ter prednosti oziroma slabosti tehnologije.

# 2 Ogrodje MAUI

## 2.1 Predstavitev

Ogrodje .NET MAUI (ang. Multi-platform App UI) je multi-platformno usmerjeno ogrodje za razvoj nativnih mobilnih ter namiznih aplikacij z uporabo programskih jezikov C# ter XAML [1]. Namenjeno je razreševanju več izzivov, s katerimi se razvijalci med razvojem programske opreme soočajo, kot so:

- konstantno spreminjajoče se zahteve aplikacije,
- nove poslovne priložnosti in izzivi,
- kontinuirane povratne informacije med razvojem, ki pogosto vplivajo na obseg in zahteve aplikacije.

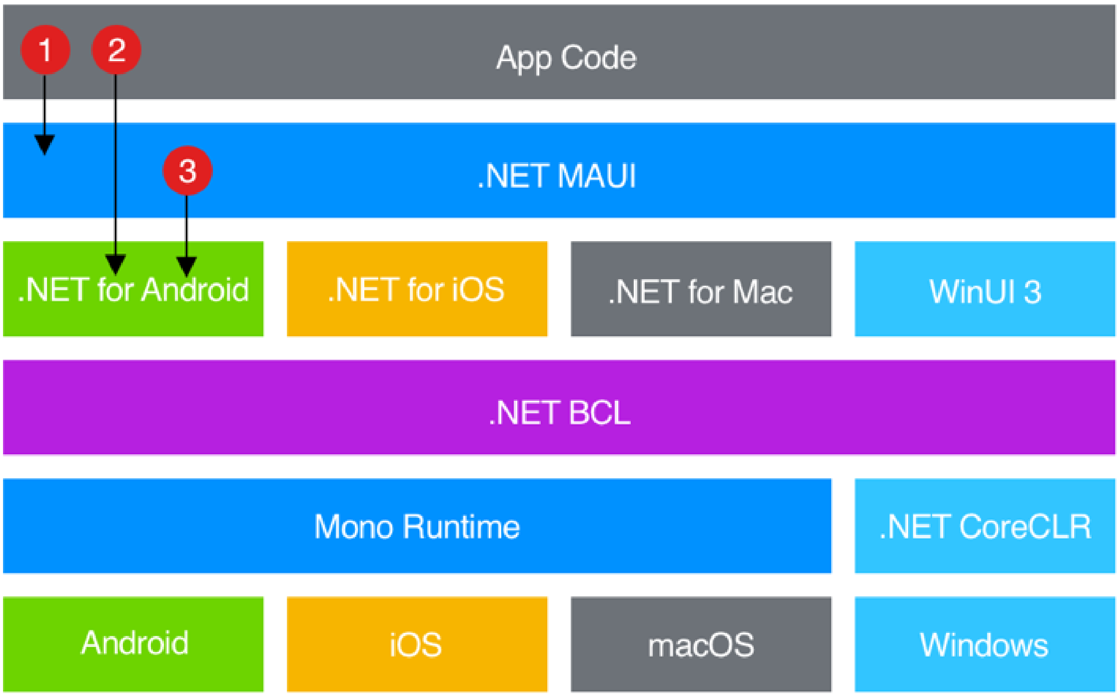
S tem v mislih je pomembno razvijati aplikacije, ki jih je mogoče sčasoma preprosto spremeniti ali razširiti. Oblikovanje takšne prilagodljivosti je lahko težavno, saj zahteva arhitekturo, ki omogoča neodvisen razvoj in testiranje posameznih delov aplikacije v izolaciji, ne da bi to vplivalo na preostali del aplikacije. Takšna arhitektura razvoja programske opreme zahteva razbitje aplikacije na več posameznih, šibko povezanih komponent, katere je možno enostavno integrirati v celoto. S tem pridobimo več prednosti [2]:

- preprost razvoj, testiranje in razširjanje funkcionalnosti aplikacije s pomočjo več ekip razvijalcev,
- delegacija opravil razvijalcem, ki imajo na posameznem področju največ izkušenj,
- zanesljiv nadzor nad posameznimi deli aplikacije zaradi ločitve odvisnosti med gradniki aplikacije, kot so poslovna logika, dostop do podatkov, avtentikacija ipd.

Ogrodje MAUI nam omenjene prednosti omogoča z uporabo več abstrakcij, ki združijo aplikacijske programske vmesnike operacijskih sistemov Android, iOS, macOS ter Windows v enoten vmesnik. Posledično je podprt razvoj deljene programske kode kot tudi razvoj enotnega uporabniškega vmesnika, ki deluje na vseh omenjenih platformah. Razvoj temelji na skupni bazi programske kode, ki je enaka za vse platforme in posledično pripomore k hitrejšemu in cenejšemu razvoju programske opreme. Ogrodje je odprtokodno in razvito na podlagi popularnega ogrodja za razvoj nativnih mobilnih aplikacij Xamarin.

Iz diagrama poteka interpretiranja programske kode na Slika opazimo, da nam ogrodje zraven abstrakcijskega nivoja, ki samodejno poskrbi za klic ustreznih metod v izhodiščnem operacijskem sistemu (označeno s številkama 1 ter 3 na sliki), ponuja tudi direkten dostop do aplikacijskih programskih vmesnikov naloženega operacijskega sistema, v kolikor bi to potrebovali (označeno s številko 2 na sliki 1). Natančneje, gradnja paketov aplikacije za željene operacijske sisteme poteka na sledeč način [1]:

- Android: izvorna programska koda se iz programskega jezika C# prevede v vmesni jezik (ang. intermediate language), ki se ob zagonu aplikacije na napravi prevaja instantno (ang. just-in-time),
- iOS: celoten projekt se prevede vnaprej (ang. ahead-of-time) v zbirno kodo ARM,
- macOS: ogrodje v tem primeru koristi tehnologijo Mac Catalyst s par dodatnimi posegi,
- Windows: v ta namen ogrodje MAUI koristi knjižnico Windows UI 3.



**Slika 1: Diagram poteka interpretiranja programske kode, ki komunicira s programskim aplikacijskim vmesnikom ogrodja MAUI**

Vir: [1].

### 2.2 Primerjava z ogrodjem Xamarin

Tako kot ogrodje MAUI, je tudi ogrodje Xamarin namenjeno razvoju nativnih aplikacij z deljeno programsko kodo, razvito v programskih jezikih C# ter XAML [3]. Omogoča grajenje aplikacij, ki se lahko izvajajo na več platformah. Uradno podprte platforme so Android, iOS ter Windows. Razvoj ogrodja Xamarin se je pričel že leta

2011 [4], nakar ga je leta 2016 kupilo podjetje Microsoft [5] in ga zelo kmalu integriralo v razvojno okolje Visual Studio za brezplačno uporabo [6].

Ogrodje Xamarin je projekt, ki je izoliran od celotnega sistema tehnologij .NET, saj deluje na podlagi izvajalnega okolja Mono. Ker je želja podjetja Microsoft unifikacija vseh njihovih orodij, izvajalnih okolij in programskih knjižnic, so se hitro začeli fokusirati v nadgradnjo ogrodja Xamarin ter tako prejšnje leto maja [9] širši javnosti predstavili ogrodje .NET MAUI kot njegovega naslednika. Prihod nove generacije razvoja multi-platformnih poslovnih rešitev je poleg integracije v celotno okolje družine .NET in posledično uporabe abstrakcijskega nivoja .NET BCL (ang. .NET Base class library) prinesel več zanimivih sprememb, ki so povzete v tabeli 1 [1, 7, 8, 10].

**Tabela 1: Primerjava ogrodij MAUI ter Xamarin.**

| | MAUI | Xamarin |
|---|---|---|
| Podprte platforme | Android 5.0+, iOS 10+, macOS 10.15+, WinUI, Tizen[1], Linux[2]. | Android 4.4+, iOS 9+, UWP, Tizen[1], macOS, GTK#[2], WPF[2]. |
| Struktura projekta | Enotna. | Projekt za posamezno platformo. |
| Ukazna vrstica | .NET CLI | .NET Framework |
| Osnovna tehnologija | .NET 6, .NET 7 | .NET Framework |
| Upravljanje virov | Enotni viri za vse platforme. | Viri za posamezno platformo. |
| Vroče posodabljanje (ang. hot reload) | Da. | Ne, vendar je eksperimentalno podprto. |
| Razvojne paradigme | MVVM, RxUI, MVU. | MVVM, RxUI. |
| Arhitektura nadzornih gradnikov | Arhitektura upodabljalnika (ang. renderer architecture). | Arhitektura upravljalnika (ang. handler architecture). |

Vir: [1, 7, 8, 10]

Posledica tovrstne integracije je med drugim tudi enovita uporaba grafičnih programskih knjižnic za razvoj uporabniškega vmesnika ter uporaba multi-platformnih programskih vmesnikov za dostop do nativnih funkcionalnosti naprav, kot so signali GPS, pospeškometer, stanje baterije, dostop do interneta ipd. [1].

Ena izmed prednosti napram ogrodju Xamarin je prav tako podpora tehnologiji Blazor Hybrid [21]. Tehnologija omogoča izvajanje gradnikov spletnega vmesnika nativno na končni napravi. To stori tako, da gradnike upodablja v vgrajenem nadzorniku spletnega ogleda (ang. embedded Web View), ki se izvaja neodvisno od spletnega brskalnika ali tehnologije WebAssembly [21]. Posledično lahko združimo programsko kodo spletnega vmesnika s programsko kodo nativnih aplikacij in poenostavimo razvoj poslovnih rešitev.

Za namen avtomatizacije posodobitev aplikacij je najpomembnejša novost podpora uporabi vmesnika ukazne vrstice .NET (ang. .NET CLI). Orodje omogoča razvoj, gradnjo, zagon ter izdajo aplikacij .NET na več platformah. Posledično ga lahko uporabimo v sklopu neprekinjene integracije in postavitve mobilnih aplikacij v vsebnikih, ki se izvajajo pod operacijskim sistemom Linux. Bolj podrobno je cevovod obrazložen v poglavju priprave praktičnega primera.

### 2.3 Učinkovitostni vidiki

Eden izmed ključnih motivacij za razvoj naslednika ogrodja Xamarin je tudi izboljšava učinkovitosti delovanja aplikacije ter zmanjšanje njene velikosti. Ob javni naznanitvi ogrodja MAUI so v ta namen objavili tudi izčrpno poročilo primerjave učinkovitosti aplikacij, razvitih na podlagi ogrodja Xamarin napram aplikacijam, zgrajenih na podlagi njegovega naslednika. Če povzamemo nekaj ključnih izboljšav ogrodja MAUI, ki temelji na tehnologiji .NET 6 [11]:

- pohitritev zagona zaradi kombinacije uporabe profiliranega grajenja strojne kode vnaprej (ang. profiled AOT),

---

[1] https://www.defect

[2] seznam sestavnih

- izboljšave v interoperabilnosti s programskim jezikom Java:
    - optimizirano klicanje konstruktorjev in metod programskega jezika C# iz programskega jezika Java z minimizacijo klica sistemske metode System.Reflection.Emit(),
    - reduciranje klicev programske kode v jeziku Java,
    - menjava klicev metode LayoutInflater.Inflate() s programskimi metodami,
- menjava splošnega gostitelja .NET (ang. .NET general host) za konfiguracijo, upravljanje dnevnika ter vključitve odvisnosti s specifično implementacijo za mobilne naprave,
- izogib uporabi knjižnice Newtonsoft.Json zaradi koriščenja interne knjižnice za serializacijo in deserializacijo System.Text.Json.

S prihodom tehnologije .NET 7 so v podjetju Microsoft šli še nekoliko dalje in predstavili še nekaj dodatnih izboljšav, kot so [12]:

- učinkovitostne izboljšave zaradi zmanjšanja nepotrebnih klicev metode View.Context(), minimizacije preskokov med programskima jezikoma C# in Javo, uporaba nativne implementacije kalkulacije orientacije smeri teksta,
- izboljšane metode za primerjavo nizov, pohitritev izvajanja regularnih izrazov z uporabo atributa [RegexGenerator],
- izboljšano prevajanje gradnikov XAML v vmesni jezik,
- ignoriranje metapodatkovnih datotek, namenjenih programskemu jeziku Kotlin, ki so posledica vse bolj pogoste implementacije Google-ovih programskih knjižnic družine AndroidX v programskem jeziku Kotlin.

Kljub omenjenim izboljšavam in analizam je tehnologija še mlada in jo je potrebno obravnavati z rezervo. Čeprav je že dobro leto dni na voljo širši javnosti, se še vedno razvija in izboljšuje zahvaljujoč skupnosti, ki tehnologijo uporablja in sproti obvešča razvijalce o težavah. Napake se sproti z razvojem tehnologije .NET 8 odpravljajo in dokumentirajo na blogu razvijalcev tehnologije .NET [13], [14].

# 3 Neprekinjena integracija in postavitev aplikacij (CI/CD)

## 3.1 Predstavitev

Neprekinjena integracija in postavitev aplikacij (CI/CD) avtomatizira večino potrebnega ročnega dela za prenos sprememb iz programske kode v produkcijo [16]. Ročno grajenje, nalaganje in postavitev aplikacij je zamudno, še posebej kadar pozabimo postopek cevovoda. Z definiranim avtomatskim cevovodom CI/CD razbremenimo razvijalca od procesa integracije novo razvitih funkcionalnosti ali popravkov v produkcijske aplikacije in posledično poskrbimo, da se osredotoči le na razvoj programske opreme.

V kategorijo neprekinjene integracije spadajo avtomatsko grajenje, testiranje in vključitev sprememb v glavne veje repozitorija. Tovrsten način razvoja programske opreme spodbuja pogosto združevanje vej repozitorija, kar zmanjša možnost konfliktov v programski kodi ter poveča njeno zanesljivost zaradi pogostega testiranja programske kode. Posledično nam neprekinjena integracija prinaša zgodnje razreševanje morebitnih napak programske kode ali varnostnih ranljivosti.

Za procesom neprekinjene integracije sledi še avtomatizacija postavitve aplikacij h končnim uporabnikom. Proces iz zgrajene in testirane programske kode pripravi končne artefakte, ki so pripravljeni za prenos v produkcijsko okolje. Poleg tega v to kategorijo spada še zagotavljanje potrebne infrastrukture za zagon aplikacije, npr. nastavitev in namestitev paketov v vsebniku (ang. container).

V praksi se je uporaba CI/CD uveljavila predvsem zaradi več ključnih prednosti, kot so [15]:

- hitrejši razvoj novih funkcionalnosti in posledično obvladovanje konkurenčnosti na trgu,
- pomaga pri pridobivanju novega kadra, zadrževanju talentiranih razvijalcev ter zmanjšanju izgorelosti, saj se lahko razvijalci fokusirajo le na razvoj programske opreme,
- izboljšana kvaliteta programske kode zaradi specializacije razvijalcev,
- zadovoljni uporabniki zaradi manjšega števila napak v produkciji in hitre predstavitve novih funkcionalnosti.

### 3.2 CI/CD v ogrodju MAUI

Ena izmed slabosti poslovnih rešitev razvitih v ogrodju Xamarin je slaba podpora grajenju projektov s pomočjo ukazne vrstice, še posebej v smislu vsebnikov, ki tečejo na operacijskem sistemu Linux. Posledično ni mogoče avtomatizirati procesa integracije in postavitve aplikacij. Vsaka sprememba v programski kodi, nova funkcionalnost ali predstavitev popravka se mora ročno prevesti, zagnati, testirati in šele ob potrditvi primernosti delovanja oddati uporabniku v testiranje. Postopek je zamuden, izpostavljen napakam in sili razvijalca k pogostemu spreminjanju konteksta. Koncentracija in produktivnost se zmanjšata, razvijalec postane nezadovoljen. Posledično se lahko hitro zgodi, da se v celotnem postopku kakšna izmed nalog pozabi ali naredi napačno, kar poveča možnost napak v programski opremi in nezadovoljstvo uporabnika.

Izpostavljena pomanjkljivost je s prihodom ogrodja MAUI ublažena, saj so v podjetju Microsoft s popolno integracijo ogrodja v ekosistem okolja .NET poskrbeli tudi za integracijo z vmesnikom ukazne vrstice .NET (ang. .NET CLI). Z njegovo pomočjo je mogoče na podlagi ukazne vrstice graditi, izvajati in objavljati aplikacije na več platformah. V naslednjem poglavju prikažemo praktični primer implementacije cevovoda CI/CD na podlagi orodja .NET CLI v platformi GitLab za demonstracijski projekt, ki je razvit v ogrodju MAUI.

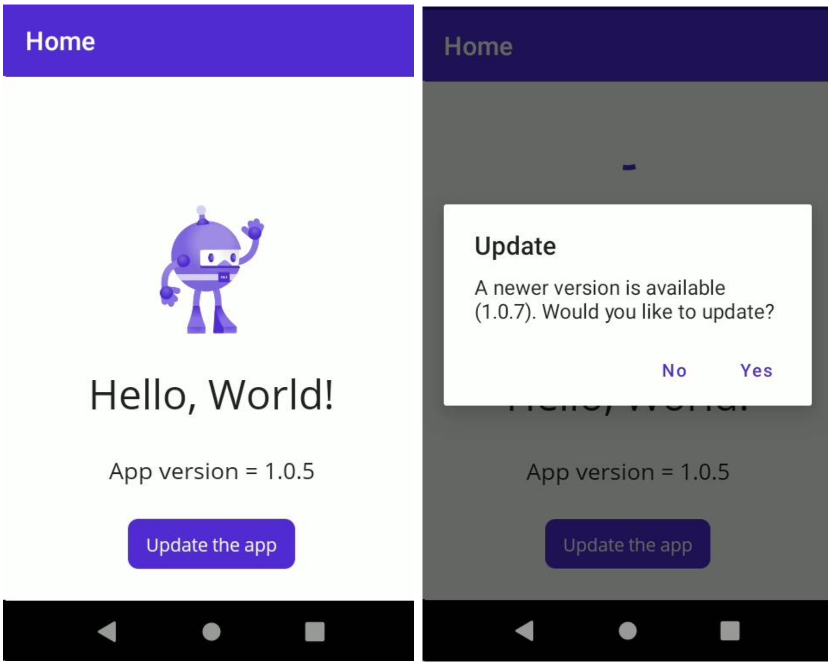
## 4 Praktični primer cevovoda CI/CD na vsebniku Linux ter platformi GitLab

Čeprav je implementacija neprekinjene integracije in postavitve aplikacij v ogrodju MAUI mogoča, v literaturi še ni moč zaznati veliko primerov uporabe. V [17] so prikazali način, kako sestaviti cevovod CI/CD v oblačnem sistemu Azure ter v orodju verzioniranja programske kode GitHub za aplikacije pripravljene v ogrodju MAUI. Oba primera izvedeta cevovod na virtualnem računalniku z operacijskim sistemom macOS ali Windows. Ker so dandanes vse bolj uporabni in popularni vsebniki, ki delujejo na osnovi poljubne verzije operacijskega sistema Linux, nas je zanimalo, ali je mogoče pripraviti cevovod CI/CD, ki bi se zagnal na tovrstnem vsebniku.

V nadaljevanju predstavimo praktični primer neprekinjene integracije in postavitve demonstracijske mobilne aplikacije, razvite v ogrodju MAUI. Mobilna aplikacija je namenjena operacijskemu sistemu Android in skrbi za samodejno posodabljanje, v kolikor zazna novo verzijo v registru paketov. Cevovod CI/CD je pripravljen za platformo GitLab, ampak ga je mogoče enostavno reproducirati na njemu podobne platforme. Ob izdaji nove verzije produkcijske aplikacije se cevovod zažene, ustvari paket mobilne aplikacije in ga objavi v registru paketov. Gradnja paketa aplikacije Android poteka v celoti na vsebniku pod operacijskim sistemom Linux. Dodatno še prikažemo kako nastaviti ukaz za generiranje podpisanih paketov mobilne aplikacije Android.

### 4.1 Android mobilna aplikacija

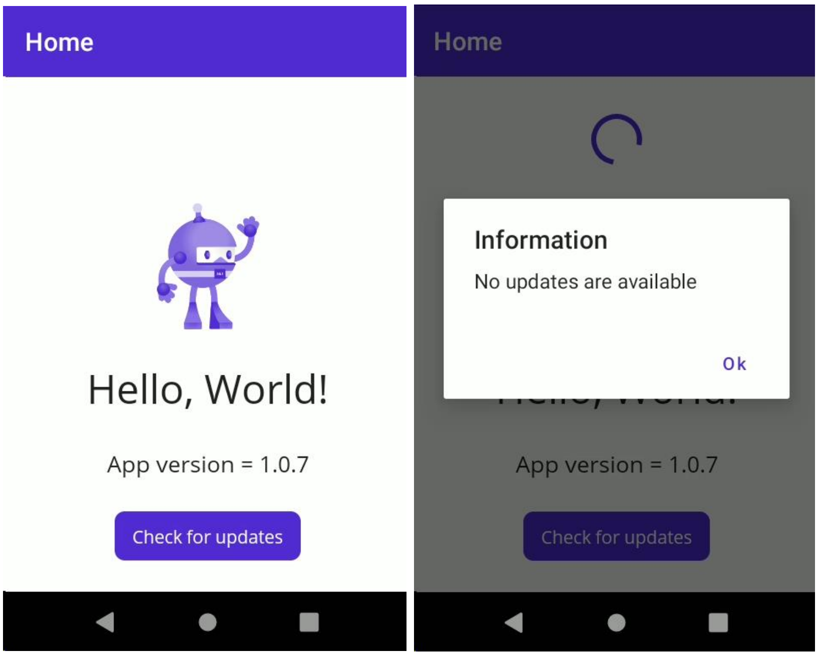
Za demonstracijski primer smo pripravili kar preprosto mobilno aplikacijo, ki je zasnovana po primeru iz [18] (Slika). Aplikacija temelji na tehnologiji .NET 7 in je namenjena operacijskemu sistemu Android. Na začetno stran smo dodali gumb, ki ob kliku preveri, ali so na voljo posodobitve (slika 2). To preverimo s pomočjo poizvedbe na končno točko registra paketov, ki jo ponuja platforma GitLab [22]. V kolikor obstaja paket z novejšo verzijo mobilne aplikacije, uporabniku ponudimo možnost avtomatske posodobitve. V pritrdilnem primeru se paket prenese na mobilno napravo in mobilna aplikacija se samodejno posodobi (slika 3).



Update

A newer version is available (1.0.7). Would you like to update?

Slika 2: Začetna stran mobilne aplikacije in predlog posodobitve.



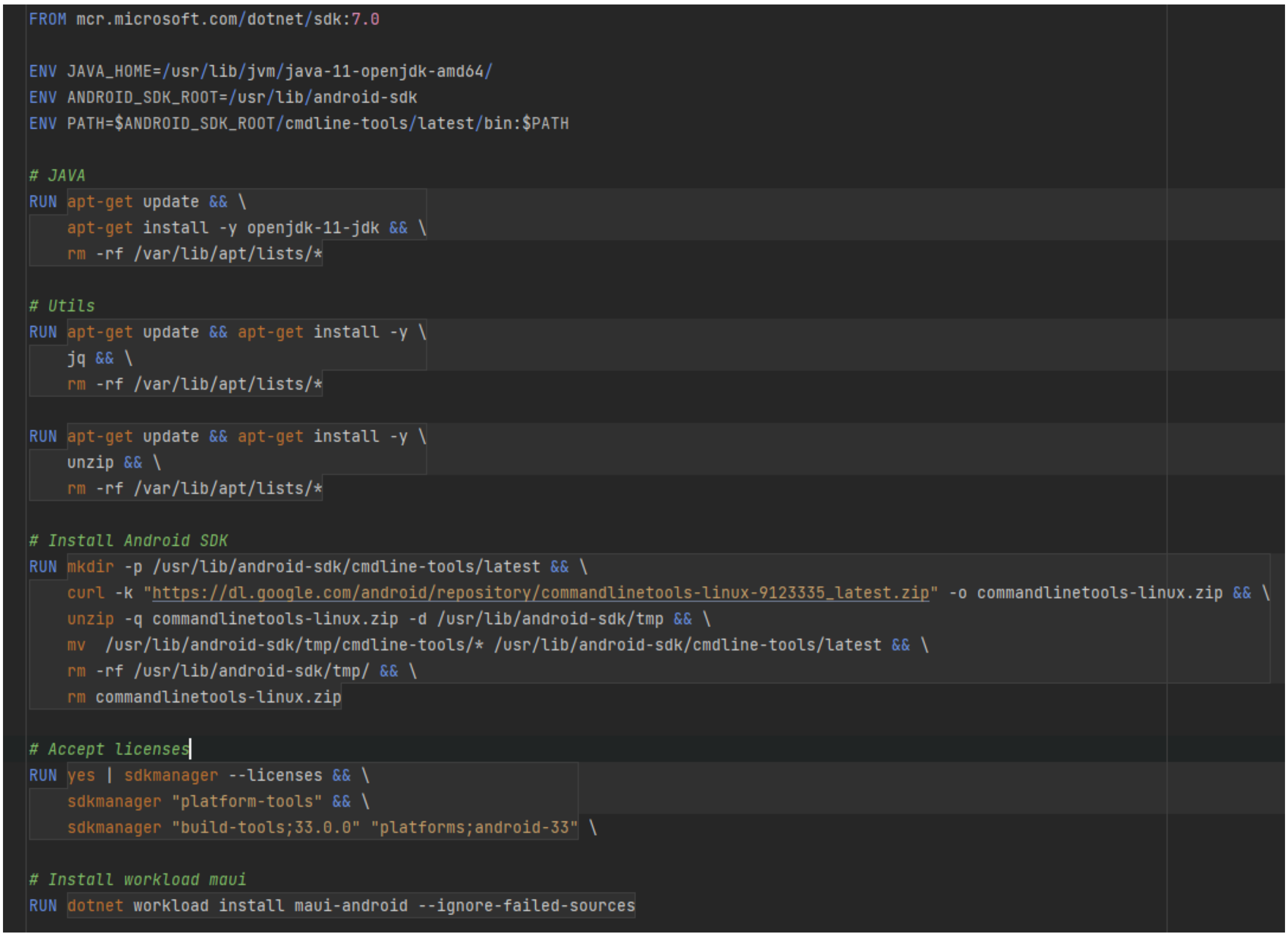
Information

No updates are available

Slika 3: Posodobljena aplikacija in ponovna preverba novih posodobitev.

### 4.2 Predloga vsebnika

Vhodna predloga vsebnika (ang. Docker image) naše nastavitvene datoteke *Dockerfile* je predloga uradnega kompleta za razvoj programske opreme na osnovi tehnologije .NET [19]. Ta nam omogoča uporabo vmesnika ukazne vrstice .NET. Nato s pomočjo upravljalca paketov naložimo razvojni komplet programskega jezika Java verzije 11, komplet za razvoj programske opreme Android, sprejmemo licence uporabe ter nazadnje naložimo še izbirno razširitev tehnologije .NET za gradnjo mobilnih aplikacij Android imenovano „maui-android”. Opisan postopek grafično prikazuje slika 4.



```dockerfile
FROM mcr.microsoft.com/dotnet/sdk:7.0

ENV JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64/
ENV ANDROID_SDK_ROOT=/usr/lib/android-sdk
ENV PATH=$ANDROID_SDK_ROOT/cmdline-tools/latest/bin:$PATH

# JAVA
RUN apt-get update && \
    apt-get install -y openjdk-11-jdk && \
    rm -rf /var/lib/apt/lists/*

# Utils
RUN apt-get update && apt-get install -y \
    jq && \
    rm -rf /var/lib/apt/lists/*

RUN apt-get update && apt-get install -y \
    unzip && \
    rm -rf /var/lib/apt/lists/*

# Install Android SDK
RUN mkdir -p /usr/lib/android-sdk/cmdline-tools/latest && \
    curl -k "https://dl.google.com/android/repository/commandlinetools-linux-9123335_latest.zip" -o commandlinetools-linux.zip && \
    unzip -q commandlinetools-linux.zip -d /usr/lib/android-sdk/tmp && \
    mv  /usr/lib/android-sdk/tmp/cmdline-tools/* /usr/lib/android-sdk/cmdline-tools/latest && \
    rm -rf /usr/lib/android-sdk/tmp/ && \
    rm commandlinetools-linux.zip

# Accept licenses
RUN yes | sdkmanager --licenses && \
    sdkmanager "platform-tools" && \
    sdkmanager "build-tools;33.0.0" "platforms;android-33" \

# Install workload maui
RUN dotnet workload install maui-android --ignore-failed-sources
```

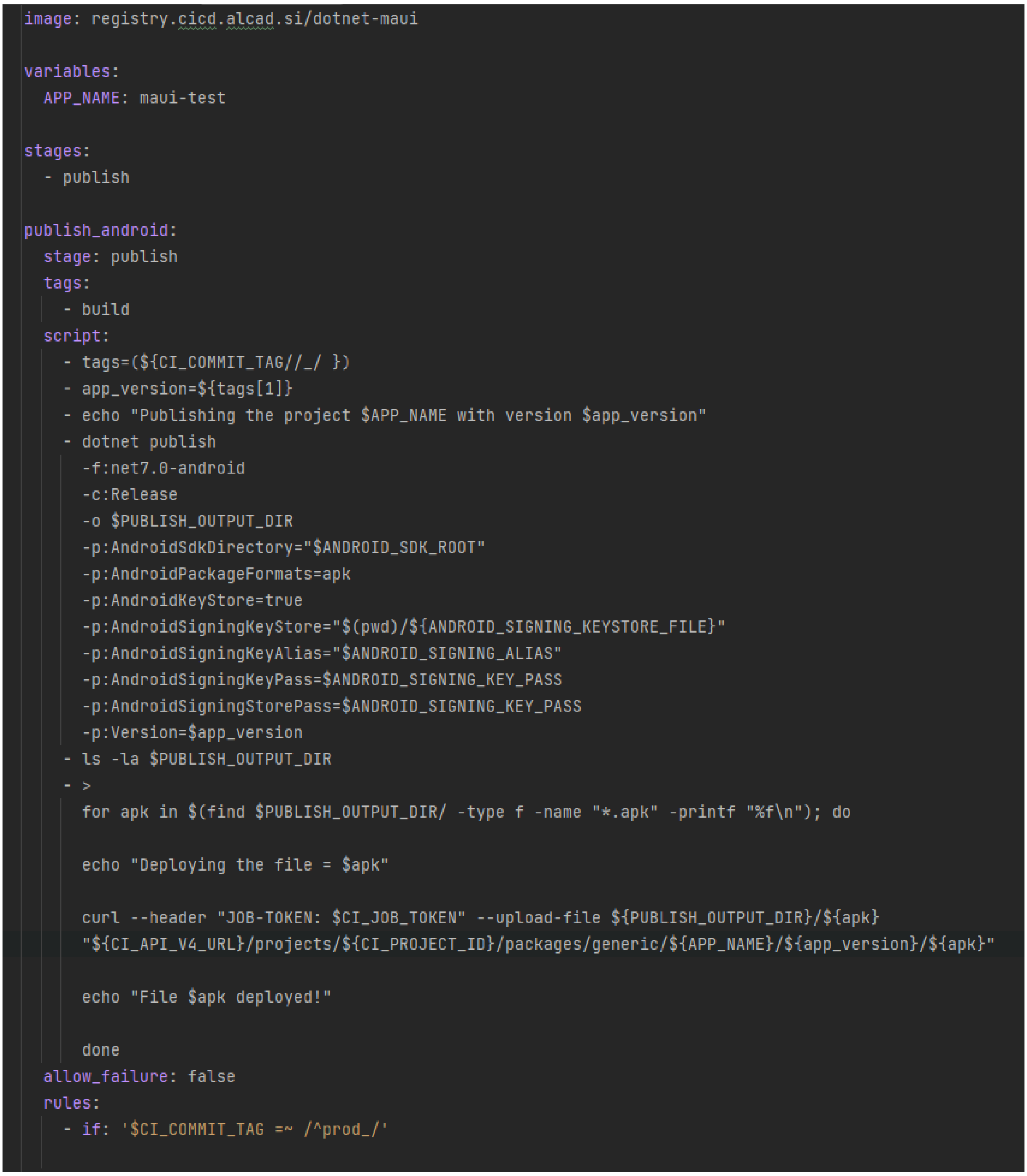
**Slika 4: Nastavitev predloge vsebnika.**

### 4.3 Nastavitev cevovoda v platformi GitLab

Za demonstracijske namene smo si cevovod zamislili na sledeč način. Najprej kot bazno predlogo vsebnika uporabimo predlogo, predstavljeno v 4.2. Nato zgradimo podpisan paket mobilne aplikacije Android, pri čemer je certifikat za podpisovanje predhodno pripravljen s pomočjo ukaza *keytool* [23]. Certifikat nato podamo kot parameter ukazu *dotnet publish* [23] skupaj z geslom in vzdevkom. Ukaz izda paket mobilne aplikacije Android. Paket shranimo v register paketov na repozitoriju platforme GitLab tako, da ga dodamo v telo končne točke za shranjevanje datotek v registru paketov. Platforma GitLab namreč omogoča hrambo, verzioniranje in nalaganje poljubnih datotek za vsak projekt posebej [20]. Pripravljen cevovod se zažene ob vsaki stvaritvi značke iz produkcijske veje. Značka je oblike „prod_<verzija>”, na podlagi česa lahko na enostaven način identificiramo novo različico programske opreme, ki jo prenašamo uporabniku v produkcijo. Implementacijo opisanega postopka prikazuje slika 5.

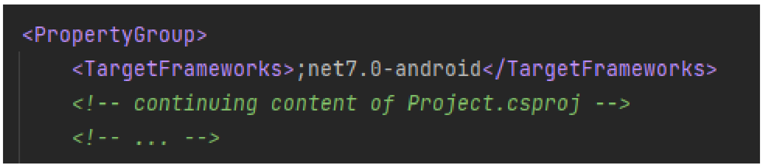
## 4.4 Koristni predlogi

Ob inicializaciji projekta mobilne aplikacije so kot privzete platforme nastavljene vse možne, ki jih ogrodje MAUI podpira. Žal grajenje projektov MAUI pod operacijskim sistemom Linux še ne vključuje podpore za vse platforme. Če v vsebniku, ki se izvaja nad predstavljeno vsebniško predlogo, zaženemo ukaz *dotnet workload search*, opazimo, da so trenutno dostopne razširitve vmesnika ukazne vrstice .NET razširitev za operacijski sistem Android, Windows ter Tizen. Zato moramo v datoteki projekta .csproj eksplicitno nastaviti oziroma odstraniti platforme, ki trenutno še niso podprte za gradnjo. V nasprotnem primeru gradnja ne bo uspešna. Ker smo demonstracijski primer predstavili za platformo Android, smo torej morali prilagoditi značko XML, ki opredeli ciljne platforme (slika 6). V kolikor bi želeli dodati še kakšno platformo, bi ustrezno posodobili XML značko.



```yaml
image: registry.cicd.alcad.si/dotnet-maui

variables:
  APP_NAME: maui-test

stages:
  - publish

publish_android:
  stage: publish
  tags:
    - build
  script:
    - tags=(${CI_COMMIT_TAG//_/ })
    - app_version=${tags[1]}
    - echo "Publishing the project $APP_NAME with version $app_version"
    - dotnet publish
      -f:net7.0-android
      -c:Release
      -o $PUBLISH_OUTPUT_DIR
      -p:AndroidSdkDirectory="$ANDROID_SDK_ROOT"
      -p:AndroidPackageFormats=apk
      -p:AndroidKeyStore=true
      -p:AndroidSigningKeyStore="$(pwd)/${ANDROID_SIGNING_KEYSTORE_FILE}"
      -p:AndroidSigningKeyAlias="$ANDROID_SIGNING_ALIAS"
      -p:AndroidSigningKeyPass=$ANDROID_SIGNING_KEY_PASS
      -p:AndroidSigningStorePass=$ANDROID_SIGNING_KEY_PASS
      -p:Version=$app_version
    - ls -la $PUBLISH_OUTPUT_DIR
    - >
      for apk in $(find $PUBLISH_OUTPUT_DIR/ -type f -name "*.apk" -printf "%f\n"); do

      echo "Deploying the file = $apk"

      curl --header "JOB-TOKEN: $CI_JOB_TOKEN" --upload-file ${PUBLISH_OUTPUT_DIR}/${apk}
      "${CI_API_V4_URL}/projects/${CI_PROJECT_ID}/packages/generic/${APP_NAME}/${app_version}/${apk}"

      echo "File $apk deployed!"

      done
  allow_failure: false
  rules:
    - if: '$CI_COMMIT_TAG =~ /^prod_/'
```

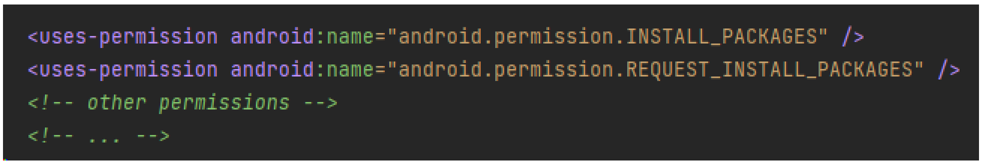
**Slika 5: Nastavitev cevovoda CI/CD v platformi GitLab.**



```
<PropertyGroup>
    <TargetFrameworks>;net7.0-android</TargetFrameworks>
    <!-- continuing content of Project.csproj -->
    <!-- ... -->
```

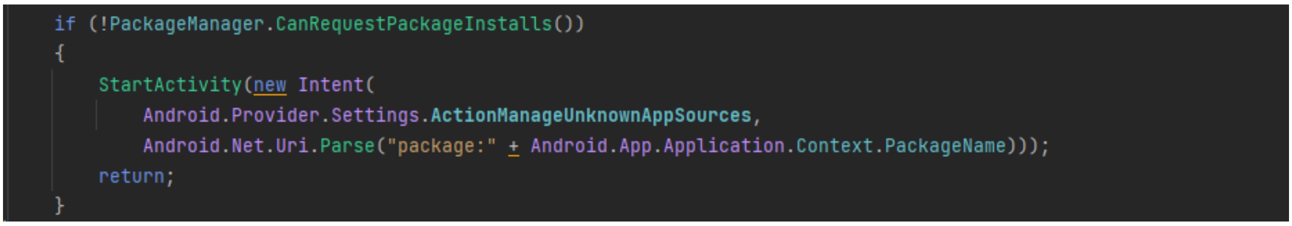
**Slika 6: Nastavitev značke XML v datoteki projekta .csproj za opredelitev ciljne platforme.**

Za uspešno izvedbo samodejnega posodabljanja mobilne aplikacije moramo v datoteki AndroidManifest.xml nastaviti dovoljenji za nalaganje ter za pojavitev zahteve o nalaganju novih paketov (slika 7). Dodatno moramo še programsko preveriti, ali je uporabnik že potrdil dovoljenje aplikaciji za zahtevek o nalaganju aplikacij. To lahko storimo s pomočjo izrezka programske kode na slika 8, kjer v primeru nepotrjenega dovoljenja uporabnika preusmerimo k dialogu za odobritev.



```
<uses-permission android:name="android.permission.INSTALL_PACKAGES" />
<uses-permission android:name="android.permission.REQUEST_INSTALL_PACKAGES" />
<!-- other permissions -->
<!-- ... -->
```

**Slika 7: Nastavitev dovoljenj mobilni aplikaciji Android za nalaganje novih paketov na mobilno napravo.**



```
if (!PackageManager.CanRequestPackageInstalls())
{
    StartActivity(new Intent(
        Android.Provider.Settings.ActionManageUnknownAppSources,
        Android.Net.Uri.Parse("package:" + Android.App.Application.Context.PackageName)));
    return;
}
```

**Slika 8: Validacija dovoljenja za nalaganje aplikacij.**

## 5 Zaključek

V primerjavi s svojim predhodnikom nam ogrodje MAUI prinaša mnogo nadgradenj in znatno olajša delo razvijalca. Je popolnoma integrirano v ekosistem tehnologij .NET in ponuja enakovredne aplikacijske programske vmesnike za razvoj aplikacij na različnih platformah. Izboljšave so zaznane tudi v učinkovitosti ter v platformni neodvisnosti. Za namen prispevka je najpomembnejša nadgradnja integracije z vmesnikom ukazne vrstice .NET, kar nam omogoča uporabo neprekinjene integracije in postavitve mobilnih poslovnih rešitev na več platformah. Slednje smo utemeljili s praktičnim primerom na platformi GitLab in mobilno aplikacijo, ki omogoča samodejno posodabljanje na podlagi samodejnih izdaj novih verzij na registru paketov.

Kljub vsem prednostim ne smemo pozabiti, da je tehnologija še mlada in se še razvija. Pomanjkljivosti in napake se še vedno pojavljajo ter se s prihodom vsake nove verzije tehnologije .NET popravijo, kar morebiti spremeni tudi aplikacijski programski vmesnik. Omenjeno moramo vzeti v obzir, če imamo namen nadgraditi obstoječe mobilne rešitve, razvite v ogrodju Xamarin, vsaj dokler ima le-to še uradno podporo. Nedvomno ima ogrodje MAUI potencial biti učinkovit naslednik ogrodja Xamarin, katerega razvoj se splača spremljati.

# Literatura

[1] https://learn.microsoft.com/en-us/dotnet/maui/what-is-maui?view=net-maui-7.0, What is .NET MAUI?, obiskano 20. 06. 2023.

[6] STONIS Michael, »Enterprise Application Patterns using .NET MAUI«. Redmond, Washington, 2022.

[7] https://learn.microsoft.com/en-us/xamarin/get-started/what-is-xamarin-forms, What is Xamarin.Forms?, obiskano 20. 06. 2023.

[8] https://tirania.org/blog/archive/2011/May-16.html, Miguel de Icaza, Announcing Xamarin, obiskano 20. 06. 2023.

[9] https://blogs.microsoft.com/blog/2016/02/24/microsoft-to-acquire-xamarin-and-empower-more-developers-to-build-apps-on-any-device/, GUTHRIE Scott, Microsoft to acquire Xamarin and empower more developers to build apps on any device, obiskano 20. 06. 2023.

[10] https://techreport.com/news/xamarin-now-comes-free-with-visual-studio/, FERREIRA Bruno, Xamarin now comes free with Visual Studio, obiskano 20. 06. 2023.

[11] https://learn.microsoft.com/en-us/xamarin/get-started/supported-platforms?tabs=windows, Xamarin.Forms supported platforms, obiskano 20. 06. 2023.

[12] https://learn.microsoft.com/en-us/dotnet/maui/supported-platforms, Supported platforms for .NET MAUI apps, obiskano 20. 06. 2023.

[13] https://devblogs.microsoft.com/dotnet/learn-dotnet-maui/, SOUCOUP Matt, New Resources to Get Started with .NET MAUI, obiskano 20. 06. 2023.

[14] https://www.syncfusion.com/blogs/post/xamarin-versus-net-maui.aspx, KATHIRESAN G. Selva, Xamarin Versus .NET MAUI, obiskano 20. 06. 2023.

[15] https://devblogs.microsoft.com/dotnet/performance-improvements-in-dotnet-maui/, PEPPERS Jonathan, Performance Improvements in .NET MAUI, obiskano 20. 06. 2023.

[16] https://devblogs.microsoft.com/dotnet/dotnet-7-performance-improvements-in-dotnet-maui/, PEPPERS Jonathan, .NET 7 Performance Improvements in .NET MAUI, obiskano 20. 06. 2023.

[17] https://devblogs.microsoft.com/dotnet/announcing-dotnet-maui-in-dotnet-8-preview-3/, ORTINAU David, Announcing .NET MAUI in .NET 8 Preview 3, obiskano 20. 06. 2023.

[18] https://devblogs.microsoft.com/dotnet/announcing-dotnet-maui-in-dotnet-8-preview-5/, ORTINAU David, Announcing .NET MAUI in .NET 8 Preview 5, obiskano 20. 06. 2023.

[19] https://about.gitlab.com/resources/ebook-fuel-growth-cicd/, Modernize your CI/CD, obiskano 21. 06. 2023.

[20] https://about.gitlab.com/topics/ci-cd/, What is CI/CD, obiskano 21. 06. 2023.

[21] https://devblogs.microsoft.com/dotnet/devops-for-dotnet-maui/, SATPATHY Sweeky, Getting Started with DevOps and .NET MAUI, obiskano 21. 06. 2023.

[22] https://learn.microsoft.com/en-us/dotnet/maui/get-started/first-app, Build your first app, obiskano 21. 06. 2023.

[23] https://hub.docker.com/_/microsoft-dotnet-sdk/, .NET SDK | Docker Hub, obiskano 21. 06. 2023.

[24] https://docs.gitlab.com/ee/user/packages/generic_packages/#publish-a-generic-package-by-using-cicd, Publish a generic package by using CI/CD, obiskano 21. 06. 2023.

[25] https://learn.microsoft.com/en-us/aspnet/core/blazor/hybrid/?view=aspnetcore-7.0, ASP.NET Core Blazor Hybrid, obiskano 22. 06. 2023.

[26] https://docs.gitlab.com/ee/api/packages.html, Packages API, obiskano 23. 06. 2023.

[27] https://learn.microsoft.com/en-us/dotnet/maui/android/deployment/publish-cli?view=net-maui-7.0, Publish an Android app using the command line, obiskano 23. 06. 2023.

# Kaj je Blazor Hybrid in kako nam lahko pomaga pri nadgradnji programske opreme?

Matjaž Prtenjak,
Endava, Ljubljana, Slovenija
matjaz.prtenjak@endava.com

Blazor je Microsoftova tehnologija za razvoj spletnih strani v okolju .NET. Kot taka se seveda uporablja za razvoj klasičnih spletnih strani, ki tečejo v brskalniku in katerih koda se lahko izvaja na strežniku ali odjemalcu. Blazor Hybrid pa je Microsoftova komponenta, s pomočjo katere lahko spletne strani izvajamo znotraj obrazca v grafičnem uporabniškem vmesniku. S pomočjo tehnologije MAUI (Multi-platform Application UI) lahko potem takšna koda teče v obrazcih na različnih okoljih (Windows, iOS, Android ali Mac). V tem prispevku pa se bomo fokusirali izključno na Windows okolja in nadgradnjo obstoječih WinForms aplikacij v novejša spletna ogrodja.

**Ključne besede:**
Blazor Hybrid
.NET
WinForms
spletne aplikacije
namizne aplikacije
C#

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.11

# 1 Uvod

Serijo prispevkov na OTS konferenci sem začel v letu 2019 s prikazom, kako lahko nek starejši razvijalec/razvijalka, ki pozna jezike kot so C, C++, C# in podobne, začne razvijati spletne aplikacije. **Predstavil sem programsko okolje VUE.**

Po koronskem času, v letu 2022, se je situacija spremenila in osebam iz prvega odstavka se ni bilo potrebno več učiti JavaScript-a in VUE, temveč so lahko s pomočjo tehnologije Blazor razvijali spletne aplikacije v znanem okolju .NET/C#.

V tem prispevku imamo pred očmi še vedno iste osebe, ki so skozi desetletja razvile obstoječe produkte v WinForm-s tehnologiji, zdaj pa bi želeli to tehnologijo, vezano samo in izključno na Okna, vsaj malce posodobiti s spletnimi tehnologijami in **predstavil bom Blazor Hybrid**.

V tej nadgradnji lahko takšni stari, vendar delujoči aplikaciji, vdihnemo novo preobleko in začnemo uporabljati spletne tehnologije kot so HTML, CSS in JavaScript. Nakar jo lahko kdaj kasneje dvignemo še višje in tako počasi iz WinForm-s aplikacije preidemo v spletno aplikacijo.

V najavi prispevka sem omenil tudi Android aplikacijo, ki bi bila razvita z uporabo MAUI ogrodja. Vendar pa ima MAUI svoj lasten prispevek izpod peresa Alena Grande (*Neprekinjena integracija in postavitev MAUI mobilnih aplikacij*), zato se bom tukaj osredotočil na WinForm-s aplikacije in kako jih lahko s pomočjo Blazor Hybrid tehnologije nadgradimo. Je pa sicer zadeva v Androidu oz. okolju MAUI enakovredna.

## 1.1 Zakaj ravno WinForms

Blazor Hybrid nikakor ni vezan na WinForm-s aplikacije, je pa v njih vsekakor uporaben. Kaj je torej tako posebnega na WinForm-s aplikacijah, da jih je potrebno posebno omeniti?

WinForm-s je ena izmed najbolj priljubljenih tehnologij in je tudi tehnologija, ki je povzdignila Visual Basic ter razmah Windows aplikacij. Windows aplikacije so resničen razcvet doživele od leta 1993 dalje, ko se je pojavil Visual Basic 3.0, s katerim se je bilo preprosto povezati s podatkovno bazo in zgraditi grafični vmesnik.

WinForm-s tehnologija se je potem transparentno prenesla v .NET Framework in je tam ostala najpomembnejša. Skozi čas je Microsoft v .NET Framework sicer uvedel tudi druge tehnologije, kot so Microsoft Silverlite ali WPF (Windows Presentation Foundation), vendar nobena izmed njih ni v resnici ogrozila primata WinForm-s aplikacij.

Situacija je torej takšna, da v svetu obstaja mnogo programov, ki uporabljajo WinForm-s tehnologijo in nerealno je pričakovati, da se vse to vrže stran ter začne pisati stvari »na novo«. Uporabnejše in boljše je torej, če lahko te aplikacije nekako nadgradimo.

# 2 Kaj je Blazor Hybrid?

Preden opišem Blazor Hybrid, moram opisati nekaj osnov in sicer zmedo okoli poimenovanj .NET ogrodij, Blazor, MAUI in šele nato Blazor Hybrid

## 2.1 .NET Framework ali .NET Core ali .NET

Na žalost so v podjetju Microsoft kratico .NET uporabili tolikokrat, da se je zelo težko znajti in razumeti, kaj trenutno .NET sploh označuje. Tukaj vam bom torej zelo na hitro in poenostavljeno poskušal razčistiti zmedo s kratico .NET.

### *2.1.1 .NET Framework*

.NET Framework je ogrodje, ki ga je podjetje MS lansiralo leta 2002, z verzijo 1.0. To je bil Microsoftov odgovor na programski jezik Java in z Javo si deli mnogo skupnih točk. Oba uporabljata vmesni jezik in programe prevajata v vmesni jezik, ki jih nato *prevajalnik ob izvajanju* (JIT – just-in-time compiler) prevede v strojno kodo računalnika.

Seveda si z Javo deli tudi praktično vse druge značilnosti, se pa razlikujeta v dveh pomembnih lastnostih:

1. Java teče na mnogih operacijskih sistemih in vedno uporablja programski jezik Java
2. .NET Framework teče samo na sistemih Windows in lahko uporablja različne programske jezike (v praksi pa predvsem C# in VB)

Čeravno je bil .NET Framework zamišljen kot ogrodje, ki bi lahko teklo v različnih operacijskih sistemih, je vedno teklo samo in izključno v Oknih. Kot zanimivost lahko omenim, da je bila tehnologija WinForms ravno ena izmed teh težav, ki so .NET Framework vezala na Okna.

### *2.1.2 .NET Core*

.NET Framework ima nekaj odprtokodnih elementov, sicer pa je to zaprta tehnologija podjetja Microsoft. Z .NET Frameworkom se je hkrati, vendar neodvisno, razvijala tudi odprtokodna različica Mono, ki je lahko tekla na različnih operacijskih sistemih.

Na podlagi .NET Framework in Mono projekta so začeli v podjetju Microsoft odpirati tehnologijo in nekje v letu 2017 se je pojavila odprtokodna različica .NET, ki je tekla na različnih operacijskih sistemih in se je imenovala .NET Core

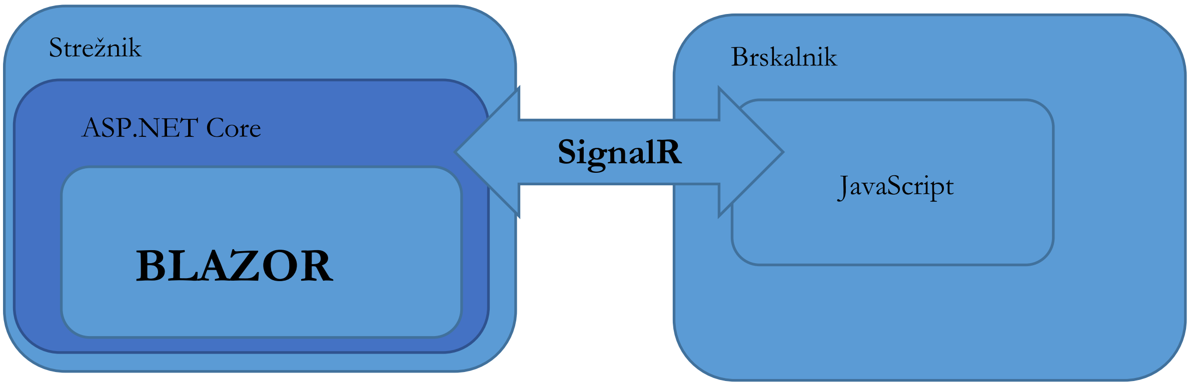
### *2.1.3 .NET*

In da je zmeda popolna, se je potem poimenovanje spet spremenilo in iz različice .NET Core 3.1 smo prešli v različico .NET 5.0

Danes se torej uporablja samo kratica .NET in to pomeni odprtokodno različico ogrodja, ki teče na več operacijskih sistemih. Med razvijalci pa je še vedno prisotno staro ime .NET Core, kar povzroča določeno zmedo.

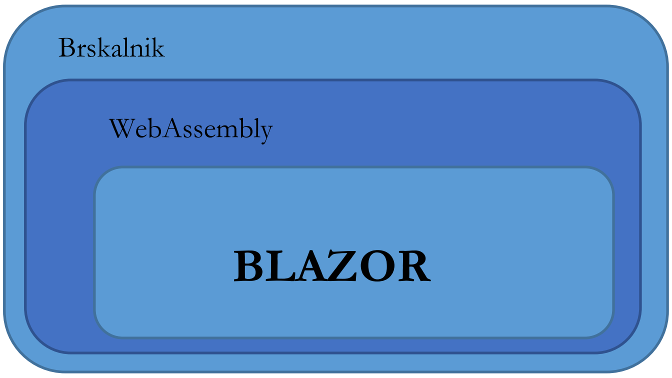
## 2.2 Blazor

O tehnologiji Blazor sem govoril na lanski konferenci OTS, zato tukaj ponovim samo nekaj osnov. Blazor je Microsoftova odprtokodna spletne tehnologija, kjer lahko spletne aplikacije razvijamo z uporabo C# programskega jezika. Blazor lahko teče na strežniku in se z odjemalci pogovarja preko SignalR knjižnice v realnem času, lahko pa v celoti teče tudi na uporabnikovem brskalniku.



**Slika 1: Blazor Server Side.**

V slednjem primeru, ki sem ga podrobneje opisoval lani, se celoten program prenese v brskalnik, kjer potem teče v WASM oz. WebAssembly-u. V tem primeru imamo torej v brskalniku programski jezik (WebAssembly), v katerega se prevede naš Blazor program in se nato izvaja v brskalniku.



**Slika 2: Blazor WebAssembly.**

### 2.2.1 *Razor*

Beseda Razor je zelo podobna besedi Blazor in to namenoma. Razor je namreč MS programski jezik za »nadzor« HTML kode. Blazor za obdelavo HTML uporablja Razor. Od tod v resnici tudi izhaja njegovo ime. Razor je namreč MS že imel in potem si je nad njim pač »izmislil« Blazor.

Primer Razor kode:

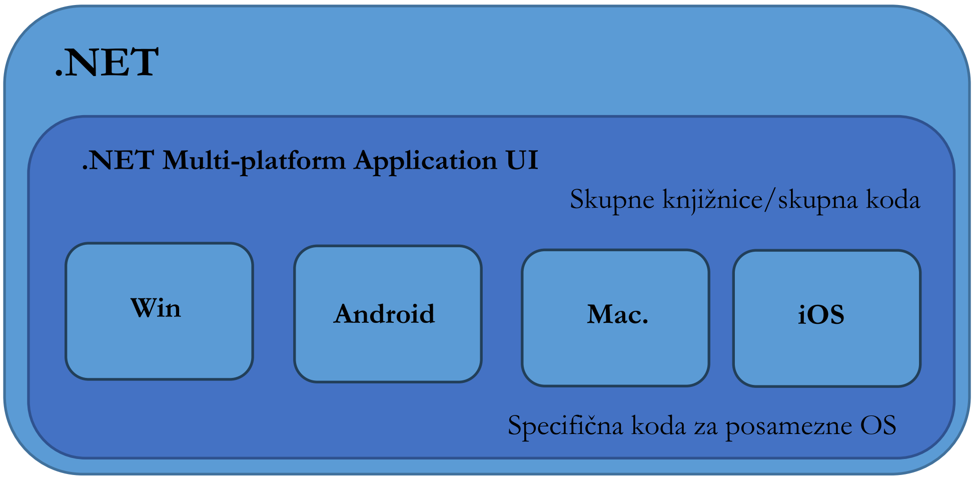
```
<div @onclick="ToogleMessage">
  @if (seen)
  {
    <span>Now you see me</span>
  }
  else
  {
    <span>Click Me!</span>
  }
</div>
```

## 2.3 MAUI

MAUI ali Multi-platform Application UI je Microsoftova tehnologija, s pomočjo katere lahko razvijamo programe, ki se potem izvajajo v Oknih, na Androidu, na Macintosh-u ali iOS.

Tukaj lahko takoj pogrešamo Linux, razlog pa je dokaj preprost. Pri tehnologiji MAUI govorimo o grafičnih uporabniških vmesnikih in pri Linux sistemih slednji niso standardizirani, zato jih MAUI (zaenkrat) ne podpira.

Na kratko je torej MAUI tehnologija, s pomočjo katere lahko v .NET-u razvijamo grafične uporabniške vmesnike za različne operacijske sisteme:



**Slika 3: MAUI.**

## 2.4 Blazor Hybrid

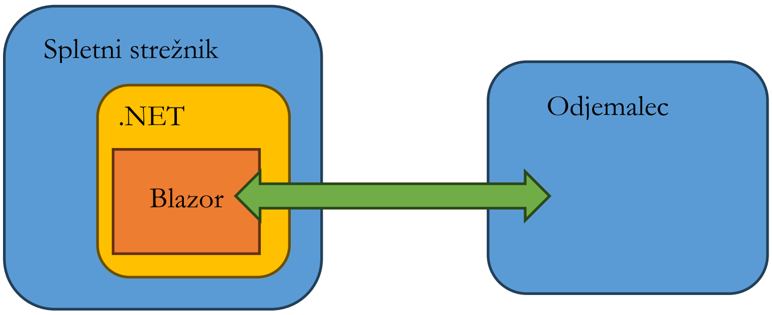
In sedaj, ko smo spoznali posamezne gradnike, se lahko povrnemo k originalnemu vprašanju: »Kaj je Blazor Hybrid?«

Microsoft na spletnih straneh pravi: »*Uporabite Blazor Hybrid za razvoj namiznih in mobilnih aplikacij z .NET in Blazor.*« To nam ne pove kaj dosti. Če na isti spletni strani beremo dalje, lahko preberemo sledeče (poskusil sem izbrati čimbolj dobeseden prevod):

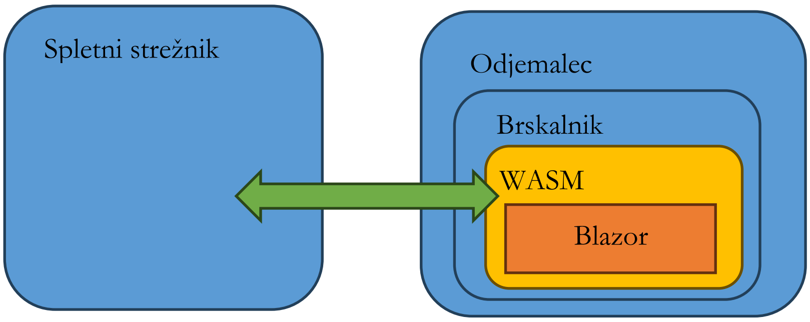
*V Blazor hibridni aplikaciji se Razor komponente izvajajo na napravi sami. Komponente se prikažejo v vgrajenem spletnem pogledu (WebView) prek lokalnega vmesnega kanala. Komponente ne delujejo v brskalniku in ne vključujejo WebAssembly-ja. Razor komponente se hitro naložijo in izvedejo kodo ter imajo popoln dostop do naravnih zmogljivosti naprave prek .NET platforme. Slogi komponent, ki se prikažejo v spletnem pogledu, so odvisni od platforme in lahko zahtevajo prilagoditve zaradi razlik v prikazu med platformami z uporabo prilagojenih slogovnih datotek.*

Pravzaprav je tudi iz tega kar težko razbrati, o čem je govora, zato je lažje, če se spet povrnemo na razlago o Blazor tehnologiji in se spomnimo sledečega:

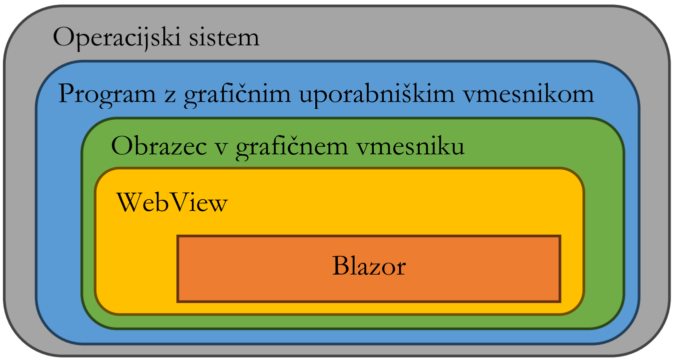
1. **Blazor Server Side** teče na spletnem strežniku in tam programsko kodo Blazor komponent izvaja .NET ogrodje, ki je instalirano na spletnem strežniku
2. V primeru **Blazor WASM**, se celoten program prenese na odjemalca in na odjemalcu WebAssembly izvaja programsko kodo Blazor komponent
3. Zdaj pa imamo še tretjo različico **Blazor Hybrid**, kjer pa Blazor izvaja posebna komponenta na vašem uporabniškem vmesniku.



**Slika 4: Blazor Server Side; Blazor teče v .NET okolju, ki teče na strežniku in preko njega ima dostop do vsega na strežniku, do česar ime dostop .NET.**



**Slika 5: Blazor WAMS; Blazor teče v WASM (WebAssembly) in ima dostop do vsega, do česar lahko znotraj brskalnika dostopa WASM.**



**Slika 9: Blazor Hybrid; Ni spletnega strežnika! Blazor kodo izvaja komponenta WebView in preko nje ima Blazor dostop do vsega, do česar ima pravico dostopati program, ki teče na operacijskem sistemu.**

Torej je razlika v tem, **kje teče Blazor koda in kdo jo izvaja ter posledično tudi, do česa ima dostop.**

Blazor Hybrid programska koda teče v nekem oknu vašega programa (bodisi v Windows, na Androidu, na iOS-u…) in tam posebna komponenta skrbi zanjo.

Na okno dodamo WebView komponento in potem v tej komponenti teče naša Blazor koda, ki ima tako dostop tudi do elementov obrazca, na katerem teče in preko obrazca tudi do ostalih delov sistema.

### *2.4.1 Prenosljivost Blazor kode*

Ena izmed prednosti, ki jo avtomatično pridobimo, pa je zmožnost prenosa Blazor kode. V kolikor ne uporabljate specifičnih elementov operacijskega sistema, lahko kodo, ki ste jo razvili kot Blazor Hybrid prenesete tudi v Blazor WASM in stvar bo delovala.

## 3 WinForms in Blazor Hybrid

Sedaj, ko smo obdelali celoten nabor tehnologij, pa lahko preidemo na praktično uporabo Blazor Hybrid tehnologije za nadgradnjo obstoječe programske opreme, pisane v tehnologiji WinForms.

### 3.1 Obstoječe WinForms Aplikacije

Pri obstoječih WinForms aplikacijah lahko pričakujemo:

- da so kompleksne,
- jih je razvijalo več razvijalcev,
- ne uporabljajo priporočenih programerskih vzorcev,
- nimajo testov.

Vse to so povsem realna pričakovanja, saj tukaj govorimo o večjih aplikacijah, ki so se razvijale skozi daljše obdobje in bi jih želeli še uporabljati.

V kolikor gre za manjše aplikacije, potem je zadeva še toliko bolj preprosta, saj jih lahko pač napišemo ponovno.

### 3.2 Ali moramo WinForms aplikacije nadgraditi

To je povsem legitimno vprašanje in odgovor nanj je NE. Aplikacij ne potrebujemo nadgrajevati zaradi kakšnih posebnih zunanjih vzrokov. Microsoft še vedno razvija .NET Framework, knjižnice so dosegljive…

Pri nadgradnji nas torej mora voditi neka večja želja, saj nadgradnja ne bo trivialna in bo zahtevala prilagajanje in določeno mero truda.

Zakaj bi torej sploh želeli nadgraditi aplikacije? Prilagam nekaj mogočih vzrokov:

- Razvijalci imajo rajši spletne tehnologije in tudi uporabniki so jih bolj navajeni.
- Del aplikacije je lahko moderen, svež, srž aplikacije pa še zmeraj ostaja in ga ne želimo spreminjati.
- V aplikaciji si želimo več avtomatičnih testov in modernejše programerske vzorce.

### 3.3 Kako se lotiti nadgradnje?

Tu pa je odgovorov seveda mnogo in noben ni nujno (povsem) pravilen, kot tudi noben zagotovo ni povsem napačen. Kot vse v svetu razvoja programske opreme, je tudi to odvisno od vaših konkretnih želja in potreb.

Pot, ki jo predstavljam v tem prispevku, je pot skozi uporabo Blazor Hybrid tehnologije, saj boste na ta način dobili aplikacijo, ki bo imela (vsaj del) kode pisane v .NET Blazor tehnologiji in ko imate kodo pisano v tej tehnologiji, jo lahko lažje prenesete tudi na splet – v kolikor to seveda želite.

### 3.4 Nadgradnja

Bistveno je, da se zavedate omejitve, ki sem jo postavil na prvo mesto predlagane nadgradnje **in to je čim manj spremembe obstoječe kode.**

Razlog za čim manj sprememb je jasen in leži v kompleksnosti kode in pomanjkanju testov.

#### *3.4.1 Kaj dobimo po nadgradnji*

Še vedno imamo WinForms aplikacijo, ki še vedno teče v Windows okolju, le da aplikacija teče v .NET (in ne več v .NET Framework), kar pomeni, da je (vsaj v teoriji) prenosljiva.

### 3.5 Postopek nadgradnje

Postopek nadgradnje sestoji iz treh korakov, med katerimi prosto prehajate:

1. Priprava aplikacije na nadgradnjo
2. Nadgradnja aplikacije s pomočjo orodja za nadgradnjo
3. Popravek aplikacije po avtomatični nadgradnji

### *3.5.1 Priprava aplikacije za nadgradnjo*

Ta korak je zelo neotipljiv in je pravzaprav odvisen od naslednjega koraka. Oba sta namreč prepletena in iz drugega koraka se boste vračali na prvega.

V drugem koraku bomo namreč uporabili orodje, ki avtomatično nadgradi WinForms Framework aplikacijo v .NET 7.0 WinForms aplikacijo. In kot vsa druga avtomatična orodja tudi slednje ni brez težav in seveda na zmore kar brez težav prenesti vse kode v .NET 7.0 (ali novejši).

Najbolj se zatakne pri nadgradnji knjižnic. V kolikor namreč uporabljate knjižnice, ki v .NET 7.0 ne obstajajo, jih boste morali zamenjati ali pa preprogramirati kodo, da jih boste zamenjali s katero drugo knjižnico.

To je tudi najtežji korak nadgradnje, ki nima magične paličice. Ta korak je poseben za vsako aplikacijo. Microsoft je pripravil oporne točke, ki jih najdete na spletni strani (Overview of porting from .NET Framework to .NET)

### *3.5.2 Uporaba avtomatičnega orodja za nadgradnjo*

Korak 2 poteka avtomatično s pomočjo orodja in če/ko se zatakne, se povrnete v korak 1 ter spremenite zadeve ali pa spreminjate kodo v koraku 3.

V Visual Studio naložite dodatek **.NET Upgrade Assistant**, s pomočjo katerega lahko avtomatično nadgradite vaš projekt.

Kot že omenjeno v prvem koraku, vam to po vsej verjetnosti ne bo uspelo v prvem poskusu. V takšnem primeru se vrnete na korak 1, popravite/odpravite težavo in se zopet vrnite na korak 2.

Tu bi še enkrat želel poudariti, da magična paličica ne obstaja.

### *3.5.3 Popravek aplikacije po avtomatični nadgradnji*

Tudi ta korak je podobno kot korak 1 zelo neotipljiv in je povsem odvisen od konkretne aplikacije, zato je sprehod med koraki 1, 2 in 3 poljuben ter ponovljiv.

## 3.6 Uporaba Blazor Hybrid tehnologije

Po uspešni nadgradnji imamo torej aplikacijo, ki namesto v .NET Framework-u teče v .NET in v kolikor je bila to edina želja, se lahko tukaj ustavimo.

Blazor Hybrid tehnologija vam pomaga samo v toliko, da lahko začnete vašo WinForms aplikacijo pisati s spletnimi tehnologijami.

Vi imate torej v tem trenutku še vedno pred sabo vašo staro aplikacijo v vsem njenem sijaju, torej delujoč program, ki ga uporabniki poznajo, toda kode se bojite, ker je stara, nima testov in bojite se jo dotikati.

### *3.6.1 WebView*

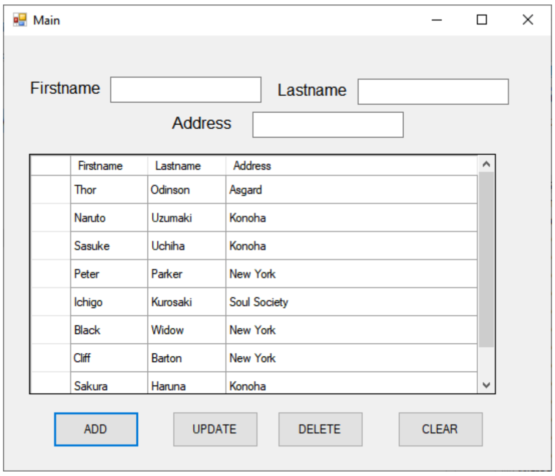
Podobno kot pri nadgradnji, tudi tukaj zadeva ni problematična s tehnološkega vidika. Najprej se morate namreč vi, kot razvijalci/uporabniki odločiti, zakaj bi želeli to uvesti in ko poznate razlog, potem je tudi lažje določiti, kam bi to uvedli.

Blazor Hybrid uporabite tako, da na obrazec dodate komponento WebView in potem v tej komponenti gostite spletne strani, ki so pisane v Blazor tehnologiji in imajo dostop do vsega, do česar lahko dostopa vaša aplikacija.

To slednje je najpomembnejše, spletna stran, ki teče v WebView komponenti ima torej dostop do vsega, do česar ima dostop uporabnik konkretnega programa na konkretnem operacijskem sistemu.

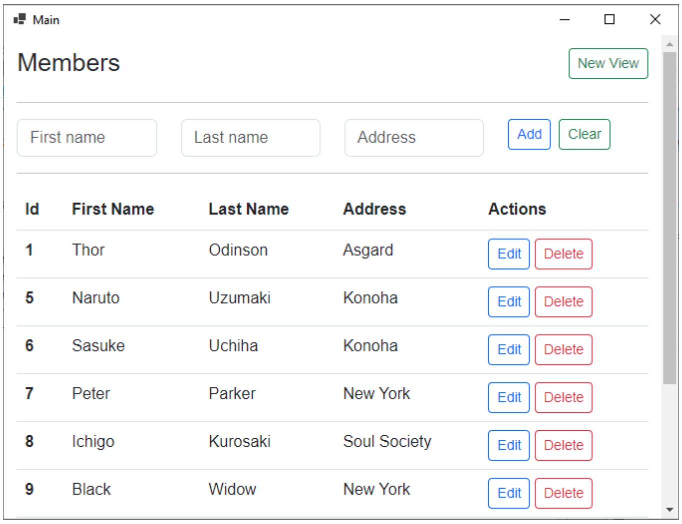
# 4 Praktični primer

Za potrebe povsem praktičnega prikaza nadgradnje obstoječe WinForms aplikacije v modernejšo obliko sem pripravil podrobno vadnico, kjer v 20-korakih nadgradim obstoječo aplikacijo, ki izgleda takole:



| Firstname | Lastname | Address |
|---|---|---|
| Thor | Odinson | Asgard |
| Naruto | Uzumaki | Konoha |
| Sasuke | Uchiha | Konoha |
| Peter | Parker | New York |
| Ichigo | Kurosaki | Soul Society |
| Black | Widow | New York |
| Cliff | Barton | New York |
| Sakura | Haruna | Konoha |

V aplikacijo, ki izgleda takole:



| Id | First Name | Last Name | Address | Actions |
|---|---|---|---|---|
| 1 | Thor | Odinson | Asgard | Edit Delete |
| 5 | Naruto | Uzumaki | Konoha | Edit Delete |
| 6 | Sasuke | Uchiha | Konoha | Edit Delete |
| 7 | Peter | Parker | New York | Edit Delete |
| 8 | Ichigo | Kurosaki | Soul Society | Edit Delete |
| 9 | Black | Widow | New York | Edit Delete |

Celoten postopek, vključno s podrobnim opisom korakov, sprememb in podobnega najdete na GitHub spletni strani: https://github.com/MPrtenjak/WinForms_2_BlazorHybrid

Seveda pa je ob tej tranziciji pomembno, da:

- Se obstoječe kode nisem dotikal.
- Nove kode je čim manj.
- Vse akcije se še vedno izvajajo v stari kode, za katero predvidevamo, da deluje pravilno.
- S spremembo torej nismo povzročili nedelovanja ali kakšnih posebnih novih hroščev, saj tudi nova aplikacija še vedno uporablja kodo stare aplikacije.

# Literatura

[1] WinForms_2_BlazorHybrid, Nadgradnja obstoječe aplikacije, https://github.com/MPrtenjak/WinForms_2_BlazorHybrid, obiskano 10.07.2023

[2] ASP.NET Core Blazor Hybrid, https://learn.microsoft.com/en-us/aspnet/core/blazor/hybrid/?view=aspnetcore-7.0, obiskano 10.07.2023

[3] Overview of porting from .NET Framework to .NET, https://learn.microsoft.com/en-us/dotnet/core/porting/, obiskano 10.07.2023

[4] Prerequisites to porting code, https://learn.microsoft.com/en-us/dotnet/core/porting/premigration-needed-changes, obiskano 10.07.2023

[5] Bring your apps to the latest .NET, https://dotnet.microsoft.com/en-us/platform/upgrade-assistant, obiskano 10.07.2023

[6] How to upgrade a Windows Forms desktop app to .NET 7, https://learn.microsoft.com/en-us/dotnet/desktop/winforms/migration/?view=netdesktop-7.0&WT.mc_id=dotnet-35129-website, obiskano 10.07.2023

[7] Install the .NET Upgrade Assistant, https://learn.microsoft.com/en-us/dotnet/core/porting/upgrade-assistant-install#install-the-visual-studio-extension, obiskano 10.07.2023

# OWASP za vse, ne samo za razvijalce

Milan Gabor

Inštitut za varnost podatkov in informacijskih sistemov (IVPIS), Viris d. o. o.,
Maribor, Slovenija
milan@viris.si

OWASP (Open ~~Web~~ Worldwide Application Security Project) je mednarodna neprofitna organizacija, ki se ukvarja z izboljšanjem varnosti programske opreme. Projekt je zasnovan kot odprtokodna skupnost, ki omogoča podjetjem, razvijalcem ter posameznikom dostop do varnostnih virov in orodij. Najbolj znan projekt OWASP je seznam »TOP 10«, ki identificira najpogostejše spletne varnostne grožnje. OWASP nudi tudi številne druge vire, kot so vodniki za razvoj varnih kod, orodja za testiranje penetracije, in izobraževalne vire za izboljšanje varnostne zavesti. Prednosti uporabe OWASP so številne. Prvič – OWASP spodbuja razvoj bolj varne programske opreme z izobraževanjem razvijalcev o najboljših praksah in pogostih napakah. Drugič – OWASP orodja omogočajo podjetjem, da preizkusijo svojo programsko opremo za morebitne varnostne luknje, kar zmanjšuje tveganje za napade. Tretjič – ker je OWASP odprtokoden, lahko organizacije prilagodijo OWASP vire svojim specifičnim potrebam. Skozi članek bomo izpostavili nove ranljivosti na seznamu OWASP TOP 10 in pregledali ostale projekte, ki so lahko koristni tako razvijalcem kot tudi drugim deležnikom v ciklu razvoja programske opreme. Prikazali bomo tudi našo statistiko odkritih pomanjkljivosti in dali poudarek na odpravi teh napak. Na koncu bomo podali praktične nasvete za še bolj varen razvoj programske opreme.

**Ključne besede:**
OWASP
varnost
spletne aplikacije
kibernetska varnost
testiranje

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.12

# 1 Uvod

V digitalni dobi, ko so spletna mesta in aplikacije temeljni del vsakodnevnega življenja, postaja vprašanje varnosti spletnih aplikacij vse bolj pereče. V tem kontekstu je OWASP ali Open Worldwide Application Security Projekt pomemben gradnik znanja in izkušenj za vse, ki se ukvarjajo z razvojem, testiranjem ter vzdrževanjem spletnih in drugih aplikacij. V začetku letošnjega leta je OWASP spremenil Web kratico v Worldwide, saj dandanes pokriva bistveno večje področje kot samo spletne aplikacije.

Zakaj pa je OWASP tako pomemben? V svetu, kjer so kibernetski napadi stalnica in kjer lahko ena sama ranljivost povzroči izgubo milijonov, je imeti zanesljiv, dosleden in ažuren vir informacij o spletni varnosti ključnega pomena. In prav tu OWASP izstopa s svojo širino in precejšnjo ponudbo na različnih področjih.

V nadaljevanju članka bomo podrobneje raziskali specifična področja, ki jih OWASP ponuja, in kako ti dokumenti in druge dobre prakse oblikujejo boljšo ter varnejšo digitalno prihodnost za vse nas.

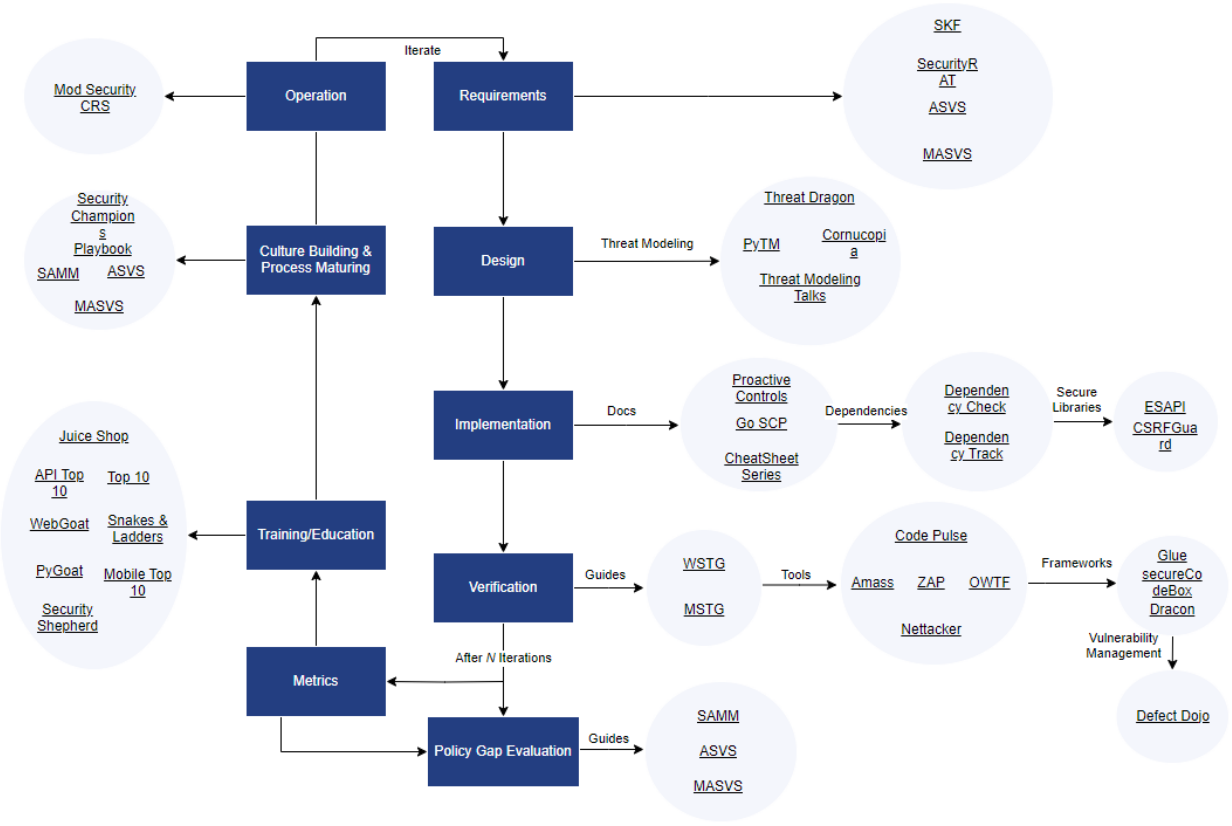
# 2 OWASP

OWASP je mednarodna neprofitna organizacija, ki se posveča izboljšanju varnosti spletnih in drugih aplikacij. Organizacija deluje na globalni ravni in združuje strokovnjake s področja informacijske varnosti, ki delijo svoje znanje in izkušnje. Glavni cilj OWASP je zagotoviti organizacijam, posameznikom in skupnosti sredstva za izboljšanje varnosti v vseh fazah razvojnega cikla programske opreme.

Nekatere ključne značilnosti OWASP so:

- **Odprtokoden in skupnostno usmerjen projekt**: OWASP vedno spodbuja transparentnost in sodelovanje. Njihovi projekti, orodja in viri so brezplačno dostopni vsem, kar omogoča široko uporabo in priznavanje v industriji.
- **Globalna prisotnost**: OWASP deluje na globalni ravni z lokalnimi poglavji (angl. chapter) v številnih državah, kjer člani redno organizirajo srečanja, delavnice in konference. V Sloveniji imamo tako aktivni dve poglavji, in sicer v Mariboru in Ljubljani.
- **OWASP Top 10:** Ta dokument navaja deset najpogostejših varnostnih tveganj za spletne aplikacije. Namenjen je ozaveščanju razvijalcev in varnostnih strokovnjakov o teh tveganjih. Vsakih nekaj let se ta seznam prilagodi trenutnim tveganjem in je tako vedno v koraku s trenutno situacijo v digitalnem ekosistemu.
- **Projekti in orodja:** OWASP financira in podpira številne odprtokodne projekte in orodja, ki pomagajo pri zaznavanju, preprečevanju in odpravljanju varnostnih tveganj. Primeri takšnih orodij vključujejo OWASP ZAP (Zed Attack Proxy) in Mod Security CRS.
- **Dokumentacija in viri:** OWASP ponuja številne vire, vključno z vodiči, priročniki in standardi za varno razvijanje, testiranje in vzdrževanje spletnih aplikacij.
- **Konference in dogodki:** OWASP redno organizira srečanja, usposabljanja in konference po vsem svetu, kjer se strokovnjaki in skupnost srečujejo, razpravljajo in izmenjujejo znanje o najnovejših trendih in najboljših praksah na področju varnosti aplikacij.
- **Izobraževanje:** Preko seminarjev, srečanj, delavnic in konferenc OWASP širi zavedanje o varnostnih vprašanjih in usposablja strokovnjake s področja IT-varnosti.
- **Podpora skupnosti:** Z vzpostavitvijo globalne mreže strokovnjakov OWASP spodbuja sodelovanje in izmenjavo znanja, kar pospešuje inovacije ter napredek na področju varnosti spletnih aplikacij.

Spekter projektov OWASP je širok in pokriva celotno področje SDLC (Software Development Lifecycle), kar pomeni, da se lahko začnejo stvari iz nabora OWASP uporabljati že pri sami specifikaciji in dizajnu ter skozi celotne faze cikla.



**Slika 1: OWASP projekti**

Vir: [1].

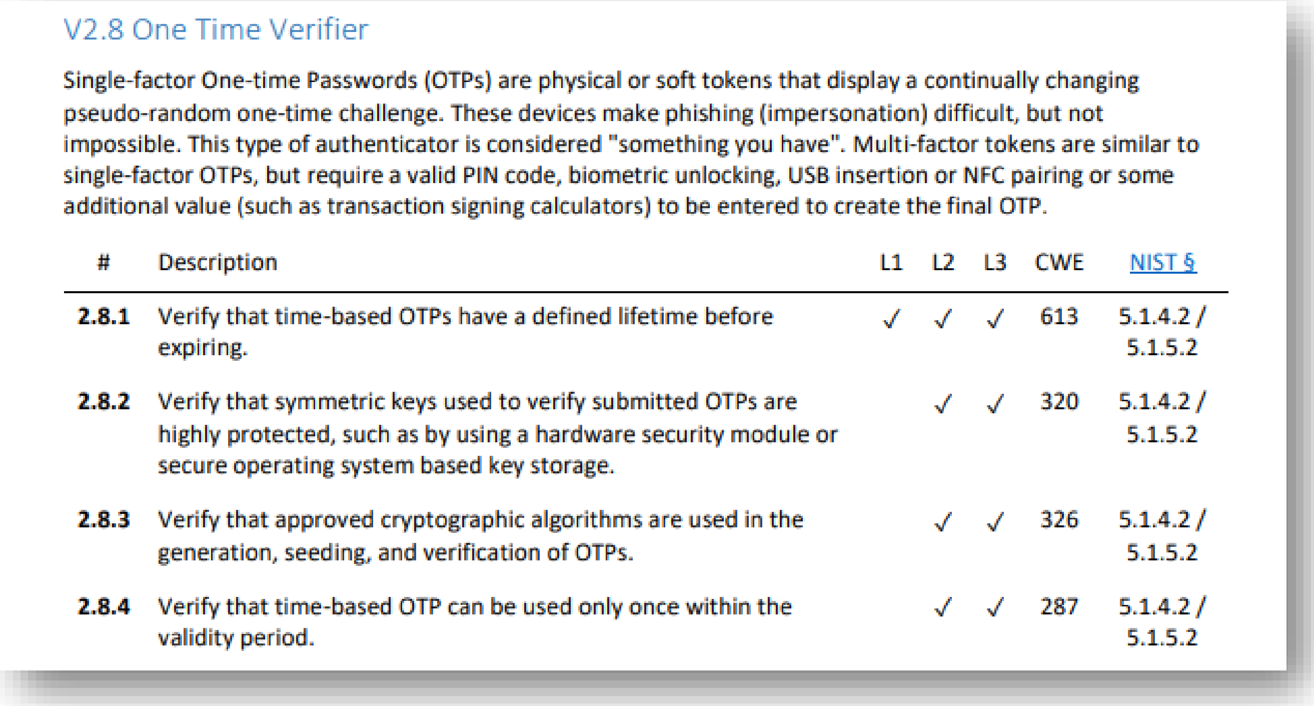
## 3 OWASP ASVS

OWASP ASVS (Application Security Verification Standard) je okvir, ki ga je razvila organizacija OWASP, da bi zagotovila podrobno osnovo za preverjanje varnosti spletnih aplikacij. Namenjen je pomoči organizacijam, razvijalcem in varnostnim strokovnjakom pri ugotavljanju in vzpostavljanju potrebnih varnostnih kontrol za spletne ter druge aplikacije. ASVS postavlja merila za različne ravni varnosti, ki so potrebne glede na občutljivost in kritičnost aplikacije ali podatkov, ki jih aplikacije obdelujejo.

Ključne značilnosti in vidiki OWASP ASVS vključujejo:

- **Podrobne zahteve:** ASVS vsebuje podrobne zahteve za varno kodiranje in oblikovanje za različne komponente in funkcije spletnih aplikacij, vključno z avtentikacijo, avtorizacijo, sejami, upravljanjem podatkov, konfiguracijo in drugimi področji na aplikativni ravni.
- **Tri ravni preverjanja:** ASVS določa tri različne ravni preverjanja (1, 2, 3), pri čemer je raven 1 osnovna, raven 2 standardna in raven 3 napredna. Organizacije lahko izberejo raven glede na kritičnost in občutljivost aplikacije.
- **Prilagodljivost:** Organizacije lahko uporabljajo ASVS kot vodnik za notranje preverjanje ali ga prilagodijo svojim specifičnim potrebam. Lahko pa ga tudi uporabljajo pri naročanju aplikacij pri zunanjih dobaviteljih.

- **Spodbujanje najboljših praks:** Poleg tega, da služi kot okvir za preverjanje, ASVS služi tudi kot orodje za izobraževanje in ozaveščanje o najboljših praksah na področju varnosti aplikacij.
- **Odprtokodni in skupnostno podprt:** Kot večina OWASP projektov je tudi ASVS odprtokodni, kar pomeni, da je na voljo za brezplačno uporabo, distribucijo in prilagajanje. Projekt je podprt s strani skupnosti, ki redno prispeva k izboljšavam in posodobitvam. V kolikor imate kakšen predlog za izboljšavo, je vedno dobrodošel.



V2.8 One Time Verifier

Single-factor One-time Passwords (OTPs) are physical or soft tokens that display a continually changing pseudo-random one-time challenge. These devices make phishing (impersonation) difficult, but not impossible. This type of authenticator is considered "something you have". Multi-factor tokens are similar to single-factor OTPs, but require a valid PIN code, biometric unlocking, USB insertion or NFC pairing or some additional value (such as transaction signing calculators) to be entered to create the final OTP.

| # | Description | L1 | L2 | L3 | CWE | NIST § |
|---|---|---|---|---|---|---|
| **2.8.1** | Verify that time-based OTPs have a defined lifetime before expiring. | ✓ | ✓ | ✓ | 613 | 5.1.4.2 / 5.1.5.2 |
| **2.8.2** | Verify that symmetric keys used to verify submitted OTPs are highly protected, such as by using a hardware security module or secure operating system based key storage. |  | ✓ | ✓ | 320 | 5.1.4.2 / 5.1.5.2 |
| **2.8.3** | Verify that approved cryptographic algorithms are used in the generation, seeding, and verification of OTPs. |  | ✓ | ✓ | 326 | 5.1.4.2 / 5.1.5.2 |
| **2.8.4** | Verify that time-based OTP can be used only once within the validity period. |  | ✓ | ✓ | 287 | 5.1.4.2 / 5.1.5.2 |

**Slika 2: Primer za uporabo OTP-žetonov**
Vir: [2].

Če povzamemo, OWASP ASVS ponuja strukturiran pristop k verifikaciji varnosti spletnih aplikacij, ki omogoča organizacijam, da razvijajo in vzdržujejo aplikacije, ki ustrezajo določenim varnostnim standardom. Za tiste, ki ne vedo, kako se lotiti postavljanja zahtev, je to zelo dobra odskočna deska za pripravo zahtev, ki jim aplikacije morajo zadoščati.

## 4 OWASP TOP 10

OWASP TOP 10 je eden najbolj priznanih in uporabljenih standardov v industriji kibernetske varnosti, ki izpostavlja najbolj kritične spletne varnostne ranljivosti. Namen tega projekta je informirati organizacije o tveganjih in motivirati razvijalce, da vključijo varnostne prakse v proces razvoja aplikacij. Predvsem pa je pomemben dejavnik, kako razvijalce prepričati, da se začnejo teh nevarnosti zavedati.

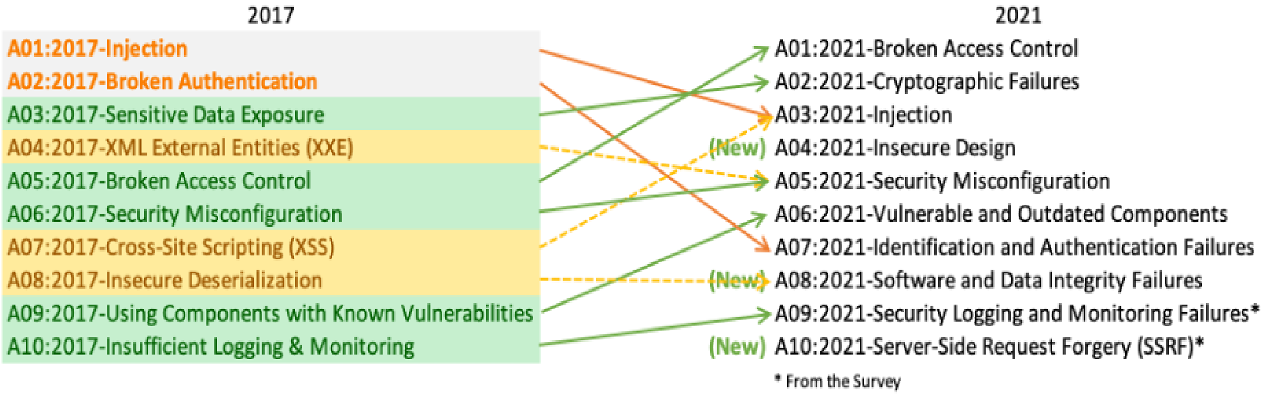
Zelo zanimiv je način, ki ga OWASP uporablja za pripravo seznama TOP 10. V grobem bi lahko ta postopek skrčili na naslednje aktivnosti:

- **Zbiranje podatkov:** OWASP sodeluje s številnimi organizacijami, strokovnjaki za varnost in raziskovalci, da zbira podatke o varnostnih incidentih in ranljivostih v spletnih aplikacijah. Večja podjetja prispevajo sezname in opise ranljivosti, na katere so naletele pri izvedbi varnostnih testiranj.

- **Analiza podatkov:** Zbrani podatki se analizirajo, da se ugotovijo najbolj razširjene in kritične ranljivosti.
- **Razvrstitev tveganj:** Ranljivosti se razvrstijo glede na njihovo kritičnost, razširjenost in potencialno škodo. Pri tem se upoštevajo različni dejavniki, kot na primer, kako lahko organizacije obravnavajo in odpravijo te ranljivosti.
- **Povratne informacije skupnosti:** Preden je nova verzija OWASP TOP 10 dokončno objavljena, se osnutek običajno deli s širšo skupnostjo. Tako OWASP zbira povratne informacije od strokovnjakov za varnost, razvijalcev in drugih zainteresiranih strani, ki jim pomagajo pri končnem oblikovanju seznama.
- **Končna izdaja:** Po zbiranju in analizi povratnih informacij skupnosti OWASP dokonča in objavi novo različico TOP 10. Dokument vključuje podrobne informacije o vsaki ranljivosti, primere, kako se te ranljivosti pojavljajo, in priporočila za njihovo odpravo.
- **Promocija in izobraževanje:** Ko je nova različica objavljena, OWASP izvaja številne izobraževalne in promocijske aktivnosti, da bi spodbudil njegovo uporabo in prepoznavnost v industriji. To vključuje delavnice, predavanja, spletne seminarje in druge izobraževalne vire.

Pomembno je razumeti, da se OWASP TOP 10 sčasoma spreminja, saj se pojavljajo nove ranljivosti in tveganja. Zato je za vse organizacije in posameznike ključnega pomena, da redno spremljajo posodobitve tega dokumenta in prilagajajo svoje varnostne prakse.

Kot lahko vidimo na spodnji sliki, se je seznam iz leta 2017 do 2021 nekoliko spremenil. Morda najbolj opazna in odmevna sprememba je ta, da je po letih kraljevanja vrivanje (angl. Injection) padlo s prvega mesta na tretje mesto. Nekatere ranljivosti so se združile, na seznamu pa so se pojavile nove ranljivosti, ki so bile predlagane in izglasovane s strani skupnosti.



**Slika 3: Prikaz starejšega in najnovejšega seznama OWASP TOP 10**
Vir: [3].

V nadaljevanju je na kratko opisana vsaka pomanjkljivost, ki je bila dodana na OWASP TOP 10 seznam za leto 2021:

- **A01:2021 – Broken Access Control** – Pomanjkljivo upravljanje dostopa omogoča napadalcem, da zaobidejo avtorizacijske mehanizme, kar lahko pripelje do neavtoriziranega izvajanja funkcij ali nepooblaščenega dostopa do podatkov.
- **A02:2021 – Cryptographic Failures** – Slabo upravljanje s kriptografskimi ključi ali pomanjkljivosti kriptografskih praks lahko pripelje do kompromitiranja zaupnih informacij, kot so gesla, ključi ali osebni podatki.
- **A03:2021 – Injection** – Varnostne ranljivosti zaradi vrivanja omogočajo napadalcem, da pošljejo zlonamerno kodo, ki se izvaja v okolju aplikacije. Najpogostejši primeri so SQL, OS in LDAP injekcije. Med vrivanje po novem sodijo tudi različni XSS-napadi.

- **A04:2021 – Insecure Design** – Varnostne pomanjkljivosti, ki izhajajo iz slabe arhitekturne zasnove aplikacije ali sistema. V kolikor je bila zasnova slaba, lahko posledično nastanejo visoki stroški s ponovnim načrtovanjem aplikacij ali sistemov in dodatnim preverjanjem zasnove.
- **A05:2021 – Security Misconfiguration** – Neprimerna, pomanjkljiva ali napačna konfiguracija varnostnih parametrov in nastavitev na vseh ravneh aplikacije ali sistema lahko še tako varno aplikacijo naredi precej nevarno.
- **A06:2021 – Vulnerable and Outdated Components** – Uporaba komponent, kot so knjižnice ali odprtokodni deli, ki vsebujejo znane varnostne ranljivosti, lahko povzroči kompromitacijo celotne aplikacije. Ugotovljeno je bilo, da se po vključitvi komponent, te v poznejših fazah ne preverjajo več tako pogosto in razvijalci tudi ne sledijo pomanjkljivostim, odkritih v njih.
- **A07:2021 – Identification and Authentication Failures** – Pomanjkljivosti v mehanizmih identifikacije in avtentikacije omogočajo napadalcem neavtoriziran dostop. Napačna uporaba avtentikacijskih mehanizmov ali nerazumevanja, kako le-ti delujejo, lahko tudi povzročijo takšen tip napak.
- **A08:2021 – Software and Data Integrity Failures** – Nezmožnost preverjanja celovitosti in pristnosti programske opreme ali podatkov lahko privede do izvrševanja nepooblaščene kode ali manipulacije podatkov. Razlog za to pomanjkljivost tiči v tem, da je bilo v zadnjih nekaj letih odkritih kar nekaj primerov, kjer so se napadalci lahko vrinili v dobaviteljsko verigo in tako infiltrirali škodljivo kodo v različne sisteme širom sveta.
- **A09:2021 – Security Logging and Monitoring Failures** – Pomanjkanje ustreznega beleženja in nadzora varnostnih dogodkov ali nezmožnost pravočasne detekcije in odziva na varnostne incidente lahko privede do velikih incidentov. Ta del je sicer povezan s konfiguracijo in postavitvijo sistemov in ni samo odvisen od razvoja programske opreme, ampak se dotika tudi DevOps področja.
- **A10:2021 – Server-Side Request Forgery (SSRF)** – Varnostna ranljivost omogoča napadalcem, da sprožijo zahteve iz zanesljivega okolja strežnika proti notranjim virom, do katerih sicer ne bi smeli imeti dostopa. Na seznam je bila uvrščena zaradi tega, ker se dostikrat pozabi na takšne tipe napadov.

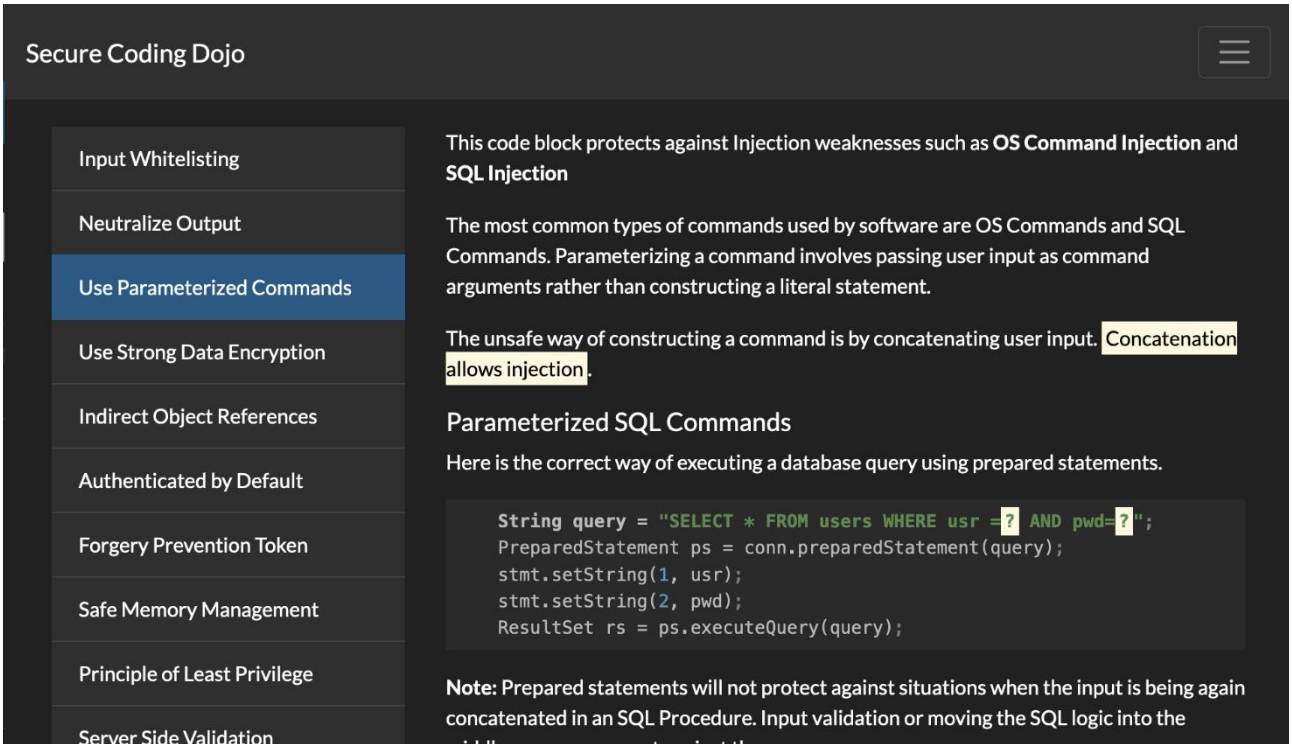
## 5 OWASP Secure Coding Dojo

Za varen razvoj programske opreme je ključnega pomena tudi varno kodiranje ter razvoj z uporabo dobrih in varnih razvojnih vzorcev. »Naštrikana« koda, ki ne sledi dobrim praksam, je največkrat kriva za odkrite varnostne pomanjkljivosti. Zato je projekt OWASP Secure Coding Dojo dober primer, kako razvijalce učiti in potem naknadno tudi trenirati v varnem razvoju programske opreme.

OWASP Secure Coding Dojo je zasnovan kot niz izzivov, s katerimi lahko uporabniki preizkusijo in izpopolnijo svoje znanje o varnem kodiranju. Nekatere prednosti tega projekta lahko strnemo v naslednje točke:

- **Modularna zasnova:** Dojo vsebuje različne modulirane treninge, ki pokrivajo široko paleto varnostnih tematik, vključno z mnogimi ranljivostmi iz OWASP TOP 10.
- **Praktično učenje:** Namesto zgolj teoretičnih vaj je Dojo zasnovan tako, da omogoča razvijalcem pridobivanje praktičnih izkušenj s testiranjem in odpravljanjem ranljivosti v resničnem okolju.
- **Samostojno učenje:** Razvijalci lahko delajo v lastnem tempu, kar pomeni, da lahko izbirajo med različnimi izzivi in moduli glede na svoje predhodno znanje in zanimanja.
- **Zasnovan za skupine:** Dojo je primeren tudi za skupinsko delo, kar omogoča skupinam, da se skupaj učijo in sodelujejo pri reševanju izzivov.

- **Odprtokodni:** Kot mnogi projekti OWASP je tudi Secure Coding Dojo na voljo kot odprtokodna rešitev, kar pomeni, da ga lahko organizacije prilagodijo in razširijo glede na svoje specifične potrebe.
- **Podpora skupnosti:** OWASP Secure Coding Dojo ima podporo širše skupnosti, ki prispeva k njegovi nadaljnji rasti in izboljšavam.



This code block protects against Injection weaknesses such as **OS Command Injection** and **SQL Injection**

The most common types of commands used by software are OS Commands and SQL Commands. Parameterizing a command involves passing user input as command arguments rather than constructing a literal statement.

The unsafe way of constructing a command is by concatenating user input. Concatenation allows injection.

Parameterized SQL Commands

Here is the correct way of executing a database query using prepared statements.

```
String query = "SELECT * FROM users WHERE usr =? AND pwd=?";
PreparedStatement ps = conn.preparedStatement(query);
stmt.setString(1, usr);
stmt.setString(2, pwd);
ResultSet rs = ps.executeQuery(query);
```

**Note:** Prepared statements will not protect against situations when the input is being again concatenated in an SQL Procedure. Input validation or moving the SQL logic into the

**Slika 4: Primer uporabe Secure Coding Dojo**
Vir: [4].

Če povzamemo, je OWASP Secure Coding Dojo odlično sredstvo za vsakogar, ki želi izboljšati svoje veščine varnega kodiranja, ne glede na to, ali je popoln začetnik ali izkušen strokovnjak za varnost. Omogoča praktično izobraževanje in je odlična dopolnitev tradicionalnemu varnostnemu izobraževanju.

## 6 OWASP Cheat Sheet Series

Ubiranje bližnjic nekako ni dobra praksa v akademski sferi in se lahko velikokrat izkaže za tek na kratke proge, vendar pa lahko dobra in kompaktna vsebina koristi predvsem tistim, ki ne marajo obširne dokumentacije in so bolj praktično naravnani. OWASP Cheat Sheet Series je zbirka dokumentov, ki jih je razvila organizacija OWASP in so zasnovani kot hitri referenčni vodiči za različna področja varnosti spletnih in drugih aplikacij. Vsak "cheat sheet" ponuja konkretne smernice, najboljše prakse in nasvete za določen vidik varnosti ter tako omogoča razvijalcem, arhitektom in varnostnim strokovnjakom, da hitro najdejo informacije o specifični varnostni temi in to koristno uporabijo pri svojem delu.

Naj naštejemo nekaj dobrih strani OWASP Cheat Sheet Series, ki jih lahko strnemo v naslednje alineje:

- **Kratek in jedrnat format:** "Cheat sheets" so namenjeni hitremu iskanju informacij in se osredotočajo na bistvene informacije, potrebne za zagotavljanje varnosti.

- **Pokrivanje širokega spektra tem:** Serija vsebuje vodiče za različne vidike varnosti, kot so avtentikacija, CSRF-zaščita, šifriranje podatkov, varna konfiguracija, preprečevanje napadov SQL-injekcije in mnogo drugih.
- **Praktične smernice:** Vsak "cheat sheet" ponuja praktične smernice in primere, kako pravilno implementirati določene varnostne mehanizme ali kako preprečiti določene ranljivosti.
- **Redno posodobljeno:** Zbirka se redno posodablja, da odraža najnovejše varnostne standarde, ranljivosti in grožnje ter je podkrepljena s kopico strokovnjakov, ki se dnevno srečujejo s takšnimi grožnjami.
- **Podpora skupnosti:** Kot večina OWASP projektov je tudi Cheat Sheet Series odprtokodni projekt, ki ga podpira in redno posodablja široka skupnost varnostnih strokovnjakov.

Java Security Cheat Sheet

Injection Prevention in Java

This section aims to provide tips to handle *Injection* in Java application code.

Sample code used in tips is located here.

What is Injection

Injection in OWASP Top 10 is defined as following:

*Consider anyone who can send untrusted data to the system, including external users, internal users, and administrators.*

General advices to prevent Injection

The following point can be applied, in a general way, to prevent *Injection* issue:

1. Apply **Input Validation** (using "allow list" approach) combined with **Output Sanitizing+Escaping** on user input/output.

**Slika 5: Primer priporočil za Javo**

Vir: [5].

V bistvu je lahko OWASP Cheat Sheet Series zelo dragoceno in priročno sredstvo za vsakogar, ki dela na področju razvoja spletnih in drugih aplikacij in želi hitro dostopati do zanesljivih informacij o specifičnih varnostnih temah. Seveda pa lahko vsak strokovnjak doda ali dopolni področja, če misli, da so pomanjkljivo spisana ali netočna.

## 7 Rezultati naših izvedenih pregledov aplikacij

Pri pregledu rezultatov naših testiranj spletnih aplikacij v zadnjih dveh letih lahko ugotovimo, da je večina odkritih pomanjkljivosti na seznamu OWASP TOP 10. Ko smo izluščili rezultate in pripravili kratko analizo teh pomanjkljivosti, lahko iz naših odkritih pomanjkljivosti izpostavimo naslednje:

- **Vrivanje kode** – V veliki večini primerov smo odkrili različna vrivanja skripte kode, in sicer Reflected in Stored XXS, v nekaterih redkih primerih tudi primerke DOM vrivanja. V enem primeru smo uspeli na platformi pripraviti takšen vnos XSS, ki bi lahko imel posledico avtomatskega širjenja po celotni platformi za vse uporabnike (Wormable Stored XXS). V vseh primerih je težava pri pomanjkanju validacije vhodnih podatkov in nerazumevanja pomembnosti te validacije tako na odjemalčevi kot strežniški strani. SQL-vrivanje smo sicer našli, vendar so bile pri večini pregledov aplikacije starejšega izvora, ki se jih prej nikoli ni pregledalo.

- **Pomanjkljiva kontrola dostopa** – tukaj je bila večina pomanjkljivosti odkrita na način, da se na strežniku ni preverjalo, ali ima prijavljeni uporabnik pravice klica določene funkcionalnosti ali dostopa do podatkov. V nekaj primerih je bilo odkrito, da so se pravice preverjale samo na strani odjemalca, na strani strežnika pa ne in je imel uporabnik poljuben dostop do podatkov na strežniku. V nekaterih primerih so razvijalcu samo iz uporabniškega vmesnika odstranili opcije za izbiro, na strežniku pa potem niso več preverjali, ali ima uporabnik pravice za klic funkcionalnosti.

- **Poljubno preusmerjanje** – veliko število aplikacij omogoča preusmerjanje na druge URL-naslove po prijavi, odjavi ali drugih akcijah. Pri testiranju je bilo velikokrat ugotovljeno, da je možno uporabnika poljubno preusmerjati na potencialno zlonamerne strani, saj ni bilo preverjanja na strani strežnika, katere preusmeritve so dovoljene in katere ne.

- **Neustrezne kontrole za strojno izpolnjevanje obrazcev** – pri skoraj večini aplikacij smo ugotovili, da nimajo zaščite pred avtomatskimi orodji ali boti, ki so lahko brez omejitev klicali končne API-vmesnike ali spletne strani, ter tako ugibali gesla, ali povzročali večjo obremenitev na strežniku.

- **Neustrezna politika gesel** – pri večjem številu analiz smo odkrili, da aplikativni vmesniki sporočajo uporabniku, kakšna je politika gesel. Pri nastavitvi gesla se je potem samo geslo preverjalo na strani uporabnika, na strežniku pa se pri klicu API-vmesnika to ni več preverjalo in smo si lahko nastavili zelo enostavna gesla, kot na primer a ali 1.

- **Neustrezno obravnavanje napak** – ta tip pomanjkljivosti je izstopal pri vseh vrstah aplikacij in uporabljenih tehnologijah. Razvijalci imajo še vedno radi prenos in razkrivanje napak na uporabniški strani, pa naj gre to za klice API-vmesnika ali preko prikaza na spletnih straneh. Te napake razkrivajo velikokrat celotne stack-trace, tehnologije in tudi poti, kjer je koda shranjena na strežniku.

## 8 Varnostna priporočila za bolj varne aplikacije

Varen razvoj aplikacij je proces, kjer so varnostne prakse integrirane skozi celotno življenjsko dobo razvoja aplikacij in zato so priporočila, kako zagotoviti ta proces, zelo priročna. Takšna priporočila nam lahko pomagajo pri zagotavljanju, da so aplikacije odporne na zlonamerne napade in varujejo občutljive podatke. Če bi na podlagi naših izkušenj podali priporočila, bi jih lahko strnili v naslednje alineje:

- **Zaščita podatkov:** Vsa komunikacija, ki vključuje občutljive informacije (kot so podatki o plačilu ali osebne informacije), naj poteka preko šifrirane povezave (npr. HTTPS).

- **Zanesljivo shranjevanje gesel:** Uporabite močne algoritme za zgoščevanje gesel in dodajte sol. Nikoli ne shranjujte gesel v čisti obliki ali jih shranjujte v vmesnih shrambah.

- **Validacija vhodnih podatkov:** Vedno preverite, filtrirajte in očistite vhodne podatke. Nikoli ne zaupajte podatkom, ki prihajajo od uporabnikov ali drugih aplikacij in API vmesnikov brez predhodne validacije.
- **Validacija na strani strežnika**: Čeprav lahko izvajate validacijo vhodnih podatkov na strani odjemalca, je ključno, da to ponovno izvedete na strani strežnika, saj lahko napadalci obidejo preverjanje na strani odjemalca.
- **Omejitev dostopa:** Zagotovite, da so funkcije in podatki dostopni le tistim uporabnikom, ki imajo ustrezne avtorizacije. To velja tako za uporabniški vmesnik kot za podatkovne vire in API-je.
- **Redne varnostne posodobitve:** Redno posodabljajte vso programsko opremo, vključno z operacijskimi sistemi, spletnimi strežniki, bazami podatkov in vsemi uporabljenimi knjižnicami.
- **Obramba v globini:** Ne zanašajte se samo na eno varnostno rešitev. Uporabljajte večplastno obrambo, kot so požarni zidovi, WAF (Web Application Firewall), IDS/IPS sistemi in druge varnostne rešitve.
- **Zmanjšajte površino napada:** Omejite število dostopnih točk v vaši aplikaciji, onemogočite nepotrebne storitve in funkcionalnosti ter redno preverjajte konfiguracijo sistema.
- **Omejitev pravic in privilegijev:** Uporabite načelo najmanjših privilegijev; to pomeni, da uporabnikom in procesom dodelite le tiste pravice, ki jih resnično potrebujejo.
- **Pravilno upravljanje napak:** Ne razkrivajte preveč informacij v sporočilih o napakah. Tehnične podrobnosti lahko napadalcu ponudijo dragocene informacije, ki jim lahko koristijo potem pri modeliranju napadov.
- **Varnost API-jev:** Če razvijate ali uporabljate API-je, upoštevajte "OWASP API Security Top 10", da se zavedate in obravnavate specifična tveganja, povezana z API-ji.
- **Varnost zasnovana od začetka:** Vključite varnost že na začetnih stopnjah razvojnega cikla, ne pa kot naknadni dodatek, saj boste s tem prihranili čas in denar.
- **Izobraževanje in usposabljanje:** Redno usposabljajte svoje razvijalce in druge člane ekipe o varnostnih najboljših praksah ter aktualnih grožnjah.
- **Odzivanje na incidente:** Pripravite načrt odzivanja na varnostne incidente. Hitro in učinkovito odzivanje na varnostne grožnje lahko zmanjša potencialno škodo in pokaže resnost organizacije, da je pripravljena tudi na takšne izjemne dogodke.

# 9 Zaključek

V svetu, ki je vse bolj povezan in digitaliziran, OWASP igra ključno vlogo pri zagotavljanju, da spletne in druge aplikacije, ki jih vsakodnevno uporabljamo, ostanejo varne pred grožnjami. Njihovi viri, orodja in skupnostna naravnanost pomenijo, da je OWASP ne le vir informacij, ampak tudi središče globalnega gibanja za varnejši internet.

OWASP je ključni vir za vsakogar, ki je vpleten v razvoj, testiranje, vzdrževanje ali preverjanje varnosti spletnih aplikacij. Zaradi svoje odprtokodne narave in skupnostnega pristopa je postal OWASP eden izmed vodilnih virov na področju varnosti spletnih aplikacij, zato se vsekakor izplača preveriti projekte, dokumente in druge vire, ki jih OWASP ponuja posameznikom in organizacijam.

Zaradi vsega naštetega je vredno preveriti, kako lahko OWASP vključite v svoje projekte in s tem prispevate k bolj varnemu ekosistemu.

## Literatura

[1] https://owasp.org/projects/, OWASP projects spletna stran, obiskano 1. 8. 2023.

[2] https://github.com/OWASP/ASVS/raw/v4.0.3/4.0/OWASP%20Application%20Security%20Verification%20Standard%204.0.3-en.pdf, Dokument s spletne strani, obiskano 1. 8. 2023.

[3] https://owasp.org/Top10/, OWASP spletna stran, obiskano 1. 8. 2023.

[4] https://owasp.org/www-project-top-ten/, OWASP spletna stran, obiskano 1. 8. 2023.

[5] https://cheatsheetseries.owasp.org/cheatsheets/Java_Security_Cheat_Sheet.html, Spletna stran, obiskano 1. 8. 2023

# OID4VC: Izdajanje in deljenje preverljivih poverilnic na osnovi OpenID Connect

Martin Domajnko, Vid Keršič, Muhamed Turkanović

Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko,
Maribor, Slovenija
martin.domajnko@student.um.si, vid.kersic@um.si, muhamed.turkanovic@um.si

V današnjem digitalnem okolju je potreba po varnih sistemih za upravljanje identitet postala vse bolj pomembna. Ena takšna rešitev je koncept preverljivih poverilnic. Njihov podatkovni model je tudi definiran kot standard W3C. VC omogočajo dokazovanje in deljenje informacij o digitalni identiteti na kriptografsko preverljiv način. Posameznikom omogočajo tudi nadzor nad njihovimi osebnimi podatki in selektivno razkrivanje le teh za namene preverjanja. Razvoj specifikacij OpenID za izdajanje preverljivih poverilnic je odprla nove možnosti za izboljšanje sistemov za upravljanje identitet. OIDC je sodoben overitveni protokol za varno preverjanje identitete. Uporabnikom omogoča poenostavljen postopek prijave na več spletnih strani s pomočjo uporabe obstoječih računov pri ponudnikih digitalne identitete, kot je na primer Google. V prispevku bomo predstavili specifikacije in izvedbo implementacije OID4VC. Validacija integracije bo izvedena na strežniku implementiranim s pomočjo knjižnice Veramo ter denarnice Masca za sprejem in hrambo VC. S tem uporabimo varen široko razširjen protokol in omogočimo OIDC ponudnikom, da postanejo izdajalci preverljivih poverilnic, uporabnikom pa, da na preprost in varen način prejemajo svoje poverilnice. Implementacija ponuja tudi podporo decentraliziranim identitetam in identifikatorjem. Prispevek zaključimo z analizo prednosti in slabosti, ki jih ima predstavljena rešitev, in pregledom novih integracij, ki jih le ta omogoča.

**Ključne besede:**
decentralizirana identiteta
digitalna identiteta
samo-upravljana identiteta
preverljive poverilnice
OpenID Connect

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.13

# 1 Uvod

Digitalna identiteta je ena izmed najpomembnejših komponent, saj se z njo srečamo na skoraj vsaki spletni strani in služi za identifikacijo uporabnikov na spletu. Že od samega začetka svetovnega spleta ta pretežno temelji na uporabniških imenih in geslih ter se skozi desetletja ni spremenila veliko kljub številnim pomanjkljivostim. Ker naše interakcije in življenje v fizičnem svetu postaja vse bolj prepleteno z digitalnim, potreba po novih, bolj varnih in zanesljivih sistemih za upravljanje digitalnih identitet postaja vse večja. Tradicionalne metode verificiranja digitalnih identitet so pogosto neprijazne in varnostno ranljive, zato se to področje zelo hitro razvija.

Eden izmed novih naprednih načinov za implementacijo sistema za digitalno identiteto je samo-upravljana identiteta (angl. self-sovereign identity, SSI), ki temelju na principih lastništva podatkov s strani uporabnikov, popolno samokontrolo nad upravljanjem identitete in povečano zasebnost uporabnikov [22]. Jedro SSI predstavlja kriptografija in sicer asimetrična kriptografija oz. kriptografija javnega ključa [37], na kateri temelji tudi tradicionalno podpisovanje digitalnih dokumentov kot tudi podpisovanje transakcij pri tehnologiji veriženja blokov (angl. blockchain) [15]. Zaradi podobnih vrednot kot tudi osnovnih tehničnih primitov, sta SSI in blockchain oz. Web3 tehnologije obravnavne skupaj, in sicer kot nova tretja generacija svetovnega spleta, ki razvija internet v smer decentralizacije, digitalnega lastništva in zasebnosti uporabnikov [14].

Ena izmed ključnih komponent SSI so preverljive poverilnice (angl. Verifiable Credential, VC), čigar standard je priporočen tudi s strani organizacije The World Wide Web Consortium (W3C) [40]. Podatkovni model VC služi kot striktno definiran standard za vse možne tipe digitalnih dokumentov in ostale informacije o uporabnikovi identiteti, na primer vozniško dovoljenje, diploma in razni certifikati. VC prav tako omogoča uporabniku deljenje in dokazovanje lastnih informacij na interoperabilen in kriptografsko preverljiv način. Uporaba novih in naprednih kriptografskih pristopov omogoča še povečan nadzor nad podatki in njihovo delitvijo, na primer selektivno razkritje (angl. selective disclosure) in dokazovanje lastništva podatkov namesto njihovih dejanskih vrednosti z uporabo zero-knowledge proofs [40].

Medtem ko sam standard W3C VC definira strukturo digitalnih dokumentov, v njem ni določeno, kako poteka prenos teh dokumentov od uporabnika do entitet, s katerimi je potrebno deliti podatke. V ta namen je bilo razvitih več protokolov, pri čemer sta najbolj znana in uporabljena DIDComm in OpenID protokol za preverljive poverilnice (angl. OpenID for Verifiable Credentials, OID4VC) [3, 18]. Ceprav je protokol DIDComm nastal znotraj SSI skupnosti, se protokol OID4VC navadno pogosteje integrira v sisteme za digitalno identiteto, saj temelji na obstoječem in široko razširjenem protokolu OpenID Connect (OIDC), zaradi česar je njegova integracija lažja in enostavnejša. Samo specifikacijo OID4VC razvija delovna skupina OpenID Connect (angl. OpenID Connect Working Group) in je tako odprla nove možnosti za izboljšanje sistemov za upravljanje digitalne identitete. OIDC je overitvena plast, zgrajena na osnovi protokola OAuth 2.0, in je sodoben overitveni protokol za varno preverjanje identitete in funkcionalnost enkratne prijave (angl. Single Sign-On – SSO) za spletne in mobilne aplikacije [7, 16]. Uporabnikom omogoča poenostavljen postopek prijave na več spletnih strani s pomočjo uporabe obstoječih računov pri večjih ponudnikih digitalne identitete, kot so Google, Facebook in Microsoft. Z dodatno specifikacijo OID4VC je uporabnikom omogočena prijava s SSI.

V prispevku bomo raziskali in predstavili specifikacije ter izvedbo implementacije OID4VC, pri čemer se bomo osredotočili na integracijo samega protokola OIDC. Validacija integracije bo izvedena na strežniku implementiranim z odprto-kodno knjižnico Veramo in digitalno denarnico Masca, s katero bomo prejeli in v njej hranili VC-je. Pri tem uporabimo varen in široko razširjen protokol ter omogočimo OIDC ponudnikom, da postanejo izdajatelji preverljivih poverilnic, uporabnikom pa, da na preprost in varen način prejmejo svoje poverilnice. Implementacija ponuja tudi podporo decentraliziranim identitetam in identifikatorjem (angl. decentralized identifier, DID), kar še dodatno uporabnikom poveča nadzor na svojo osebno digitalno identiteto [2]. Prispevek zaključimo z analizo prednosti in slabosti, ki jih ima predstavljena rešitev in uporaba OID4VC, ter pregledom novih integracij, ki jih le ta omogoča.

# 2 Osnovni koncepti

## 2.1 Single Sign-On (SSO)

Single Sign-On (SSO) je mehanizem za avtentikacijo, ki omogoča uporabnikom dostop do več aplikacij oz. storitev z uporabo ene digitalne identitete [1]. Glavni cilj SSO je poenostaviti in izboljšati uporabniško izkušnjo tako, da uporabnikom ni potrebno uporabljati različnih uporabniških in gesel pri uporabi različnih sistemov. S SSO se uporabnik prijavi enkrat, nato pa lahko dostopa do različnih aplikacij, ne da bi moral znova vnašati svoje poverilnice oz. podatke o identiteti. SSO izboljša uporabniško izkušnjo, poveča varnost in zmanjšuje breme za uporabnike, ki ne rabijo več upravljati več prijavnih poverilnic.

Postopek SSO navadno vključuje tri glavne komponente oz. entitete: ponudnika identitete (angl. identity provider, IDP), ponudnika storitev (angl. service provider, SP) in končnega uporabnika. Ko uporabnik poskuša dostopati do storitve, SP preusmeri uporabnika na IDP za overjanje oz. avtentikacijo. Ko je uporabnik overjen, IDP vrne SP-ju žeton, ki uporabniku omogoči dostop do zahtevane storitve. Nekateri izmed najbolj priljubljenih protokolov in tehnologij za SSO so SAML (Security Assertion Markup Language), OAuth 2.0 in OpenID Connect (OIDC).

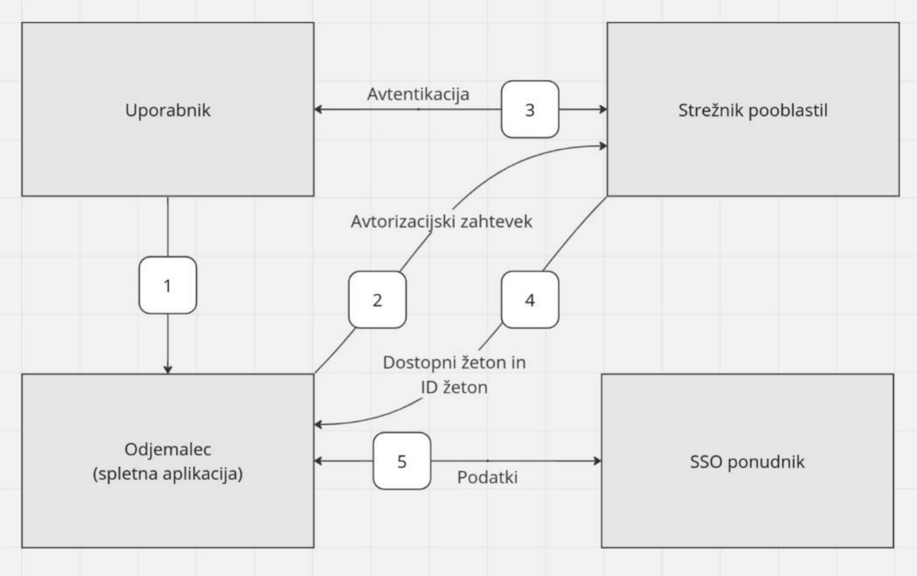
### OAuth 2.0

OAuth 2.0 je široko razširjen avtorizacijski protokol, ki omogoča varen dostop do virov v imenu uporabnika, ne da bi ta delil svoje poverilnice [16]. Zasnovan je predvsem kot način za dostop do nabora virov, na primer oddaljenih API-jev ali uporabniških podatkov.

Protokol OAuth 2.0 definira štiri glavne vloge: lastnik vira (uporabnik), odjemalec (aplikacija, ki zahteva dostop), avtorizacijski strežnik pooblastil (izda dostopne žetone po overjanju uporabnika) in strežnik virov (hrani zaščitene vire). OAuth 2.0 vključuje pridobivanje dostopnega žetona od strežnika pooblastil, ki ga odjemalec nato predstavi strežniku virov za dostop do zaščitenih virov. Protokol je široko sprejet zaradi svoje prilagodljivosti, enostavne implementacije in široke podpore za različne scenarije, kot so mobilne aplikacije in spletne storitve.

### OpenID Connect (OIDC)

OpenID Connect (OIDC) je avtentikacijski sloj, ki je zgrajen na osnovi protokola OAuth 2.0 [7]. Poleg zagotavljanja dostopa do virov, omogoča odjemalcem tudi preverjanje identitete uporabnika. Medtem ko se OAuth 2.0 osredotoča na dostop do virov, OIDC doda mehanizem za avtentikacijo z uporabo Json Web Token (JWT), ki zagotavlja identiteto uporabnikov. V protokolu OpenID Connect odjemalec zahteva avtentikacijo od strežnika pooblastil, ta pa overi uporabnika in vrne identifikacijski žeton (ID žeton) z informacijami o uporabniku. ID žeton je v formatu JWT, ki je digitalno podpisan in ga odjemalec lahko preveri ter pridobi identiteto uporabnika. OIDC zagotavlja standardiziran način izvajanja avtentikacije in se široko uporablja za vključevanje avtentikacije v aplikacije in storitve, hkrati pa izkorišča varnostne funkcije protokola OAuth 2.0. Celotna arhitektura OIDC je prikazana na sliki 1.

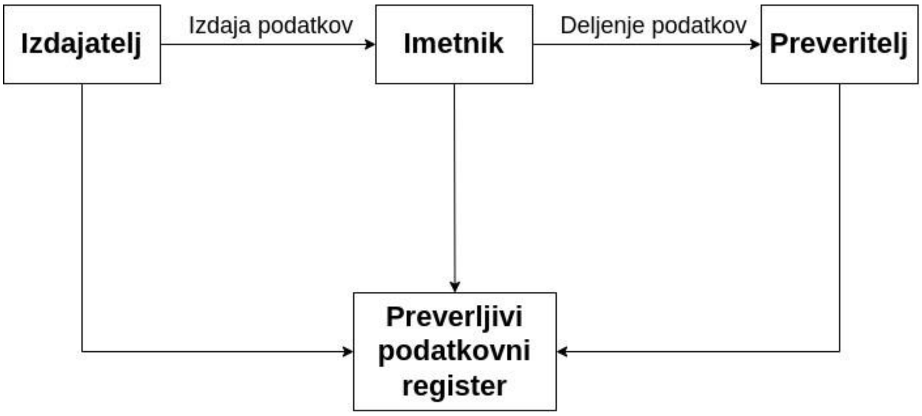


**Slika 1: Prikaz arhitekture OpenID Connect.**

## 2.2 Samo-upravljana identiteta (SSI)

Samo-upravljana identiteta (SSI) je eden izmed novih in naprednih načinov implementacije digitalne identitete in visoko-nivojsko preslikava stanje iz realnega fizičnega sveta [22]. Uporabnikom so izdani digitalni dokumenti, ki so nato shranjeni v njihovih digitalnih denarnicah. Svoje digitalnega dokumente lahko nato delijo s komerkoli želijo. SSI v ospredje postavlja lastništvo, zasebnost in popolno kontrolo nad digitalno identiteto. Vsako implementacijo digitalne identitete navadno sestavlja dve glavni komponenti in to so identifikatorji uporabnikov ter njihovi podatki. V primeru SSI sta to standarda DID in VC, pri čemer prvi definirana globalno unikatne identifikatorje uporabnikov in pravila upravljanja z njimi, medtem ko drugi defininra strukturo in lastnosti podatkovne strukture podatkov [2, 40]. Za razvoj skoraj vseh standardov s skupnosti SSI skrbita organizaciji W3C in Decentralized Identity Foundation (DIF) [41, 4].

SSI temelji na dveh glavnih procesih: izdajanje VC in njihovo deljenje. Ta dva procesa se izvjata med tremi tipi entitet in sicer izdajateljem (angl. issuer), imetnikom (angl. holder) in preveriteljem (angl. verifier) [22]. Celoten SSI cikel je prikazan na sliki 2. Na preprostem primeru visokošolske diplome bi bila v vlogi izdajatelja fakulteta oz. univerza, imetnik bi bil diplomant in preveritelj podjetje, kamor se bo po študiju zaposlil diplomant. Imetnik pridobi VC od izdajatelja in jo shrani v svojo digitalno denarnici, od koder jo nato lahko deli s preveritelji. Pri izdajanju in deljenju vse entitete verificirajo digitalne podpise podatkov in posledično tudi identifikatorje entitet, s katerimi komunicirajo. Verifikacija temelji na kriptografiji in sicer kriptografiji javnega ključa. Izdajatelj s svojim zasebnim ključem digitalno podpiše podatke, prav tako jih imetnik digitalno podpiše pri njihovem deljenju. Tako lahko vsakdo kriptografsko preveri, da so podatki bili izdani oz. deljeni s strani lastnika podatkov. Medtem ko je zasebni ključ entitet shranjen na varni lokaciji, na primer digitalni denarnici, mora biti javni ključ javno dostopen, saj preveritelj in izdajatelj ne komunicirata neposredno. V ta namen se javni ključ (z ostalimi potrebnimi podatki) registrirajo na preverljivi podatkovni register (angl. Verifiable Data Registry, VDR) [22]. Na VDR se zabeleži tudi stanje oz. veljavnost VC v obliki binarne vrednosti. VDR je pogosto implementiran kot decentralizirano omrežje, saj le v takšnem primeru SSI ostane povsem v decentralizirani obliki. Kadar uporabnik deli svoje podatke oz. dokumente s preveriteljem, generira preverljivo prezentacijo (angl. Verifiable Presentation, VP), ki je prav tako definirana v standardu VC [40]. Pri deljenju lahko uporabi tudi eno izmed naprednih kriptografskih metod, na primer selektivno razkritje ali ZKP [40].



**Slika 2: Model zaupanja pri digitalni identiteti SSI.**

Sistem in ekosistem za digitalno identiteto, ki temelji na principih SSI, razvija tudi Evropska komisija (angl. European Commission) v sodelovanju z Evropskim blockchain partnerstvom (angl. European Blockchain Partnership, EBP) v sklopu projektov European Blockchain Services Infrastructure (EBSI) in European Self-Sovereign Identity Framework (ESSIF) [6]. Evropska komisija je prav tako pred kratkim izdala navodila za razvoj digitalnih denarnic za decentalizirane digitalne identitete, pri čemer je vključenih veliko standardov iz skupnosti SSI [5]. Omenjena digitalna denarnica je namenjena prebivalcem Evropske unije (angl. European Union, EU), pri čemer je poudarek na storitvah v javni upravi.

### Decentralizirani identifikatorji (DID)

DID so globalno unikatni identifikatorje uporabnikove oz. entitetine samoupravljane identitete, pri čemer so ti generirani naključno oz. iz kriptografskega ključa. Struktura identifikatorjev temelji na široko razširjenem spletnem standardu Uniform Resource Identifier (URI) [2]. Specifikacijo DID je prav tako pod okriljem organizacije W3C. Primer DID identifikatorja je did:ebsi:zxAFkP2dnB5oL2A5iaaK5K3, kjer:

- prvi del (did) striktno izraža, da je prikazan identifikator decentraliziran
- drugi del (ebsi) sporoča izbrano DID metodo (angl. DID method); tj. podvrsta oz. specifikacija za en tip decentraliziranih identifikatorjev, ki definira pravila njihovega upravljanja in pogosto tudi omrežje, ki se uporablja za omenjene DID-e
- tretji del (zxAFkP2dnB5oL2A5iaaK5K3) predstavlja vrednost, ki mora biti globalno unikatna znotraj izbrane DID metode

Prikazan identifikator spada pod DID metodo ebsi in se uporablja znotraj omenjenega SSI ekosistema, pri čemer so dodatne informacije o identifikatorju in entiteti za njim shranjene na EBSI blockchain omrežju [6]. Vsak DID se s pomočjo namenske storitve imenovane resolver transformira v DID dokument (angl. DID document), ki vsebuje vse dodatne informacije o DID-u [2]. To so vsi javni ključi identifikatorja, trenutni administrator oz. kontroler (angl. controller), URL pri izdajateljih in preveriteljih, itd. DID dokument je navadno v obliki JSON-LD in vse njegove možne lastnosti (angl. property) so definirane v DID metodi. Na tem mestu je potrebno omeniti, da vse DID metode ne podpirajo posodabljanja DID dokumentov - to sta na primer DID metodi did:key in did:pkh. Te DID metode se navadno uporabljajo za enkrat uporabo, preprostejše primere uporabe ali so tesno povezane z uporabnikovimi blockchain naslovi. Ostale DID metode, kot so na primer did:ebsi, did:polygonid in did:web, po drugi strani omogočajo posodabljanje DID dokumentov, kar omogoča napredne uporabe identifikatorjev, na primer rotacijo ključev ali kontrolerjev.

### Preverljive poverilnice (VC in VP)

VC definira podatkovni format oz. kontejner, s katerim je mogoče predstaviti katerikoli digitalni dokument [40]. Tudi ta specifikacija je pod okriljem organizacije W3C. Medtem ko DID definirana unikatne identifikatorje uporabnikov, VC-ji predstavljajo podatke (najpogosteje podatke o njihovi identiteti). Ene izmed glavnih prednosti uporabe preverljivih poverilnic so zmožnost njihove strojne berljivosti, enoten standard za vse tipe digitalnih dokumentov in kriptografska preverljivost izdajatelja ter imetnika dokumentov. Vsak VC navadno vsebuje informacije o izdajatelju, času izteka, tipu digitalnega dokumenta, informacije o imetniku, vrsto kriptografskega podpisa in ostale dodatne informacije specifične za določen tip digitalnega dokumenta [40].

Uporabniki svoje VC-je navadno hranijo v svojih digitalnih denarnicah. Priporočen in najpogostejši format za VC je JSON, pri čemer je ta najpogosteje v obliki JWT ali JSON-LD. Izbira formata VC je v večini odvisna od potreb in zahtev po uporabi določene vrste kriptografskega digitalnega podpisa. Vsak tip VC oz. digitalnega dokumenta mora slediti vnaprej definirani JSON Schemi, v kateri se zabeleži vse zahtevane lastnosti JSON-a, kot tudi katera polja so obvezna in katera opcijska. Verifikacija digitalnega podpisa poteka z resolucijo DID izdajatelja, kjer se iz DID dokumenta pridobi njegov javni ključ, ki se uporabi ob preverjanju digitalnega podpisa. Kadar imetnik VC-jev svoje dokumente deli s preveriteljem, zgenerira preverljivo prezentacijo (VP), v katero se lahko vključi en ali več digitalnih dokumentov. Z uporabo novih in naprednih kriptografskih pristopov, se lahko varnost in zasebnost uporabnika še dodatno izboljša, predvsem z uporabo selektivnega razkritja, kjer se razkrijejo le nujno potrebne vrednosti, in ZKPs, kjer se dokazuje prisotnost vrednosti in ne njihove dejansko vrednosti.

Sam standard VC ne definira enotnega načina, kako prenesti digitalne dokumente med različni entitetami v SSI ciklu, na primer izdajanje VC od izdajatelja do imetnika in deljenje od imetnika do preveritelja. V ta namen so bili razviti različni protokoli, pri čemer po uporabi prevladujeta DIDComm in OID4VC [3, 18]. Medtem ko je prvi nastal znotraj skupnosti decentralizirane identitete in podpira naprednejše funkcionalnosti DID-ov in VC-jev, se pogosteje uporablja in integrira v obstoječe sisteme OID4VC. Razlog za to je, da ta protokol temelji na obstoječem protokolu OIDC, ki je industrijski standard za sisteme za upravljanje digitalne identitete. Oba protokola sicer vsebujeta podobne določene korake, na primer dokaz o lastništvu privatnih ključev, ki kontrolirajo DID, in zahteve po pridobitvi določenih tipov oz. shem VC-jev in VP-jev.

Skozi leta razvoja sistemov za decentralizirano identiteto je bilo ustvarjenih več odprto-kodnih knjižnic, ki poenostavijo delo in integracijo različnih komponent SSI. Ene izmed najbolj priljubljenih knjižnic so Veramo, Hyperledger Aries, DID JWT VC, itd. Zanimivost in prednost standardov SSI je tudi, da se lahko uporabijo ločeno, kar pomeni, da se v določenih primerih uporabe lahko integrira le DID, v določenih pa le VC.

## 2.3 Digitalne denarnice

Digitalna denarnica predstavlja vstopno točko za imetnike za samo-upravljano identiteto oz. decentralizirani tehnologijami nasploh [21]. Denarnice so programska ali strojno oprema, s katero se uporabniki identificirajo in njihova jedrna naloga je upravljane s kriptografskimi ključi. Delitev navadno lahko poteka na dva načina in sicer glede na namembnost ali glede na tehnične specifikacije. Prvi prvem načinu prevladujejo kripto denarnice ali Web3 denarnice, ki se uporabljajo za podpisovanje transakcij in uporaba različnih blockchain omrežij (na primer MetaMask in Trust Wallet), in na denarnice za decentralizirano identiteto, na primer Lissi in Masca. Pri drugem načinu se denarnice grobo ločijo na programske denarnice, tj. vse denarnice omenjene do sedaj, in denarnice v obliki strojne opreme, na primer Ledger in Trezor. Kot je bilo že omenjeno, ne glede na namembnost in tehnično specifikacijo, je glavna naloga varno upravljano s kriptografskimi ključi, ki se lahko uporabijo za različne namene. Ključi se lahko iz denarnic varnostno kopirajo na različne načine, najpogosteje s semensko frazo (angl. seed phrase) ali s pomočjo novejših pristopov kot so Passkeys. Poleg upravljanja s kriptografskimi ključi še denarnice podpirajo več funkcionalnosti kot so generiranje DID in blockchain naslovov, generiranje in podpisovanje VP, podpisovanje in pošiljanje blockchain transakcij, komunikacija z blockchain omrežji in podpora različnim protokolom kot sta DIDComm in OID4VC [9].

# 3 OID4VC

OID4VC je nabor treh specifikacij ki definirajo standarden način za izdajanje, predstavljanje in deljenje VC in VP. Te tri specifikacije so:

- OpenID za izdajanje VC (angl. OpenID for Verifiable Credential Issuance, OID4VCI) [19], ki določa API in ustrezne avtorizacijske mehanizme, ki temeljijo na OAuth, za izdajanje VC. Podpira tako W3C VC podatkovni model, kot tudi mobilna vozniška dovoljenja (angl. mobile driving license, mDL) predstavljena v standardu ISO/IEC 18013-5:2021 [12].
- OpenID za predstavitev VP (angl. OpenID for Verifiable Presentations, OID4VP) [20] določa mehanizem zgrajen na OAuth 2.0, ki omogoča predstavitev trditev v obliki VP kot del protokola.
- Samoizdani OpenID ponudnik v2 (angl. Self-Issued OpenID Provider v2, SIOP) [38] omogoča uporabnikom, da sami delujejo kot svoj OpenID ponudnik in opravijo avtentikacijo s samoizdanimi ID žetoni, ki so podpisani s ključi pod njihovim nadzorom, in s tem neposredno predložijo samopotrjene trditve.

## 3.1 Primeri uporabe

OID4VCI se lahko uporabi za različne namene, na primer izdajo digitalnih potrdil oz. digitalnih dokumentov, ki vsebujejo informacije o identiteti ali kvalifikacijah posameznika. Primeri takšnih potrdil so osebne izkaznice, ki vsebuje informacije o starosti, in diplome oz. drugi certifikati o opravljenih tečajih. Uporabniki lahko nato pridobljene VC-je uporabijo kot dokazilo o opravljenem študij, usposobljenosti ali za drugačne namene.

OID4VP se medtem uporablja za varno predstavitev VC v obliki preverljivih predstavitev (VP) in omejevanje dostopa do raznih virov. Lahko se uporabi za omejitev dostopa do spletne strani, ki se omejene s starostjo ali izobrazbo uporabnika. V teh primerih uporabnik ustvari VP, ki vsebuje enega ali več veljavnih VC ter selektivno dokaže trditve oz. potrebne podatke in si tako pridobi dostop do želenega vira.

## 3.2 Potek

V prispevku se zaradi omejenega časa osredotočimo na sledeči dve specifikaciji: OID4VCI in OID4VP. Ti dve specifikaciji predstavljata tudi jedro OID4VC.

### OID4VCI

Vsak izdajatelj VC za avtorizacijo dostopa uporablja OAuth 2.0 avtorizacijski strežnik, pri čemer se lahko isti strežnik uporabi s strani enega ali več izdajateljev. Izdajanje potrdil je moč izvesti na več različnih načinov, pri čemer se načini v grobem delijo na naslednje [19]:

- Avtoriziran kodni tok (angl. authorized code flow) ali pred-avtoriziran kodni tok (angl. pre-authorized code flow). Izdajatelj potrdila pridobi uporabniške podatke, ki jih pretvori v VC z uporabo uporabniške avtentikacije na OAuth 2.0 avtorizacijskem strežniku (avtoriziran kodni tok) ali z uporabo mehanizmov zunaj območja izdaje (pred-avtorizirani kodni tok).
- Sproženo s strani denarnice (angl. Wallet initiated) ali izdajatelja (agnl. Issuer initiated). Zahteva iz denarnice se lahko pošlje brez interakcije oz. akcije izdajatelja (sproženo s strani denarnice) ali kot odgovor po začetni komunikaciji z izdajateljem potrdil (sproženo s strani izdajatelja).
- Ista naprava (angl. same-device) ali več naprav (angl. cross-device). Spletna stran ali aplikacija izdajatelja lahko prebiva na isti napravi kot digitalna denarnica, ki prejema VC (ista naprava), ali na drugi napravi (več naprav).

- Pravočasno (angl. just-in-time) ali odloženo (angl. deferred). Izdajatelj potrdila lahko potrdilo izda neposredno v odgovor na zahtevo (pravočasno) ali pa z zakasnitvijo, pri čemer uporabnik s svojo digitalno denarnico prevzame kasneje (odloženo).

### OID4VP

Načini pri postopku OID4VP imajo podobne lastnosti in se v grobem delijo v iste kategorije kot pri OID4VCI. Glavna razlika je, da v tem postopku imetnik pošilja podatke, medtem ko jih pri prejšnjem prejema. Glavna razširitev specifikacije OID4VP je vpeljava VP žetona [20]. Slednji vsebuje enega ali več VP in je v vlogi kontejnerja, ki omogoča končnim uporabnikom deljenje VP. Deljenje lahko podobno kot pri OID4VCI poteka na isti napravi, kot se nahaja denarnica, ki hrani VC, ali pa se nahaja na drugi napravi.

## 3.3 Arhitektura

Tako pri OID4VCI, kot tudi pri OID4VP poteka komunikacija s strežnikom preko https povezave. Obe specifikaciji uporabljata lastne formate URI naslovov za začetek tokov izmenjave poverilnic ali predstavitev. URI za ponudbo izdaje nove poverilnice se začne z predpono $\text{openid} - \text{credential} - \text{offer} : //$, URI za zahtevo pa avtorizaciji pa z predpono $\text{openid} : //$.

### OID4VCI

V sklopu avtorizacijskega strežnika je potrebno implementirati sledeče vmesnike:

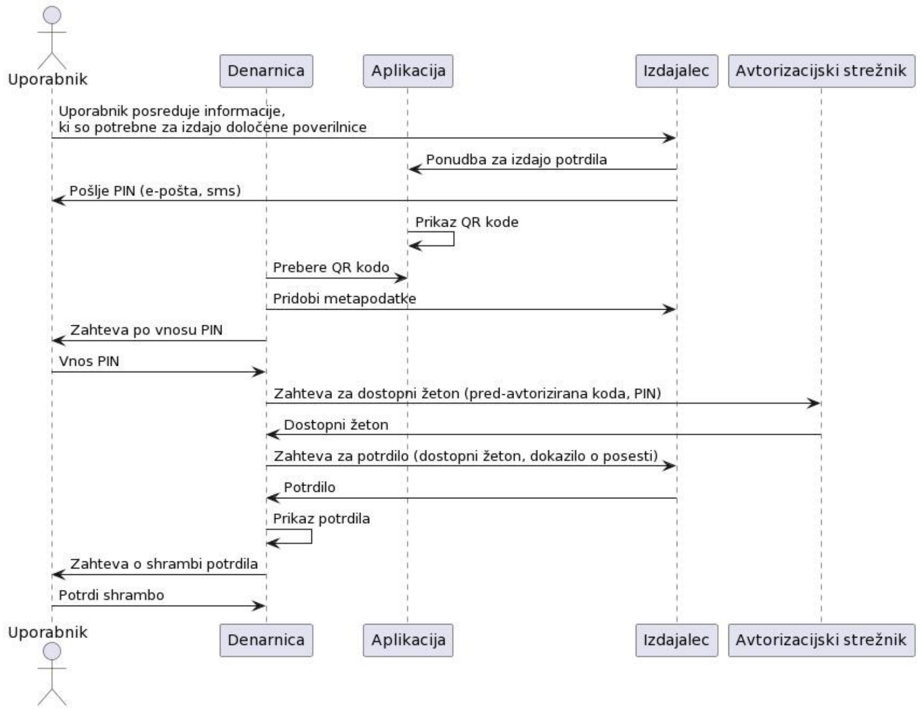
- Vmesnik za avtorizacijo (angl. authorization endpoint), ki se uporablja za avtorizacijo uporabnika pred izdajanjem VC.
- Vmesnik za pridobitev žetona (angl. token endpoint), ki se uporablja za pridobitev žetonov za dostop (angl. access token) in opcijsko tudi osvežitvenih žetonov (angl. refresh token).

V sklopu izdajatelja VC so implementirani sledeči vmesniki:

- Vmesnik za poverilnice (angl. credential endpoint), ki se uporablja za izdajanje posameznih VC v zameno za veljaven žeton. Opcijsko lahko ta vmesnik tudi veže izdan VC na kriptografski ključ uporabnikove identitete.

Opcijsko se lahko na strani izdajatelja VC implementirajo tudi sledeči vmesniki:

- Vmesnik za ponudbo potrdila (angl. credential offer endpoint), ki se uporablja za kreiranje ponudb uporabniku za izdajanje potrdila. Ponudba lahko vključuje informacije o VC, način zahtevane avtorizacije in URL izdajatelja, kjer denarnica lahko pridobi VC.
- Vmesnik za metapodatke (angl. metadata endpoint), ki se uporablja za objavljanje metapodatkov o sposobnostih izdajatelja potrdil. Ti lahko vključujejo informacije o vrstah VC, podprtih formatih VC, podprtih kriptografskih algoritmih in seznam vseh dostopnih vmesnikov.
- Vmesnik za množična potrdila (angl. batch credential endpoint), ki se uporablja za izdajanje večih potrdil v enem zahtevku. To je koristno za izdajo potrdil skupini uporabnikov ali za izdajo večih potrdil enemu uporabniku.
- Vmesnik za odloženo potrdilo (angl. deferred credential endpoint), ki se uporablja za izdajanje potrdil, ki niso takoj na voljo uporabnikom.



**Slika 3: Potek izdaje VC (uporaba kombinacije pred-avtoriziranega kodnega toka, sproženega s strani izdajatelja, pri uporabi večih naprav in s pravočasnim izdajanjem).**

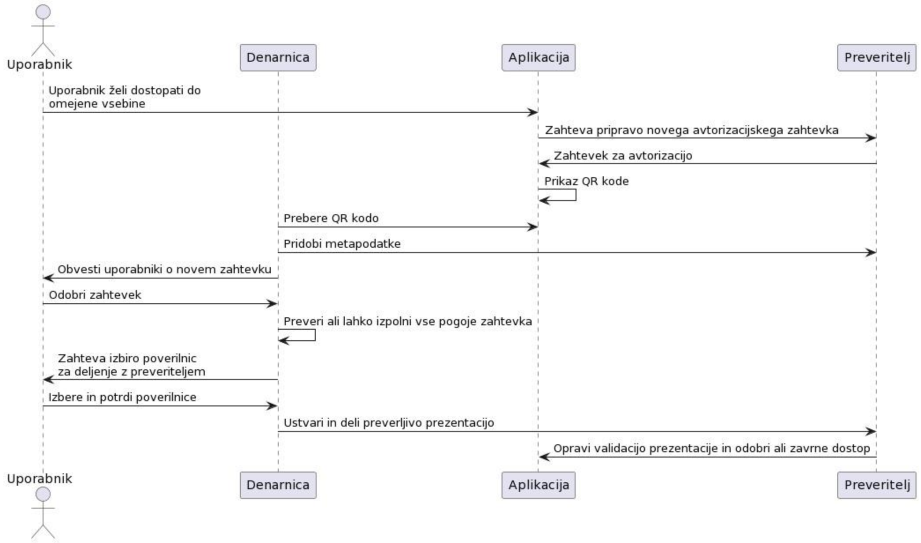
## OID4VP

Na strani preveritelja je potrebno implementirati sledeče vmesnike:

- Vmesnik za avtorizacijski zahtevek (angl. authorization request endpoint), ki se uporablja za kreiranje zahtev za predstavitev VP od uporabnika.
- Vmesnik za avtorizacijski odgovor (angl. authorization response endpoint), ki se uporablja za prejemanje in validacijo uporabnikovih VP s pomočjo preusmeritev.

Opcijsko se lahko na strani preveritelja VP implementirajo tudi sledeči vmesniki:

- Vmesnik za metapodatke (angl. metadata endpoint), ki se uporablja za objavljanje metapodatkov o sposobnostih preveritelja VP. Ti lahko vključujejo informacije o podprtih formatih VP, podprtih kriptografskih algoritmih in seznam vseh dostopnih vmesnikov.
- Vmesnik za direkten avotorizacijski odgovor (angl. direct post response endpoint), ki se uporablja za prejemanje in validacijo uporabnikovih VP preko metode HTTP POST.



**Slika 4: Potek deljenja VP pri uporabi večih naprav.**

# 4 Uvajanje protokola OID4VC v digitalno denarnico Masca

## 4.1 Masca

Masca je MetaMask Snap [13] (vtičnik), ki razširi delovanje kripto denarnice MetaMask s funkcionalnostmi SSI oz. decentralizirane identitete [10]. Glavne funkcionalnosti zavzemajo upravljanje z decentraliziranimi identifikatorji, hrambo in upravljanje z VC, generiranje VP, deljenje VP preko protokola OID4VP, pridobivanje VC preko protokolov OID4VCI in Iden3comm [8] in avtentikacijo z ZKP preko protokola Iden3comm. Dodatna prednost kripto denarnice Masca je tudi preprosta integracija v (decentralizirane) aplikacije, saj je z uporabo knjižnice imenovane masca-connector možno integracijo izvesti le z nekaj vrsticami kode [11].

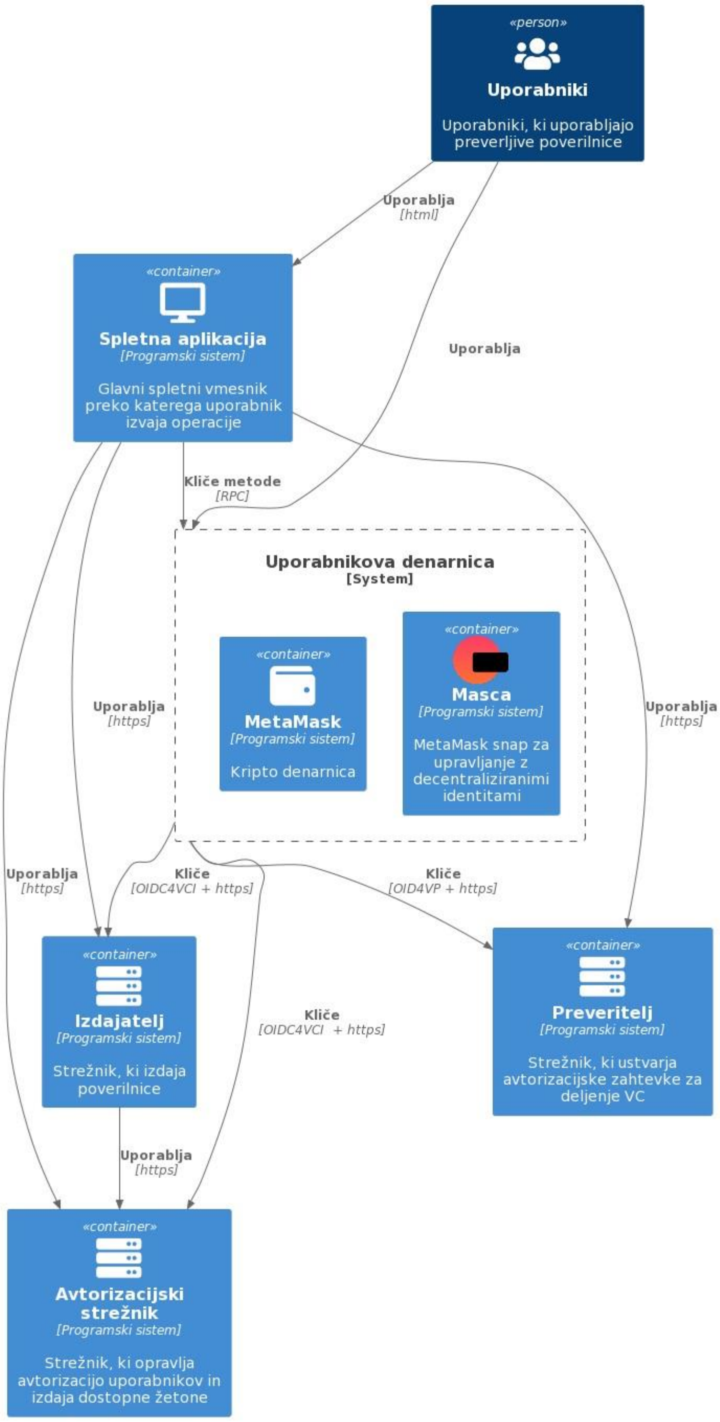
## 4.2 Veramo

Veramo je JavaScript knjižnica za SSI in na splošno preverljive podatke [39]. Njena zasnova temelji na fleksibilnost in modularnosti, kar omogoča enostavno integracijo z različnimi sistemi in v različnih okoljih. Knjižnica lahko tako teče na strežniku, v brskalniku ali na mobilnih napravah z uporabo okolja React Native. Uporaba knjižnice poteka tako, da razvijalec na začetku ustvari tako imenovanega agenta, kateremu se nato dodajo vtičniki z želenimi funkcionalnostmi, ki so nato dosegljive preko agentovega API. Nekatere izmed teh funkcionalnosti so:

- kreiranje in upravljanje s kriptografskimi ključi za namen digitalnega podpisovanja in šifriranja,
- kreiranje in upravljanje z DID,
- generiranje VC in VP,
- verificiranje VC in VP ter
- predstavljanje VP s selektivnim razkrivanjem.

## 4.3 Implementacija

Arhitektura implementacije je prikazana na sliki 5. Spletna aplikacija tukaj služi kot točka za pridobivanje ponudb za izdajo novih poverilnic, za pridobivanje zahtevkov za avtorizacijo in za avtentikacijo uporabnikov. Prav tako mora ta implementirati funkcionalnosti za klicanje RPC metod Masca vtičnika. Vtičnik od spletne aplikacije prejme pridobljene ponudbe za izdajo poverilnic in zahtevke za avtorizacijo, ter opravi vse nadaljnje korake, da se le te ti uspešno opravijo.

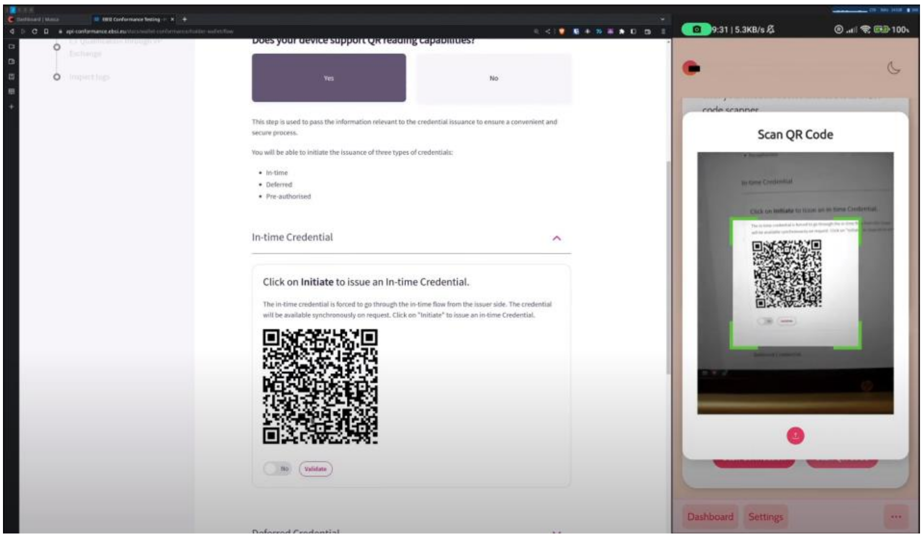


**Slika 5: Arhitektura celotnega implementiranega sistema.**

# 5 Validacija

## 5.1 Izmenjava poverilnic

Postopek izdaje nove poverilnice z uporabo pred-avtoriziranega kodnega toka se vedno prične pri izdajatelju. Slednji si je predhodno že pridobil potrebne informacije za izdajo potrdila uporabniku in v prvem koraku pripravi ponudbo za izdajo potrdila. Ta ponudba je v obliki URI naslova in se lahko prikaže uporabniku v obliki QR kode, katero mora skenirati na Masca aplikaciji, primer viden na sliki 6, ali pa se neposredno kliče RPC metoda Masca vtičnika. Hkrati izdajatelj pošlje uporabniku PIN kodo in sicer po drugem kanalu kot je poslal ponudbo za izdajo potrdila. PIN koda je potrebna za pridobitev dostopnega žetona. V naslednjem koraku denarnica pridobi metapodatke izdajatelja poverlinic, kjer je tudi zabeležen naslov avtorizacijskega vmesnika. Denarnica nato pripravi zahtevek za pridobitev dostopnega žetona, ki nato od uporabnika zahteva vnos PIN kode in vse skupaj se pošlje na avtorizacijski strežnik. Kot odgovor uporabnik pridobi dostopni žeton, kateri se v naslednjem koraku uporabi za pridobitev poverilnice s strani izdajatelja. Poverilnica se prikaže uporabniku, kateri jo nato potrdi in jo shrani v digitalno denarnico ali jo zavrne.



This step is used to pass the information relevant to the credential issuance to ensure a convenient and secure process.

You will be able to initiate the issuance of three types of credentials:

- In-time
- Deferred
- Pre-authorised

In-time Credential

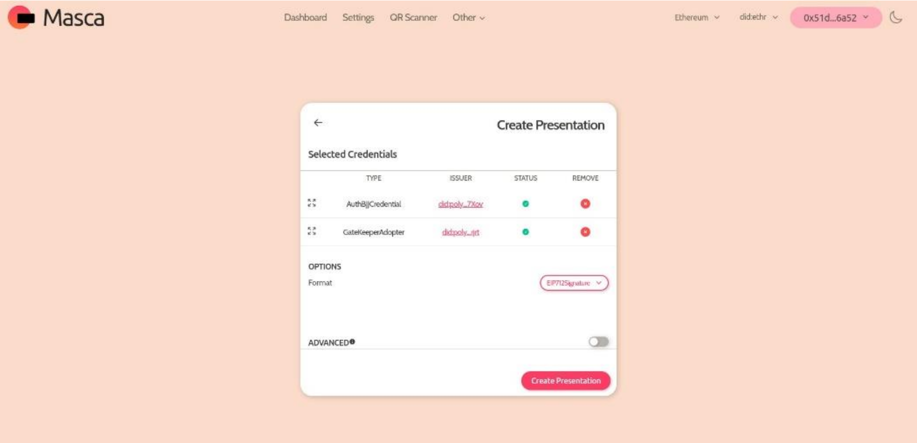
Click on **Initiate** to issue an In-time Credential.

The in-time credential is forced to go through the in-time flow from the issuer side. The credential will be available synchronously on request. Click on "Initiate" to issue an in-time Credential.

**Slika 6: Skeniranje QR kode na strani EBSI WCT, ki vsebuje ponudbo za izdajo preverljive poverilnice.**

## 5.2 Deljenje poverilnic

Postopek izmenjave preverljivih prezentacij (VP) se začne na strani preveritelja, ki pripravi zahtevek za avtorizacijo. Ta je prav tako kot v primeru izdajanja nove poverilnice v obliki URI naslova. Zahtev vsebuje vse potrebne informacije o poverilnicah, ki jih želi prejeti od uporabnika. Ponovno se lahko URI prikaže uporabniku v obliki QR kode, katero mora skenirati z Masca aplikacijo, ali pa se neposredno kliče RPC metoda Masca vtičnika. Vtičnik obvesti uporabnika o novem zahtevku za avtorizacijo in ko ta zahtevek odobri, preveri ali lahko zadosti vsem pogojem v njem (npr. deljenje specifičnega tipa poverilnice). Od uporabnika zahteva tudi, da izbere poverilnice, katere želi deliti s preveriteljem in nato ustvari preverljivo prezentacijo, ki se pošlje preveritelju. Na sliki 7 je prikazano deljenje poverilnic preko spletne aplikacije Masca.



Create Presentation

Selected Credentials

| TYPE | ISSUER | STATUS | REMOVE |
|---|---|---|---|
| AuthBJJCredential | did:poly...7Xov | ● | ✕ |
| GateKeeperAdopter | did:poly...rjrt | ● | ✕ |

OPTIONS

Format: EIP712Signature

ADVANCED

**Slika 7: Deljenje poverilnice na spletni aplikaciji Masca.**

## 5.3 Varnostni pogledi

### OID4VCI

Digitalne denarnice ne smejo pri uporabi vmesnika za ponudbo potrdil tem slepo zaupati, saj izvor ponudb ni avtoriziran in integriteta sporočil ni zavarovana. Zlonamerni akterji bi potencialno lahko uporabili te ponudbe za napad z lažnim predstavljanjem (angl. phishing attack) ali napad z vrivanjem (angl. injection attack) [19].

Prav tako se pojavi tudi vprašanje o zaupanju med denarnicami in izdajatelji VC. Izdajatelji pogosto želijo vedeti, kateri denarnici izdajajo VC in kako le ta upravlja s svojimi zasebnimi kriptografskimi ključi. Razlogi za to so sledeči:

- Izdajatelj želi zagotoviti, da so kriptografski ključi uporabnika pravilno zaščiteni pred uhajanjem (angl. exfiltration) in ponovnim predvajanjem (angl. replay). Tako je zlonamerni osebi preprečeno, da se pretvarja kot legitimen imetnik potrdila in predloži uporabnikovo potrdilo.
- Izdajatelj želi zagotoviti, da digitalna denarnica, ki upravlja potrdila, upošteva določene pravilnike in je bila po potrebi revidirana ter odobrena v skladu z določeno regulativno in/ali komercialno shemo.

Pred-avtoriziran kodni tok je ranljiv na ponovno predvajanje, saj ni vezan na specifično napravo, kot je to značilno pri avtoriziranem kodnem toku v kombinaciji s PKCE [30]. V ta namen je zato priporočljivo dodatno uporabiti naključno generirano uporabniško kodo, ki se uporabniku posreduje po drugem kanalu (npr. elektronski pošti ali SMS-u) kot ponudba za prevzem VC. Ta koda se nato zahteva v naslednjem koraku, ko denarnica želi pridobiti žeton za prevzem VC [19].

### OID4VP

Pri deljenju VP je pomembno, da preveritelji implementirajo funkcionalnosti, ki preprečujejo napadalcem uporabo prestreženih VP žetonov, za namene impersonacije uporabnika. To se doseže z uporabo dveh parametrov. Vsak avtorizacijski zahtevek mora vsebovati naključno kriptografsko število, imenovano nonce, ki ima dovolj veliko entropijo in uporabnik jo mora vključiti v VP žeton. Prav tako mora VP žeton vsebovati parameter imenovan client id, ki predstavlja identifikator preveritelja, kateremu je žeton namenjen [20].

Pri uporabi deljenja VP, kjer se namesto preusmeritev uporablja metoda HTTP POST, lahko pride do napada fiksacije seje (angl. session fixation). V tem primeru napadalec začne postopek deljenja VP na svoji napravi in posreduje prejet avtorizacijski zahtevek pravemu uporabniku. Napadalec nato periodično poskuša zaključiti tok na svoji napravi in ko uporabnik zaključi le tega, dobi napadalec dostop do uporabnikove seje. Ta napad se lahko prepreči z uporabo preusmeritve, ki vključuje naključno kodo vezano na sejo, uporabnikove denarnice na čelni (angl. frontend) del aplikacije preveriteljevega strežnika [20].

## 5.4 Prednosti in slabosti

### Prednosti

Ena večjih prednosti specifikacije OID4VCI je uporaba protokola OAuth 2.0, saj je ta široko razširjen, varen, preprost in fleskibilen. Prav tako omogoča, da se lahko obstoječe namestitve OAuth 2.0 in OIDC strežnikov razširijo s funkcionalnostmi izdajanja VC. Specifikacija s poenotenim in standardiziranim načinom izdajanja VC pripomore tudi k interoperabilnosti rešitev v tem ekosistemu.

Dobra lastnost specifikacij je tudi ta, da omogočajo razširitev implementacij z dodatnimi standardi. Na primer pri uporabi avtoriziranega kodnega toka se priporoča razširitev s standardom RFC7636 [30], ki omogoča preprečitev napada na prestrezanje avtorizacijskih kod, in standardom RFC9126 [35], ki zagotavlja celovitost in pristnost zahtev za avtorizacijo. Ostali standardi na katerih gradijo specifikacijo so sledeči: RFC6750 [23], RFC7515 [25], RFC7517 [27], RFC7519 [28], RFC7591 [29], RFC8152 [31] in RFC8725 [33].

Poglavitni doprinos specifikacij OID4VP je, da omogoča višjo zasebnost in varnost uporabnikov, uvede poenoten in standardiziran način deljenja in verifikacije VP, ter omogoča nadgradnjo že obstoječih rešitev. Standardi, na katerih specifikacija gradi oz. se lahko dodatno razširi so sledeči: RFC6819 [24], RFC7516 [26], RFC7591 [29], RFC8252 [17], RFC8414 [32], RFC9101 [34], RFC9126 [35] in RFC9207 [36].

**Slabosti**

Skupna slabost OID4VCI in OID4VP specifikacij je to, da sta še vedno v razvoju in se pogosto spreminjata. To povzroča nekompatibilnosti med različnimi implementacijami in nameščenimi rešitvami. Nobena od specifikacij prav tako ne omenja, kako se lahko v postopek izdajanja VC in preverjanja VP vključi uporaba ZKP.

## 6 Zaključek

V prispevku smo predstavili komponento novega in inovativnega sistema za upravljanje digitalne identitete, ki temelji na konceptih in principih SSI. Na začetku smo izpostavili potrebo bo varnih protokolih za izdajanje in deljenje preverljivih poverilnic (VC) ter predstavili prednosti uporabe protokolov OID4VC. V nadaljevanju smo predstavili specifikacijo in standarde OID4VC v podrobnosti, pri čemer smo se osredotočili na protokola OID4VCI in OID4VP.

Predstavljene koncepte smo validirali na lastni implementaciji izdajatelja in preveritelja v kombinaciji s kripto denarnico Masca, v katero smo implementirali podporo za protokole OID4VC. Med validacijo smo prišli do več zaključkov in povzeli prednosti ter slabosti omenjenega pristopa, pri čemer je ena izmed glavnih prednosti uporaba preverjenih standardov kot je OAuth 2.0 in velika zmožnost razširitve jedrnega postopka, medtem ko je ena izmed večjih slabosti hitro spreminjajoče specifikacije, saj še le te niso v dalj časa stabilni različici.

## Literatura

[1] CLERCQ Jan De “Single sign-on architectures”, International Conference on Infrastructure Security. Springer. 2002, str. 40–58.

[2] https://www.w3.org/TR/did-core/, Decentralized Identifiers (DIDs) v1.0, obiskano 7. 6. 2023.

[3] https://identity.foundation/didcomm-messaging/spec/, DIDComm Messaging Specification, obiskano 7. 6. 2023.

[4] https://identity.foundation/, DIF - Decentralized Identity Foundation, obiskano 7. 6. 2023.

[5] https://digital-strategy.ec.europa.eu/en/library/ european-digital-identity-architecture-and-reference-frameworkoutline, European Digital Identity Architecture and Reference Framework – Outline, obiskano 7. 6. 2023.

[6] https://ec.europa.eu/digital-buildingblocks/wikis/display/ebsi, Home - EBSI -, obiskano 7. 6. 2023.

[7] https://openid.net/developers/ how-connect-works/, How OpenID Connect works, obiskano 7. 6. 2023.

[8] https://0xpolygonid.github.io/tutorials/ wallet/wallet-sdk/polygonid-sdk/iden3comm/overview/, Iden3comm Overview, obiskano 7. 6. 2023.

[9] KERSIC Vid, et al. “Orchestrating Digital Wallets for On-and Off-chain Decentralized Identity Management”, IEEE Access 2023.

[10] KERSIC VID et al. “Dodajanje poljubnih funkcionalnosti digitalni kripto denarnici MetaMask”. In: Nasl. z nasl. strani. Univerza v Mariboru, Univerzitetna založba; Fakulteta za elektrotehniko, računalništvo in informatiko, 2022, pp. 141–154. url: https://dk.um.si/IzpisGradiva. php?id=82880.

[11] https://www.npmjs.com/package/@blockchain-lab-um/mascaconnector, Knjižnica za integracijo Masca (@blockchain-lab-um/masca-connector ), obiskano 7. 6. 2023.

[12] https://www.iso.org/standard/ 69084.html, mDL - ISO/IEC 18013-5:2021, obiskano 7. 6. 2023.

[13] https://metamask.io/snaps/, MetaMask Snaps, obiskano 7. 6. 2023.

[14] MURRAY Alex et al. “The promise of a decentralized internet: What is Web3 and how can firms prepare?”, Business Horizons 66.2 (2023), str. 191–202.

[15] https://bitcoin.org/bitcoin, Satoshi Nakamoto. "Bitcoin whitepaper", obiskano 7. 6. 2023.

[16] https://oauth.net/2/, OAuth 2.0 - OAuth, obiskano 7. 6. 2023.

[17] https://www.rfc-editor.org/info/ rfc8252, OAuth 2.0 for Native Apps, obiskano 7. 6. 2023.

[18] https://openid.net/openid4vc/, OpenID for Verifiable Credentials - OpenID Foundation, obiskano 7. 6. 2023.

[19] https://openid.net/specs/openid-4-verifiable-credential-issuance-1_0.html, OpenID for Verifiable Credentials Issuance, obiskano 7. 6. 2023.

[20] https://openid.net/specs/openid-4-verifiable-presentations-1_0.html, OpenID for Verifiable Presentations, obiskano 7. 6. 2023.

[21] PODGORELEC Blaz, et al. "What is a (digital) identity wallet? a systematic literature review", 2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC). IEEE. 2022, str. 809–818.

[22] PREUKSCHAT Alex, REED, Drummond. "Self-sovereign identity", Manning Publications, 2021.

[23] https://www.rfc-editor.org/info/rfc6750, RFC 6750: The OAuth 2.0 Authorization Framework: Bearer Token Usage, obiskano 7. 6. 2023.

[24] https://www.rfc-editor.org/info/rfc6819, RFC 6819: OAuth 2.0 Threat Model and Security Considerations, obiskano 7. 6. 2023.

[25] https://www.rfceditor.org/info/rfc7515, RFC 7515: JSON Web Signature (JWS), obiskano 7. 6. 2023.

[26] https://www.rfceditor.org/info/rfc7516, FC 7516: JSON Web Encryption (JWE, obiskano 7. 6. 2023.

[27] https://www.rfc-editor. org/info/rfc7517, RFC 7517: JSON Web Key (JWK), obiskano 7. 6. 2023.

[28] https://www.rfc-editor. org/info/rfc7519, RFC 7519: JSON Web Token (JWT), obiskano 7. 6. 2023.

[29] https://www.rfc-editor.org/info/rfc7591, RFC 7591: OAuth 2.0 Dynamic Client Registration Protocol, obiskano 7. 6. 2023.

[30] https://www.rfc-editor.org/info/rfc7636, RFC 7636: Proof Key for Code Exchange by OAuth Public Clients, obiskano 7. 6. 2023.

[31] https://www.rfc-editor.org/info/rfc8152, RFC 8152: CBOR Object Signing and Encryption (COSE), obiskano 7. 6. 2023.

[32] https:// www.rfc-editor.org/info/rfc8414, RFC 8414: OAuth 2.0 Authorization Server Metadata, obiskano 7. 6. 2023.

[33] https://www.rfc-editor.org/info/rfc8725, RFC 8725: JSON Web Token Best Current Practices, obiskano 7. 6. 2023.

[34] https://www.rfc-editor.org/info/rfc9101, RFC 9101: The OAuth 2.0 Authorization Framework: JWT-Secured Authorization Request (JAR), obiskano 7. 6. 2023.

[35] https://www.rfc-editor.org/info/rfc9126, RFC 9126: OAuth 2.0 Pushed Authorization Requests, obiskano 7. 6. 2023.

[36] https://www.rfc-editor.org/info/rfc9207, RFC 9207: OAuth 2.0 Authorization Server Issuer Identification, obiskano 7. 6. 2023.

[37] SALOMAA Arto, "Public-key cryptography". 1996.

[38] https://openid.net/specs/openid-connect-self-issued-v2-1_0.html, Self-Issued OpenID Provider v2, obiskano 7. 6. 2023.

[39] https://veramo.io, Veramo - A JavaScript Framework, obiskano 7. 6. 2023.

[40] https://www.w3.org/TR/vc-data-model/, Verifiable Credentials Data Model v1.1, obiskano 7. 6. 2023.

[41] https://www.w3.org/, World Wide Web Consortium (W3C), obiskano 7. 6. 2023.

# Testno ogrodje za razvijalce - ali kako doseči, da razvijalci celovito testirajo

Mitja Krajnc, Tadej Ciglič, Boris Ovčjak

Databox, Ptuj, Slovenija
mitja.krajnc@databox.com, tadej.ciglic@databox.com, boris.ovcjak@databox.com

Pred izdajo rešitve v produkcijsko okolje je zagotavljanje kakovosti in testiranje pomemben korak, vendar je hkrati tudi postopek, ki se ga razvijalci z vso silo otepamo. Če je testiranje enot že ustaljen del razvojnega procesa pa se drugih vrst testiranj, kot sta npr. dimno testiranje in integracijski testi, pogosto že na daleč izognemo in to predamo testerjem. Da bi celoten proces testiranja izboljšali, smo se odločili pripraviti ogrodje, ki bi razvijalcem omogočilo pisanje in izvajanje kompleksnih testov na enostaven način - podobno kot testiranje enot. V članku bomo predstavili ogrodje za testiranje, ki je v prvi vrsti namenjeno prav razvijalcem za enostavno uporabo pri razvoju. Ogrodje temelji na odprtokodni rešitvi Playwright, ki je namenjena avtomatskemu testiranju spletnih rešitev v brskalniku in omogoča izdelavo testov v različnih programskih jezikih. Ker smo želeli zagotoviti čim širšo uporabo ogrodja v podjetju, smo uporabili različico, ki temelji na jeziku TypeScript. Struktura ogrodja je sestavljena iz abstrakcij, ki so že znane iz objektno orientiranega programiranja in so zato razvijalcem blizu, s čimer omogoča, da preko zaporedne uporabe razredov in metod ustvarijo testni scenarij, ki ga lahko na preprost način večkrat zapored hitro in učinkovito uporabijo.

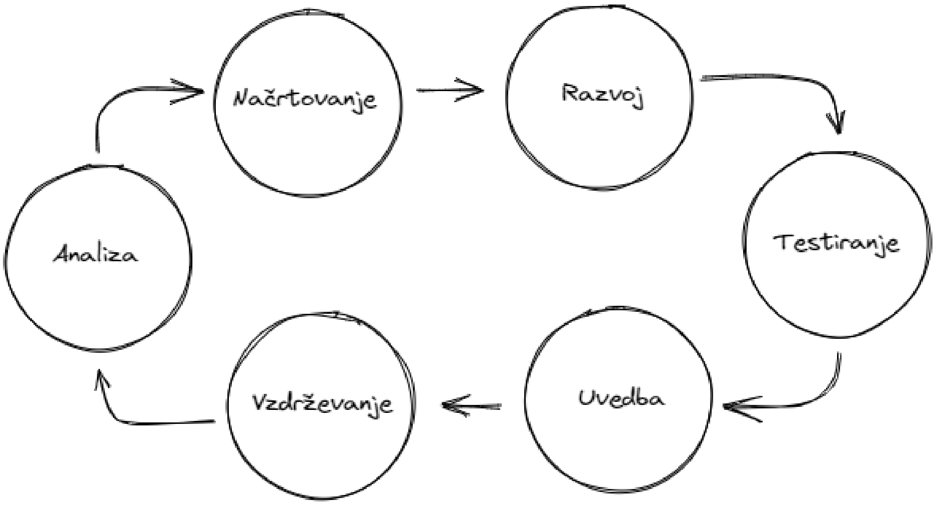
**Ključne besede:**
testiranje
avtomatizacija testiranja
Playwright
ogrodje
razvoj

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.14

# 1 Uvod

Pri razvoju programske opreme se je tekom let v praksi izoblikoval nek tipičen razvojni cikel, ki ga z manjšimi prilagoditvami in spremembami poznamo in uporabljamo vsi. Sestavljen je iz korakov kot so analiza, načrtovanje, razvoj, testiranje, uvedba in vzdrževanje. Na sliki 1 so omenjeni koraki postavljeni v ciklično strukturo, kjer se po vzdrževanju že prične analiza za nadaljnji razvoj.



**Slika 1: Tipičen razvojni cikel.**

V našem članku se bomo podrobneje lotili faze testiranja, ki bi se naj izvajala po razvoju oziroma že v sami fazi razvoja. Razvijalci bi naj tukaj pisali in izvajali teste za preverjanje pravilnega delovanja kode. Testiranje lahko vključuje enote testiranja (unit testing), integracijsko testiranje (integration testing), funkcionalno testiranje (functional testing) in sprejemno testiranje (acceptance testing). V pomoč je zagotovo tudi dimno testiranje (smoke test), pri katerem imajo glavno vlogo testerji in ekipa za zagotavljanje kakovosti. Kljub temu, da se na papirju vse to lepo bere pa se je potrebno zavedati, da pisanje testov s strani razvijalcev ni ravno najbolj priljubljena aktivnost. Čeprav ni res, da vsi razvijalci ne marajo pisanja testov, obstaja več razlogov, zakaj nekateri razvijalci morda ne uživajo v pisanju testov ali jim ne namenjajo prednosti v svojem razvojnem procesu. Tu je nekaj pogostih razlogov:

- **Omejenost časa**: Razvijalci se pogosto soočajo s tesnimi roki in pritiskom, da hitro dostavijo funkcionalnosti ali posodobitve. Pisanje obsežnih testov lahko vzame veliko časa, še posebej pri kompleksnih sistemih. Nekateri razvijalci morda menijo, da lahko prihranijo čas tako, da preskočijo ali zmanjšajo pisanje testov ter se osredotočijo le na pisanje kode.
- **Pomanjkanje takojšnjega zadovoljstva**: Pisanje testov običajno ne zagotavlja takoj vidnih ali oprijemljivih rezultatov. Zahteva trud in pozornost do podrobnosti, ne da bi neposredno prispevala k funkcionalnosti ali vidnim spremembam za končne uporabnike. Posledično se zdi, da pisanje testov deluje manj nagradno v primerjavi s pisanjem kode, ki proizvaja vidne spremembe.
- **Kompleksnost in učna krivulja**: Učinkovito pisanje testov zahteva znanje in razumevanje testnih okolij, orodij ter najboljših praks. Osvojiti zapletenosti testiranja je lahko izziv, še posebej za razvijalce, ki so relativno novi na tem področju ali jim manjka izkušenj pri pisanju zanesljivih testnih paketov.
- **Nadležnost vzdrževanja**: Teste je treba vzdrževati in posodabljati, ko se koda razvija in spreminja skozi čas. Dodajanje novih funkcionalnosti ali preoblikovanje obstoječe kode lahko zahteva ustrezne posodobitve testov. To stalno vzdrževanje lahko razvijalcem predstavlja dodatno breme, še posebej, ko se že ukvarjajo z drugimi prioritetami.

- **Prioritizacija drugih nalog**: V nekaterih primerih se razvijalci morda počutijo pritisnjene v kot, da dajo prednost razvoju funkcionalnosti ali popravkom napak pred pisanjem testov. Vodstvo ali zahteve projekta lahko poudarjajo hitro dostavo novih funkcionalnosti, kar vodi k temu, da razvijalci ne postavljajo pisanja testov na prednostno mesto, da bi izpolnili te zahteve.

V ta namen smo se odločili da s sodelovanjem med razvijalci in testerji razvijemo ogrodje, ki bi naslovilo zgoraj omenjene točke. Se pravi bi omogočilo hitro pripravo kompleksnih testov, s čimer bi že sami razvijalci prevzeli del kontrole pred izdajo. Takšno ogrodje bi moralo biti tudi del neprekinjene integracije, pri čemer se testi izvedejo v skladu z vnaprej definiranimi scenariji, ki jih definirajo testerji. Na podlagi teh zahtev smo izdelali ogrodje za testiranje »Platform Testing Service for Databox« ali PTSD, ki ga vzdržuje ekipa za zagotavljanje kakovosti in je s svojim širokim spektrom razvijalcem prijaznih operacij namenjen prav njim za uporabo pri razvoju.

## 2 Testno ogrodje za razvijalce PTSD

Ena izmed prvih odločitev pri izdelavi ogrodja PTSD je bila, kakšno ogrodje vzeti za osnovo. Znotraj podjetja smo že uporabljali ogrodje Katalon [2], ki omogoča avtomatizacijo testiranja spletnih aplikacij, API-jev, mobilnih aplikacij in namiznih aplikacij ter vključuje funkcionalnosti, kot so snemanje in predvajanje testov, upravljanje podatkov, generiranje poročil o testiranju. Orodje ponuja uporabniku prijazen vmesnik, ki olajša ustvarjanje, izvajanje in upravljanje testnih primerov in podobno. Kaj kmalu smo ugotovili, da sam Katalon Studio sicer ima uporabniku prijazen vmesnik, vendar pa za naše razvijalce ni bil ugodna rešitev. Zato smo iskali rešitev, ki bila bolj pisana na roko razvijalcem, bila lahkotnejša in po načinu uporabe bolj podobna razvijalčevemu vsakdanu. V ta namen smo ocenili priljubljena orodja kot so Selenium, Cyprsess, Puppeteer in Playwright [3,4]. Za naš primer je bila še posebej pomembna podpora različnim brskalnikom, več zavihkom in oknom, hitrost, stabilnost, uporaba lokatorjev in kompatibilnost z obstoječim programskimi jeziki. Po pregledu ogrodij in preizkusu smo se za to odločili za ogrodje Playwright.

### 2.1 Playwright

Playwright je razmeroma novo odprtokodno orodje za avtomatizacijo testiranja v brskalniku, njegovo prvo različico je Microsoft izdal leta 2020. Izdelala ga je ekipa, ki stoji za ogrodjem Puppeteer. To je ogrodje brezglavega testiranja za Chrome/Chromium. Playwright. Playwright sam po sebi presega Puppeteer in med drugimi spremembami zagotavlja podporo za več brskalnikov. Zasnovan je za samodejno testiranje spletnih aplikacij od konca do konca (end-to-end). Posebej je prilagojen za sodoben splet in na splošno deluje zelo hitro, tudi pri kompleksnih projektih testiranja. Čeprav je precej novejši od Seleniuma, Playwright hitro pridobiva na moči in ima vse več privržencev. Zaradi svoje mladosti sicer podpira manj brskalnikov/jezikov kot Selenium, vendar po istem principu vključuje tudi novejše funkcije in zmogljivosti, ki so bolj usklajene s sodobnim spletom. Aktivno ga razvija Microsoft. Med značilnosti ogrodja Playwright, ki so ključne za naše potrebe spadajo:

- **Podpora za več brskalnikov**: Kot že omenjeno Playwright omogoča avtomatizacijo brskanja in testiranje spletnih aplikacij v več brskalniki. To so Chrome, Firefox, Safari in Edge. S čimer nam omogoča, da preverimo, ali naša aplikacija deluje pravilno v vseh brskalnikih, ki jih tudi uradno podpiramo.
- **Podpora več platformam:** Poleg podpore za različne brskalnike, Playwright omogoča tudi avtomatizacijo na različnih platformah, vključno z operacijskimi sistemi Windows, macOS in Linux.
- **Hitrost in zanesljivost**: Playwright je zasnovan tako, da omogoča hitro in zanesljivo izvajanje testov. Avtomatizacija brskalnikov je optimizirana, kar pripomore k učinkovitemu izvajanju testov in zmanjšanju časa potrebnega za zaključek testnih sklopov. Pri evaluaciji ogrodij smo pri prepisu nekaterih testov iz Katalona zaznali občutno povečano hitrost, in sicer tudi do 300 odstotkov.

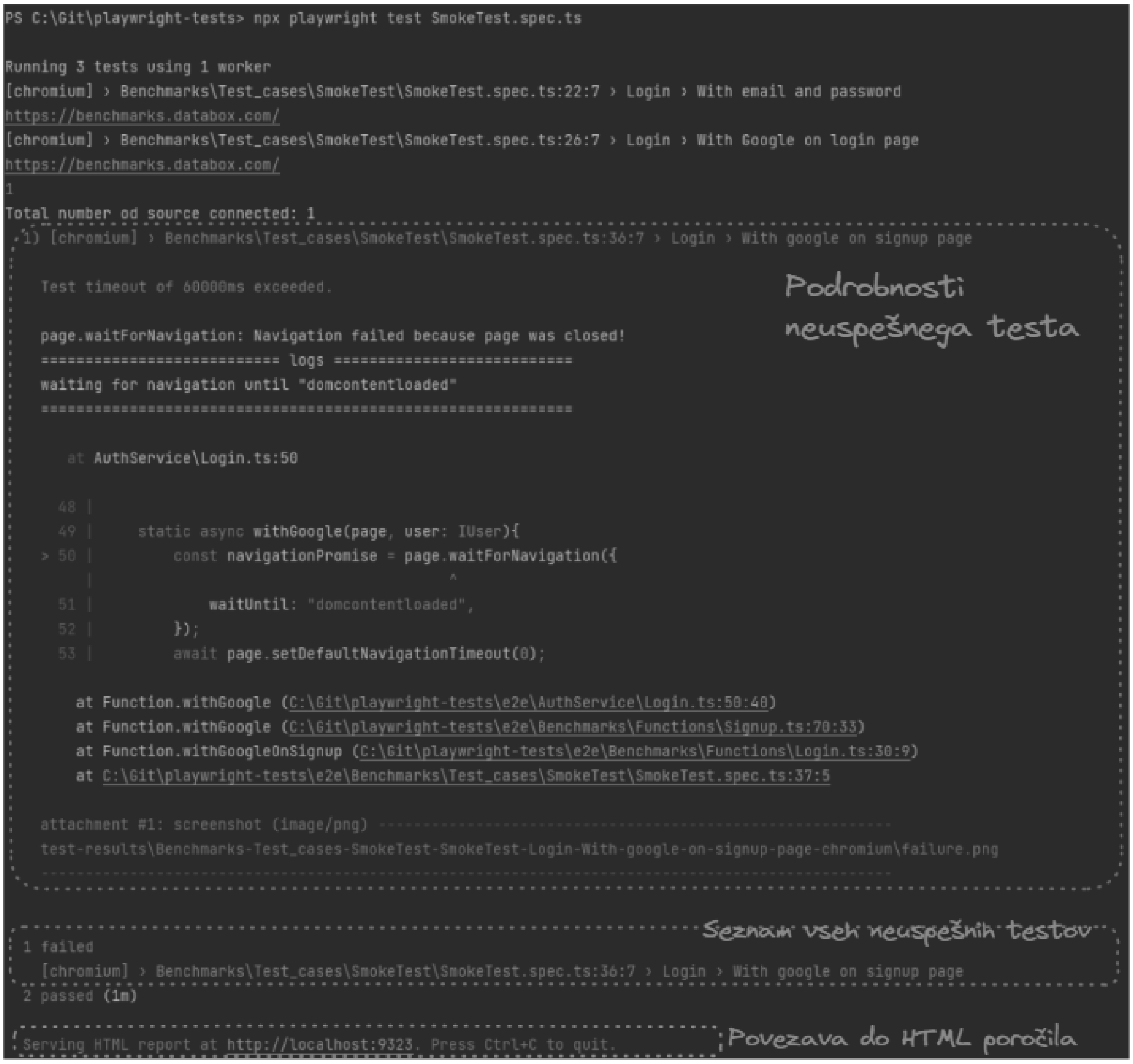
- **Podpora za več jezikov:** Playwright podpira več programskih jezikov, kot so JavaScript, TypeScript, Python in C#. Zaradi te podpore, smo lahko omogočili, da smo izbrali jezik, ki je najbolj ustrezal našim razvijalcem.

Glede na uporabo programskih jezikov v različnih ekipah v podjetju smo se odločili za izdelavo ogrodja v jeziku TypeScript. S tem smo našli najbližji skupni imenovalec tako ekipam, ki delajo na čelnem delu sistema (frontend) kot ekipam, ki delajo na zaledju sistema (backend). Za poganjanje Playwright potrebuje samo okolje Node.js in namestitev paketa preko enega izmed paketnih upraviteljev (npm, yarn, pnpm). Poganjanje poteka preko konzole z ukazom, kjer določimo kateri test v kateri zbirki poženemo. Tekom poteka testa se v konzolo lahko izpisujejo tudi kakšne dodatne informacije, ki smo si jih pripravili v sklopu priprave testa. Primer poganjanja testov je viden na sliki 2.

```
PS C:\Git\playwright-tests> npx playwright test SmokeTest.spec.ts

Running 3 tests using 1 worker
[chromium] › Benchmarks\Test_cases\SmokeTest\SmokeTest.spec.ts:22:7 › Login › With email and password
https://benchmarks.databox.com/
[chromium] › Benchmarks\Test_cases\SmokeTest\SmokeTest.spec.ts:26:7 › Login › With Google on login page
https://benchmarks.databox.com/

Total number od source connected: 1
[chromium] › Benchmarks\Test_cases\SmokeTest\SmokeTest.spec.ts:36:7 › Login › With google on signup page

Total number od source connected: 1

  3 passed (40s)

To open last HTML report run:

  npx playwright show-report
```

**Slika 2: Poganjanje testov znotraj konzole z ogrodjem Playwright.**

V kolikor testi niso bili uspešni, lahko pridobimo poročilo z ukazov “show-report”, pri čemer dobimo detajlni pogled kaj je šlo narobe. Eno izmed takih poročil je vidno na sliki 3.



```
PS C:\Git\playwright-tests> npx playwright test SmokeTest.spec.ts

Running 3 tests using 1 worker
[chromium] › Benchmarks\Test_cases\SmokeTest\SmokeTest.spec.ts:22:7 › Login › With email and password
https://benchmarks.databox.com/
[chromium] › Benchmarks\Test_cases\SmokeTest\SmokeTest.spec.ts:26:7 › Login › With Google on login page
https://benchmarks.databox.com/
1
Total number od source connected: 1
  1) [chromium] › Benchmarks\Test_cases\SmokeTest\SmokeTest.spec.ts:36:7 › Login › With google on signup page

    Test timeout of 60000ms exceeded.

    page.waitForNavigation: Navigation failed because page was closed!
    =========================== logs ===========================
    waiting for navigation until "domcontentloaded"
    ============================================================

      at AuthService\Login.ts:50

      48 |
      49 |     static async withGoogle(page, user: IUser){
    > 50 |         const navigationPromise = page.waitForNavigation({
         |                                        ^
      51 |             waitUntil: "domcontentloaded",
      52 |         });
      53 |         await page.setDefaultNavigationTimeout(0);

        at Function.withGoogle (C:\Git\playwright-tests\e2e\AuthService\Login.ts:50:40)
        at Function.withGoogle (C:\Git\playwright-tests\e2e\Benchmarks\Functions\Signup.ts:70:33)
        at Function.withGoogleOnSignup (C:\Git\playwright-tests\e2e\Benchmarks\Functions\Login.ts:30:9)
        at C:\Git\playwright-tests\e2e\Benchmarks\Test_cases\SmokeTest\SmokeTest.spec.ts:37:5

    attachment #1: screenshot (image/png) ----------------------------------------------------
    test-results\Benchmarks-Test_cases-SmokeTest-SmokeTest-Login-With-google-on-signup-page-chromium\failure.png
    ------------------------------------------------------------------------------------------------


  1 failed
    [chromium] › Benchmarks\Test_cases\SmokeTest\SmokeTest.spec.ts:36:7 › Login › With google on signup page
  2 passed (1m)

  Serving HTML report at http://localhost:9323. Press Ctrl+C to quit.
```

Slika 3: Prikaz neuspelega testa.

Iz takega poročila lahko razvijalec izlušči kaj je šlo narobe in kaj mora popraviti oziroma če je tekom razvoja prišlo do neželenih sprememb na nepričakovanih delih aplikacije.

## 2.2 Struktura ogrodja PTSD

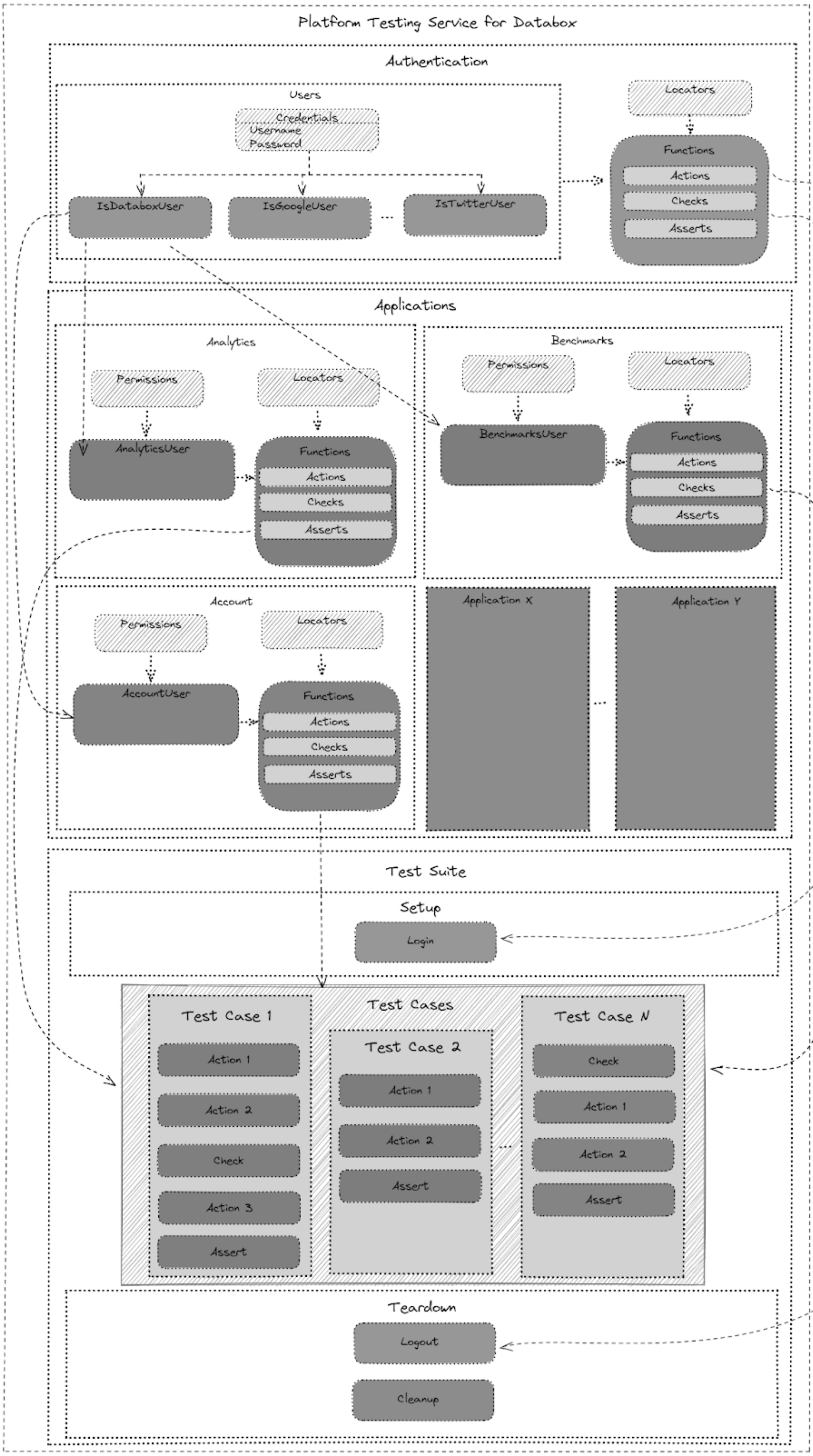
Da bi uporabo ogrodja PTSD čim bolj približali razvijalcem, smo ga zasnovali tako, da je sestavljen iz abstrakcij, ki so že znane iz objektno orientiranega programiranja in so zato blizu razvijalcem in sami po sebi niso revolucionarni. Testi oziroma testni scenariji so sestavljeni s pomočjo razredov, vmesnikov, metod, kot da bi uporabljali kakšno programsko ogrodje, s to izjemo da na koncu, ko kodo poženejo, se izvede celotni test v brskalniku. S pomočjo zastavljene strukture smo poskrbeli da je celotno ogrodje na takšnem nivoju abstrakcije, da se razvijalec ne rabi zavedati specifik samega ogrodja (Playwright). Vse specifike, kot so lokatorji, metode za manipulacije s kontrolami, stranmi, so skrite za primerno poimenovanimi metodami.

Da smo dosegli smiseln nivo abstrakcije, smo najprej ogrodje razdelili na več logičnih enot in sicer na avtentikacijo in na aplikacije. Na tak način smo ločili komponente, ki so skupne našim produktom od specifik, ki jih ima vsaka aplikacija:

- **Komponenta avtentikacije** omogoča uporabo celotnega avtentikacijskega dela naših produktov, od prijave, odjave do funkcionalnosti pozabljenega gesla itd.. V ta namen izpostavlja nabor funkcionalnosti, ki logično uporabniku ogrodja povedo, kaj bodo dosegli s katero akcijo v sistemu. Ta del ogrodja vzpostavi tudi prve vmesnike za nadaljnjo uporabo. Tukaj znotraj so definirani tipi uporabnikov glede na način prijave. Ločimo tako uporabnika, ki se prijavi z uporabniškim imenom in geslom ali preko katerega od ponudnikov identifikacijskih storitev (google, twitter, facebook,...).
- **Komponente aplikacij** predstavljajo posamezne aplikacije znotraj našega ekosistema. Sestavljene so iz funkcionalnosti, ki zagotavljajo nabor akcij v posamezni aplikaciji in manipulacijo z njimi. Vsaka aplikacija ima tako nabor svojih funkcionalnosti in tudi svoj nabor vmesnikov za uporabnike, saj s tem zagotovimo pravilno obnašanje glede na tip uporabnika.

Kot je že lahko zgoraj razvidno so torej končni testni scenariji sestavljeni iz definicije uporabnikov in uporabo funkcionalnosti posameznih aplikacij ter interakciji med uporabniki in funkcionalnostmi, kar je tudi prikazano na sliki 4. Uporabnike in funkcionalnosti znotraj ogrodja definirali sledeče:

- **Uporabniki (angl. Users)** so razredi in vmesniki, ki predstavljajo uporabnike sistema in so na voljo za uporabo v funkcijah. Vsak uporabnik vsebuje nabor dovoljenj in poverilnic, ki vplivajo na tokove in interakcijo s komponentami. Če vedenje komponente temelji na vrsti uporabnika, ki z njo komunicira, je treba uporabnika takega tipa tudi posredovati funkciji. Dovoljenja za uporabnika so vnaprej določena za vsako komponento znotraj aplikacije, zato je za uporabo na voljo več vrst uporabnikov (npr. skrbnik, urednik itd.)
- **Funkcije (angl. Functions)** so metode, ki predstavljajo komponente aplikacije, s katerimi je mogoče komunicirati. Temeljijo na strukturi uporabniškega vmesnika aplikacije. Če vedenje temelji na specifičnem uporabniškem profilu, mora funkcija prejeti ustreznega uporabnika, kar je definirano v sami funkciji. Podobno, če želimo biti v interakciji na določeni komponenti, moramo posredovati identifikator v funkcijo (npr. uporabniško ime, če iščemo uporabnika v iskalnem polju). Glede na način interakcije smo funkcije razdelili na tri skupine:
    - **Akcija (angl. Actions):** To so funkcije, ki predstavljajo dejanja na uporabniškem vmesniku.. Predstavljajo navigacije, klike na gumbe, pisanje besedila v vnosna polja, izbiranje elementov s spustnega seznama itd. Predstavljajo jedro testnega primera.
    - **Preverjanja (angl. Checks):** Funkcije, ki predstavljajo logiko za preverjanje stanja v toku, da se določijo nadaljnji koraki. To je funkcija za nadzor poteka testnega primera. Te funkcije se uporabljajo za ugotavljanje, ali je nekaj v pravilnem stanju za nadaljevanje testa (npr. uporabnik obstaja).
    - **Preizkusi (angl. Asserts):** Funkcija, ki jo je mogoče preizkusiti in zajema logiko enega preizkusa dejanja. Predstavlja pričakovanje, kaj se mora zgoditi. V testnem primeru je treba uporabiti vsaj eno preizkusno metodo. Uspešnost ali neuspeh te metode pa potem določa uspeh testnega primera.



**Slika 4: Struktura ogrodja PTSD.**

## 3 Uporaba ogrodja PTS

Kot že omenjeno so,  zaradi nivoja abstrakcije, uporabnikom ogrodja (razvijalcem), prikrite specifike testov. Primer take abstrakcije lahko vidimo na sliki 5, kjer smo znotraj razreda *LoginPage* za metodo *actionLoginWithCredentials* skrili specifike ogrodja Playwright, na kak način pridemo do primernih tekstovnih polj in kateri gumb se stisne.

```
export class LoginPage {
    2 usages
    static async actionLoginWithCredentials(page: Page, user: IDataboxUser) : Promise<void> {
        await page.locator(authLoginLocators.INPUT_LOGIN_EMAIL).fill(user.getCredentials().email);
        await page.locator(authLoginLocators.INPUT_LOGIN_PASSWORD).fill(user.getCredentials().password);
        await page.locator(authLoginLocators.BUTTON_SIGN_IN).click();
        await page.waitForLoadState();
    }
```

**Slika 5: Prikaz izseka kode iz razreda LoginPage skupaj z metodo actionLoginWithCredentials.**

Predpriprava metod in razredov je v domeni testerjev in ekipe za zagotavljanje kakovosti, ki poskrbijo, da se napovedane spremembe uporabniškega vmesnika zrcalijo tudi v samem ogrodju.  S tem pristopom smo skrili tudi kompleksnost avtomatskega testiranja v brskalniku pred razvijalci in poskrbeli, da se ne rabijo ubadati z morebitnimi spremembami na uporabniškem vmesniku. To je še posebej dobrodošlo za razvijalce zaledja (backend), ki niso tako dovzetni za spremembe uporabniškega vmesnika.

Nek testni scenarij, lahko razvijalec spiše sam ali uporabi že vnaprej sestavljen testni scenarij. Tak je denimo scenarij uporabljen pri dimnem testiranju, je enostavno berljiv in samo razlagalen kaj se kje dogaja in kaj se bo testiralo. Dokler nismo tega skrili za abstrakcijo pa je bilo to za razvijalca neprimerno težje razbrati. Na sliki 6 vidimo izsek testnega scenarija brez uporabe abstrakcij v primerjavi z izsekom enakovrednega scenarija z uporabo abstrakcij, tako kot ga uporablja razvijalec. Že na prvi pogled je spodnji izsek kode, ki predstavlja naše ogrodje, krajši in bolj berljiv. Kljub temu, da samo ogrodje ne predstavlja nič drugega kot uporabo znanih konceptov objektnega programiranja pa smo dosegli izboljšavo procesa, predvsem za to, ker so testerji že prej vključeni v proces načrtovanja in ogrodje te posledično testi pripravljeno dosti prej kot je to bilo pred vpeljavo tega ogrodja.

```
export const EditorPermission =async (page) => {
    //Login
    await page.goto(Url.BENCHMARKS_PROD);
    await page.locator(BenchmarksLoginLocators.REDIRECT_LOGIN_BUTTON).click();
    await page.locator(BenchmarksLoginLocators.LOGIN_EMAIL_INPUT).fill(BenchmarksLoginCredentials.USER_EDITOR);
    await page.locator(BenchmarksLoginLocators.LOGIN_PASSWORD_INPUT).fill(BenchmarksLoginCredentials.USER_PASSWORD);
    await page.locator(BenchmarksLoginLocators.LOGIN_SIGN_IN_BUTTON).click();
    await expect(page.locator(BenchmarksHomeLocators.NAVIGATION_BAR_HOME)).toBeVisible();

    //Check permissions on account
    await page.locator(BenchmarksHomeLocators.BUTTON_ACCOUNT_OPTIONS).click();
    await page.waitForTimeout(500);
    await expect(await page.locator(BenchmarksHomeLocators.ACCOUNT_OPTION_PROFILE)).toBeVisible();
    await expect(await page.locator(BenchmarksHomeLocators.ACCOUNT_OPTION_DATA_MANAGEMENT)).toBeVisible();
    await expect(await page.locator(BenchmarksHomeLocators.ACCOUNT_OPTION_USER_MANAGEMENT)).toBeVisible();
    await expect(await page.locator(BenchmarksHomeLocators.ACCOUNT_OPTION_COMPANY_INFORMATION)).toBeHidden();

    //Check permissions on Data source management page
    await page.locator(BenchmarksHomeLocators.ACCOUNT_OPTION_DATA_MANAGEMENT).click();
    await page.locator(DataSourceManagerLocators.BUTTON_EDIT_SOURCE_OPTIONS).click();
    await expect(await page.locator(DataSourceManagerLocators.EDIT_DATA_SOURCE)).toBeVisible();
    await expect(await page.locator(DataSourceManagerLocators.EDIT_DATA_SOURCE_METADATA)).toBeVisible();
    await expect(await page.locator(DataSourceManagerLocators.DELETE_DATA_SOURCE)).toBeVisible();

    //Check permissions on group that was created by Admin role
    await page.goto(Url.BENCHMARKS_PROD + '/groups/joined');
    await page.locator(PrivateGroupLocators.ADMIN_GROUP_CHECK).click();

    async function isFinished(response) {
        return response.url().includes('/benchmarks/') && response.status() === 200 || response.status() === 400
    }
    const response = await page.waitForResponse(async (response) => await isFinished(response));
    console.log(response.status());

    await page.locator(PrivateGroupLocators.GROUP_OPTIONS).click();
    await expect(await page.locator(PrivateGroupLocators.SHARE_GROUP)).toBeVisible();
    await expect(await page.locator(PrivateGroupLocators.LEAVE_GROUP)).toBeVisible();
    await expect(await page.locator(PrivateGroupLocators.DELETE_GROUP)).toBeHidden();

    await expect(await page.locator(PrivateGroupLocators.METRICS)).toBeVisible();
    await expect(await page.locator(PrivateGroupLocators.MEMBERS)).toBeHidden();
    await expect(await page.locator(PrivateGroupLocators.DATA_SOURCES)).toBeVisible();

    await page.locator(PrivateGroupLocators.DATA_SOURCES).click();
    await expect(await page.locator(PrivateGroupLocators.BUTTON_ADD_DATA_SOURCES)).toBeEnabled();
    await expect(await page.locator(PrivateGroupLocators.BUTTON_REMOVE_DATA_SOURCE)).toBeVisible();

    await page.locator(PrivateGroupLocators.EDIT_GROUP).click();
    await page.locator(PrivateGroupLocators.SAVE_GROUP).click();
    await expect (await page.locator(BenchmarksHomeLocators.INFO_TOAST)).toBeVisible();

    //Check permissions on group that was created by Editor role
    await page.goto(Url.BENCHMARKS_PROD + '/groups/joined');
    await page.locator(PrivateGroupLocators.EDITOR_GROUP_CHECK).click();

    async function isFinished1(response) {
        return response.url().includes('/benchmarks/') && response.status() === 200 || response.status() === 400
    }
    const response1 = await page.waitForResponse(async (response) => await isFinished1(response));
    console.log(response1.status());

    await page.locator(PrivateGroupLocators.GROUP_OPTIONS).click();
    await expect(await page.locator(PrivateGroupLocators.SHARE_GROUP)).toBeVisible();
    await expect(await page.locator(PrivateGroupLocators.LEAVE_GROUP)).toBeVisible();
    await expect(await page.locator(PrivateGroupLocators.DELETE_GROUP)).toBeVisible();

    await expect(await page.locator(PrivateGroupLocators.METRICS)).toBeVisible();
    await expect(await page.locator(PrivateGroupLocators.MEMBERS)).toBeVisible();
    await expect(await page.locator(PrivateGroupLocators.DATA_SOURCES)).toBeVisible();

    await page.locator(PrivateGroupLocators.DATA_SOURCES).click();
    await expect(await page.locator(PrivateGroupLocators.BUTTON_ADD_DATA_SOURCES)).toBeEnabled();
    await expect(await page.locator(PrivateGroupLocators.BUTTON_REMOVE_DATA_SOURCE)).toBeVisible();

    await page.locator(PrivateGroupLocators.EDIT_GROUP).click();
    await page.locator(PrivateGroupLocators.SAVE_GROUP).click();
    await expect (await page.locator(BenchmarksHomeLocators.INFO_TOAST)).toBeVisible();
}
```

```
export const testPermissions =async (page: Page, user: User.IBenchmarkUser) => {
    await Benchmarks.LoginPage.actionLoginWithCredentials(page, user);
    await Benchmarks.ProfileMenu.assertPermissions(page, user);
    if(user instanceof User.Admin || user instanceof User.Editor){
        await Benchmarks.SourcesPage.assertPermissions(page, user);
        await Benchmarks.GroupsJoinedPage.assertAdminPermissionsInGroup(page, user, 'Admin Permission Group');
        await Benchmarks.GroupsJoinedPage.actionEditPrivateGroup(page, 'Admin Permission Group');
        await Benchmarks.GroupsJoinedPage.assertEditorPermissionsInGroup(page, user , 'Editor Permission Group');
        await Benchmarks.GroupsJoinedPage.actionEditPrivateGroup(page, 'Editor Permission Group');
        await Benchmarks.GroupsJoinedPage.assertUserPermissionsInGroup(page, user, 'User Permission Group');


    }
    if(user instanceof User.User){
        await Benchmarks.SourcesPage.assertPermissions(page, user);
        await Benchmarks.GroupsJoinedPage.assertUserPermissionsInGroup(page, user, 'User Permission Group');


    }
    if(user instanceof User.Viewer){
        await Benchmarks.GroupsJoinedPage.assertAdminPermissionsInGroup(page, user, 'User Permission Group');
    }
}
```

**Slika 6: Primerjava testnih scenarijev napisan v čistem Playwrightu in v našem ogrodju po abstrakciji.**

Poleg razvoja samega ogrodja smo čas posvetili tudi vpeljavi ogrodja v sam proces neprekinjene integracije (continuous integration). Vnaprej pripravljeni scenariji poskrbijo, da se pred uvedbo novih funkcionalnostih avtomatično zaženejo čim bolj podrobni testi, ki so nastali pod okriljem ekipe za zagotavljanje kakovosti. Tudi tukaj se je izkazalo ogrodje PTSD več kot primerno, saj je pisanje scenarijev kot zlaganje sestavljanke in lahko na enostaven način veliko programske kode uporabimo ponovno, kar pa ni presenetljivo, glede na to da so zadaj koncepti objektnega programiranja. Pri vpeljavi nove aplikacije v ekosistem podjetja smo lahko vse kar se tiče avtentikacije v aplikacijo preprosto ponovno uporabili iz prej omenjene komponente avtentikacije. Tudi pri testnih scenarijih, ki so zahtevali takšno sosledje dogodkov, da je bilo potrebno prehajanje iz ene v drugo aplikacijo, smo lahko enostavno ponovno uporabili že obstoječe funkcije.

## 4 Zaključek

Z vpeljavo ogrodja PTSD v podjetju smo razvijalcem ponudili možnost pisanja celovitih in dimnih testov na enak način kot bi pisali teste enot. Kakovost rešitev preden pridejo v roke ekipi za zagotavljanje kakovosti se je v ekipah, kjer smo to ogrodje vpeljali občutno dvignila. Zaznali smo več kot 50% zmanjšanje zaznanih napak, ki jih najde ekipa za zagotavljanje kakovosti. Odziv razvijalcev je prav tako bil pozitiven, saj se je zmanjšal čas pri pridobivanju prvih informacij glede testiranja. S tem ko so sami dobili možnost, da lahko enostavno in hitro poženejo celovite teste ter si sami sestavijo testni scenarij brez potrebnega znanja o testnih ogrodij se je povečala tudi njihova samozavest pri prehodu na nadaljnje faze razvojnega cikla.

Potrebno je pa tudi omeniti še eno pozitivno stran, ki jo je prineslo ogrodje PTSD. Sodelovanje med ekipo za zagotavljanje kakovosti, razvijalci in projektnimi vodji se je okrepilo, saj je prišlo do pretoka informacij do vseh akterjev prej že v fazi analize in načrtovanja. Ekipa za zagotavljanje kakovosti je zaradi zagotavljanja skladnosti ogrodja z želenimi načrti tako že vnaprej sodelovala pri načrtovanju in nudila povratne informacije glede na vpliv na celotno rešitev in morebitne posledice, ki jih razvijalci in projektni vodje niso predvideli.

## Literatura

[1] https://www.softwaretestinghelp.com/software-development-life-cycle-sdlc/, SDLC (Software Development Life Cycle) Phases, Process, Models, obiskano 4.7.2023

[2] https://katalon.com/, Katalon, obiskano 4.7.2023

[3] https://www.testim.io/blog/puppeteer-selenium-playwright-cypress-how-to-choose/ , Puppeteer, Selenium, Playwright, Cypress – how to choose?, obiskano 5.7.2023

[4] https://www.browserstack.com/guide/cypress-vs-selenium-vs-playwright-vs-puppeteer, Cypress vs Selenium vs Playwright vs Puppeteer: Core Differences, obiskano 5.7.2023

[5] https://playwright.dev/, Playwright, obiskano 5.7.2023

# Optimizacija spletne rešitve v ogrodju Next.js z Google Lighthouse

Leon Pahole

Povio Inc., San Francisco, United States of America
leon.pahole@povio.com

Raziskave kažejo, da so strani, ki se hitro naložijo, bolj optimizirane za spletne iskalnike, prijaznejše za uporabnike in bolj profitabilne. Google Lighthouse je orodje za analizo kvalitete spletnih strani, s katerim lahko med drugim izmerimo učinkovitost nalaganja strani. Učinkovitost je izmerjena na podlagi metrik, ki izražajo uporabniško izkušnjo ob nalaganju. Next.js je ogrodje za razvoj spletnih aplikacij, ki temelji na knjižnici React, pri čemer še posebej poudarja optimizacijo delovanja spletnih aplikacij, predvsem nalaganje le-teh. V prispevku smo na kratko razložili performančne metrike, ki jih meri Google Lighthouse v sklopu ocene učinkovitosti spletne strani, nato pa smo na podlagi praktičnih izkušenj predstavili naslednje Next.js tehnike za optimizacijo teh metrik: izrisovanje na strežniku, statično izrisovanje, inkrementalno statično izrisovanje, ločevanje kode, optimizacija slik, optimizacija pisav, strežniške komponente in nalaganje na zahtevo. Izkaže se, da lahko v praksi s pomočjo navedenih tehnik, v kombinaciji z meritvami iz Google Lighthouse, na enostaven in intuitiven način izboljšamo nalaganje naše spletne strani, kar vodi k boljši uporabniški izkušnji.

**Ključne besede:**
Next.js
Google Lighthouse
optimizacija nalaganja
kakovost informacijskih rešitev
spletne aplikacije

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.15

# 1 Uvod

V modernem razvoju spletnih rešitev ni pomembno le, da so naše spletne strani privlačne in funkcionalne, temveč tudi to, da so učinkovite. Raziskave kažejo, da so strani, ki se hitro naložijo, bolj optimizirane za spletne iskalnike (angl. search engine optimization, v nadaljevanju SEO), prijaznejše za uporabnike in bolj profitabilne [1,2,3].

Google Lighthouse (v nadaljevanju Lighthouse) je eno izmed orodij za analizo učinkovitosti nalaganja spletnih strani. Deluje na podlagi ocen metrik, ki so bile iz strani Lighthouse ekipe določene kot pomembne za hitro nalaganje in dobro uporabniško izkušnjo.

V tem prispevku smo predstavili lastne izkušnje in s tem povezano znanje za optimizacijo spletnih strani, napisanih v ogrodju Next.js, glede na metrike iz Google Lighthouse:

- V prvem delu smo opisali metrike, ki jih Lighthouse definira kot pomembne za učinkovitost.
- V drugem delu smo na podlagi analize in lastnih izkušenj predstavili tehnike v ogrodju Next.js, s katerimi lahko izboljšamo Lighthouse oceno in posledično pohitrimo delovanje spletne strani.

Kljub temu da v prispevku obravnavamo ogrodje Next.js, je prispevek zasnovan konceptualno, zato upamo, da bodo podane informacije koristile tudi spletnim razvijalcem, ki ne uporabljajo ogrodja Next.js.

# 2 Google Lighthouse

Lighthouse je odprtokodno orodje za analizo kvalitete spletnih strani. Kvaliteta je izražena v petih kategorijah: učinkovitost (angl. performance), SEO, dostopnost (angl. accessibility), dobre prakse (angl. best practices) in progresivne spletne aplikacije (angl. progressive web apps ali PWA) [4].

Rezultat Lighthouse analize je numerična ocena (med 1 in 100) za vsako izmed kategorij in seznam predlogov za izboljšavo ocene. Ker je teste možno izvesti hitro in pogosto, lahko razvijalci Lighthouse uporabijo kot pripomoček pri izboljšavi kvalitete spletne strani.

V tem prispevku smo se osredotočili na oceno učinkovitosti, s pomočjo katere lahko merimo hitrost nalaganja spletne strani. Pri tem posebej poudarjamo, da gre za optimizacijo začetnega nalaganja spletne strani in ne optimizacijo delovanja strani, ko je le-ta že naložena.

## 2.1 Izračun Lighthouse ocene učinkovitosti

Kriterij izračuna Lighthouse ocene1 ni stalen, temveč se lahko skozi čas spreminja glede na to, katere metrike se na podlagi raziskav ekipi Lighthouse zdijo najprimernejše kot izraz dobre uporabniške izkušnje [5].

V času pisanja tega prispevka je aktualna Lighthouse verzija 10 [6], ki se osredotoča na naslednje metrike2: First Contentful Paint (FCP), Speed Index (SI), Largest Contentful Paint (LCP), Total Blocking Time (TBT) in Cumulative Layout Shift (CLS).

Vsaka izmed metrik prispeva h končni oceni na podlagi uteži, ki predstavlja njeno pomembnost. Uteži so določene na podlagi podatkov iz realnih strani in so porazdeljene po logaritemsko normalni porazdelitvi. Za lažjo predstavo si lahko pomagamo s kalkulatorjem uteži in ocene [7].

---

[1] seznam sestavnih delov – izvorno "Software Bill of Materials", okrajšava "SBOM"
[1] Repozitorij, https://www.sona

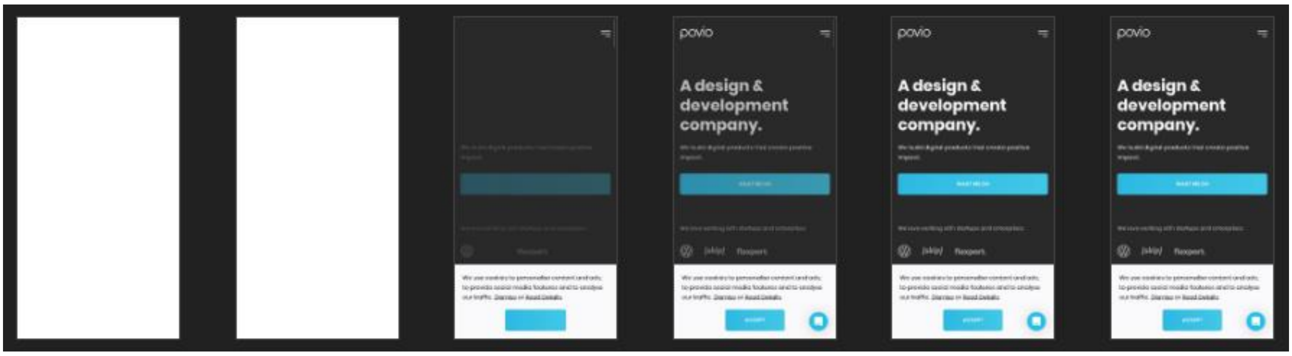
V nadaljevanju so predstavljene performančne metrike. Ker se v prihodnosti lahko spremenijo, smo metrike opisali na kratko, bralce pa pozivamo k rednemu spremljanju novih verzij Lighthouse [6]. Prav tako je pomembno razumevanje dejstva, da so vse metrike izbrane na podlagi vizualne uporabniške izkušnje med procesom nalaganja.

### 2.1.1 *First Contentful Paint (FCP)*

FCP meri čas (v sekundah), v katerem se na strani prikaže prvi vizualni element - tekst, slika, ali katerikoli drug element, ki je viden [8].

Slika 1 prikazuje korake nalaganja spletne strani - v tem primeru se FCP zgodi na tretjem koraku, ko se na zaslon izrišejo prvi elementi.

Metrika FCP je pomembna, saj izraža dobro uporabniško izkušnjo pri nalaganju strani - prej, ko se na strani začne izrisovati katerikoli vizualni element, prej uporabnik dobi indikacijo, da se stran dejansko nalaga.



**Slika 1: Izris spletne strani.**

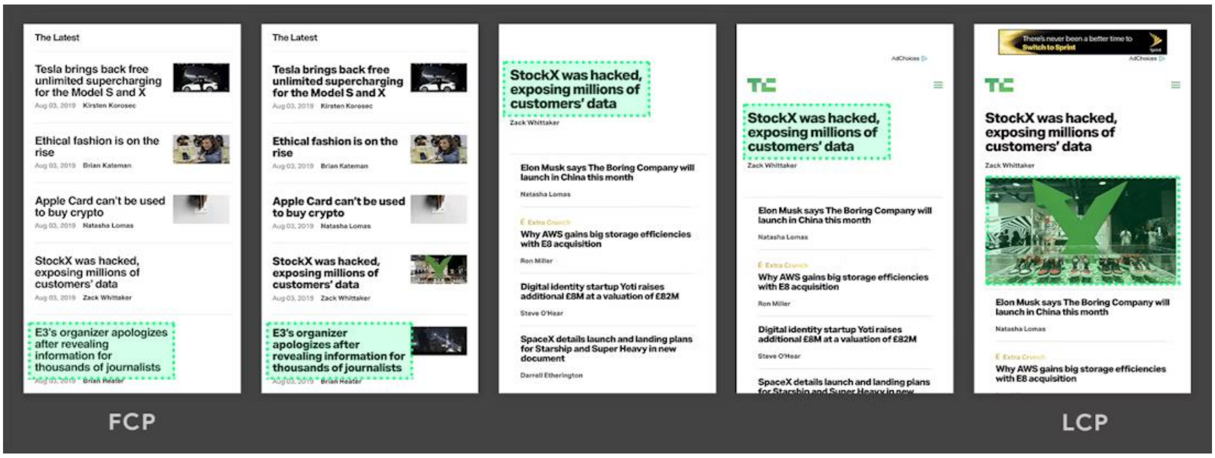
### 2.1.2 *Speed Index (SI)*

SI meri hitrost (v sekundah), s katero se vizualni elementi na strani izrisujejo med nalaganjem in kako hitro se na strani izrišejo vsi vizualni elementi. V nasprotju s FCP, je SI bolj celovita metrika, saj obravnava celotno časovnico nalaganja spletne strani, vključno s postopnim nalaganjem elementov in enakomernostjo nalaganja [9,10]. Metrika se izračuna na podlagi videa nalaganja strani, nad katerim se izvede algoritem Speedline [11].

### 2.1.3 *Largest Contentful Paint (LCP)*

LCP meri čas (v sekundah), v katerem se na strani prikaže največji in najpomembnejši vizualni element. Ta element določi hevristični algoritem [12] in je lahko, na primer slika, video, ali velik tekst s pomembnimi informacijami [12]. Pri tem je pomembno poudariti, da se skozi nalaganje strani ta element lahko spremeni, pri merjenju LCP pa je pomemben samo zadnji tak element. Slika 2 prikazuje primer spreminjanja najpomembnejšega elementa (označen s pravokotnikom) skozi nalaganje spletne strani.

Podobno kot FCP in SI, tudi LCP poudarja dobro uporabniško izkušnjo pri nalaganju spletne strani; uporabniki želijo, da se najpomembnejši del spletne strani naloži čimprej.



**Slika 2: Izris spletne strani in spreminjanje najpomembnejšega elementa (označen s pravokotnikom)**
Vir: [12].

### *2.1.4 Total Blocking Time (TBT)*

Glavna nit (angl. main thread) v brskalniku je zadolžena za izrisovanje spletne strani, izvajanje JavaScript kode in obravnavanje uporabniškega vnosa [13]. Če se ob nalaganju prenese veliko JavaScript kode, je treba le-to prevesti (angl. parse) in izvesti. Med tem časom je glavna nit blokirana in ni zmožna obravnavati uporabniškega vnosa. TBT meri trajanje (v milisekundah), v katerem je bila glavna nit na tak način blokirana, in sicer med FCP in trenutkom, ko je bila stran naložena in je postala interaktivna [14].

TBT meri interaktivnost spletne strani med nalaganjem. Velik TBT pomeni, da je stran med nalaganjem neinteraktivna in se vsi uporabniški vnosi obravnavajo z zamikom, kar nakazuje na slabo uporabniško izkušnjo.

### *2.1.5 Cumulative Layout Shift (CLS)*

CLS meri vizualno stabilnost spletne strani na podlagi nepričakovanih zamikov v postavitvi strani (angl. layout shift, v nadaljevanju zamik). Zamik se zgodi, ko se vsebina spletne strani nepričakovano spremeni ali prestavi, kar povzroči zamik celotne vsebine strani. Takšen zamik negativno vpliva na uporabniško izkušnjo, saj ni vizualno prijeten, prav tako pa lahko pride do klikov na nepredvidene elemente. Metrika je izražena z oceno med 0 in 1, kjer 0 predstavlja stran brez zamikov, 1 pa stran z veliko motečimi zamiki [15].

### *2.1.6 Velikost JavaScript paketa*

Velikost JavaScript paketa (v kB), ki se ob nalaganju prenese na odjemalca (angl. bundle size) ni metrika, ki se neposredno uporablja pri merjenju ocene učinkovitosti spletne strani, jo je pa kljub temu smiselno omeniti, saj posredno vpliva na večino metrik. Hkrati so v sodobni praksi popularna ogrodja, ki uporabljajo veliko količino JavaScript kode, zato je treba velikost JavaScript paketa nenehno spremljati.

V poglavju 2.1.4 smo omenili, da je potrebno JavaScript kodo ob nalaganju prenesti, prevesti in izvesti. Večji, kot je JavaScript paket, dlje časa bo ta proces trajal, kar bo posledično povzročilo vpliv na metrike FCP, LCP, SI in TBT. Kot bomo spoznali v poglavju 3, je veliko število tehnik za optimizacijo namenjenih ravno zmanjševanju velikost JavaScript paketa.

## 2.2 Merjenje metrik

Učinkovitost naše strani lahko ocenimo na podlagi dveh tipov podatkov: laboratorijski (angl. lab) in terenski (angl. field) [16].

Laboratorijski podatki so pridobljeni s strani razvijalcev v kontroliranem okolju - ponavadi v brskalniku na lokalni napravi razvijalca. Takšni testi so zelo hitri in ponovljivi, zato jih je smiselno zaganjati pogosto, za namene spremljanja in izboljševanja učinkovitosti strani.

Kljub temu da nam laboratorijski podatki lahko podajo približno oceno učinkovitosti spletne strani, pa le-ti niso dovolj reprezentativni za vse uporabnike, ki obiščejo našo spletno stran; uporabniki prihajajo iz različnih geografskih lokacij, z različnimi napravami in različnimi internetnimi hitrostmi. Da dobimo vpogled v to, kako učinkovita je naša stran za realne uporabnike, je potrebno analizirati terenske podatke. Ti podatki se zberejo, medtem ko uporabniki uporabljajo stran in se pošljejo v centralizirano analitiko (npr. Google Analytics).

V praksi predlagamo, da razvijalci uporabljajo oba tipa podatkov.

- Laboratorijski podatki so uporabni v začetnih fazah razvoja, ko še nimamo terenskih podatkov ali pa takrat, ko na podlagi terenskih podatkov vemo, da imamo težavo z učinkovitostjo in moramo težavo podrobneje analizirati v lokalnem razvoju.
- Terenski podatki so nujni za realno sliko o tem, kako učinkovita je naša spletna stran. Dobra Lighthouse ocena še ne pomeni, da je naša stran popolnoma učinkovita za vse uporabnike. Metrike iz terenskih podatkov je potrebno nenehno spremljati.

### *2.2.1 Izvedba Lighthouse testa*

Lighthouse test najlažje izvedemo neposredno v brskalniku Google Chrome ali pa s pomočjo vtičnika Google Lighthouse [4,17]. Obstajajo tudi spletna orodja, s katerimi lahko Lighthouse test zaženemo iz različnih lokacij in v različnih simuliranih okoljih (npr. mobilna naprava s slabo povezavo). Pri testiranju je potrebno omeniti slednje:

- Lighthouse test bo zbral laboratorijske podatke.
- Test je potrebno zagnati v brskalniku v privatnem načinu (angl. incognito), saj s tem preprečimo, da bi predpomnjeni podatki ali vtičniki vplivali na oceno.
- Če teste zaganjamo na lokalni strani (angl. localhost), je smiselno uporabiti Google Lighthouse v terminalu, s čimer lahko simuliramo omrežno latenco [18].
- Testiramo lahko v mobilnem ali navadnem načinu. Mobilni način simulira določene omejitve mobilnih naprav, kot so znižana procesorska zmogljivost in slaba omrežna povezava. Večina spletnih strani cilja tako mobilne naprave kot tudi računalnike, zato sta pomembni obe oceni, pri čemer pa lahko mobilno oceno interpretiramo kot najslabši možni scenarij in stran optimiziramo glede na le-to.
- Poleg hitrosti nalaganja spletne strani je s pomočjo izbire načine (angl. mode) možno testirati tudi druge faze delovanja strani [19]. V tem prispevku se na druge načine nismo osredotočili.
- Lighthouse analizira zgolj eno stran. Večina spletnih strani vsebuje več strani in idealno je, da test zaženemo na vseh. To lahko naredimo z orodjem, kot je Unlighthouse [36].

## 3 Tehnike za izboljšanje učinkovitosti v Next.js

Next.js je popularno (več kot 4 milijoni prenosov na teden v času pisanja prispevka [20]) ogrodje za razvoj spletnih aplikacij, ki temelji na ogrodju React. Ogrodje postavlja velik poudarek na dobri razvijalski izkušnji (angl. developer experience ali DX) in velikemu naboru tehnik za izboljšanje učinkovitosti spletnih aplikacij, predvsem nalaganja le-teh. V tem poglavju smo predstavili nekaj izmed teh tehnik, analizirali njihov vpliv na performančne metrike in podali praktične nasvete za uporabo.

Za namene jedrnatosti smo vsako tehniko opisali zgolj na konceptualnem nivoju. Ogrodje Next.js se namreč vseskozi razvija, tako da bo lahko v prihodnosti koda drugačna, kot je v času pisanja prispevka, koncepti pa bodo ostali enaki.

## 3.1 Izrisovanje na strežniku

Izrisovanje na strežniku (angl. server-side rendering ali SSR) je tehnika, pri kateri se vsebina strani v obliki HTML kode generira na strežniku, nato pa se generirana HTML koda pošlje na odjemalca [21]. To je v nasprotju z izrisovanjem na odjemalcu (angl. client-side rendering ali CSR), kjer se spletna aplikacija naloži kot prazna stran, nato pa se izrisuje s pomočjo JavaScripta - to počnejo ogrodja za enostranske aplikacije (angl. single-page application ali SPA), kot so React, Angular in Vue. Next.js privzeto uporablja SSR za vse strani, ki so odvisne od zunanjih podatkov.

SSR v ogrodju Next.js deluje drugače kot v tradicionalnih ogrodjih, kot je npr. PHP. Ko se stran v Next.js aplikaciji naloži na odjemalca, bo zagonsko okolje React prevzelo delovanje spletne strani. To pomeni, da bo React obravnaval uporabniške vnose, izrisovanje in navigacijo, kar praktično pomeni, da se stran prične obnašati enako, kot se obnaša SPA stran. Lahko bi torej rekli, da je Next.js hibriden pristop med SSR in SPA - ob nalaganju se uporabi SSR, ob nadaljnji interakciji pa SPA. To je ponavadi zaželjeno, saj so SPA aplikacije po nalaganju hitrejše in bolj interaktivne, ker ni potrebe po osveževanju strani.

Proces, v katerem Next.js preda izvajanje aplikacije ogrodju React, ni zastonj. Potrebno je namreč inicializirati React zagonsko okolje, zgraditi vse podatkovne strukture (na primer drevo komponent) in povezati vse poslušalce dogodkov (angl. event handler). Ta proces se imenuje hidracija (angl. hydration) [22,23]. Med hidracijo je stran neodzivna za uporabniško interakcijo (šteje se kot eno izmed opravil, ki povečujejo TBT). To pomeni tudi, da je ob nalaganju strani potreben JavaScript.

### *3.1.1 Vpliv SSR na performančne metrike*

Tabela 1 prikazuje vpliv SSR na performančne metrike v primerjavi s SPA. Pojasnimo:

- FCP, LCP in SI so manjše pri SSR, saj se iz strežnika pošlje že pripravljen HTML, ki se lahko takoj prične izrisovati. Zaradi tega se prvi vizualni izris in izris najpomembnejšega elementa zgodita prej. V primeru SPA pa se naloži prazna stran, ki se mora nato še prvič izrisati.
- TBT je manjši za SSR. Čeprav mora SSR pristop izvesti hidracijo, pa mora mora SPA poleg tega izvesti še prvi izris.
- Opomniti moramo, da SSR primer uporablja ločevanje kode (poglavje 3.4), ki dodatno pozitivno vpliva na rezultate.

### *3.1.2 Kdaj uporabiti SSR*

V preteklosti je bilo izbiro med SSR in SPA nujno opraviti na začetku projekta ter izjemno pomembna, saj je napačna odločitev lahko pomenila dolgotrajno migracijo v prihodnosti. Next.js to odločitev nekoliko poenostavi, saj lahko na nivoju posamezne strani (in celo komponente) določimo, katero metodo izrisa bomo uporabili.

SSR ima nekaj prednosti pred SPA:

- Je optimiziran za SEO, saj strežnik ob nalaganju vrne izrisan HTML, namesto prazne strani.
- Pospeši nalaganje strani, saj ni potrebno izvesti začetnega izrisa na odjemalcu. To je pomembno predvsem za mobilne naprave, ki imajo manj procesorske moči.
- Če je naša aplikacija odvisna od podatkov, lahko le-te pridobimo že na strežniku, kar lahko pohitri izvajanje, saj je komunikacija med strežniki ponavadi hitrejša kot komunikacija med odjemalcem in strežnikom. Prav tako je tak način komunikacije varnejši, saj lahko dostop do vira podatkov omejimo zgolj na naš Next.js strežnik.

SPA ima pa naslednje prednosti pred SSR:

- SSR za delovanje potrebuje relativno močan strežnik, medtem ko SPA lahko serviramo na omrežju za dostavo vsebine (angl. content delivery network ali CDN). To pomeni, da je vzdrževanje SPA aplikacij potencialno cenejše, prav tako pa se lahko hitreje naložijo s strežnika (saj je CDN geografsko bližje uporabniku).
- Ker SPA izrisuje zgolj na odjemalcu, je strežnik manj obremenjen kot pri SSR. To lahko naredi razliko pri veliki količini uporabnikov.

Naš predlog za izbiro tehnike izrisovanja temelji predvsem na tem, koliko denarnih sredstev imamo na razpolago in kako pomemben je SEO. Poudariti je treba, da SSR optimizira zgolj nalaganje spletne strani.

- Če razvijamo javno stran, kjer je pomemben SEO in hitrost nalaganja, uporabimo SSR.
- Če razvijamo interno stran (na primer nadzorno ploščo), ki ne potrebuje SEO ali hitrega nalaganja, uporabimo SPA.
- Next.js aplikacijo je možno pretvoriti v SPA aplikacijo, zato lahko v vsakem primeru uporabimo Next.js [24].

## 3.2 Statično izrisovanje

Statično izrisovanje (angl. static-site generation ali SSG) je tehnika, pri kateri se statične HTML datoteke naše strani generirajo v času prevajanja (angl. build time) naše aplikacije. Generirane HTML datoteke se nato v času izvajanja servirajo statično, brez potrebe po SSR [25]. Če naša stran vsebuje podatke iz zunanjih virov, se podatki pridobijo v času prevajanja. Prav tako se generirajo vse dinamične strani (npr. /blog/2 za drugo stran bloga).

Pomembno je poudariti, da je edina razlika med SSR in SSG hitrost, pri kateri strežnik odgovori na zahtevo odjemalca - pri SSR mora strežnik najprej generirati HTML, pri SSG pa lahko samo vrne vnaprej generiran HTML. Ko se stran naloži, se SSG stran prične obnašati enako, kot SSR (torej, kot SPA), zato hidracija je potrebna.

Next.js privzeto uporablja SSG za vse strani, ki niso odvisne od zunanjih podatkov.

### *3.2.1 Vpliv SSG na performančne metrike*

Tabela 1 prikazuje vpliv SSG na performančne metrike v primerjavi s SSR. Pojasnimo:

- Pričakovano je, da so FCP, LCP in SI boljše v primeru SSG, saj se strežnik hitreje odzove na uporabniško zahtevo in se posledično stran lahko prične hitreje izrisovati.
- Velikost paketa JavaScript in TBT ste enaka, saj tako SSR in SSG izvedeta hidracijo in potrebujeta enako količino JavaScripta za delovanje. Razlika je torej samo v hitrosti odgovora.

### *3.2.2 Kdaj uporabiti SSG*

V Next.js lahko za vsako stran izberemo, ali želimo, da se izrisuje s tehniko SSG. Kako sprejeti to odločitev?

SSG ima naslednje prednosti pred SSR:

- Je hitrejši način izrisovanja, saj serviramo zgolj statični HTML.
- Statične strani lahko serviramo iz CDN, kar dodatno pohitri zahtevke.

SSR ima kljub temu nekaj prednosti pred SSG:

- Največja slabost SSG je nespremenljivost; strani se generirajo med fazo prevajanja in se med izvajanjem ne morejo spreminjati. Če se na naši strani vsebina spreminja (npr. socialno omrežje, kot je Twitter, ali stran, ki je povezana na sistem za upravljanje vsebin /angl. content management system ali CMS/), SSG najverjetneje ni primeren. Ena izmed rešitev za to težavo je, da v času prevajanja generiramo HTML za statične elemente (navigacija, ikone, ipd.), nato pa dinamične podatke naložimo v času izvajanja na odjemalcu, po tem, ko se zgodi hidracija, vendar ima to slab SEO in je počasnejše od SSR.

- Če imamo veliko strani, se bo SSG stran prevajala dolgo časa, prav tako pa bodo te strani potrebovale veliko prostora na strežniku.

Izbira SSG je torej pogojena z naravo vsebine spletne strani:

- Če se vsebina ne spreminja (npr. statična informativna stran), uporabimo SSG.
- Če se vsebina spreminja, ampak dovolj redko, da si lahko privoščimo ponovni zagon prevajanja ob vsaki spremembi (npr. osebni blog, na katerega objavljamo 1-krat na teden), uporabimo SSG.
- Če se podatki spreminjajo zelo hitro (npr. stran s spremljanjem športnega rezultata v živo) ali pa so podatki personalizirani na uporabnika (npr. Facebook časovnica), uporabimo SSR.
- Če se podatki spreminjajo relativno redko (npr. enkrat na 10 minut), ampak si ne moremo privoščiti ponovnega prevajanja, lahko uporabimo ISR (opisan v poglavju 3.3).

## 3.3 Inkrementalno statično izrisovanje

V poglavju 3.2 smo omenili, da lahko SSG uporabimo za osebni blog, ker si lahko privoščimo ponovno prevajanje strani ob vsaki objavi. Kaj pa, če ima naš blog več urednikov, ki občasno pišejo objave, ampak nimajo tehničnega znanja, da bi zagnali ponovno prevajanje? V tem primeru moramo uporabiti SSR, da so objavljeni podatki ažurno prikazani na strani.

Vendar ker je SSG tako hiter, bi bilo kljub temu optimalno, da bi ga lahko nekako uporabili v primerih, ko se podatki ne spreminjajo zelo hitro (nekajkrat na minuto). To nam omogoča tehnika inkrementalnega statičnega izrisovanja.

Inkrementalno statično izrisovanje (angl. incremental static rendering ali ISR) nam omogoča, da med izvajanjem aplikacije selektivno posodobimo statične HTML datoteke, ki smo jih generirali v času prevajanja. Prav tako lahko na novo generiramo strani, ki sploh niso bile generirane v času prevajanja: ko uporabnik prvič zahteva stran, ki še ni generirana, jo izrišemo s pomočjo SSR, rezultat shranimo in ob naslednjih zahtevkih serviramo to HTML datoteko iz predpomnilnika. ISR torej v bistvu kombinira SSR (prvi zahtevek) in SSG (naslednji zahtevki) [26].

V ISR je pomembna nastavitev, kako pogosto (če sploh) bomo posodabljali predpomnilnik:

- Na interval, npr. po 5 minutah, bo stran označena kot neažurna in se bo ob naslednjem zahtevku ponovno zgenerirala (s pomočjo SSR).
- Na določen dogodek, npr., ko se spremenijo podatki v CMS. V začetku tega poglavja smo omenili primer z blogom - v tem primeru je dogodek objava.

Vidimo lahko, kako ISR kombinira prednosti SSR (dinamičnost) in SSG (hitrost nalaganja).

### *3.3.1 Vpliv ISR na performančne metrike*

ISR ne počne nič novega, kar nismo analizirali že v prejšnjih poglavjih - ob prvem zahtevku bodo metrike podobne, kot pri SSR (poglavje 3.1), ob naslednjih zahtevkih (do posodobitve predpomnilnika) pa bodo podobne, kot pri SSG (poglavje 3.2).

### *3.3.2 Kdaj uporabiti ISR*

Uporabo ISR predlagamo vedno, ko strani niso statične, ampak se podatki ne spreminjajo tako pogosto, da bi bil strežnik preobremenjen ob ponovnem generiranju HTML datotek.

Dober primer strani, kjer lahko uporabimo ISR, so strani, ki so povezane na CMS ali blogi. Tam pričakujemo relativno redke posodobitve, npr. enkrat na 10 minut.

ISR ni dobra izbira za strani, ki vsebujejo zelo hitro spreminjajoče se podatke (npr. spremljanje lokacije vozila), ali pa so podatki personalizirani za uporabnika (npr. seznam predlaganih izdelkov za uporabnika). Prav tako moramo v obzir vzeti dejstvo, da ISR potrebuje nekaj časa, da izvede generiranje datotek, zato spremenjeni podatki ne bodo takoj vidni na strani. V določenih primerih si tega ne moremo privoščiti (npr. realno-časovno spremljanje novic).

## 3.4 Ločevanje kode

Naše spletne strani ponavadi sestavlja več strani. V tipični SPA (npr. React) aplikaciji se JavaScript koda za celotno spletno stran (torej za vse strani) prenese ob nalaganju. To je seveda precej potratno, saj stran, ki se nalaga, najverjetneje ne potrebuje celotne kode.

Ločevanje kode je tehnika, pri kateri v času prevajanja JavaScript kodo razdelimo na več datotek (angl. chunk), ponavadi eno za vsako stran. Cilj je zmanjšati nalaganje spletne strani tako, da prenesemo zgolj JavaScript kodo, ki je potrebna za izvajanje zahtevane strani - koda za ostale strani pa se lahko prenese v ozadju, ko se nalaganje zaključi [27].

### *3.4.1 Vpliv ločevanja kode na performančne metrike*

Ločevanje kode prispeva k razliki v metrikah, ki je vidna v tabeli 1, kjer smo primerjali React (brez ločevanja kode) in Next.js (z ločevanjem kode). Ločevanje kode zagotovo izboljša vse metrike, povezane z nalaganjem strani (predvsem TBT), saj zniža količino kode, ki se mora ob nalaganju prenesti in izvesti.

### *3.4.2 Kdaj uporabiti ločevanje kode*

Next.js privzeto uporablja ločevanje kode in v času pisanja tega prispevka ne obstaja način, da bi ga lahko izklopili. Menimo, da je ločevanje kode uporabno na vseh spletnih straneh, zato se strinjamo s tem, da je vedno vključeno.

## 3.5 Optimizacija slik

Slike so pomemben del vsebine spletnih strani. Ponavadi jih vključimo z značko img, v Next.js pa lahko uporabimo tudi komponento Image, ki je zgrajena nad img značko in za sliko omogoči naslednje optimizacije [28]:

- Nalaganje na zahtevo (angl. lazy loading): slike se naložijo šele, ko so v vidnem polju.
- Optimizacija velikosti: Next.js za vsako sliko generira več velikosti in nato odjemalcu pošlje tisto, ki je najprimernejša glede na velikost in zmogljivost naprave. Na primer, na mobilnih napravah slike v visoki ločljivosti niso potrebne, zato se pošljejo manjše slike, kar pohitri nalaganje.
- Uporaba modernih formatov: v kolikor ga uporabnikov brskalnik podpira, bo Next.js na odjemalca poslal sliko v modernem slikovnem formatu, kot je na primer WebP. Moderni formati so namreč pri isti kvaliteti manjši kot tradicionalni formati, kot sta npr. PNG ali JPEG [29].
- Preprečitev zamikov: Image komponenta preprečuje razvijalcem, da sliki ne bi določili ustreznih dimenzij, kar pripomore k boljši oceni za metriko CLS.

Image komponenta deluje tako za statične, vnaprej določene slike, kot tudi za dinamične, oddaljene (angl. remote) slike. V ozadju Next.js pridobi sliko, jo pretvori v ustrezen format in velikost, jo predpomni in nato pošlje na odjemalca (v tem pogledu deluje podobno kot ISR za HTML datoteke).

### 3.5.1 Vpliv optimizacije slik na performančne metrike

Tabela 1 prikazuje vpliv optimizacije slik na performančne metrike, v primerjavi z img značko brez dimenzij. Pojasnimo:

- Slikam je vedno potrebno definirati dimenzije. Če uporabimo značko img brez dimenzij (atributa width in height), brskalnik zaradi manjkajočih dimenzij ne bo rezerviral prostora za sliko, kar povzroči zamik ob nalaganju. Zaradi tega je CLS večji, kot je pri Image komponenti, ki nam preprečuje, da slikam ne bi določili dimenzij.
- Image komponenta kaže izboljšavo v vseh metrikah, povezanih s hitrostjo, saj so dostavljene slike manjše in v optimalnem formatu.
- Image komponente se prikažejo hitreje, kar izboljša FCP.
- Image komponenta zahteva več logike na odjemalcu, zato se velikost JavaScript paketa poveča.

### 3.5.2 Kdaj uporabiti optimizacijo slik

Izgleda, kot da Image komponenta nima slabih lastnosti in se lahko vedno uporablja namesto img značke. S tem se strinjamo, ko gre za statične slike (na primer logotipi, ozadja in ostale dekorative slike).

Ko pa gre za dinamične, oddaljene slike (na primer slike iz strežnika, profilne slike ali druge slike, ki so bile naložene s strani uporabnikov), moramo biti previdni. Spomnimo, da mora vsaka slika v Image komponenti biti prenešena z izvornega strežnika na predpomnilnik Next.js strežnika, kar bo postopoma povečevalo količino shranjenih podatkov in obremenilo strežnik. Zaradi tega slike na večini spletnih ponudnikov za gostovanje Next.js aplikacij niso zastonj. Na primer, v času pisanja prispevka, Vercel gostovanje ponuja zgolj 1000 slik na periodo zastonj [37].

Če stran vsebuje veliko število slik, je potrebno pretehtati naslednje:

- Če razvijamo javno stran, kjer je hitrost nalaganja zelo pomembna, moramo za optimalno nalaganje zagotovo uporabiti Image komponento.
- Če razvijamo interno nadzorno ploščo, ki vsebuje veliko slik, nalaganje pa ni bistvenega pomena, je verjetno bolje uporabiti img značko, s čimer znižamo stroške in obremenjenost strežnika.

### 3.5.3 Kako izbrati pravilno dimenzijo slik

Image komponenta razvijalce prisili v to, da za sliko definiramo dimenzije. Sliki lahko nastavimo statično velikost (preko width in height atributov) ali pa ji povemo, da se razširi čez celotnega starša.

V kolikor se dimenzije slike ne spreminjajo skozi različne dimenzije zaslonov (npr. logotipi, profilne slike in ikone), vedno uporabimo statične dimenzije. S tem bomo zagotovili optimalno nalaganje brez zamika.

Ko delamo s slikami, katerih dimenzije se spreminjajo na podlagi širine zaslona, moramo dimenzije nastaviti na drug način. Slika 3 prikazuje referenčno implementacijo za primer, kjer želimo, da je slika širine 500px na računalniku, polovico širine zaslona (50vw) na tablici in celotno širino zaslona (100vw) na telefonih. Storili smo slednje:

- Sliko smo nastavili tako, da se raztegne čez celotnega starša (element div).
- Starša smo s pomočjo CSS nastavili na ustrezne širine za vsako izmed širin zaslona.
- Na sliki smo uporabili atribut sizes, da za vsako širino zaslona odjemalec zahteva ustrezno velikost slike.

```
<div className={crewMemberPageStyles.imageWrapper}>
  <Image
    src={crewMember.images.png}
    alt={`Image of ${crewMember.name} crew member`}
    sizes={`
      (max-width: ${Breakpoints.mobile}) 100vw,
      (max-width: ${Breakpoints.tablet}) 50vw,
      500px
    `}
    fill
    className={crewMemberPageStyles.image}
  />
</div>
```

```
.imageWrapper {
  position: relative;

  width: 100vw;
  height: 400px;

  @include breakpoints.tablet {
    width: 50vw;
    height: 200px;
  }

  @include breakpoints.desktop {
    width: 500px;
    height: 300px;
  }
}
```

**Slika 3: Koda za določitev dimenzij slike, ki ima spreminjajoče se dimenzije na različnih širinah zaslonov.**

V tem primeru je treba posebej poudariti atribut sizes. Atribut določa širino slike, ki jo bo odjemalec zahteval na podlagi širine zaslona. Če se bo stran npr. naložila na telefonu, bo odjemalec od strežnika zahteval sliko s širino, ki je enaka širini zaslona (100vw; npr. če je telefon širine 450px, bo to 450px). Na tablici bo polovico zaslona (50vw), na računalniku pa 500px. Višina slike se na strežniku samodejno izračuna iz širine na podlagi razmerja med širino in višino slike (angl. aspect ratio). Pomembno je poudariti, da Next.js najverjetneje ne bo vrnil slike s točno zahtevano dimenzijo, ampak z dimenzijo, ki je podobna, ampak večja od zahtevane. Next.js si namreč v ozadju zgenerira končno število slik z dimenzijami in ne generira dodatnih slik za vsak zahtevek.

## 3.6 Optimizacija pisav

Pisave so vključene v praktično vsaki moderni spletni strani, vendar pa optimalno nalaganje pisav ponavadi povzroči določene izzive za razvijalce [30]:

- Blokiranje nalaganja: če nismo previdni, lahko nalaganje pisav blokira glavno nit brskalnika, kar pomeni, da se stran ne bo prikazala, dokler se pisava ne bo naložila. To lahko rešimo za asinhronim nalaganjem.
- Prikaz nestiliziranega besedila (angl. flash of unstyled text ali FOUT): če pisave nalagamo asinhrono, se lahko zgodi, da se bo ob nalaganju za nekaj trenutkov prikazala rezervna pisava (angl. fallback font), preden se bo zamenjala s pravo pisavo. To povzroči vidni zamik, ki poslabša CLS. To lahko rešimo tako, da za rezervno pisavo uporabimo čim bolj podobno pisavo, kot je pisava, ki jo nalagamo, nato pa jo še dodatno približamo s pomočjo CSS lastnosti, kot so line-height, letter-spacing in font-size.

Optimizacija pisav (angl. Font optimization), ki jo ponuja Next.js, samodejno reši vse zgoraj opisane težave. Kot razvijalci moramo samo določiti ustrezno pisavo, ki je lahko lokalna ali pa naložena iz Google fonts, Next.js pa bo poskrbel za pravilno nalaganje in ustrezno prilagojeno rezervno pisavo [31].

### *3.6.1 Vpliv optimizacije pisav na performančne metrike*

Tabela 1 prikazuje vpliv optimizacije pisav na performančne metrike, v primerjavi z načinom, ki blokira glavno nit (blokiranje) in z asinhronim nalaganjem s FOUT (async). Pojasnimo:

- V primerjavi z blokiranjem se izboljšajo metrike, povezane s hitrostjo nalaganja, saj pri blokiranju nalaganje strani blokiramo, dokler pisava ni naložena.
- V primerjavi z asinhronim načinom se izboljša predvsem CLS metrika, saj optimizacija pisav preprečuje pojav FOUT. Poleg tega so hitrostne metrike boljše, če uporabljamo Google fonts, saj Next.js v času prevajanja lokalno shrani pisavo, zato povezava z Google fonts strežnikom ni potrebna.

### *3.6.2 Kdaj uporabiti optimizacijo pisav*

Smo mnenja, da optimizacija pisav nima slabosti, saj ne povzroča dodatnih stroškov, temveč zgolj poenostavi vključevanje pisav, ki bi ga razvijalci v vsakem primeru morali narediti. Zato priporočamo, da se vedno uporabi.

## 3.7 Strežniške komponente

Spletne strani ponavadi vsebujejo več vizualnih komponent. Glede na obnašanje, lahko komponento ločimo na:

- Interaktivne: komponente, ki se odzivajo na uporabniške interakcije in hranijo stanje, npr. vnosno polje.
- Neinteraktivne: komponente, ki zgolj prikazujejo določeno vsebino, npr. objava na socialnem omrežju. Ne hranijo stanja in ne obravnavajo uporabniških vnosov.

V Next.js se vse komponente ponavadi izrišejo na strežniku (SSR), nato pa se hidrirajo na odjemalcu. To zahteva prenos JavaScript kode za vsako izmed komponent iz strežnika na odjemalca. Tukaj pa obstaja priložnost za optimizacijo; v kolikor je komponenta neinteraktivna, namreč hidracija ne bi bila potrebna, saj neinteraktivne komponente ne hranijo stanja in ne obravnavajo uporabniških vnosov. To je motivacija za idejo strežniških komponent.

Next.js nam omogoča, da za vsako komponento določimo, ali je strežniška (angl. server component) ali odjemalčeva (angl. client component). V kolikor je komponenta neinteraktivna, jo lahko označimo kot strežniško, kar bo pomenilo, da se JavaScript koda za komponento ne bo prenesla na odjemalca, saj ni potrebe po hidraciji. To lahko občutno pohitri nalaganje, saj večina spletnih strani vsebuje več neinteraktivnih komponent, kot interaktivnih [32,33].

### *3.7.1 Vpliv strežniških komponent na performančne metrike*

Tabela 1 prikazuje vpliv strežniških komponent na performančne metrike, v primerjavi z implementacijo, kjer so vse komponente odjemalčeve. Pričakovano je JavaScript paket manjši, prav tako pa so posledično izboljšane vse metrike, povezane s hitrostjo nalaganja.

### *3.7.2 Kdaj uporabiti strežniške komponente*

Next.js nam omogoča, da se na nivoju vsake komponente odločimo, ali bo le-ta strežniška ali odjemalčeva. Seveda je zaželjeno, da je čim več komponent strežniških.

Pomembno je poudariti, da potomci odjemalčevih komponent ne morejo biti strežniške komponente. To pomeni, da si moramo za optimalno delovanje drevo React komponent ustrezno izoblikovati tako, da so odjemalčeve komponente čim bližje listom drevesa [33]. To morebiti zahteva nekaj preurejanja kode.

## 3.8 Nalaganje na zahtevo

Glede na naravo naše spletne strani, določenih komponent morda ni treba naložiti že ob nalaganju strani. Kot primer vzamimo modalno okno, ki se prikaže le, če uporabnik klikne na gumb. JavaScript kode za takšno komponento ne bi bilo potrebno nalagati takoj, ampak bi nalaganje lahko preložili na kasnejši trenutek (npr. ko je stran dokončno naložena, ali pa šele, ko uporabnik postavi kurzor na gumb).

Nalaganje na zahtevo (angl. lazy loading) nam omogoča, da preložimo nalaganje določenih komponent [34]. V zgornjem primeru bi torej lahko preložili nalaganje kode za modalno okno, kar bi kodo za to komponento izločilo iz začetnega JavaScript paketa. S tem pohitrimo nalaganje in posledično izboljšamo performančne metrike.

### *3.8.1 Vpliv nalaganja na zahtevo na performančne metrike*

Tabela 1 prikazuje vpliv nalaganja na zahtevo na performančne metrike, v primerjavi z implementacijo, kjer se nobena komponenta ne nalaga na zahtevo. Rezultati so pričakovani: nalaganje na zahtevo izboljša metrike hitrosti in zmanjša velikost JavaScript paketa, saj se koda za komponente prenese šele po tem, ko se stran naloži.

### *3.8.2 Kdaj uporabiti nalaganje na zahtevo*

Nalaganje na zahtevo lahko občutno izboljša hitrost nalaganja strani, ampak ima nekaj slabosti:

- Režija: prenašanje dodatne kode po nalaganju strani je bolj potratno, kot če bi celoten paket naložili takoj, saj moramo po tem, ko se stran naloži, poslati zahtevke še za ostale komponente. Če je koda za komponento, ki jo nalagamo na zahtevo, zelo majhna, se to morda ne splača.
- Med nalaganjem komponente lahko pride do napake, ki jo moramo nato ustrezno obravnavati.
- Komponenta se ne naloži takoj, kar lahko vodi v počasno delovanje za uporabnike.
- SEO: komponente, ki se nalagajo na zahtevo, ne bodo del vsebine, ki jo vidijo spletni iskalniki.

Odločitev, ali uporabimo nalaganje na zahtevo ali ne, je torej odvisna od narave vsebine naše spletne strani:

- Komponente, ki morajo biti vidne takoj ob nalaganju, ali pa vsebujejo pomembno vsebino za SEO, se naj ne nalagajo na zahtevo.
- Primeri komponent, ki se lahko nalagajo na zahtevo, so modalna okna ali prikazna okna, ki se prikažejo šele nekaj časa po tem, ko se stran naloži (na primer obvestilo o piškotkih).
- Opomba: strežniških komponent (poglavje 3.7) ni potrebno nalagati na zahtevo, saj niso del JavaScript paketa, ki se pošlje na odjemalca. Gre torej zgolj za komponente na odjemalcu.

### 3.9 Povzetek: vpliv tehnik na performančne metrike

Tabela 1 povzema vpliv opisanih Next.js tehnik na performančne metrike na podlagi testov, izvedenih na različnih velikostih projektov. Vsako tehniko smo primerjali z referenčno implementacijo in zabeležili razliko v metrikah. Legenda znakov:

- =: tehnika ne vpliva na metriko
- +: metrika je večja v primerjavi z referenčno implementacijo
- -: metrika je manjša v primerjavi z referenčno implementacijo

**Tabela 1: Povzetek vpliva Next.js tehnik na performančne metrike.**

| | SSR | SSG | Loč. kode | Opt. slik | Opt. pisav | Opt. pisav | Strežniške komp. | Nal. na zahtevo |
|---|---|---|---|---|---|---|---|---|
| Primerjava z | SPA | SSR | SPA brez loč. kode | Img značka brez dimenzij | Blokiranje | Async | Odjemalčeve komp. | Pristop brez nal. na zahtevo |
| FCP (s) | - | - | - | - | - | - | - | - |
| SI (s) | - | - | - | - | - | - | - | - |
| LCP (s) | - | - | - | - | - | - | - | - |
| TBT (ms) | - | = | - | = | - | - | - | - |
| CLS | = | = | = | - (dimenzije slik) | = | - (ni FOUT) | = | = |
| Velikost JavaScript paketa (kB) | = (hidracija) | = (hidracija) | - | +(dodatna logika) | = | = | - | - |

### 3.10 Spremljanje terenskih podatkov

Next.js omogoča enostavno zbiranje terenskih podatkov performančnih metrik (slika 4) [35].

```
export function reportWebVitals(metric) {
  console.log(metric);
}
```

**Slika 4: Koda za pridobivanje terenskih podatkov performančnih metrik (web vitals) v Next.js.**

Funkcija reportWebVitals bo samodejno klicana, ko bo brskalnik izmeril metrike. Nato jih lahko pošljemo v poljubno orodje za analizo podatkov, npr. Google Analytics ali Datadog.

## 4 Zaključek

V prispevku smo predstavili praktično znanje in izkušnje na področju optimizacije učinkovitosti nalaganja spletnih strani, razvitih v ogrodju Next.js. Opisali smo metrike, ki jih Google Lighthouse izmeri in uporabi za izračun performančne ocene spletne strani. Nato smo navedli tehnike v ogrodju Next.js, ki jih v praksi uporabljamo za izboljšanje performančnik metrik, pri čemer smo vsako tehniko na kratko opisali, poročali vpliv na metrike in navedli lastno mnenje glede prednosti in slabosti uporabe tehnik.

Tehnike v ogrodju Next.js se v praksi izkažejo za zelo uporabne pri optimizaciji nalaganja spletnih strani, saj so intuitive, enostavne za uporabo in efektivne. V praksi optimizacija poteka tako, da spremljamo terenske podatke, z Lighthouse preverimo, kje lahko izboljšamo nalaganje naše spletne strani, nato pa uporabimo ustrezno tehniko za optimizacijo.

## Literatura

[1] https://web.dev/tags/case-study/, Case Study, obiskano 23. 7. 2023.

[2] https://web.dev/why-speed-matters/, Why speed matters, obiskano 23. 7. 2023.

[3] https://www.conductor.com/academy/page-speed-resources/, Steven van Vessum, Why Page Speed Matters: 10 Case Studies Show How, obiskano 23. 7. 2023.

[4] https://developer.chrome.com/docs/lighthouse/overview/, Lighthouse overview, obiskano 23. 7. 2023.

[5] https://developer.chrome.com/en/docs/lighthouse/performance/performance-scoring/, Lighthouse performance scoring, obiskano 23. 7. 2023

[6] https://github.com/GoogleChrome/lighthouse/releases, Lighthouse releases, obiskano 23. 7. 2023

[7] https://googlechrome.github.io/lighthouse/scorecalc/, Lighthouse Scoring Calculator, obiskano 23. 7. 2023

[8] https://web.dev/i18n/en/fcp/, Philip Walton, First Contentful Paint (FCP), obiskano 23. 7. 2023

[9] https://developer.mozilla.org/en-US/docs/Glossary/Speed_index, Speed index, obiskano 23. 7. 2023

[10] https://developer.chrome.com/en/docs/lighthouse/performance/speed-index/, Speed index, obiskano 23. 7. 2023

[11] https://github.com/paulirish/speedline, Speedline, obiskano 23. 7. 2023

[12] https://web.dev/i18n/en/lcp/, Philip Walton, Barry Pollard, Largest Contentful Paint (LCP), obiskano 23. 7. 2023

[13] https://developer.mozilla.org/en-US/docs/Glossary/Main_thread, Main thread, obiskano 23. 7. 2023

[14] https://web.dev/i18n/en/tbt/, Philip Walton, Total Blocking Time (TBT), obiskano 23. 7. 2023

[15] https://web.dev/i18n/en/cls/, Philip Walton, Milica Mihajlija, Cumulative Layout Shift (CLS), obiskano 23. 7. 2023

[16] https://web.dev/lab-and-field-data-differences/, Philip Walton, Why lab and field data can be different (and what to do about it, obiskano 23. 7. 2023

[17] https://chrome.google.com/webstore/detail/lighthouse/blipmdconlkpinefehnmjammfjpmpbjk, Lighthouse, obiskano 23. 7. 2023

[18] https://github.com/GoogleChrome/lighthouse#cli-options, Lighthouse, obiskano 23. 7. 2023

[19] https://github.com/GoogleChrome/lighthouse/blob/HEAD/docs/user-flows.md, User Flows in Lighthouse, obiskano 23. 7. 2023

[20] https://www.npmjs.com/package/next, Tim Neutkens, Naoyuki Kanezawa, Guillermo Rauch, Arunoda Susiripala, Tony Kovanen, Dan Zajdband, Next.js,, obiskano 23. 7. 2023

[21] https://nextjs.org/docs/pages/building-your-application/rendering/server-side-rendering, Server-side Rendering (SSR), obiskano 23. 7. 2023

[22] https://nextjs.org/learn/foundations/how-nextjs-works/rendering, What is Rendering?, obiskano 23. 7. 2023

[23] https://qwik.builder.io/docs/concepts/resumable/, Resumable vs. Hydration, obiskano 23. 7. 2023

[24] https://gist.github.com/gaearon/9d6b8eddc7f5e647a054d7b333434ef6, Next.js client-only SPA example, obiskano 23. 7. 2023

[25] https://nextjs.org/docs/pages/building-your-application/rendering/static-site-generation, Static Site Generation (SSG), obiskano 23. 7. 2023

[26] https://nextjs.org/docs/pages/building-your-application/rendering/incremental-static-regeneration, Incremental Static Regeneration (ISR), obiskano 23. 7. 2023

[27] https://nextjs.org/learn/foundations/how-nextjs-works/code-splitting, What is Code Splitting?, obiskano 23. 7. 2023

[28] https://nextjs.org/docs/pages/building-your-application/optimizing/images, Image Optimization, obiskano 23. 7. 2023

[29] https://www.adobe.com/creativecloud/file-types/image/raster/webp-file.html, WebP files, obiskano 23. 7. 2023

[30] https://css-tricks.com/how-to-load-fonts-in-a-way-that-fights-fout-and-makes-lighthouse-happy/, Adrian Bece, How to Load Fonts in a Way That Fights FOUT and Makes Lighthouse Happy, obiskano 23. 7. 2023

[31] https://nextjs.org/docs/pages/building-your-application/optimizing/fonts, Font Optimization, obiskano 23. 7. 2023

[32] https://adhithiravi.medium.com/what-are-server-components-and-client-components-in-react-18-and-next-js-13-6f869c0c66b0, Adhithi Ravichandran, Understanding Server Components in React 18 and Next.js 13, obiskano 23. 7. 2023

[33] https://nextjs.org/docs/getting-started/react-essentials, React Essentials, obiskano 23. 7. 2023

[34] https://nextjs.org/docs/app/building-your-application/optimizing/lazy-loading, Lazy Loading, obiskano 23. 7. 2023

[35] https://nextjs.org/docs/pages/building-your-application/optimizing/analytics, Analytics, obiskano 23. 7. 2023

[36] https://unlighthouse.dev/, Unlighthouse - Site-wide Google Lighthouse, obiskano 26. 7. 2023

[37] https://vercel.com/pricing, Pricing - Vercel, obiskano 26. 7. 2023

# Lažna božanstva in njih besede

Matej Šprogar

Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko,
Maribor, Slovenija
matej.sprogar@um.si

Znanost je religija in kot taka potrebuje svoje cerkve in vernike. Dejansko je v vsakdanji vojni za slednje z ostalimi cerkvami. V bitko za vernike pa je pritlehno vstopil še kapital, ki skriva svoj namen, pa vendar deluje najbolj grobo in brezobzirno. Glavno orodje delovanja vseh cerkva so besede, ki določajo človeka, človeško civilizacijo. Določajo, kaj in kako razmišljamo, določajo, kaj je prav in dobro že s tem, ko obstajajo - domače besede predstavljajo »normalno«, vse neubesedljivo pa je avtomatsko tuje in »nenormalno«. Če potem dodamo še napačno uporabo in zlorabo, hitro pridemo do trenutka, ko narobe postane prav… Vse to se seveda pozna tudi v virtualnem svetu, kjer besede vodijo razvoj (in vojne) med programskimi jeziki, platformami, s katerimi ukazujemo idealnemu sužnju – stroju. Monopol nad jeziki prinaša moč in dobiček, kar pa še ni konec zgodbe: sedaj so namreč stroji že prišli do stopnje, ko so pričeli tvoriti besede! Vse v imenu napredka; kaj sploh pomeni beseda »napredek«?

**Ključne besede:**
beseda
korporacije
programski jeziki
napredek
religija

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.16

## 1 Uvod

Vzemimo obrobno misel in jo s pomočjo pretiravanja, napihovanja in potvarjanja dejstev napihnimo do skrajnosti, potem pa se vprašajmo, kako se je to zgodilo in kam nas je pripeljalo; prihodnosti zaradi slabega stanja sedanjosti raje ne bomo prerokovali.

Znanost je religija, osnovana okoli vere v preverljivost in dokazljivost znanstvenih teorij. Znanost prevzema vernike, vpliv in sredstva tradicionalnim religijam, ki navidezno zahtevajo ravno obratno - slepo vero, vero brez dokazov. Seveda ima tudi znanost svoje dogme (znanstvene metode, aksiome, meritve, podatke), vernike (znanstvenike) in katedrale (univerze). Da je mera polna, ima celo svoje "antikriste" (npr. kreacioniste), ki jih skuša z metodo logičnega sklepanja (uporabniku prijazen nadomestek grmade) vsakodnevno spreobrniti na pravo pot. In kot vsaka religija ima tudi znanost kopico svetnikov, če ne celo "božanstev". To so postali izjemni posamezniki, ki so se dvignili nad povprečje in spremenili tok zgodovine: Platon, Newton...

Kot v stari Grčiji, kjer je vsak bog poosebljal določene človeške lastnosti, tudi ti posamezniki predstavljajo prelomnice v človekovem dojemanju stvarstva. Ugibanje so zamenjali z vedenjem in do takrat nedoumljivo je postalo samoumevno. Danes pa smo priča novemu fenomenu: med božanstva so se vrinile brezosebne korporacije. Kako je to mogoče?

## 2 Priznavanje božanstva

Najprej preizkusimo hipotezo o božanskosti recimo Einsteina. Skladno s krščanskimi nazori je za svetnika potrebno izpolniti pet pogojev: (1) ja, bil je "veren" znanstvenik, (2) ves čas je delal po znanstvenih načelih, (3) imel je navdih in (4) "čudežno" je našel opis zakonov vesolja. Znanstvena skupnost ga po smrti sicer ni razglasila za blaženega (5), je pa postal ikona in znanilec neke dobe.

Kdo torej priznava Einsteinu status božanstva? Znanstvena cerkev ne, saj uradno ne obstaja. Birokratsko gledano ni papirja, ki bi to dokazoval, se pa ta misel, občutek, nezavedno širi med ljudmi. In nekaj podobnega se (na žalost) dogaja tudi s korporativnimi božanstvi.

Vse korporacije dnevno darujejo v cerkvi kapitala - za vsakdanji profit, stabilno gospodarsko rast in nesrečo konkurence. Nič kaj drugače ni pri za nas zanimivih informacijsko osnovanih korporacijah. Čeprav se mogoče zdi, da korporacija sledi neki vsebinski niti (recimo operacijski sistem), pa dejansko promovira najbolj tisti produkt, ki ji prinaša največ dobička. In spodnaša konkurenco, ker v poslu je dovoljeno še več kot v ljubezni in vojni. Če bi bilo profitabilno, bi z veseljem poleg programov prodajala tudi krompir (ali telefone). Hkrati z ogromno zalogo kapitala pobira smetano znanstvenikov in raziskovalcev, ki pa se v okviru korporacije razosebijo.

V kapitalističnem svetu takšna korporacija veruje v denar (kriterij 1), dela po finančno utemeljenih principih (kriterij 2), ima navdih, kako povečati profit (kriterij 3) in čudežno preraste iz male firme v megakorporacijo (kriterij 4). Za razliko od človeških svetnikov ji borza še za časa obstoja (kriterij 5) dodeli status božanstva. In ker smo vsi odvisni od njenih produktov, se z vsakim nakupom/klikom priklanjamo in darujemo na oltarju novega člana družine nesmrtnih.

Zanimivo je, da lahko z manjšimi izjemami klasična cerkev lepo sobiva z znanstveno cerkvijo. Konflikt nastopi šele, če znanost poruši nasprotno resnico. Pač pa obe, klasična in znanstvena cerkev, brez težav izvajata ritual denarnih tokov...

## 3 Napredek v imenu napredka?

Tradicionalne religije naslavljajo temeljno človekovo vprašanje o smislu, o obstoju, in nudijo nekakšno (tolažečo) razlago s takšnim ali drugačnim namenom. Znanost sicer tudi skuša odgovoriti na to isto vprašanje, hkrati pa spreminja svet tako korenito in hitro, da se človek za časa življenja ni sposoben prilagoditi vsem spremembam. Na področju računalništva in informatike je to še posebej očitno.

Problem pravzaprav izhaja iz človeka, ki usmerja neskončno zapleteno zaporedje dogodkov, ki na dolgi rok pripeljejo v neko, prepogosto nepričakovano, stanje. Želje po optimizaciji in dobičku so iz "umetnosti" programiranja naredile obrt, iz obrti pa serijsko proizvodnjo, tekoči trak. Pomembna ni več kvaliteta, pomembna je reklama. Vse to je sprožil ravno znanstveno pogojen tehnološki napredek, ki je postal gonilna sila kaotičnega kapitalističnega sveta; kaotičnim sistemom pa je po definiciji nemogoče napovedati prihodnost, lahko jo zgolj doživimo.

Človekova sposobnost prilagajanja na še tako neugodno okolje izhaja iz naše inteligence. V vedno hitrejšem svetu ljudje že dolgo več nimamo dovolj časa naučiti se vseh potrebnih znanj, zato se specializiramo in drug drugemu prepuščamo določena opravila. Posledično lahko preživimo le še kot skupnost. Skupnost pa je določena z množico omejitev/zakonov, ki določajo (moralne) norme obnašanja in s tem omogočajo naše sobivanje.

Otroke učimo skozi stoletja izoblikovane vzorce socialnega obnašanja, ne moremo pa jih naučiti preživeti v neznanem digitalnem svetu prihodnosti. Mladostniki bodo torej svoje norme ustvarjali sproti, po potrebi, kar je zlovešča napoved za prihodnost človeštva. Namreč že sedaj opažamo, kako se je mnogo dobrih tehnologij že obrnilo proti človeštvu. Če je še pred kratkim internet veljal za rešitelja, ki bo povezal nepovezane in izučil neuke... se je žal pokazal tudi v nasprotni luči: razdvaja, poneumlja in nasploh zavira napredek, ki naj bi ga omogočil. Kaj se je zgodilo?

Virtualni svet ima svoje zakonitosti in lekcije, ki jih otroci ponotranjijo in ki se jačajo iz leta v leto. Dejansko prihaja do samoojačitvene zanke, ko nek pojav vzpodbuja samega sebe. Primer je recimo vedno večja apatija do nedelujočih programov, ki so vedno bolj dojemani kot nekaj "samoumevnega": "Ah, to je normalno, resetiraj, pa bo!", ali "Hej, naloži že novo verzijo, to je itak zastarelo, kar imaš" so namreč klasični komentarji na spletnih omrežjih. Finale dosežejo v stavkih kot je "Program X že pol leta ni bil posodobljen, ziher je zastarel". Kar se je pričelo kot izgovor za majhno napako, je postalo norma: programi se sesuvajo in to ni narobe, to je pričakovano. Na spletni strani npmcompare.com je to celo zaželeno, saj kot boljši šteje program z več verzijami (Total Versions [1])! Nove verzije resda prinesejo nove funkcionalnosti, ampak tudi nove napake... Trenutna poplava slabih aplikacij uči mlade, da ni nič narobe, če se aplikacija sesuje, če stvar ne deluje. Zadovoljni naj bodo, itak bo vse popravljeno v naslednji verziji (ki se bo avtomatsko naložila ob najslabšem možnem trenutku). Ko bo ta miselnost iz mobilnih platform in osebnih računalnikov pljusknila še v industrijo, bo katastrofa. Se pa že kaže, tudi z vedno nižjo kvaliteto programov vodilnih svetovnih proizvajalcev programske opreme.

Prihodnost so torej generacije programerjev, ki si morajo v mladosti same ustvariti norme digitalnega obnašanja in ki imajo hkrati na voljo enostavno preveč - predobre računalnike, predobre prevajalnike, predobra razvojna okolja, preveč ogrodij in orodij... Od obilja so razvajeni, ne rabijo več poznati bitov in bajtov in registrov in prekinitev in pač vsega, kar je procesor. Dejansko cela generacija mladih programerjev živi v vatiranem vesolju višjih programskih jezikov in pripadajočih ogrodij, ki so reklamirano vsemogočna in krasna in oh in ah. Ker je Microsoft tako rekel. Ker je Google tako rekel. Ker je Facebook tako rekel... Te mega korporacije so moderna božanstva, otroci pa verniki, ki ne znajo drugače, kot da slepo sledijo pridigani reklami.

Tipična maša v novodobni cerkvi: Tehnologija X "significantly reduces development and deployment costs and provides enhanced...". Da so programerji dejansko ovčice, ki jih pastir vodi na pašo, je razvidno tudi v tem, da si praktično vsi pastirji lahko dovolijo velike zdrse v svojih obljubah. Ki jih niti ne poskušajo opravičiti, ampak jih lahkotno odpravijo s press-releasom in besedami "deprecated", "outdated" in "obsolete". Še huje, novodobni pastirji nam zavestno in nekaznovano lažejo, demonizirajo nasprotnike, ker ko bomo čez 20 let spoznali resnico, je že itak prepozno. Pač nova realnost. Dejstvo je, da jih uporabnik, ki je zabredel že (pre)globoko, težko kaznuje s prestopom h konkurenci; pa tudi se ne splača iz dežja pod kap...

In seveda se ta miselnost odraža tudi v programski kodi. Vedno več tehničnega dolga, zanašanja na nepreverjene tehnologije in čarobne zmožnosti prevajalnikov in frameworkov ter splošen upad kvalitete programske kode. Kar je nujno, saj so programi vedno obsežnejši z vedno več odvisnostmi; pravzaprav smo srečni že, če jih sistemski update ne poruši. Minili so časi, ko so bile napake obravnavane s pozornostjo in zaskrbljenostjo. Danes sprijaznjeni z izgubo resignirano resetiramo računalnik.

Megakorporacije podpirajo takšno miselnost zaradi svojih finančnih interesov. Saj ni treba dokazovati, pa vendar. Recimo Microsoft - v času Korona krize je izkoristil svoj finančni monopol, da je vsilil škandalozno slab program MS Teams veliki količini "svojih" uporabnikov. No, vsilil je premočna beseda, recimo "lansiral". In prijazno sugeriral, da on ve več, da je on boljši... Ker to je bistvo reklame - prepričevanje neodločenih, da je alternativa slabša! Si znamo sploh predstavljati drugačno reklamo za MS Edge brskalnik od resnične? Slogan pravi: "Choose the web browser that puts you first. Microsoft Edge is the fast and secure browser that helps you protect your data and save time and money." Subtilno, če je Edge hiter in varen, kakšna je potem šele konkurenca? In kje smo že slišali frazo "<someone> first"? Nadalje, podatke moramo varovati - aha, internet je nevaren in rabimo mogočnega(!) varuha, sicer je čas za paniko!? In čas je denar - pobrenkajmo še na to struno! Koliko spretnih besed, koliko insinuacij, podtonov, koliko namigov, kako malo eksplicitnega pa kako veliko implicitnega hvalisanja... Ker resnica - "Another browser from Microsoft" - je seveda mala siva miška. Marketing zlorablja besede v najekstremnejši obliki.

## 4 Bistvo civilizacije je v besedi

Princip publish or perish je vzpodbudil odkrivanje tople vode - če bi vse (akademske) objave dejansko izpolnjevale kriterij originalnosti, bi moralo človeštvo že zdavnaj odkriti smisel življenja, vesolja in sploh vsega... Ampak ne, sedaj vsaka nova generacija objavi in patentira staro znanje pod novim imenom ali dela infinitezimalno majhne korake. Resničnih prebojev je bore malo. Še huje, staro znanje se izgublja, saj v močvirju besed ne vemo več, kaj neka beseda sploh še pomeni. In to je bistvo vsega - beseda.

Božanstva imajo moč, da ustvarijo besede. Ali celo spremenijo pomen besed drugih božanstev. In to jim uspe v takšni meri, da tega ljudje sploh ne opazimo, ampak smatramo kot "normalno" stanje stvari; smo kuhane žabe v kropu. Za primer poglejmo glagol "kupiti". Če kupim recimo računalnik, sem lastnik. Imam uradno pravico, da ga uporabljam, razstavim, spremenim, posodim, prodam, uničim. Kaj pa sem, če kupim program? Dejansko sem najemnik. Lahko ga le uporabljam in uničim, ne smem pa ga ne razstaviti, ne spremeniti, ne prodati! Zakaj torej "kupujemo" programsko opremo, zakaj ne rečemo bobu bob? Ker najemništvo izraža podrejen, suženjski položaj. Ljudje smo večji potrošniki, če imamo iluzijo svobode, kot pa če se zavedamo realnega položaja. Dejansko smo ujetniki kapitala.

Najtežja stvar pri programiranju je pravzaprav poimenovanje. Slabo ime naredi kodo nerazumljivo, to vnese napake, katerih odpravljanje je težavno... Še huje, problem poimenovanja se ne končna samo pri imenih spremenljivk, funkcij, razredov, modulov, knjižnic. Slabo poimenovanje namreč povzroči nerazumevanje, ki kulminira v iluziji o tem, kako program deluje. Slaba imena ustvarjajo iluzijo ne samo pri poslušalcu/bralcu kode, ampak celo pri avtorju. Vsaka beseda ima namreč svoj kontekst, v katerem jo človek razume. Če jo uporabljamo skladno s pričakovanji, ni presenečenj, recimo beseda objekt. Če pa uporabljamo visokoleteča imena, ki v kodi ne odražajo svojega pravega pomena, lahko v ekstremu doživimo celo psihološki efekt samozanikanja. Kor primer vzemimo trenutno popularno področje umetne inteligence. Dejansko ne razumemo principa, kako se možgani učijo in kako sklepamo; ne vemo točno, zakaj smo, kar smo. Ampak večina kvazi AI programov vsebuje rutine poimenovane learn, train... In celo programerji dejansko pričenjajo verjeti, da so sprogramirali nekaj, česar v bistvu ne razumejo! Korporacije pa potem seveda pričnejo prodajati najnovejši inteligentni sistem za avtonomno vožnjo avtomobilov ali pametne osebne pomočnike ali avtomatski diagnostični sistem ali super sposobno pametno iTehnologijo... Industrija potrebuje nova imena ali vsaj pridevnike za svoje produkte, ker kako pa naj sicer ovčke prepriča, da prodaja nekaj boljšega od konkurence!? Tako, da staro preobleče v nove obleke, preimenuje in reklamira kot zveličavno novost!

Pozoren bralec je v prejšnjem odstavku trznil ob uporabi besede "objekt" kot simbola dobro razumljene besede. In imel je prav; beseda objekt, ki je dobro znana vsem programerjem, pa je popolna neznanka za začetnike, ki ne poznajo te besede. Kot primer, kako besede zapletejo človeško življenje, poglejmo enega izmed odgovorov na omrežju Quora na vprašanje "Kaj je objekt in zakaj v Javi ustvarimo objekt?" [2]. Še pred tem naj omenim, da avtor odgovora zaseda položaj Senior Consultant - Developer pri firmi Thoughtworks in da je njegov odgovor zbral 296 "upvote-ov", 33 zadovoljnih komentarjev in bil 8 krat deljen. Torej, po njegovi interpretaciji je razred(!) kot soba z željeno vsebino (metodami), do katerih pa izven sobe nimamo dostopa. Za dostop rabimo "objekt", ki si ga lahko predstavljamo kot vrata, ki nam omogočijo dostop do vsebine sobe. Nihče drug ne more uporabljati naših vrat, če pa smo že v sobi, vrat seveda ne rabimo. Enostavno, intuitivno, narobe. "Izkušen" programer smatra objekt kot vrata v razred??? Primerjava ustreza konceptu reference, ne pa objekta. Do sem nas je torej pripeljala demonizacija kazalcev.

Vrhunec pa trenutno doživljamo v preboju tim. pametnih pogovornih botov, kot je chatGPT. Njihovo bistvo so namreč ravno besede (in verjetnosti). Nič pameti, nič razumevanja, zgolj verjetnost, številka, da bo neka beseda pravilna. In zadeva deluje dejansko zastrašujoče dobro! ChatGPT, ki se je verjetno učil tudi na odgovoru iz Quore, sicer vrne suhoparen šolski odgovor na dobesedno vprašanje "What is an object and why do we create an object in java?" Če ga še dodatno spodbudimo, da poišče preprosto analogijo s sobo in vrati, pa proizvede opisanemu odgovoru podoben odziv. Če ga nadalje opozorimo na nelogičnost, se vljudno opraviči in ponudi pravilno analogijo, kjer je razred načrt za izdelavo sobe, objekt pa soba kot taka.

## 5 Zaključek

Vse to nakazuje, da človeški um mogoče temelji na besedi in da lahko ima v procesu razmišljanja napačna beseda napačne posledice. Ker beseda povzroča in krmili nadaljnji tok misli - poplavo novih besed - je dejansko nemogoče razmišljati pravilno, če štartamo z napačnimi besedami. Človeški možgani imajo, hvala bogu, mnogo večjo stopnjo tolerance kot ChatGPT, pa vendar niso imuni na problem slabega poimenovanja. Ni čudno, da je ena poglavitnih prednosti programskih vzorcev predvsem poenotenje besedišča.

Človeški kaos, ki sicer nastane zaradi neobstoja skupnega besedišča, je izkoristil že sam Bog, da nam je preprečil gradnjo babilonskega stolpa. Če nam je on premešal jezike nekoč davno, si jih vozlamo danes kar sami. Vsak mesec nov programski jezik, vsak spet "boljši", "elegantnejši", "hitrejši"... In vsak redefinira stare besede pa seveda prinese debelejši priročnik. In seveda goreče pridigarje, ki v imenu napredka rušijo stare templje. In za vsakim novim jezikom, ogrodjem, orodjem, programčkom, platformo stoji mala cerkvica, ki upa na svoj trenutek slave, na svoj delež denarja. In ki bo pojedla svoje otroke, ko bo potrebno.

## Literatura

[1] https://npmcompare.com/, marec 2023.

[2] K. Balathandayuthapani, What is an object and why do we create an object in java? Quora, https://qr.ae/prUljq, maj 2023.

# Dekompozicija sistema za preračunavanje zavarovalno-tehničnih vrednosti

Martin Potrč, Dušan Bratuša, Vida Primožič, Nejc Maleš

Novum-RGI Germany GmbH Podružnica Maribor, Maribor, Slovenija
vida.primozic@novum-rgi.si, dusan.bratusa@novum-rgi.si,
martin.potrc@novum-rgi.si, nejc.males@novum-rgi.si

V našem podjetju smo se odločili za preobrazbo sistema za preračunavanje zavarovalno-tehničnih vrednosti, ki smo ga poskusili prestaviti iz aplikacije za informacijsko podporo življenjskih zavarovalniških poslovnih procesov (imenovano V'ger Life) v samostojno komponento. Za ta projekt smo izbirali med različnimi tehnologijami. Za implementacijo uporabniškega vmesnika smo uporabili ogrodje Quasar, ki temelji na Vue.JS, za zaledni sistem smo izbrali aplikacijski strežnik Quarkus, za implementacijo jedra sistema za preračunavanje pa smo izbrali Node.js. Komponento smo poimenovali Product Development Workbench –oziroma s kratico – PDW.

**Ključne besede:**
Quasar
Node.js
Product Development Workbench
PDW
Quarkus
Chrome
JavaScript
Aktuar

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.17

# 1 Uvod

Izračunava zavarovalno-tehničnih vrednosti iz vhodnih podatkov, pa naj si gre za izračun ponudbe zavarovanje, izračun vrednosti ob sklenitvi ali pa za izračune ob spremembah zavarovalne pogodbe, zna biti programerski izziv tako za načrtovalca aplikacije, programerja/testerja ter koordinatorja med aktuarji in konceptom rešitve. Vsi vpleteni želimo težiti k hitrejšemu lansiranju po eni strani novih zavarovalnih produktov na trg, kakor tudi implementacijo novih generacij obstoječih produktov ob rednem spreminjanju razmer na zavarovalniškem trgu.

V ta namen je potrebno izdelati sistem za preračunavanje zavarovalno-tehničnih vrednosti, ki omogoča izpolnitev omenjenih teženj.

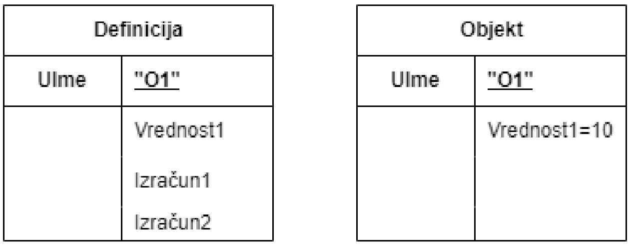
Obstoječa rešitev, ki bo opisana v naslednjih poglavjih, se je sicer izkazala za uporabno, a smo z analizo ugotovili nekatere pomanjkljivosti ter poskušali najti rešitev, ki bi aktuarju na prijaznejši način omogočala razvoj in testiranje produktov še preden preidejo v produkcijske sisteme.

# 2 Koncept implementacije v obstoječem sistemu

Na tem mestu bomo na kratko opisali osnovne principe in algoritme, katere uporabljamo v sistemu za izračunavanje vrednosti ter prikazali primer definicije vzorčnega zavarovalnega produkta.

## 2.1. Osnovni principi izračunavanja

Za osnovo si predstavljajmo definicijo, ki na podlagi vhodne vrednosti izračuna dve drugi vrednosti. Definicija in primer objekta sta povezana preko unikatnega imena oziroma notacije.

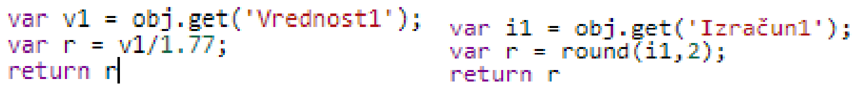


| Definicija | |
|---|---|
| UIme | "O1" |
| | Vrednost1<br>Izračun1<br>Izračun2 |

| Objekt | |
|---|---|
| UIme | "O1" |
| | Vrednost1=10 |

**Slika 1: Preprost primer definicije in konkretnega objekta.**

Recimo, da želimo definirati izračun na sledeči način: *Izračun1 = Vrednost1 / 1.77* ter *Izračun2=zaokroži(Izračun1, 2 decimalni mesti)*. Na primeru iz slike 1 bi definiciji za oba izračuna izgledali takole:



```
var v1 = obj.get('Vrednost1');
var r = v1/1.77;
return r
```

```
var i1 = obj.get('Izračun1');
var r = round(i1,2);
return r
```

**Slika 2: Izračun.**

Za izgradnjo formule za izračun so na voljo sledeči konstrukti:

- Referenca na objekt – *obj*
- Pridobitev vrednosti v objektu – *obj.get(ime)*
- Klic vgrajene funkcije s parametri – *func(p1)*
- Referenca na globalno knjižnico funkcij (npr. GKF) ter klic funkcije na tej knjižnici – *GKF.func1(p1,p2,p3)*

Objekti so lahko povezani tudi v objektne drevesne strukture s povezavo *starš* (angl. parent) in *otrok* (angl. child). S tem lahko na eni strani pridobimo objekt, ki je trenutnemu objektu starš, s funkcijo *obj.getParent()* ter seznam otrok objekta s funkcijo *obj.getChildren(childRelationName)*. Ta funkcija nam omogoča definicijo agregatnih funkcij kot so vsota, povprečje in podobno.

Dodatna in tudi zelo uporabna funkcionalnost je izpeljevanje (angl. derivation). Definiciji lahko definiramo, da je izpeljana iz neke druge definicije in s tem pridobimo možnost, da neko vrednost definiramo na predlogi. Ta vrednost je nato dosegljiva s prej omenjeno funkcijo *obj.get(imeNaPredlogi)*.

Ko nek vhodni objekt oziroma drevo objektov pošljemo skozi naš "računski stroj" nam sistem vrne novo izračunano drevo. Algoritem je zastavljen tako, da če se dostopa do neke vrednosti, ki še ni izračunana, s funkcijo *obj.get(ime)*, se ta vrednost izračuna samo enkrat, nato je dostopna vsem drugim funkcijam kot že izračunana vrednost.

Še ena od funkcionalnosti, ki je v tem članku ne bomo podrobneje opisali, je validacija vrednosti. Vsaka vrednost se lahko razširi s formulo za validacijo izračunane vrednosti. Rezultat validacije je po eni strani diskretna vrednost, ki prikazuje pravilnost, opozorilo ali napako, po drugi strani pa opis opozorila ali napake, ki se lahko doda v različnih jezikih. Najpreprostejši primer bi bil, da preverimo, če je vrednost v nekih osnovnih mejah (npr. min < x < max).

### 2.2. Primer produkta za izračun vrednosti življenjskega zavarovanja

Za primer in lažje razumevanje bomo predstavili formule za primer preprostega življenjskega zavarovanja z opcijo priključitve invalidnine za pokrivanje premije življenjskega zavarovanja v času invalidnosti.

Formule so definirane tako, da so razporejene v štiri-nivojsko drevesno strukturo:

- Skupina produktov (DL01EU2017)
    - Življenjski produkt (DL01EU2017)
        - Življenjski rizik (DL01EU2017)
            - Glavni del življenjskega rizika (DL01EU**M**2017)
            - Del živ. rizika za prilagoditev premije (DL01EU**A**2017)
            - Del živ. rizika po prenehanju plačevanja premije (DL01EU**P**2017)
            - Del živ. rizika za prilagoditev po prenehanju plačevanja premije (DL01EU**R**2017)
    - Rentni produkt za invalidnost (WL01EU2013)
        - Rizik rente za invalidnost (WL01EU2013)
            - Del rizika rente za invalidnost v dobi zavarovanja (WL01EU**R**2013)
            - Del rizika rente za invalidnost v dobi izplačevanja (WL01EU**L**2013)

Na vsakem izmed opisanih nivojev se nahajajo formule za vrednosti, kot so premije (riziko, stroški, dodatki ipd.), zavarovalne vsote, invalidnine, poročevalske vrednosti (rezervacije, premijski razrezi, amortizacije) ter še veliko drugih vrednosti, ki jih ne bomo naštevali, ker to ni relevantno za naš primer. Na vseh nivojih je možno uporabiti že izračunane vrednosti na drugih nivojih, kot na primer sešteti vse točno določene vrste premije. Primer: na najnižjem nivoju je definirana formula za riziko premijo, nakar je na višjih nivojih definirana formula za vsoto riziko premij v spodnjih nivojih, kar rezultira v riziko premiji na trenutnem nivoju, ter tako rekurzivno po nivojih navzgor.

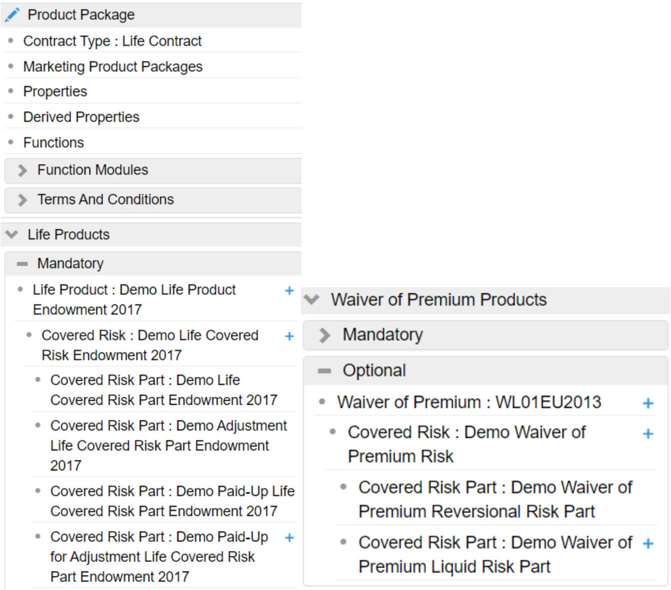
Za primere ponovne uporabe enakih formul obstajajo knjižnice funkcij, ki so v sistemu globalno dostopne po imenu in se lahko uporabijo v formulah štiri nivojskega primera s preprostim klicem na unikatno oznako knjižnice in definicijo metode za izračun (npr. *FN.round(x, factor, mode)*). Vse formule so definirane v skriptnem jeziku JavaScript.

Za primere podobnih produktov ali generacije produktov mora obstajati tudi koncept izpeljevanja. Koncept izpeljevanja spominja na razredno dedovanje, saj je možno definirati formule za vrednosti na nivoju predloge za določen nivo produkta, ki uporabljajo lastnosti konkretnega nivoja produkta. S tem se, kolikor se le da, izognemo podvajanju algoritmov.

### 2.3. Primer definicije produkta v obstoječem integriranem sistemu

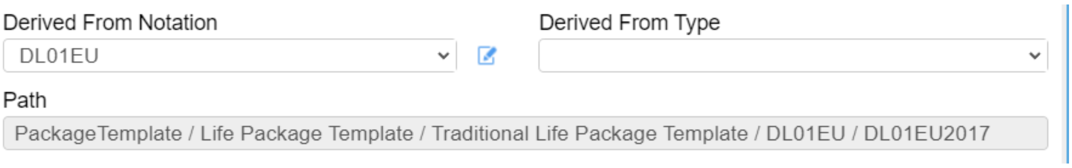
Na spodnjih slikah so prikazani deli uporabniškega vmesnika obstoječega integriranega sistema, kjer je mogoče pregledati ali spremeniti formule skupine produktov.

Prikaz štiri nivojske strukture v V'ger Life:



- Product Package
  - Contract Type : Life Contract
  - Marketing Product Packages
  - Properties
  - Derived Properties
  - Functions
  - Function Modules
  - Terms And Conditions
- Life Products
  - Mandatory
    - Life Product : Demo Life Product Endowment 2017
      - Covered Risk : Demo Life Covered Risk Endowment 2017
        - Covered Risk Part : Demo Life Covered Risk Part Endowment 2017
        - Covered Risk Part : Demo Adjustment Life Covered Risk Part Endowment 2017
        - Covered Risk Part : Demo Paid-Up Life Covered Risk Part Endowment 2017
        - Covered Risk Part : Demo Paid-Up for Adjustment Life Covered Risk Part Endowment 2017
- Waiver of Premium Products
  - Mandatory
  - Optional
    - Waiver of Premium : WL01EU2013
      - Covered Risk : Demo Waiver of Premium Risk
        - Covered Risk Part : Demo Waiver of Premium Reversional Risk Part
        - Covered Risk Part : Demo Waiver of Premium Liquid Risk Part

Slika 3: Prikaz strukture produkta v integriranem sistemu.

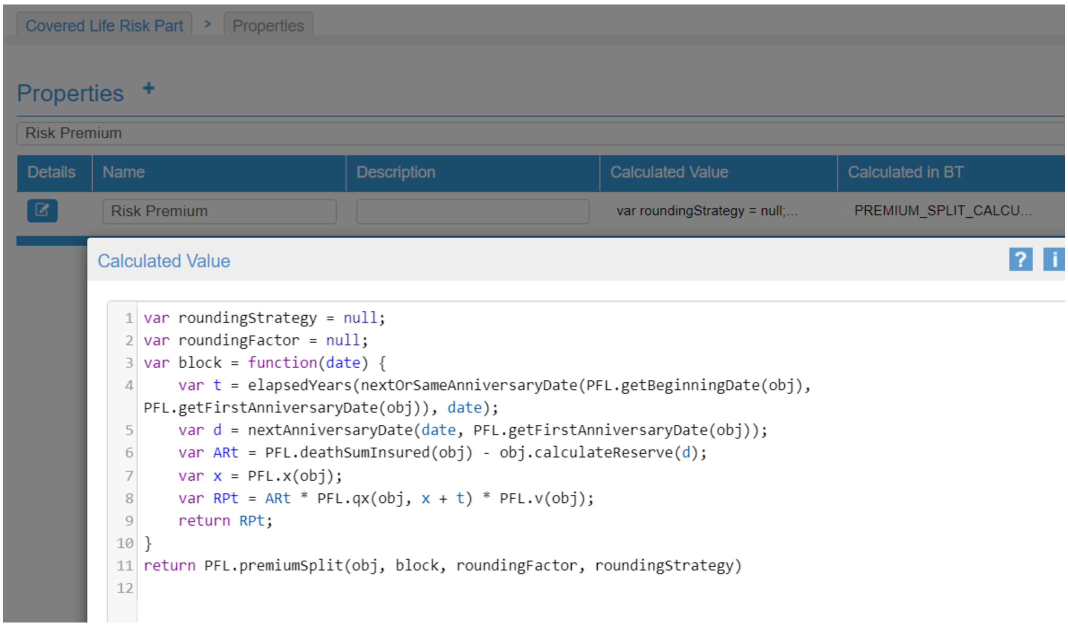


Derived From Notation: DL01EU

Derived From Type:

Path: PackageTemplate / Life Package Template / Traditional Life Package Template / DL01EU / DL01EU2017

Slika 4: Prikaz izpeljevanja iz predlog v atributu '*Path'.*



Risk Premium

| Details | Name | Description | Calculated Value | Calculated in BT |
|---|---|---|---|---|
| | Risk Premium | | var roundingStrategy = null;... | PREMIUM_SPLIT_CALCU... |

```
var roundingStrategy = null;
var roundingFactor = null;
var block = function(date) {
    var t = elapsedYears(nextOrSameAnniversaryDate(PFL.getBeginningDate(obj),
PFL.getFirstAnniversaryDate(obj)), date);
    var d = nextAnniversaryDate(date, PFL.getFirstAnniversaryDate(obj));
    var ARt = PFL.deathSumInsured(obj) - obj.calculateReserve(d);
    var x = PFL.x(obj);
    var RPt = ARt * PFL.qx(obj, x + t) * PFL.v(obj);
    return RPt;
}
return PFL.premiumSplit(obj, block, roundingFactor, roundingStrategy)

```

**Slika 5: Prikaz definiranja formule za riziko premijo**

# 3 Analiza pomanjkljivosti

V nadaljevanju naštejmo pomanjkljivosti, ki so se pokazale pri uporabi samih definicij produktov, kot na primer iskanje napak, iskanje odvisnosti med formulami, iskanje razlogov za zmanjšanje zmogljivosti, kot so poraba pomnilnika in počasnost izvajanja. Hkrati smo želeli tudi iskati možnosti, kako bi aktuarji, tako v našem podjetju, kakor tudi pri poslovnih partnerjih, lahko na enostavnejši način in v čim krajšem času definirali nove generacije produktov iz obstoječih ter, kar je po navadi še pomembneje, testirali kalkulacije v sistemih izven produkcije ter jih po pozitivnih testnih rezultatih kar se da hitro vpeljali v produkcijo. Hkrati pa bi želeli ohraniti osnovne koncepte izračunavanja vrednosti, kot so izračuni po nivojih, uporaba lokalnih (znotraj gradnikov) ter globalnih (splošnih in neodvisnih od tipa produktov) funkcij ter koncept izpeljevanja konkretnih produktov iz predlog. Le-ti so se izkazali za največjo moč in uporabnost obstoječega sistema.

## 3.1. Ugotovljene pomanjkljivosti

Analiza je pokazala sledeče pomanjkljivosti oziroma izzive, s katerimi bi se želeli soočiti in jih premostiti v boljše rešitve:

- Priprava vhodnih podatkov za testiranje s strani razvijalcev zavarovalnih produktov je preveč kompleksna za nekoga, ki ni vešč programiranja v Javi – kot na primer aktuarji zaposleni pri zavarovalnicah
- Sledenje napakam s pomočjo razhroščevanja je bilo pri večini primerov možno le z uporabo tako imenovanih "štrcljev" – izpisov v konzole oziroma dnevnike
- Produkti razviti v paralelnem testnem okolju so težko prenosljivi v produkcijo
- Za razvoj in testiranje produktov je potrebna popolna paralelna testna aplikacija
- Preglednost formul in iskanje uporabljenih formul je v obstoječem sistemu praktično nemogoča oziroma zahteva preveč dodatnega programiranja
- Preglednost zgradbe štiri nivojske strukture vključno z izpeljavami iz predlog je zelo pomanjkljiva
- Sodelovanje med Java in JavaScript povzroča zmedo v razvoju in vzdrževanju algoritmov za izračunavanje vrednosti

- Obstoječa rešitev je temelji na JavaScript izvajalnem okolju implementiranem v Javi, kar je Java ponujala do verzije 15 [1][2]. Zaradi težav z vzdrževanjem in nadgradnjami JavaScript implementacije v Javi, ki so posledica hitrega razvoja JavaScript standardov in pomanjkanja vzdrževalcev implementacije, je bila podpora za to odstranjena iz JDK-ja. S to potezo se jasno kaže velika težava obstoječe rešitve, saj je to osnova za celotno njeno delovanje.

### 3.2. Ideje za izboljšavo

Ugotovljene pomanjkljivosti so napeljale na nadaljnjo analizo možnosti izboljšav. Zastavili smo si cilje, s katerimi bi pomanjkljivosti odpravili ali pa vsaj omilili njihov vpliv na sistem, razvijalce in uporabnike.

- Ločitev razvojnega dela zavarovalnega produkta od same integrirane aplikacije
- Možnost uporabe ločenega sistema za izračunavanje preko REST vmesnikov iz integrirane aplikacije
- Možnost vpeljave v ločenem sistemu razvitega produkta v integrirano aplikacijo, če povezava preko vmesnikov ne bi bila zaželena
- Preprostejša priprava testnih vhodnih podatkov ter izvajanje testnih scenarijev – tudi za aktuarje brez večjih programerskih znanj in izkušenj
- Preprostejši način programiranja testih scenarijev za integrirano testiranje
- Možnost razhroščevanja in možnost vpogleda v vmesne vrednosti med razhroščevanjem
- Izboljšana zmogljivost – izboljšava hitrosti izračunov ter brez nepotrebnega tratenja pomnilniškega prostora ter
- Preprostejše iskanje funkcij ter programiranje z uporabo samodejnega izpolnjevanja klicev funkcij (angl. code completion)
- Prekinitev sodelovanja med Javo in JavaScript – amputacija v Javi razvitega algoritma izračunavanja ter premik tega v JavaScript

## 4 Prehod na samostojno aplikacijo za izračun vrednosti

Za namen prehoda smo preko pomanjkljivosti začeli s pregledom možnih orodij na trgu, s katerimi bi lahko pomanjkljivosti odpravili ter vzpostavili zadovoljivo arhitekturo, ki omogoča, da ohranimo vse bistvene funkcionalnosti, ki so glavne prednosti naših algoritmov.

### 4.1. Izbira tehnologij

Za ta projekt smo izbirali med različnimi tehnologijami. Za implementacijo uporabniškega vmesnika smo uporabili ogrodje Quasar [3], ki temelji na Vue.JS, za zaledni sistem smo izbrali aplikacijski strežnik Quarkus [5], za implementacijo jedra sistema za preračunavanje pa smo izbrali NodeJS [4].

Izbira teh orodij ni bila naključna. Po primerjavi orodij na trgu smo se odločili za Node.js, ker je ena izmed najbolj razširjenih rešitev za izvajanje JavaScript kode na strežniku. Quarkus smo izbrali zaradi njegove namembnosti, kajti omogoča integracijo v platformo za storitve v oblaku, kjer ponujamo tudi naš osrednji sistem. Ker je Quarkus primarno grajen za izvajanje v Kubernetes gruči, prav tako omogoča pakiranje v vsebnik (npr. Docker Image) in je primeren za hitro inštalacijo za potrebe tako testiranja kot tudi prezentacij. Quasar smo izbrali zaradi njegovih vnaprej pripravljenih gradnikov (Material Design), ki omogočajo enostavno, konsistentno in tudi hitrejšo izgradnjo uporabniških vmesnikov in spletnih aplikacij. Hkrati pa je sistem omogoča izgradnjo po uveljavljenih standardih in dobrih praksah dizajna uporabniških vmesnikov.

### 4.2. Arhitektura

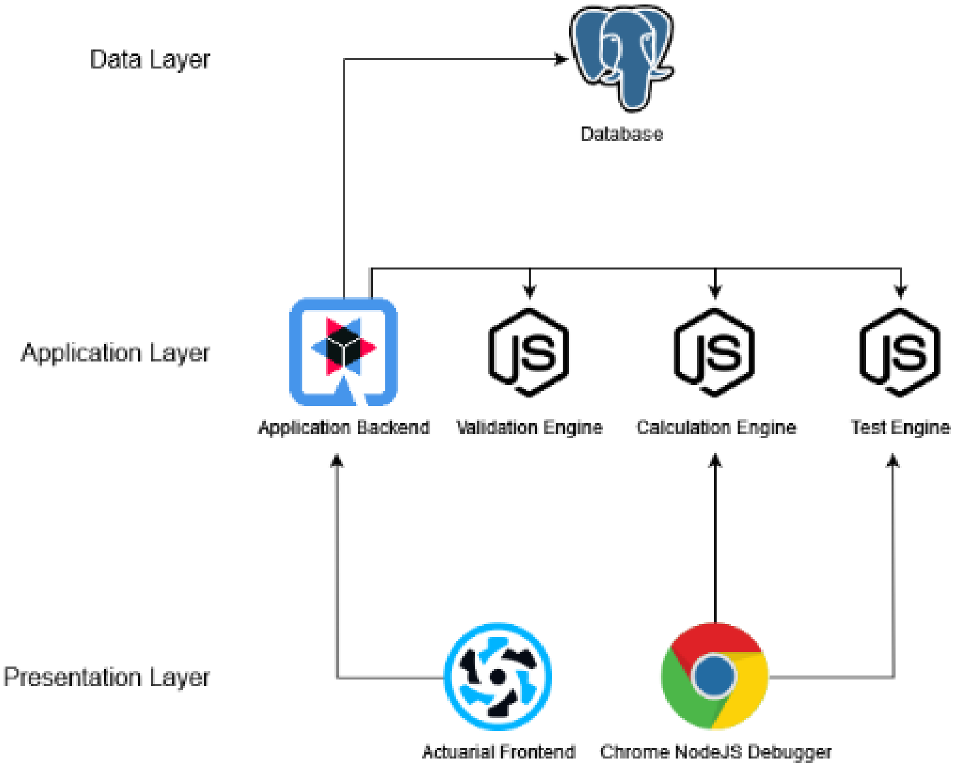
Sistem je tehnično zasnovan na klasični tri nivojski arhitekturi: podatkovni-, aplikacijski- in prezentacijski nivo.

Podatkovna baza je v komunikaciji zgolj z zalednim sistemom za razvoj zavarovalno-tehničnih vrednosti in shranjuje poslovne definicije s tehničnimi formulami. Definicije, sestavljene iz formul v JavaScript jeziku (vnesenih preko spletnega vmesnika), se pretvorijo v skupek JavaScript datotek, katere so posredovane kalkulatorju (angl. Calculation Engine). Le-te je odgovoren za samo izvajanje vseh formul v pravilnem zaporedju, glede na odvisnosti vnosov in rezultatov.

Vsaka posamezno vnesena formula je najprej preverjena preko preverjevalnika (angl. Validation Engine), ki skrbi za skladnost z vnaprej definiranimi pravili skladnosti kodiranja. Rezultat preverjanja je več nivojski in vsebuje lažje in težje kršitve pravil, npr. lažja kršitev je uporaba ne-striktne enakovrednosti ("=="), katera na nivoju JavaScript jezika ne zagotavlja popolne enakovrednosti med dvema primerjanima objektoma, medtem ko je težja kršitev manjkanje rezultata formule. Vse kršitve pravil pisanja formul so sporočene uporabniku preko vmesnika, vendar v kolikor so v formuli zaznane težje kršitve, le-ta ni sprejeta v zbirko vseh definicij.

Uporabniku je na voljo tudi ogrodje za testiranje (angl. Test Engine), ki omogoča pisanje testov posameznih formul in z njimi povezanih definicij. Vse vnesene formule lahko uporabnik preveri s testiranjem pričakovanih rezultatov glede na podane vnosne parametre. Vsak posamezen test ali skupina testov se izvede z izračunom preko kalkulatorja in preverbo rezultatov. Sam rezultat je zatem prikazan preko vmesnika uporabniku, z namenom da se čimprej najde mesto kjer je test padel.

Poleg testov lahko uporabnik, ki je vešč v kodiranju v JavaScript, uporabi tudi razhroščevalnik izvajanja tako testov kot tudi samih formul preko razvojnih orodij ponujenih s strani Chrome (Chrome DevTools [6]) brskalnika namenjenih specifično za Node.js okolje. Ta arhitekturna zasnova je prikazana na sliki 6.



Slika 6: Arhitekturna zasnova.

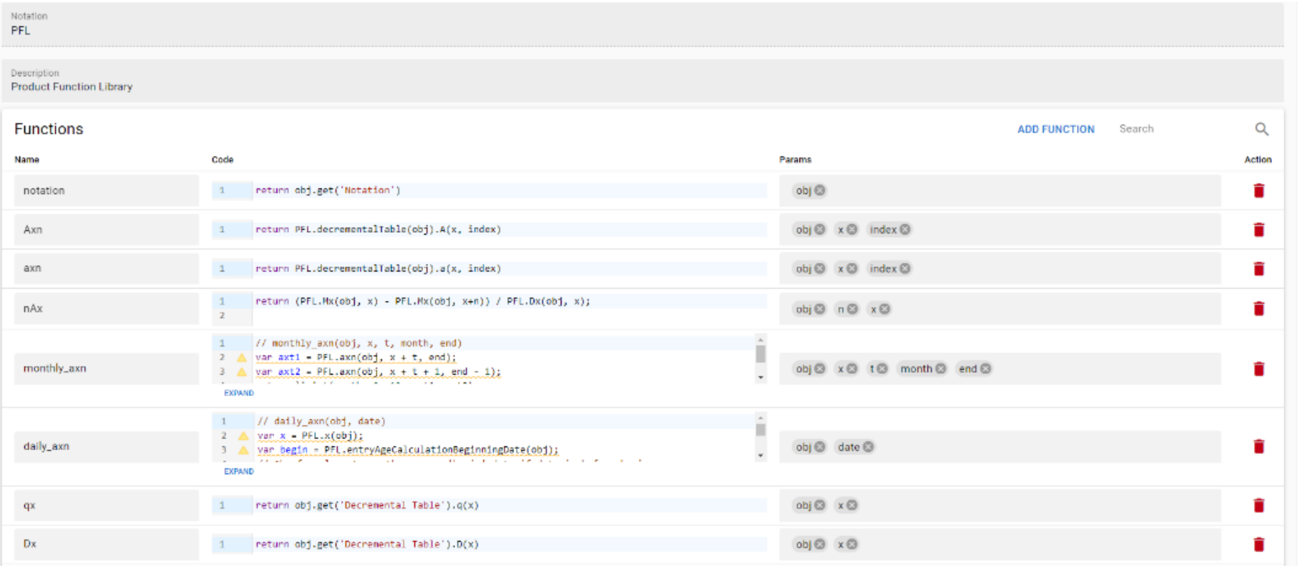
Iz pogleda produkcijskega izvajanja izračunov prezentacijski nivo, razhroščevalnik, testirano ogrodje in preverjevalnik, niso relevantni. Zato je sistem zasnovan na način, da vsak posamezen del lahko komunicira z drugimi preko REST API vmesnika in tako ponudi možnost integracije individualnih delov v druge sisteme. Tako se v produkcijskih okoljih, kjer razvijalec zavarovalnih vrednosti ni več prisoten, odstrani nepotrebne sisteme in integrira samo kalkulator. S tem je produkcijsko okolje optimizirano za izvajanje izračunu vrednosti.

# 5 Primeri uporabe sistema Product Development Workbench – PDW

V nadaljevanju bomo prikazali nekaj primerov uporabe samostojnega sistema, ki smo ga do neke zadovoljive mere implementirali. Z implementacijo smo uspeli priti tako daleč, da smo sposobni definirati zavarovalne produkte, ki jih je mogoče testirati.

## 5.1. Prikaz knjižnice funkcij ter definicij gradnikov v PDW

Tako v PDW, kot prej v V'ger Life, so formule definirane v jeziku JavaScript. V formulah se uporabljajo funkcije, ki so definirane v knjižnicah in splošne funkcije, ki so definirane v programu samem.
Na primer: knjižnica Product Function Library (PFL) ima definirano formulo *qx()*, ki vrne verjetnost umrljivosti iz tablice umrljivosti za glede na starost osebe. Knjižnica je prikazana na sliki 7.



Notation
PFL

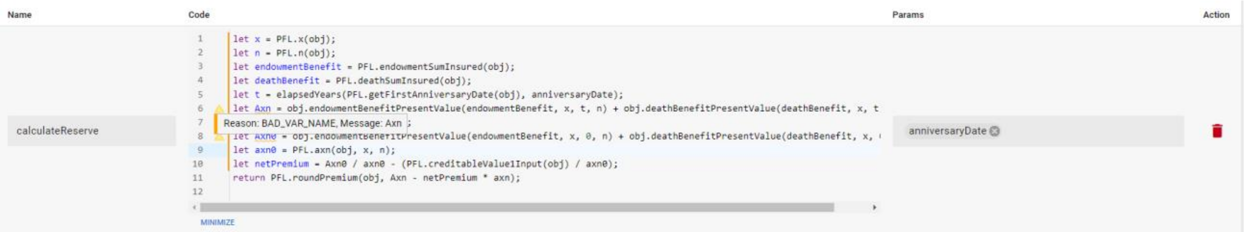
Description
Product Function Library

Functions

| Name | Code | Params | Action |
|---|---|---|---|
| notation | `return obj.get('Notation')` | obj | 🗑 |
| Axn | `return PFL.decrementalTable(obj).A(x, index)` | obj<br>x<br>index | 🗑 |
| axn | `return PFL.decrementalTable(obj).a(x, index)` | obj<br>x<br>index | 🗑 |
| nAx | `return (PFL.Mx(obj, x) - PFL.Mx(obj, x+n)) / PFL.Dx(obj, x);` | obj<br>n<br>x | 🗑 |
| monthly_axn | `// monthly_axn(obj, x, t, month, end)`<br>`var axt1 = PFL.axn(obj, x + t, end);`<br>`var axt2 = PFL.axn(obj, x + t + 1, end - 1);` | obj<br>x<br>t<br>month<br>end | 🗑 |
| daily_axn | `// daily_axn(obj, date)`<br>`var x = PFL.x(obj);`<br>`var begin = PFL.entryAgeCalculationBeginningDate(obj);` | obj<br>date | 🗑 |
| qx | `return obj.get('Decremental Table').q(x)` | obj<br>x | 🗑 |
| Dx | `return obj.get('Decremental Table').D(x)` | obj<br>x | 🗑 |

**Slika 7: Prikaz knjižnice funkcij vključno s funkcijo za umrljivost q(x).**

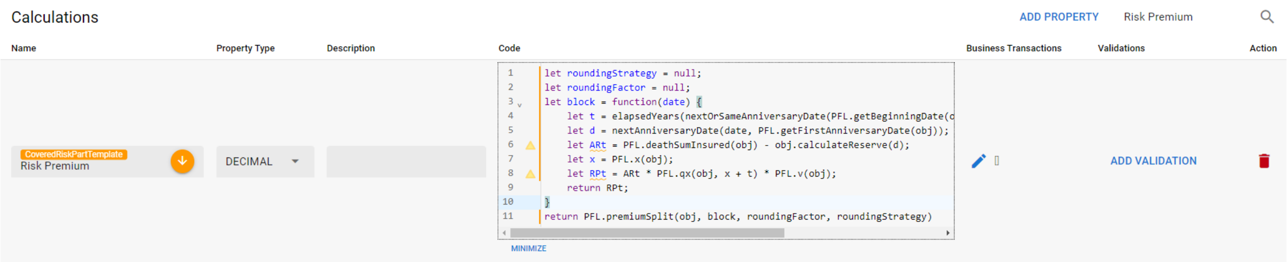
Na glavnem delu rizika DL01EUM je v formuli za riziko premijo uporabljen klic funkcije q(x). Prav tako je uporabljena splošna funkcija *elapsedYears(datefrom, dateUntil)*, ki vrne število let med dvema datumoma. Tretja možnost definiranja funkcij je definiranje le-teh na delu rizika (npr. *calculateReserve(date)* na "Traditional Life Covered Risk Part Template"), ki vrne vrednost rezervacije na določen datum. Na sliki 8 je vidno tudi opozorilo o slabi izbiri imena spremenljivke *Axn*.



```
let x = PFL.x(obj);
let n = PFL.n(obj);
let endowmentBenefit = PFL.endowmentSumInsured(obj);
let deathBenefit = PFL.deathSumInsured(obj);
let t = elapsedYears(PFL.getFirstAnniversaryDate(obj), anniversaryDate);
let Axn = obj.endowmentBenefitPresentValue(endowmentBenefit, x, t, n) + obj.deathBenefitPresentValue(deathBenefit, x, t
 Reason: BAD_VAR_NAME, Message: Axn ;
let Axn0 = obj.endowmentBenefitPresentValue(endowmentBenefit, x, 0, n) + obj.deathBenefitPresentValue(deathBenefit, x, 
let axn0 = PFL.axn(obj, x, n);
let netPremium = Axn0 / axn0 - (PFL.creditableValue1Input(obj) / axn0);
return PFL.roundPremium(obj, Axn - netPremium * axn);
```

**Slika 8: Prikaz formule za izračun rezervacije.**

Prikaz uporabe vseh treh tipov funkcij v formuli "Risk Premium" je na sliki 9.

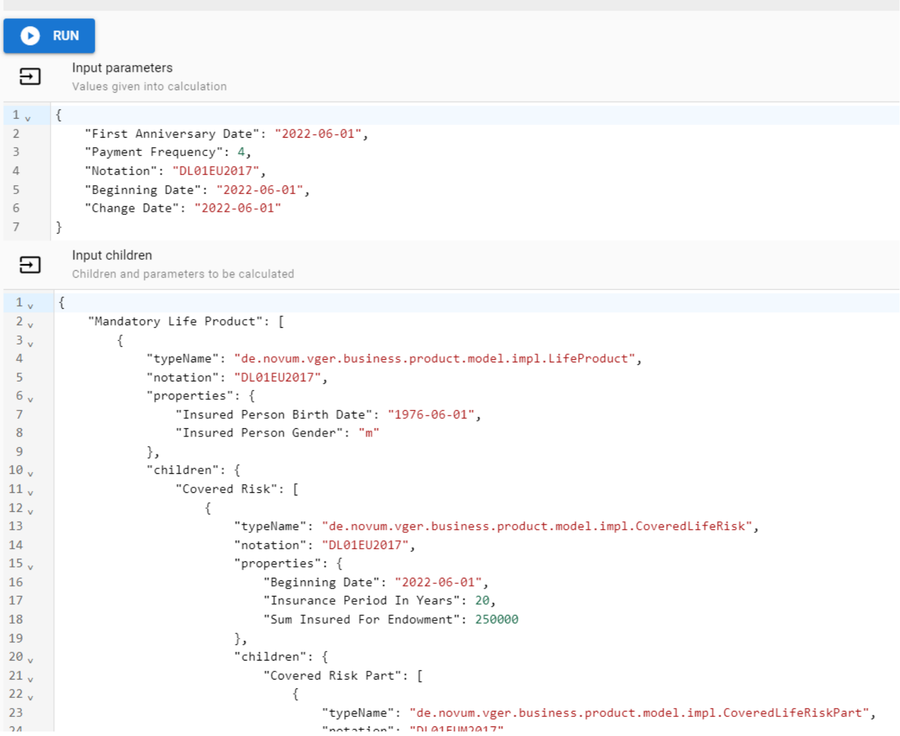


```
let roundingStrategy = null;
let roundingFactor = null;
let block = function(date) {
    let t = elapsedYears(nextOrSameAnniversaryDate(PFL.getBeginningDate(
    let d = nextAnniversaryDate(date, PFL.getFirstAnniversaryDate(obj));
    let ARt = PFL.deathSumInsured(obj) - obj.calculateReserve(d);
    let x = PFL.x(obj);
    let RPt = ARt * PFL.qx(obj, x + t) * PFL.v(obj);
    return RPt;
}
return PFL.premiumSplit(obj, block, roundingFactor, roundingStrategy)
```

Slika 9: Prikaz formule za riziko premijo.

## 5.2. Testiranje produktnih definicij v PDW

Prednost definiranja produktov v PDW je sprotno testiranje pravilnosti izračunov in sledenje morebitnim napakam. Primer testiranja skupine produktov DL01EU2017 z definiranjem štiri nivojske strukture:



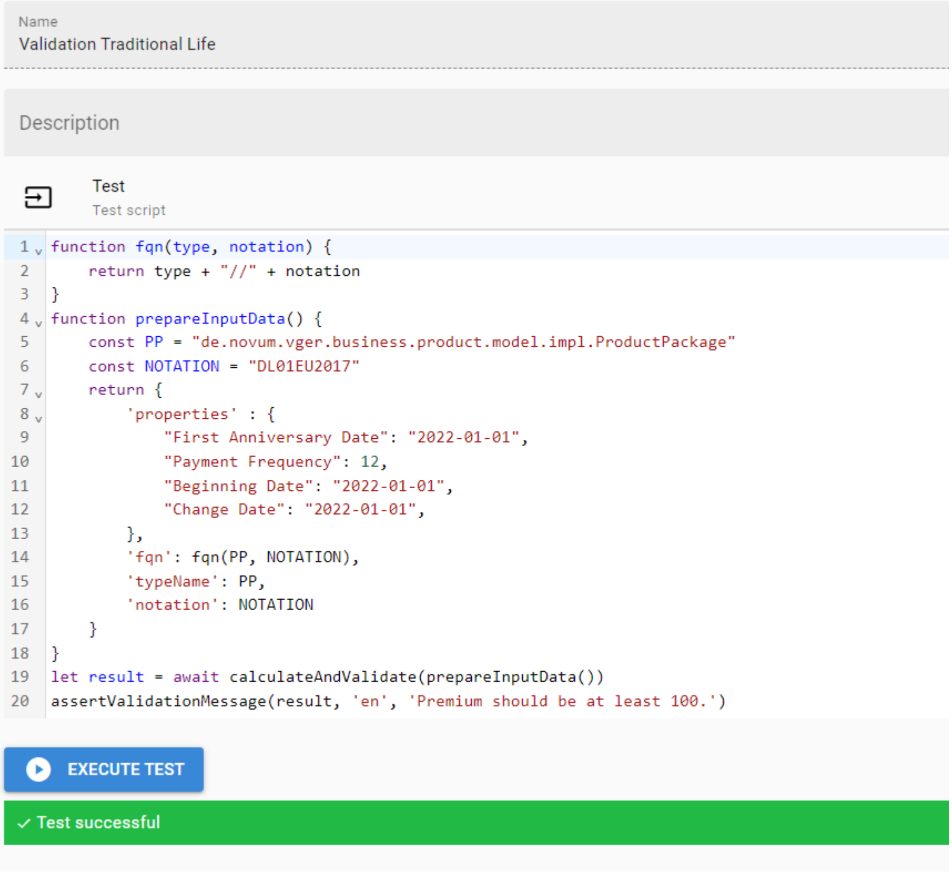
```
{
    "First Anniversary Date": "2022-06-01",
    "Payment Frequency": 4,
    "Notation": "DL01EU2017",
    "Beginning Date": "2022-06-01",
    "Change Date": "2022-06-01"
}
```

```
{
    "Mandatory Life Product": [
        {
            "typeName": "de.novum.vger.business.product.model.impl.LifeProduct",
            "notation": "DL01EU2017",
            "properties": {
                "Insured Person Birth Date": "1976-06-01",
                "Insured Person Gender": "m"
            },
            "children": {
                "Covered Risk": [
                    {
                        "typeName": "de.novum.vger.business.product.model.impl.CoveredLifeRisk",
                        "notation": "DL01EU2017",
                        "properties": {
                            "Beginning Date": "2022-06-01",
                            "Insurance Period In Years": 20,
                            "Sum Insured For Endowment": 250000
                        },
                        "children": {
                            "Covered Risk Part": [
                                {
                                    "typeName": "de.novum.vger.business.product.model.impl.CoveredLifeRiskPart",
```

Slika 10: Prikaz definiranja testa za skupino produktov.

## 5.3. Testiranje in validacija produktnih definicij v PDW

Robni pogoji produktnih definicij so definirani z pravili, ki preverjajo, ali so parametri posameznega produkta ustrezni. Primer takega pravila je, da preverimo, ali je premija zavarovanja nad določeno minimalno vrednostjo in to preverimo s testom. Na slikah 11 in 12 je prikaz uspešnega in neuspešnega testa validacije.



Name
Validation Traditional Life

```
function fqn(type, notation) {
    return type + "//" + notation
}
function prepareInputData() {
    const PP = "de.novum.vger.business.product.model.impl.ProductPackage"
    const NOTATION = "DL01EU2017"
    return {
        'properties' : {
            "First Anniversary Date": "2022-01-01",
            "Payment Frequency": 12,
            "Beginning Date": "2022-01-01",
            "Change Date": "2022-01-01",
        },
        'fqn': fqn(PP, NOTATION),
        'typeName': PP,
        'notation': NOTATION
    }
}
let result = await calculateAndValidate(prepareInputData())
assertValidationMessage(result, 'en', 'Premium should be at least 100.')
```

✓ Test successful

Slika 11: Prikaz uspešne validacije produkta.

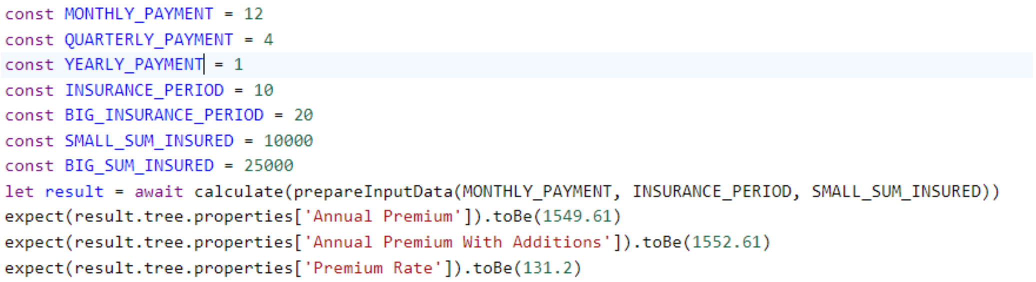
```
function fqn(type, notation) {
    return type + "//" + notation
}
function prepareInputData() {
    const PP = "de.novum.vger.business.product.model.impl.ProductPackage"
    const NOTATION = "DL01EU2017"
    return {
        'properties' : {
            "First Anniversary Date": "2022-01-01",
            "Payment Frequency": 12,
            "Beginning Date": "2022-01-01",
            "Change Date": "2022-01-01",
        },
        'fqn': fqn(PP, NOTATION),
        'typeName': PP,
        'notation': NOTATION
    }
}
let result = await calculateAndValidate(prepareInputData())
assertValidationMessage(result, 'en', 'Premium should not be at least 100.')
```

× Test failed

```
[
    {
        "line": 24,
        "matcherName": "toContain",
        "message": "Expected [ Object({ messages: [ [ 'en', 'Premium should be at least 100.' ] ] }) ] to contain Object({ messages: [ [ 'en', 'Premium should not be at least 100.' ] ] })."
    }
]
```

Slika 12: Prikaz neuspešne validacije produkta.

V PDW lahko poleg osnovni testov testiramo tudi izračune produkta ob različnih vrednostih parametrov. Na primer izračun premije za različne starosti zavarovancev, različne dobe zavarovanja ali različne zavarovalne vsote. Ta način testiranja je namenjen za ugotavljanje obnašanja formul v robnih pogojih. S takimi testi zagotavljamo pravilnost kalkulacij pri mejnih vrednostih, preverjamo robustnost formul na neznane ali neobstoječe vrednosti parametrov, kakor tudi časovno in prostorsko zmogljivost neke definicije produkta.



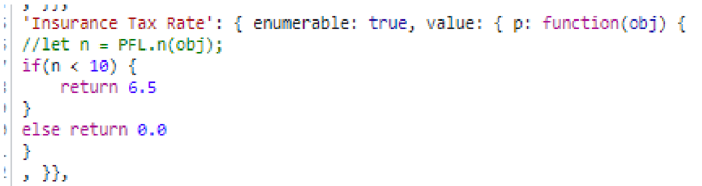
```
const MONTHLY_PAYMENT = 12
const QUARTERLY_PAYMENT = 4
const YEARLY_PAYMENT = 1
const INSURANCE_PERIOD = 10
const BIG_INSURANCE_PERIOD = 20
const SMALL_SUM_INSURED = 10000
const BIG_SUM_INSURED = 25000
let result = await calculate(prepareInputData(MONTHLY_PAYMENT, INSURANCE_PERIOD, SMALL_SUM_INSURED))
expect(result.tree.properties['Annual Premium']).toBe(1549.61)
expect(result.tree.properties['Annual Premium With Additions']).toBe(1552.61)
expect(result.tree.properties['Premium Rate']).toBe(131.2)
```

Slika 13: Prikaz testiranja s pričakovanimi vrednostmi.

## 5.4. Razhroščevanje z razvojnimi orodji Chrome (DevTools)

Dodatna pridobitev glede na integriran V'ger Life sistem je možnost razhroščevanja (angl. debugging) s pomočjo orodij, ki jih ponuja Chrome [6]. PDW omogoča povezavo na Chrome Node.js Debugger preko agenta, podobno, kot smo tega navajeni pri razvojnih okoljih za razvoj Java aplikacij.

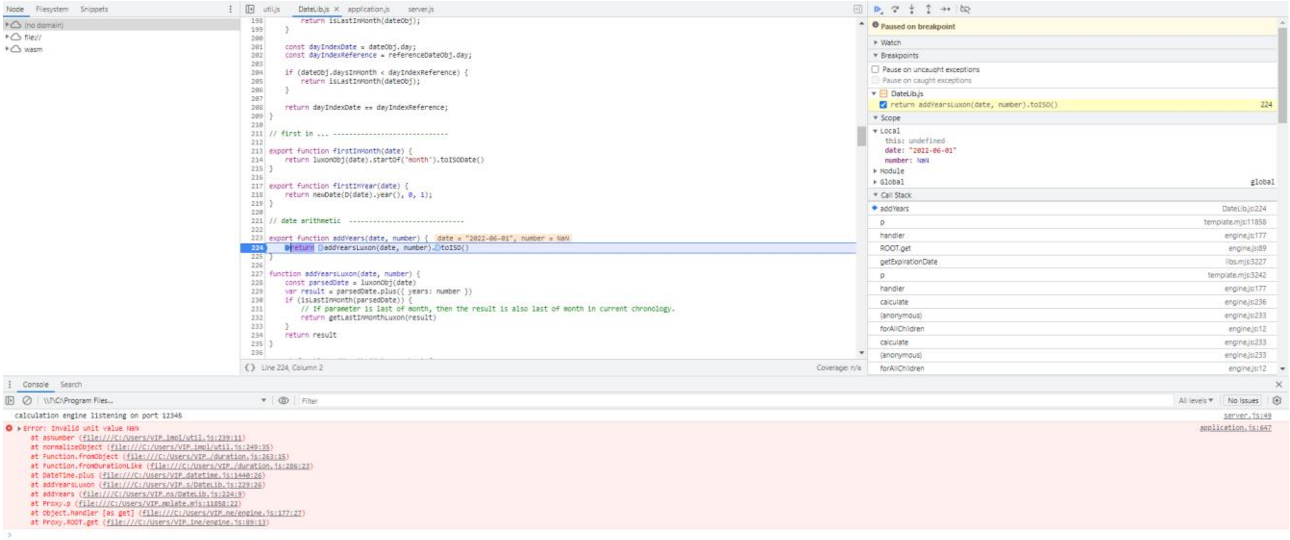
V definiciji formule za "Insurance Tax Rate" smo uporabili spremenljivko "n", ki je nismo definirali, kar lahko vidimo na sliki 14.



```
'Insurance Tax Rate': { enumerable: true, value: { p: function(obj) {
//let n = PFL.n(obj);
if(n < 10) {
    return 6.5
}
else return 0.0
}
, }},
```

Slika 14: Namerna povzročitev napake - neobstoj spremenljivke *n*.

Pri zagonu testa lahko program na določenem mestu ustavimo in preverimo vrednosti spremenljivk sredi programa, kar je vidno na sliki 15.



```
        return isLastInMonth(dateObj);
    }

    const dayIndexDate = dateObj.day;
    const dayIndexReference = referenceDateObj.day;

    if (dateObj.daysInMonth < dayIndexReference) {
        return isLastInMonth(dateObj);
    }

    return dayIndexDate == dayIndexReference;
}

// first in ... ----------------------------

export function firstInMonth(date) {
    return luxonObj(date).startOf('month').toISODate()
}

export function firstInYear(date) {
    return newDate(D(date).year(), 0, 1);
}

// date arithmetic  ----------------------------

export function addYears(date, number) {  date = "2022-06-01", number = NaN
    return addYearsLuxon(date, number).toISO()
}

function addYearsLuxon(date, number) {
    const parsedDate = luxonObj(date)
    var result = parsedDate.plus({ years: number })
    if (isLastInMonth(parsedDate)) {
        // If parameter is last of month, then the result is also last of month in current chronology.
        return getLastInMonthLuxon(result)
    }
    return result
}
```

☑ return addYearsLuxon(date, number).toISO() 224

```
this: undefined
date: "2022-06-01"
number: NaN
Module
Global                     global
```

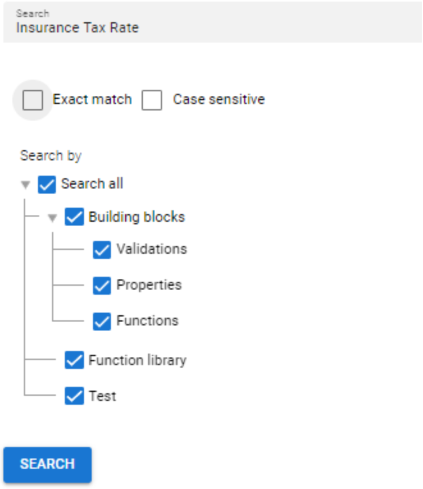
```
addYears              DateLib.js:224
p                     template.mjs:11858
handler               engine.js:177
ROOT.get              engine.js:89
getExpirationDate     libs.mjs:3227
p                     template.mjs:3242
handler               engine.js:177
calculate             engine.js:236
(anonymous)           engine.js:233
forAllChildren        engine.js:12
calculate             engine.js:233
(anonymous)           engine.js:233
forAllChildren        engine.js:12
```

```
calculation engine listening on port 12345                                    server.js:49
▸ Error: Invalid unit value NaN                                               application.js:647
    at asNumber (file:///C:/Users/VIP…impl/util.js:239:11)
    at normalizeObject (file:///C:/Users/VIP…impl/util.js:249:35)
    at Function.fromObject (file:///C:/Users/VIP…/duration.js:263:15)
    at Function.fromDurationLike (file:///C:/Users/VIP…/duration.js:286:23)
    at DateTime.plus (file:///C:/Users/VIP…datetime.js:1440:26)
    at addYearsLuxon (file:///C:/Users/VIP…s/DateLib.js:229:26)
    at addYears (file:///C:/Users/VIP…ns/DateLib.js:224:9)
    at Proxy.p (file:///C:/Users/VIP…mplate.mjs:11858:22)
    at Object.handler [as get] (file:///C:/Users/VIP…ne/engine.js:177:27)
    at Proxy.ROOT.get (file:///C:/Users/VIP…ine/engine.js:89:13)
```

Slika 15: Prikaz napake v chrome:inspect.

### 5.5. Preglednost formul in iskanje po formulah

Z razhroščevalnikom smo torej ugotovili, da imamo napako v formuli "Insurance Tax Rate". Z globalnim iskanjem lahko iščemo definicijo te formule po vseh produktih, knjižnicah in testih.



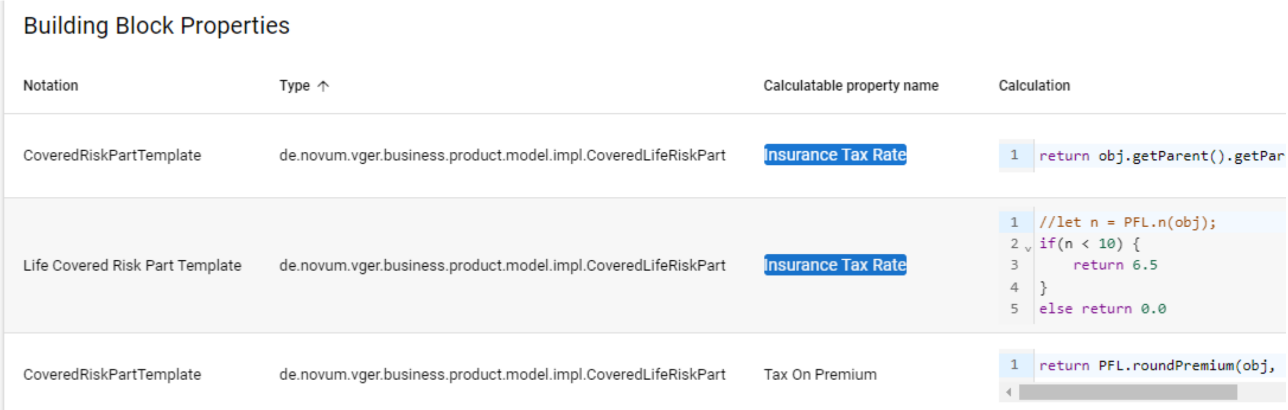
Search: Insurance Tax Rate

☐ Exact match ☐ Case sensitive

Search by
- ☑ Search all
  - ☑ Building blocks
    - ☑ Validations
    - ☑ Properties
    - ☑ Functions
  - ☑ Function library
  - ☑ Test

**Slika 16: Prikaz globalnega iskanja.**

Iz rezultata iskanja (slika 17) lahko vidimo, da se formula z napako nahaja na "Life Covered Risk Part Template"



Building Block Properties

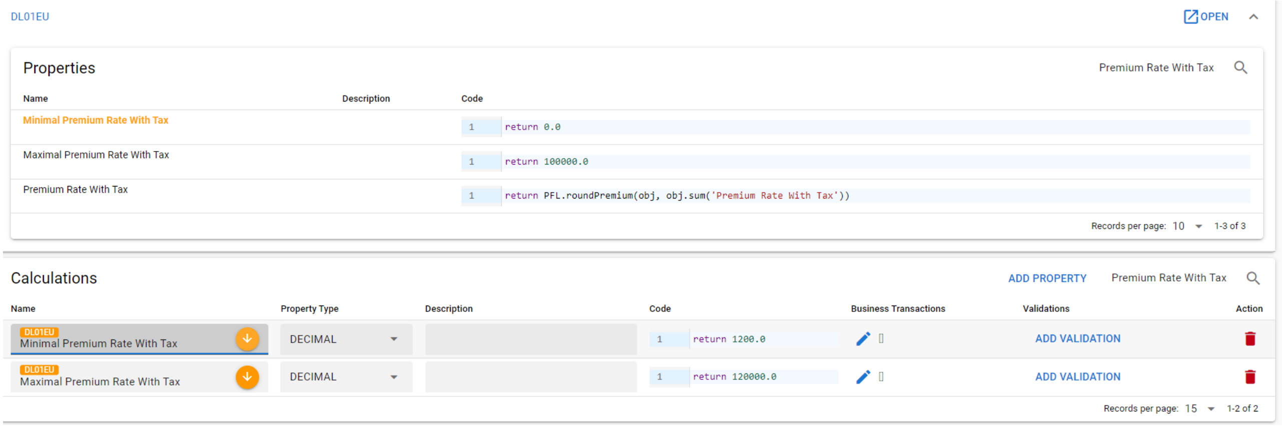
| Notation | Type ↑ | Calculatable property name | Calculation |
|---|---|---|---|
| CoveredRiskPartTemplate | de.novum.vger.business.product.model.impl.CoveredLifeRiskPart | Insurance Tax Rate | `return obj.getParent().getPar` |
| Life Covered Risk Part Template | de.novum.vger.business.product.model.impl.CoveredLifeRiskPart | Insurance Tax Rate | |
| CoveredRiskPartTemplate | de.novum.vger.business.product.model.impl.CoveredLifeRiskPart | Tax On Premium | `return PFL.roundPremium(obj,` |

Life Covered Risk Part Template

```
//let n = PFL.n(obj);
if(n < 10) {
    return 6.5
}
else return 0.0
```

**Slika 17: Prikaz rezultatov globalnega iskanja.**

Kadar so formule definirane na različnih predlogah in se tako definicije "prepišejo", lahko v PDW to vidimo tako, da poiščemo definicijo formule na različnih predlogah. Primer: iščemo formule za "Premium Rate With Tax" v PDW. Na sliki 18 je "Premium Rate With Tax" definiran na predlogi DL01EU in ni prepisan na DL01EU2017, medtem ko sta "Minimal Premium Rate With Tax" in "Maximal Premium Rate With Tax" definirani tako na predlogi DL01EU kot tudi na DL01EU2017, kar lahko v PDW vidimo na ekranu hkrati.



Properties

| Name | Description | Code |
|---|---|---|
| Minimal Premium Rate With Tax | | `return 0.0` |
| Maximal Premium Rate With Tax | | `return 100000.0` |
| Premium Rate With Tax | | `return PFL.roundPremium(obj, obj.sum('Premium Rate With Tax'))` |

Calculations

| Name | Property Type | Description | Code | Business Transactions | Validations | Action |
|---|---|---|---|---|---|---|
| DL01EU<br>Minimal Premium Rate With Tax | DECIMAL | | `return 1200.0` | | ADD VALIDATION | |
| DL01EU<br>Maximal Premium Rate With Tax | DECIMAL | | `return 120000.0` | | ADD VALIDATION | |

**Slika 18: Pregled formul.**

## 6 Zaključek

Ugotavljamo, da smo nekatere cilje prehoda na samostojen sistem izračunavanja dosegli. Orodje smo tudi testirali tako, da smo vključili naše aktuarje v izgradnjo definicij naših zavarovalniških produktov, ki jih uporabljamo pri prezentacijah našega demo sistema. Izkazalo se je, da je mogoče iz stališča aktuarja s sistemom delati bolje kot je bilo to možno v integriranem sistemu.

Glede na to, da je PDW še vedno v fazi razvoja, pa je potrebno vložiti še nekaj truda v sam izgled orodja, kakor tudi dodati še nekatere funkcije, ki bi pripomogle k lažji in hitrejši implementaciji zavarovalniških produktov.

Manjkajoče funkcionalnosti, ki so že v fazi razvoja ali jih imamo namen v bližnji prihodnosti še razviti

- Samodejno izpolnjevanje programske kode (ang. code completion)
- Analiza pokritosti formul s testi (angl. code coverage)
- Moderna dokumentacija z vključenimi primeri uporabe ter dobre prakse
- Uvoz in izvoz definiranih gradnikov produktov ter knjižnic funkcij
- Možnost razširitve kalkulacijskega sistema z vtičniki (angl. plugins)
- Razvoj oziroma nadgradnja naših obstoječih rešitev z integracijo v PDW

## Literatura

[1] https://openjdk.org/jeps/335: JEP 335: Deprecate the Nashorn JavaScript Engine, obiskano 6. 6. 2023

[2] https://openjdk.org/jeps/372: JEP 372: Remove the Nashorn JavaScript Engine, obiskano 6. 6. 2023

[3] https://quasar.dev Quasar: The enterprise-ready cross-platform VueJs framework, obiskano 8. 6. 2023

[4] https://nodejs.org/en/docs Node.js, obiskano 4. 6. 2023

[5] https://quarkus.io/ Quarkus, obiskano 1. 6. 2023

[6] https://developer.chrome.com/docs/devtools/ Chrome DevTools, obiskano 10. 6. 2023

# IIoT v oblaku za optimizacijo delovanja strojev brez večjih začetnih investicij

Marjan Kaligaro, Jure Habjan

Metronik d.o.o., Ljubljana, Slovenija
marjan.kaligaro@metronik.si, jure.habjan@metronik.si

V vse bolj tehnološko zahtevnem in konkurenčnem poslovnem okolju številna industrijska podjetja iščejo skrite rezerve za izboljšanje učinkovitosti poslovanja. Eno od takšnih področij je zagotavljanje visoke razpoložljivosti strojev, saj rezultati kažejo, da lahko s tem produktivnost povečamo tudi do nekaj deset odstotkov. Stroji so pri svojem obratovanju izpostavljeni stalnim obremenitvam, obrabi in staranju, kar vpliva na njihovo delovanje in povzroča okvare. Redno vzdrževanje in odzivanje na okvare sicer rešita del težav, a še zdaleč ne vseh. Za doseganje bistvenih izboljšav v razpoložljivosti, ki po eni strani ustvarjajo večjo dodano vrednost strojev za industrijska podjetja, po drugi strani pa prinašajo več finančnih in nefinančnih pozitivnih učinkov za proizvajalce strojev, je potrebno sodelovanje obeh strani (tako uporabnika stroja kot njegovega proizvajalca). Z vzpostavitvijo ustrezne platforme za pridobivanje, hrambo ter obdelavo operativnih podatkov lahko zagotavljamo omenjeno visoko razpoložljivost ter pravočasen odziv na težave.

**Ključne besede:**
IIoT, Industrija 4.0
edge naprave
enovit dostop do procesnih podatkov
legacy sistemi in podatki
digitalizacija

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.18

# 1 Uvod

V praksi se je prva pojavila tehnologija IoT, za njo pa je sledila še tehnologija IIoT. Pomembno je poudariti, da v nasprotju s precej razširjenim mnenjem pri omenjenih tehnologijah ne gre samo za senzorje, temveč za celoten sistem, od zajema podatkov v geografsko razpršenih sistemih do njihovega centralnega shranjevanja in pretvorbe v uporabne informacije za različne namene.

IoT, kar prevajamo kot internet stvari, srečujemo na različnih področjih in predvsem za široko uporabo. Najbolj znane so na primer aplikacije za pametna mesta, v prometu, pri oskrbi z energijo, v stanovanjskih zgradbah in podobno. IIoT kot industrijski internet stvari pa je namenjen industriji za spremljanje in izboljševanje industrijskih procesov.

Poenostavljeno povedano je IIoT v industriji prenos podatkov med stroji in nadzornimi delavci, ki spremljajo, kako delujejo posamezni stoji in celoten obrat. Če velike količine takšnih podatkov shranjujemo in obdelujemo z ustreznimi orodji, pa lahko pridobimo ogromno koristnih informacij, ki podjetjem pomagajo pri izboljšavah in sprejemanju raznih odločitev. Najbolj značilne koristi uporabe IIoT v industriji so na primer povečanje produktivnosti, zmanjšanje porabe energije, pametno vzdrževanje strojev, preden pride do okvar in zastojev, možnost nadzora proizvodnje na daljavo, kar je bilo še posebej aktualno v času koronavirusa in podobno.

IIoT360 je modularna oblačna rešitev, ki je namenjena strojegradnikom, industrijskim podjetjem ter njihovim kupcem za spremljanje in optimizacijo delovanja. Kot nakazuje že ime, temelji na tehnologiji IIoT in s svojimi moduli zagotavlja celovito podporo vpogledu v delovanje strojev in izboljšanju njihovega delovanja (360).

Proizvajalci imajo možnost centraliziranega spremljanja delovanja svojih strojev različnih proizvodnih podjetjih oziroma lokacijah, kjer ti obratujejo. Tako pridobijo veliko uporabnih podatkov, na podlagi katerih lahko uporabnikom svetujejo, kako lahko povečajo njihovo razpoložljivost ter posledično produktivnost in znižajo stroške izpadov proizvodnje zaradi okvar. Tako svojim kupcem ponudijo storitve z večjo dodano vrednostjo, kot je zgolj dobava, servis in vzdrževanje strojev. Hkrati pa pridobijo ključne informacije za optimizacijo strojev ter razvojne rešitve, s katerimi povečujejo konkurenčnost in ugled svojih blagovnih znamk.

Uporabniki strojev pridobijo predvsem večjo izkoriščenost strojev ter nižje stroške zaradi manj zastojev proizvodnje in cenejšega vzdrževanja strojev. Prav tako pomembne pa so tudi informacije, ki jim pomagajo pri odločanju o nakupih strojev in investicijah v proizvodnji nasploh.

# 2 Poglavitni izzivi proizvajalcev strojev

Današnje industrijsko okolje se sooča z izzivom hitrega pospeševanja poslovanja in uvajanja sprememb. Skoraj polovice podjetij, ki so danes uvrščena na seznam Fortune 500, leta 2000 na njem še ni bilo.

Tradicionalne proizvajalce vse bolj izpodrivajo inovatorji, ki svoje poslovne priložnosti in rast gradijo na digitalnih tehnologijah. To se pogosto odraža tudi v cenovnem pritisku, kar proizvajalce spodbuja k iskanju novih virov dodane vrednosti za kupce.

Pri proizvajalcih strojev to pomeni, da svoje proizvode vse pogosteje nadgrajujejo z digitalnimi storitvami. Razvoj digitalnih orodij, ki so temelj za nove storitve (ter optimizacijo obstoječih procesov), pa je zamuden in drag ter terja zaposlitev specializiranega kadra, kar majhnim in srednje velikim proizvajalcem še otežuje konkurenčnost.

Za preskok pri izboljšanju svojega poslovnega modela in posledično dvig konkurenčnosti morajo proizvajalci strojev najti ustrezno digitalno rešitev za premagovanje izzivov, ki danes to onemogočajo:

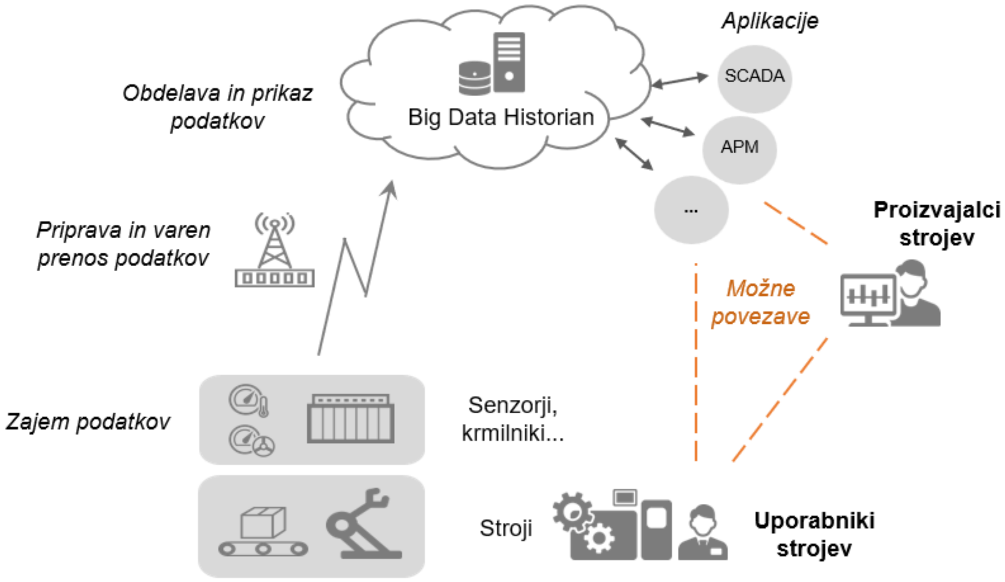
- pomanjkljiv vpogled v delovanje strojev v realnem času ter otežene analize zgodovinskih podatkov za doseganje izboljšav,
- razpršenost podatkov zaradi lokalnih zapisov o delovanju strojev (npr. dnevniki izpadov in okvar, obratovalni podatki),
- dolgi odzivni časi pri kompleksnejših zastojih (obveščanje, identifikacija in odprava napake),

- oteženo in drago zagotavljanje kakovostnega vzdrževanja,
- pomanjkljive informacije za razvoj inovacij in novih generacij strojev,
- pomanjkanje dodatnih storitev oziroma razvoj poslovnih modelov za povečanje prihodkov in razlikovanje od konkurence (npr. svetovanje za optimalno uporabo, prediktivno vzdrževanje, najemni modeli).

## 3 Arhitektura platforme

Za vzpostavitev platforme, ki omogoča realizacijo funkcionalnosti, ki rešujejo zgoraj navedene težave je potrebno vzpostaviti naslednje komponente (slika 1):

- Edge naprava, ki se vgradi kot del samega stroja in zagotavlja zajem procesnih podatkov, začasno hrambo teh (v primeru nedostopnosti oblačne infrastrukture) ter njihovo posredovanje
- prenosni kanal, ki omogoča varen prenos podatkov
- centralno platformo, ki skrbi za hrambo podatkov ter implementira nabor API-jev za nadaljnjo obdelavo podatkov



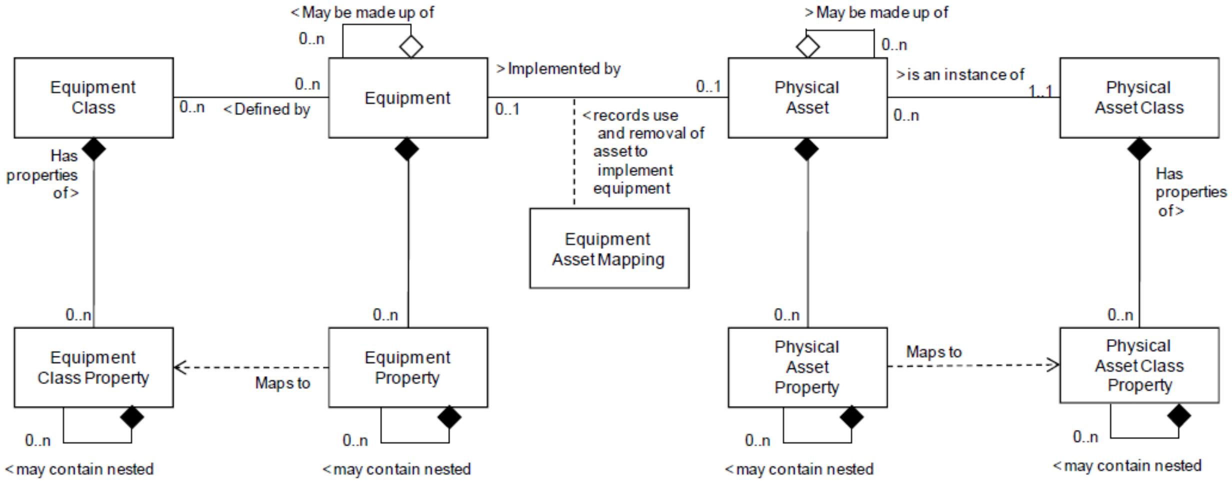
**Slika 1: Idejna zasnova platforme.**

### 3.1 Aplikativni vidik platforme

Kot Edge napravo lahko pojmujemo vsak kos strojne opreme, ki se umešča med zunanjim omrežjem ter notranjim omrežjem (oz. strojem) in skrbi za kontroliran pretok podatkov. Z vidika funkcionalnosti platforme gre za poglavitni kos strojne opreme, ki mora (v odvisnosti od dejanskega projekta) zagotavljati nabor funkcionalnosti:

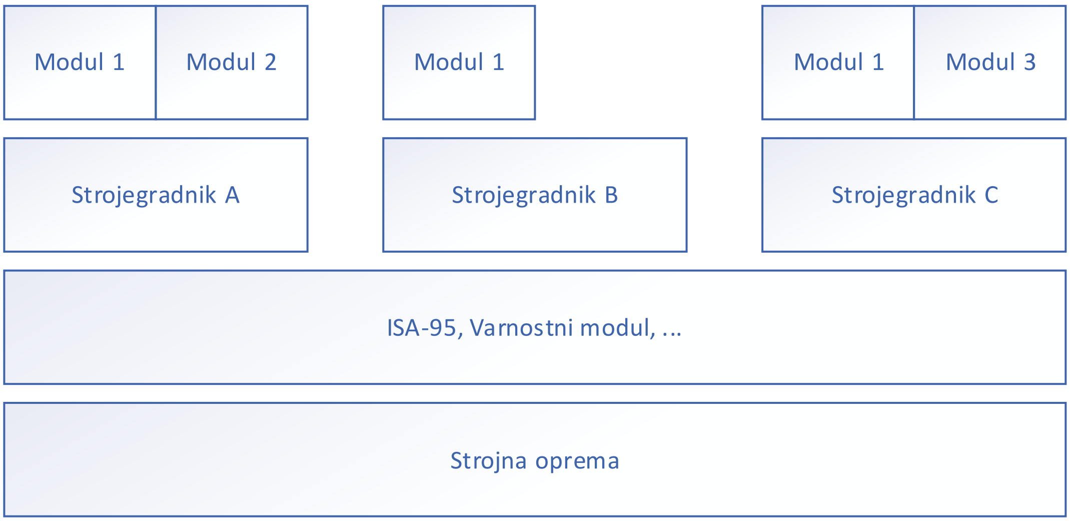
- Zmožnost hrambe procesnih podatkov določeno časovno obdobje pri čemur je potrebno upoštevati tako količino podatkov, ki se zbirajo kot maksimalno retenzijo teh
- Komunikacijski vmesniki (npr. tako LAN, WiFi kot 4G/5G vmesnik)
- Možnost priklopa zunanjih anten
- Količina digitalnih ter analognih priključkov
- Temperaturni razpon delovanja
- Odpornost na tresljaje

V sami IIoT360 platformi je kot osnovna funkcionalnost za obvladovanje podatkov zagotovljena implementacija ISA-95 modela (slika 3), ki omogoča strojegraditeljem pripravo digitalnega dvojčka okolja.



**Slika 3: Standardni podatkovni model (izsek)**
Vir: [7].

Centralna komponenta platforme temelji na multitenant pristopu (slika 4), kjer zagotavljamo enovito postavitev aplikacijskih komponent, ki so skupne vsem strojegradnikom z upoštevanjem pravic dostopa do podatkov ter funkcionalnosti, ki jih ima zakupljen posamezni strojegradnik.

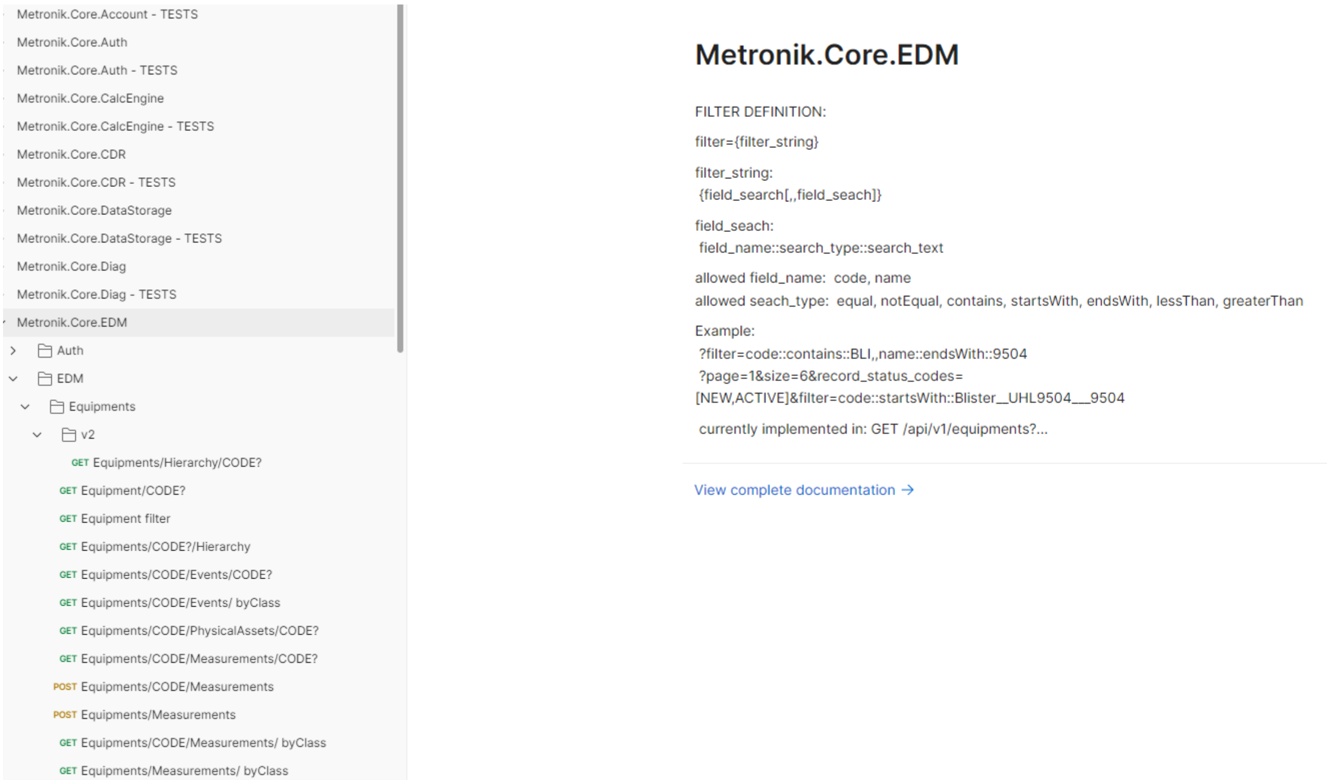


**Slika 4: Ločevanje strojegradnikov ter funkcionalnih modulov v platformi**

## 4 Uporabniški vidik platforme

Z zagotovitvijo prenosa ter hrambe (surovih) podatkov se širok nabor možnosti njihove nadaljnje obdelave ter vizualizacije, kar omogočamo na dva načina:

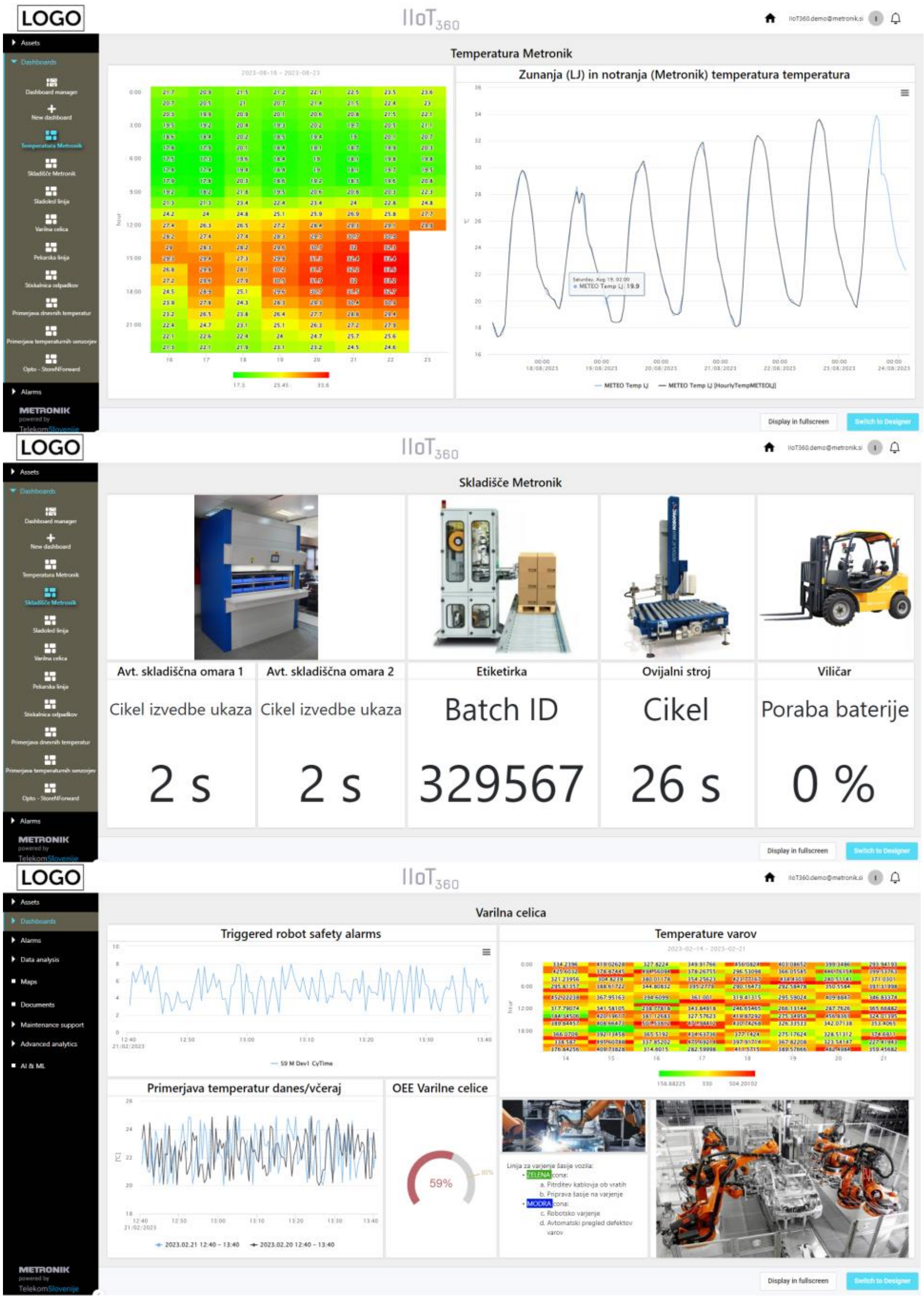
- Dostop preko REST API vmesnika za potrebe integracije s 3rd party aplikacijami, kot so npr. ticketing sistemi pri strojegraditeljih (slika 5),
- Uporaba namenske uporabniške aplikacije, ki omogoča t.i. whitelabeling ter kustomizacijo nabora funkcionalnosti (slika 6).



**Metronik.Core.EDM**

```
FILTER DEFINITION:
filter={filter_string}
filter_string:
 {field_search[,,field_seach]}
field_seach:
 field_name::search_type::search_text
allowed field_name:  code, name
allowed seach_type:  equal, notEqual, contains, startsWith, endsWith, lessThan, greaterThan
Example:
 ?filter=code::contains::BLI,,name::endsWith::9504
 ?page=1&size=6&record_status_codes=
[NEW,ACTIVE]&filter=code::startsWith::Blister__UHL9504___9504
 currently implemented in: GET /api/v1/equipments?...
```

**Slika 5: Nabor REST API-jev za dostop do podatkov.**



**Slika 6: Primeri nadzornih plošč.**

Sama uporabniška aplikacije je v nadaljevanju razdeljena v module, ki jih lahko posamezni strojegradnik naroči na nivoju posameznega stroja:

- **Osnovni (Core)**

  Zagotavlja osnovne pogoje, ki so potrebni za ustrezno delovanje sistema in je predpogoj za nadgradnjo funkcionalnosti platforme z ostalimi moduli.

  Zagotavlja programsko rešitev za:
  - zajem podatkov s strojev na strukturiran način, njihovo pripravo in prenos v oblak Telekoma Slovenije preko internetne povezave ali mobilnega omrežja
  - upravljanje sredstev (npr. strojev), upravljanje uporabnikov in pravic ter upravljanje Edge naprav
  - kontekstualizacijo podatkov in ustrezno shranjevanje v oblaku z možnostjo nadaljnje obdelave oziroma izvoza podatkov v tretje sisteme

- **Delovanje strojev (Operational Monitoring)**
  Omogoča ustrezno obdelavo zbranih podatkov o stroju ter njihovo vizualizacijo za pomoč pri obvladovanju delovanja strojev.
  Zagotavlja programsko rešitev za:
    - spremljanje statusa strojev, geolokacije strojev
    - vizualizacijo podatkov in izračun KPI-jev (ki si jih lahko končni uporabnik nastavi po svojih potrebah)
    - prilagodljivo obveščanje o dogodkih na stroju in nepravilnem delovanju (nadzorne plošče, SMS, e-pošta)
    - izvajanje analiz učinkovitosti strojev in faktorjev izgub
    - zgodovinski pregled podatkov
    - prikaz uporabniške dokumentacije po strojih (uporabniška navodila, navodila za delo...)
    - priprava poročil o delovanju strojev
- **Podpora vzdrževanju (Maintenance Support)**
  Modul omogoča lažji nadzor nad izvajanjem vzdrževalnih aktivnosti.
  Zagotavlja programsko rešitev za:
    - odpiranje in zapiranje zahtevkov, KanBan, merjenje učinkovitosti vzdrževalskih aktivnosti
    - alarmiranje vzdrževalcev (SMS, e-pošta)
    - prikaz vzdrževalske dokumentacije po strojih (tehnična dokumentacija, navodila za vzdrževanje...)
    - pregled sestavnih delov naprav in njihove zgodovine (menjave, okvare...)
    - pripravo poročil o vzdrževanju
- **Napredna analitika (Advanced Analytics)**
  Omogoča napredno obdelavo zbranih podatkov in posledično boljše izkoriščanje podatkov.
  Zagotavlja programsko rešitev za:
    - izdelavo prilagodljivih poročil (Custom Reporting)
    - primerjavo obratovanja iste vrste strojev na različnih lokacijah (Similar Asset Charting)
    - hitro raziskovanje in odpravljanje vzrokov okvar z bazo znanja (Root Cause Analysis)

# 5 Zaključek

IIoT platforma je večplasten sistem ki omogoča kontrolirano pripravo, upravljanje in digitalizacijo naprav povezanih v okviru sistema IoT. Zasnovana je tako, da poveže strojno opremo različnih proizvajalcev in tehnologij v oblak s pomočjo fleksibilnih komunikacijskih kanalov in visoko varnostnih mehanizmov. Zbirka pripravljenih funkcionalnosti omogoča pospešen razvoj aplikacij za povezane naprave brez skrbi o medsebojni nekompatibilnosti tehnologij in proizvajalcev.

Taka platforma se lahko gleda kot na vmesni sloj, ki skrbi za varno in enolično povezavo oddaljenih naprav z uporabniškimi aplikacijami, ter omogoča upravljanje vseh interakcij med njima. Drugi aspekt IoT platforme je digitalizacija strojne opreme, kjer služi kot »cloud enabler«, saj omogoči standardiziran dostop do podatkov vseh povezanih naprav kot osnovo razvoja različnih poslovno informacijskih aplikacij.

Nadaljnji razvoj bo platformo razširil z uporabo naprednih tehnoloških storitev umetne inteligence, ki uporabniku omogočijo avtomatiziranje procesov oziroma izboljšanje uporabniške izkušnje. Razvojne smernice so:

Napredno strojno učenje. Pridobivanje znanj o assetu na podlagi preteklih dogodkov in uporabniških izkušenj. Na podlagi pridobljenih podatkov se išče vzorce obnašanja sistema in predvidi prihajajoče dogodke.

Digitalni dvojček. Replikacija fizične naprave v digitalno obliko. Pri tem se naredi dvojček z vsemi njenimi opisnimi, funkcionalnimi in procesnimi lastnostmi.

Obogatena resničnost (Augmented reality). Prikaz informacij asseta in navideznih predmetov v realnem času se preko mobilne naprave doda na avdio/vizualno stanje realnega okolja.

# Literatura

[1] Metronik d.o.o. (2017 – 2023). MxWeb 4.0 razvoj (interno gradivo). Ljubljana: Metronik d.o.o.

[2] Metronik d.o.o. (2018 – 2023). Unified Data Gateway Architecture (interno gradivo). Ljubljana: Metronik d.o.o.

[3] Metronik d.o.o. (2018 – 2023). Funkcijska specifikacija (interno gradivo). Ljubljana: Metronik d.o.o

[4] Haile, N., & Altmann, J. (2016). Value creation in software service platforms. Future Generation Computer Systems, 55, 495–509.

[5] 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.

[6] 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.

[7] Bell, M. (2008). Service-Oriented Modeling (SOA): Service Analysis, Design, and Architecture. Hoboken: Wiley & Sons.

[8] ANSI/ISA-95.00.02-2010 (IEC 62264-2 Mod), Enterprise-Control System Integration – Part 2: Object Model Attributes

# Podatki v utripu srca: podatkovna platforma za zajem biosenzoričnih podatkov v realnem času

Urška Nered, Jure Jeraj

Result d.o.o., Ljubljana, Slovenija
urska.nered@result.si, jure.jeraj@result.si

Pametne naprave s senzorji, ki beležijo naše gibalne, spalne in druge navade, so potencialno obsežen vir podatkov – še posebej, če jih lahko izvlečemo in združimo s podatki iz drugih virov. V prispevku predstavljamo podatkovno platformo, ki je namenjena prav temu – zajemu, shranjevanju in obogatitvi biosenzoričnih podatkov v realnem času. Za izhodiščne komponente smo izbrali Apache Kafka kot komunikacijsko vodilo, Apache Druid za shranjevanje in ksqlDB za transformacije podatkov. Taka platforma predstavlja osnovo in izhodišče za končni produkcijski sistem, saj že temelji na ustreznih tehnologijah za delo s pretočnimi podatki v realnem času.

**Ključne besede:**
podatkovna platforma
biosenzorični podatki
pretočni podatki
Apache Kafka
Apache Druid
ksqlDB

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.19

## 1 Uvod

Večina si nas življenje deli z vsaj eno pametno napravo, ki zbira podatke o naših dejavnostih, spalnih navadah, srčnem utripu in drugih metrikah, ki so posredno ali neposredno povezane tudi z našim zdravjem. Ker so naprave vedno z nami, je zajem takih podatkov enostaven tudi dolgoročno in v velikih količinah, s čimer predstavlja velik potencial za spremljanje ne le naših športnih dosežkov ampak tudi zdravstvenega stanja v daljšem obdobju, zaznavanje sprememb, spremljanje pacienta v času okrevanja. Če veliko takih podatkov naprave že beležijo in hranijo, zakaj jih ne bi izkoristili?

Za izkoriščanje takih podatkov najprej naletimo na problem zaprtih podatkov, tipično vezanih na proizvajalca naprave ali aplikacije, ki beleži meritve. Problem ni nepomemben, ni pa nepremostljiv. Podatke torej z nekaj dela imamo. Kaj pa zdaj? Lahko jih shranjujemo, a sami po sebi brez pametne uporabe niso vredni veliko.

Za resnično obogatitev in dodano vrednost iz podatkov, bi si jih želeli zajemati, kombinirati s podatki iz drugih virov, jih imeti možnost ponovno uporabiti za drugačen namen, vse to pa početi sproti v realnem času. Tukaj nastopi podatkovna platforma kot enotna točka, kamor se lahko stekajo podatki iz vseh naših pametnih naprav skupaj s podatki iz drugih poljubnih virov. Omogoča nam njihovo shranjevanje, procesiranje, integracijo, obogatitev za sprotno uporabo, hkrati pa spremljanje trendov v daljšem časovnem obdobju. Ob primernih varnostnih mehanizmih je z dovoljenjem posameznika to lahko osnova za zbiranje podatkov za namene znanstvenih raziskav, treniranje modelov umetne inteligence za diagnostične namene, zgodnje zaznavanje sprememb v zdravstvenem stanju itd. V nadaljevanju bo prikazan zametek takšne podatkovne platforme. Njeni ključni sestavni deli so Apache Kafka za zajem podatkov s senzorjev v realnem času, ksqlDB za njihovo sprotno procesiranje in Apache Druid za shranjevanje za takojšnjo ali kasnejšo uporabo.

## 2 Na kratko o podatkih

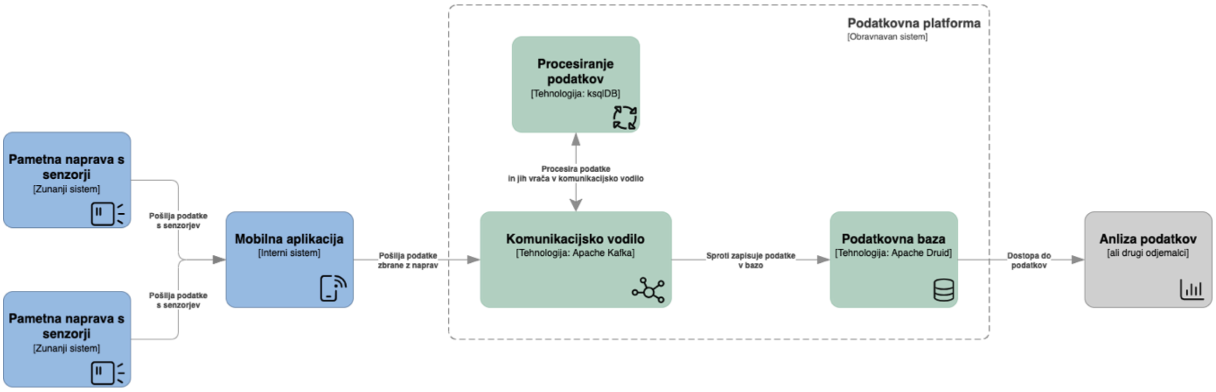
Podatkovna platforma, ki jo predstavljamo v tem članku, je namenjena zbiranju in obdelavi podatkov, zbranih s senzorji pametnih naprav – pametnih telefonov, ur, majic, tehtnic … Pametne naprave, ki nas spremljajo večino časa, o nas vejo veliko: koliko in kako se gibamo, kako spimo, koliko smo pod stresom itd., vendar so podatki, ki jih zberejo, težko dosegljivi izven naprave same ali mobilnih aplikacij proizvajalca. Kljub nekaterim standardom, ki obstajajo, pa je implementacija med proizvajalci različna.

Za pridobivanje podatkov iz različnih pametnih naprav smo razvili svojo mobilno aplikacijo z zalednim delom, ki pa bo tema za kak drug članek. V našem primeru je dovolj, da vemo, da lahko sistem podatke na platformo pošilja v realnem času, če je pametna naprava dovolj blizu telefona, da sta lahko povezana preko Bluetooth povezave (v nasprotnem primeru naprava podatke shrani in pošlje, ko se spet povežeta). Podatki so opremljeni s tipom naprave, ki jih je zbrala, časom meritve, tipom meritve (koraki, srčni utrip, teža itd.) in seveda izmerjeno vrednostjo v dogovorjeni enoti.

## 3 Arhitektura podatkovne platforme

Za osnovno podatkovno platformo za zajem in uporabo biosenzoričnih podatkov potrebujemo le nekaj komponent, ki jih kasneje lahko nadgrajujemo. O komponentah platforme govorimo od sprejema podatkov, ki jih pošilja zaledni del mobilne aplikacije naprej. Sama mobilna aplikacija skupaj z zalednim delom, pa tudi pametne naprave, ki zbirajo podatke, so z vidika platforme zunanje naprave oz. zunanji vir podatkov, kot je prikazano na sliki 1.

Ker gre za podatke s senzorjev, pričakujemo veliko število majhnih sporočil v kratkih časovnih intervalih. Kadar je pametna naprava povezana s telefonom, mu bo meritve pošiljala sproti, telefon pa jih bo sproti posredoval naprej. Zato bi želeli celo platformo prilagoditi zajemanju podatkov v realnem času. Kot osrednjo komponento naše platforme izberemo komunikacijsko vodilo, katerega glavna funkcija je sprejem podatkov iz zunanjih virov, v našem primeru mobilne aplikacije, in distribucija podatkov ostalim komponentam platforme. Podatke seveda želimo nekje hraniti, zato na komunikacijsko vodilo priključimo še podatkovno bazo, ki bo prejete podatke sproti zapisovala (shranjevala). Ker bi radi podatke sproti obdelali in obogatili že na nivoju sporočila, izberemo še komponento za procesiranje pretočnih podatkov v realnem času, ki bo brala vhodne podatke s komunikacijskega vodila, jih obogatila in ločeno zapisovala nazaj v komunikacijsko vodilo, da bodo na voljo za shranjevanje v bazo in vsem ostalim potencialnim odjemalcem. Za našo platformo smo kot komunikacijsko vodilo izbrali **Apache Kafka** [1], za podatkovno bazo **Apache Druid** [2], podatke pa bomo obdelovali s **ksqlDB** [3].



**Slika 1: Konceptualna arhitektura predstavljene podatkovne platforme.**

### 3.1 Podrobna arhitektura

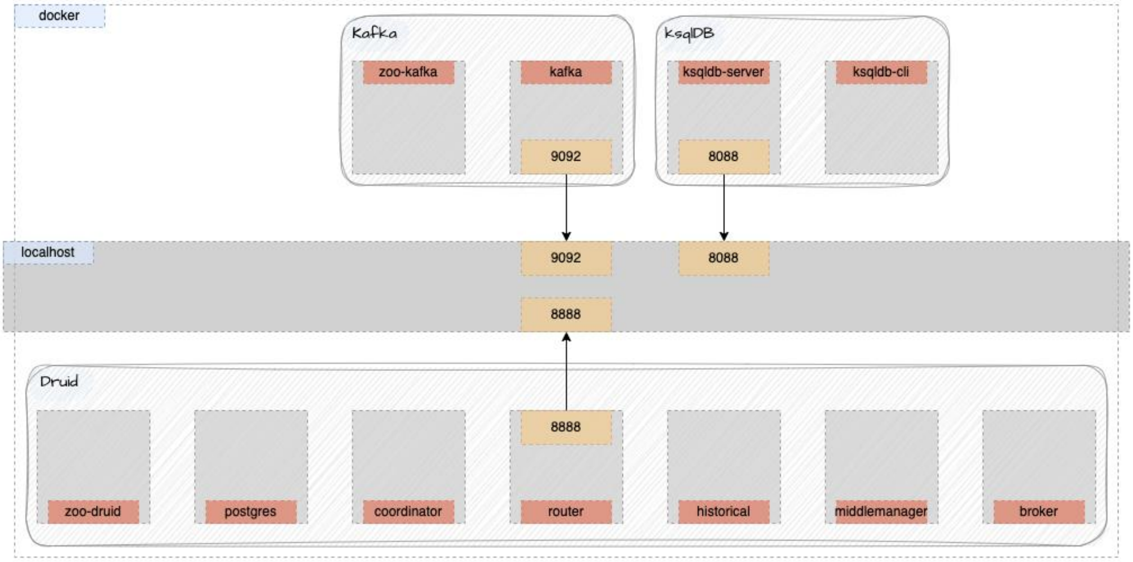
Vse komponente podatkovne platforme so za naš primer nameščene kot Docker vsebniki, ki jih poženemo z eno definirano docker-compose datoteko. Slika 2 prikazuje vsebnike, ki jih moramo zagnati za delovanje vseh delov platforme. Vsaki komponenti pripada več kot en vsebnik, saj različni deli skrbijo za različne naloge, na tak način pa jih lahko tudi neodvisno skaliramo glede na naše potrebe. Za Kafko in Druid potrebujemo še ZooKeeper [4], ki skrbi za povezovanje in komunikacijo med njunimi posameznimi vmesniki[1].

Na sliki 2 so prikazana vrata (ang. *port*), preko katerih je posamezen vsebnik dostopen. Zaradi preglednosti so prikazana le tista, ki so najbolj pomembna in jih bomo uporabljali v našem primeru.

V produkcijskem okolju bi postavitev seveda izgledala drugače. Še vedno je možna postavitev vseh komponent z vsebniki, čeprav se za izbrane komponente načeloma priporoča neposredna namestitev na strežnik. Bi pa v nasprotju z našo platformo, kjer vse komponente zaženemo na enem strežniku oz. osebnem računalniku, za produkcijsko rešitev seveda potrebovali gručo strežnikov in orodje za orkestracijo vsebnikov (Docker Swarm, Kubernetes ipd.).

---

[1] https://www.fluentd.org/
https://grafana.com/oss/loki/
https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/
CVE – izvorno "Common Vulnerabilities and Exposures", https://www.cve.org/
https://www.defectdojo.org/
seznam sestavnih delov – izvorno "Software Bill of Materials"



**Slika 2: Podrobna arhitektura predstavljene platforme s prikazom vseh uporabljenih Docker vmesnikov.**

## 3.2 Komunikacijsko vodilo – Apache Kafka

Komunikacijsko vodilo ima v podatkovni platformi pomembno funkcijo. Njegova glavna naloga je distribucija podatkov med komponentami platforme. Komponente bi si podatke lahko sicer izmenjevale tudi neposredno, vendar se na ta način stvari lahko hitro zapletejo. Komponente med sabo komunicirajo z različnimi protokoli, nekatere morda zahtevajo razvoj lastne rešitve za povezavo z drugimi komponentami itd. Z naraščanjem števila komponent kompleksnost takega sistema hitro narašča, komponente pa so, kar se tiče komunikacije, močno odvisne druga od druge. V primeru, da ena od para ne deluje, mora druga čakati, preden ji lahko pošlje podatke, menjava ene komponente z novo pa lahko zahteva veliko uskladitev v celotnem sistemu.

Za poenostavitev sistema in manjšo soodvisnost komponent je smiselno uvesti novo, ki skrbi za prenos podatkov med ostalimi – komunikacijsko vodilo. Pojem je sicer splošen in lahko pomeni marsikaj, odvisno od potreb našega sistema – komponente za integracijo, asinhroni sistemi za posredovanje sporočil, razni drugi posredniki in vmesniki itd., v našem primeru pa govorimo predvsem o sistemih za pretok podatkov v realnem času. V ta namen smo kot komunikacijsko vodilo v naši platformi izbrali orodje Apache Kafka [1].

Ključni nalogi Kafke sta prenos podatkov in zmanjšanje odvisnosti med ostalimi komponentami v sistemu, saj si podatke izmenjujejo preko Kafke, ne več neposredno. Kafka sporočila prenaša v binarni obliki, s čimer zagotavlja večje hitrosti prenosa in omogoča prenos sporočil različnih formatov. Za serializacijo in deserializacijo sporočil skrbijo ustvarjalci in odjemalci sporočil. Sporočilo je sestavljeno iz neobveznega ključa in pripadajoče vrednosti oz. vsebine sporočila. Sistem, ki ustvari sporočilo, ga zapiše v določeno temo (ang. *topic*). Vsaka vrsta sporočil tipično pripada svoji temi, podobno kot zapis v tabeli v relacijski bazi. Ni pa taka delitev obvezna, sporočila lahko npr. v teme razporedimo glede na izvorni sistem, eno sporočilo lahko zapišemo v več kot eno temo ipd. Vsaka tema je razdeljena na particije, sporočila z enakim ključem pa so vedno pripisana isti particiji. Znotraj ene particije je vrstni red sporočil zagotovljen, vsako sporočilo ima svoj identifikator, ki je vedno naraščajoč. Sporočila so shranjena za določen čas (osnovna nastavitev je 7 dni), v tem času pa jih lahko prebere poljubno število odjemalcev. Odjemalec sporočil na podlagi identifikatorja zadnjega prebranega sporočila v vsaki particiji ve, katerih sporočil še ni prebral. V primeru težav ob ponovnem zagonu nadaljuje z branjem od zadnjega prebranega sporočila dalje.

### 3.2.1 *Kafka v naši platformi*

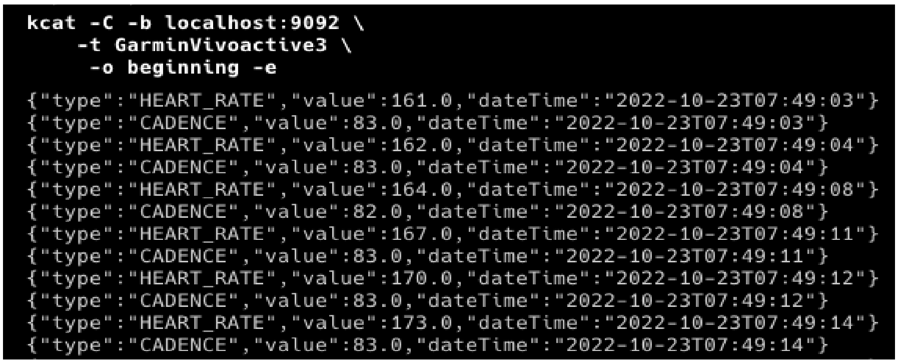
Za našo podatkovno platformo z biosenzoričnimi podatki imamo za vsak tip naprave, ki lahko pošilja podatke s svojih senzorjev, narejeno ločeno temo, katere ime je enako imenu, ki ga določimo napravi. V primeru dodajanja novih naprav se nove teme ustvarijo samodejno (taka nastavitev za produkcijski sistem sicer ni priporočljiva, saj običajno želimo imeti več nadzora).

V produkcijskem okolju bi za zagotavljanje dostopnosti sistema za Kafko uporabili vsaj tri strežnike. V našem primeru imamo postavljen le en "strežnik" (vsebnik), vseeno pa za pravilno delovanje potrebujemo tudi ZooKeeper. Kafka je z lokalnega okolja dostopna na vratih 9092 (Slika ).

Mobilna aplikacija podatke s senzorjev pošilja v formatu JSON, vsako sporočilo pa vključuje tri vrednosti:

- Type (*String*): tip meritve, npr. "HEART_RATE" (srčni utrip); hkrati tudi ključ za sporočilo,
- Value (*Float*): izmerjena vrednost v dogovorjeni enoti, za srčni utrip so to npr. udarci na minuto oz. BPM,
- DateTime (*String*): datum in čas meritve (UTC).

Primeri sporočil so prikazani na sliki 3.



```
kcat -C -b localhost:9092 \
    -t GarminVivoactive3 \
     -o beginning -e
{"type":"HEART_RATE","value":161.0,"dateTime":"2022-10-23T07:49:03"}
{"type":"CADENCE","value":83.0,"dateTime":"2022-10-23T07:49:03"}
{"type":"HEART_RATE","value":162.0,"dateTime":"2022-10-23T07:49:04"}
{"type":"CADENCE","value":83.0,"dateTime":"2022-10-23T07:49:04"}
{"type":"HEART_RATE","value":164.0,"dateTime":"2022-10-23T07:49:08"}
{"type":"CADENCE","value":82.0,"dateTime":"2022-10-23T07:49:08"}
{"type":"HEART_RATE","value":167.0,"dateTime":"2022-10-23T07:49:11"}
{"type":"CADENCE","value":83.0,"dateTime":"2022-10-23T07:49:11"}
{"type":"HEART_RATE","value":170.0,"dateTime":"2022-10-23T07:49:12"}
{"type":"CADENCE","value":83.0,"dateTime":"2022-10-23T07:49:12"}
{"type":"HEART_RATE","value":173.0,"dateTime":"2022-10-23T07:49:14"}
{"type":"CADENCE","value":83.0,"dateTime":"2022-10-23T07:49:14"}
```

**Slika 3: Primer sporočil, ki jih na platformo pošilja mobilna aplikacija za zajem podatkov s pametnih naprav.**

### 3.3 Shranjevanje podatkov in poizvedbe – Apache Druid

Apache Druid je precej nova baza, njen razvoj se je začel leta 2011, odprtokodna pa je od leta 2012 (Apache licenca od leta 2015). Danes jo uporablja kar nekaj velikih podjetij, med drugimi Airbnb, eBay, Netflix, Paypal, Slack in Reddit. Z nje zajemajo ogromne količine podatkov v realnem času (Netflix npr. v Druid shrani do 2 TB podatkov vsako uro, eBay pa zajema podatke s hitrostjo več kot 100.000 dogodkov na sekundo) [7].

Naš primer je seveda veliko manjši, za trenutno stanje bi zadostovala katerakoli podatkovna baza. Je pa smiselno že v tej fazi predvideti, kakšne podatke pričakujemo, kakšna rast je mogoča in na kakšen način jih želimo shranjevati ter nato do njih dostopati, in izbrati podatkovno bazo, ki je prilagojena taki uporabi. V našem primeru gre za meritve senzorjev, kjer pričakujemo veliko število majhnih sporočil, podatke pa bi radi zajemali v realnem času. Še ena tipična lastnost podatkov s senzorjev (in pravzaprav vseh podatkov v realnem času) je, da so časovno urejeni in je čas nastanka podatka ena od ključnih informacij. S tega vidika izbira bolj klasične relacijske baze ni najboljša, podatki bolj ustrezajo kakšni od časovno vrstičnih (ang. *timeseries*) baz. Druid je v prvi vrsti namenjen delu s pretočnimi podatki in v svoji arhitekturi združuje principe podatkovnih skladišč, časovno vrstičnih baz in sistemov za pregledovanje log datotek, tako da poleg hitrega shranjevanja pretočnih podatkov omogoča tudi dober odziv na analitične poizvedbe [8]. Velja pa omeniti, da Druid ni dobra izbira za sisteme, kjer potrebujemo hitre posodobitve obstoječih podatkov in v primerih, ko želimo v poizvedbah združevati več zelo velikih tabel dejstev (ang. *fact table*) [8].

Druid sestavlja šest vrst internih komponent, ki skrbijo za različne naloge, vsako pa lahko horizontalno skaliramo glede na svoje potrebe (slika 2):

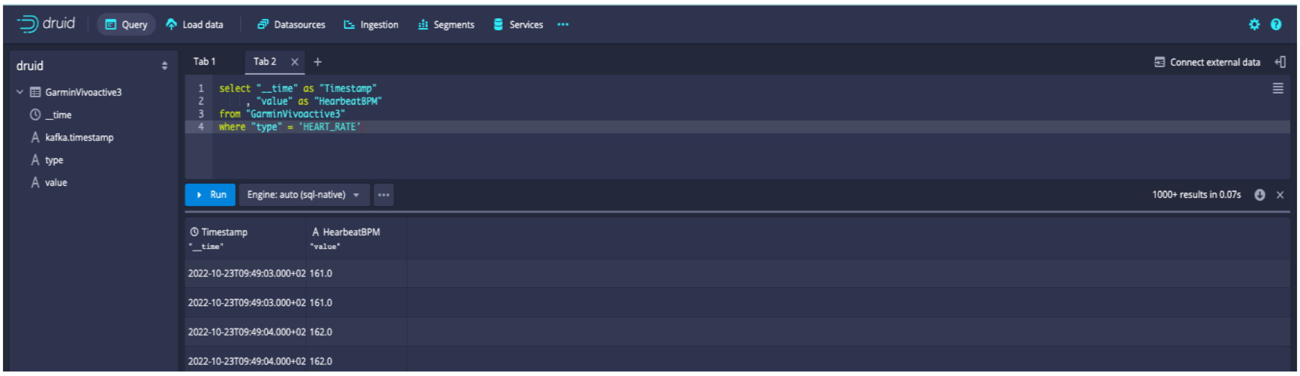
- *Coordinator* skrbi za dostopnost podatkov v sistemu,
- *Overlord* dodeljuje naloge, povezane z zajemom podatkov,
- *Broker* sprejema in streže poizvedbe,
- *Router* ni obvezen, skrbi pa za preusmeritev zahtev zgornjim trem komponentam,
- *Historical* hrani podatke za poizvedbe,
- *MiddleManager* zajema vhodne podatke.

Poleg vsega tega za delovanje Druida potrebujemo še relacijsko bazo, običajno PostgreSQL, kamor se shranjujejo metapodatki, ZooKeeper, ki skrbi za koordinacijo strežnikov in komponent, za prenos podatkov v ozadju in ponovno vzpostavitev sistema, v primeru popolne odpovedi pa potrebujemo tudi zunanje shranjevanje podatkov, tipično S3, HDFS ali omrežno povezan datotečni sistem.

### 3.3.1 *Druid v naši platformi*

V našem primeru uporabljamo vseh šest Druidovih komponent, tudi *Router*, so pa vse postavljene kot Docker vmesniki (slika 2). Poleg osnovnih Druidovih komponent imamo še ločen ZooKeeper (ne delita si ga s Kafko) in PosgreSQL za shranjevanje metapodatkov. Za zunanjo hrambo podatkov uporabljamo kar lokalni datotečni sistem, saj v tej fazi podatkov še ni zelo veliko. Za zajem podatkov s Kafke moramo pri postavitvi Druida definirati ustrezno razširitev, ki to omogoča ("*druid-kafka-indexing-service*"). Ta bo nato avtomatično vključena, s čimer bosta omogočena povezava s Kafko in neposredni zajem podatkov.

Druidov uporabniški vmesnik (slika 2) je dosegljiv preko vrat 8888, od koder *Router* skrbi za preusmeritev naših zahtev ustreznim komponentam. Preko uporabniškega vmesnika lahko urejamo zajem podatkov, kjer v našem primeru definiramo na Kafki temo, ki jo želimo brati, določimo stolpec, ki bo primarno uporabljen za particioniranje po času, določimo morebitne filtre za podatke, ki jih ne želimo zajemati (npr. neveljavne vrednosti na senzorju), in ostale nastavitve zajema. Ko nastavitve objavimo, se zažene zajemanje podatkov. Podatki so takoj po začetku zajema že na voljo za poizvedbe (slika 4). Naprej jih lahko obdelujemo v jeziku SQL, jih po želji filtriramo, agregiramo itd.



```sql
select "__time" as "Timestamp"
     , "value" as "HearbeatBPM"
from "GarminVivoactive3"
where "type" = 'HEART_RATE'
```

| Timestamp<br>"__time" | HearbeatBPM<br>"value" |
|---|---|
| 2022-10-23T09:49:03.000+02 | 161.0 |
| 2022-10-23T09:49:03.000+02 | 161.0 |
| 2022-10-23T09:49:04.000+02 | 162.0 |
| 2022-10-23T09:49:04.000+02 | 162.0 |

**Slika 4: Prikaz Druidovega uporabniškega vmesnika, kjer lahko pregledujemo podatke.**

Ker je Druid v osnovi namenjen delu s podatki v časovnih vrstah, ima veliko funkcionalnosti vezanih na časovne vrednosti. Tako lahko npr. Nastavimo, v katerem časovnem pasu smo, kar bo časovne podatke izpisovalo v našem lokalnem času, čeprav so shranjeni v UTC. Za optimizacijo poizvedb se lahko tudi odločimo, da so nam dovolj približni rezultati agregacij, kar še dodatno pohitri obdelavo podatkov.

### 3.4 Procesiranje podatkov – ksqlDB

Manj zahtevno procesiranje podatkov bi sicer lahko naredili kar v Druidu in izpustili ločeno komponento, vendar smo se zaradi ločitve nalog in lažje potencialne menjave komponent v prihodnosti odločili, da shranjevanje in procesiranje ločimo.

Zaradi enostavnosti uporabe smo izbrali ksqlDB [2], ki je del Kafka ekosistema. Podoben namen ima tudi Kafka Streams [9], je pa uporaba ksqlDB veliko enostavnejša, saj se uporablja kot ločena komponenta, transformacije podatkov pa lahko delamo v jeziku SQL, kar omogoča široko uporabo. Po drugi strani je Kafka Streams knjižnica, ki jo lahko uporabimo v lastni aplikaciji, transformacije pa so zato manj dostopne za analitike. Druga velika prednost ksqlDB so integrirani vmesniki za povezovanje najbolj razširjenih zunanjih podatkovnih sistemov s Kafko, za katere bi sicer uporabili komponento Kafka Connect [10].

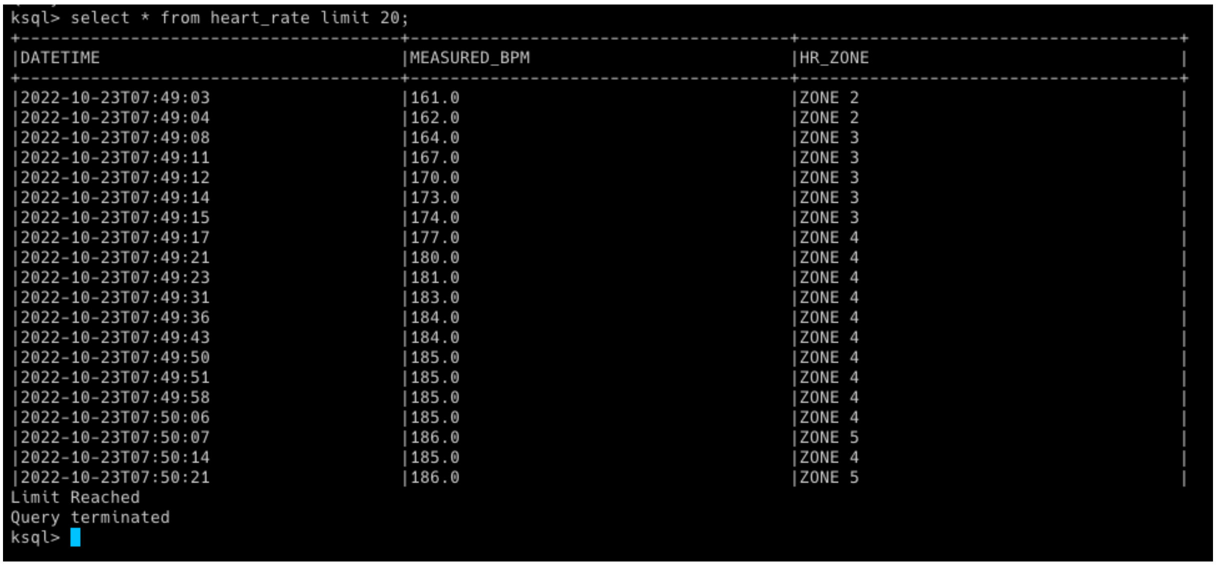
Komponenta ksqlDB omogoča sprotno procesiranje pretočnih podatkov s Kafke, pri čemer lahko podatke filtriramo, agregiramo in združujemo s podatki iz drugih Kafka tem. Končni rezultat procesiranja so pretočni podatki v ločeni temi, za agregirane podatke pa nam je na voljo prikaz v tabeli, kjer se vrednosti sproti posodabljajo glede na nove prejete podatke. V ozadju obeh je določena Kafka tema, kamor ksqlDB pošilja podatke oz. spremembe, zunanji sistemi pa jo lahko berejo kot vsako drugo temo. Poleg takih "potisnih" poizvedb (ang. *push query*), kjer so rezultati posredovani naprej kot nov tok pretočnih podatkov, so nam na voljo tudi poizvedbe v bolj klasični obliki, kjer pošljemo poizvedbo in dobimo enkraten odgovor (ang. *pull query*). Take vrste poizvedb imajo določene omejitve, ne podpirajo npr. neposrednih agregacij, saj se rezultat le-teh načeloma ves čas spreminja s prihodom novih podatkov. V takem primeru moramo v ozadju ustvariti tabelo kot potisno poizvedbo in agregacije pripraviti že v tabeli, nato pa jih s poizvedbo le prebrati.

Za delovanje ksqlDB potrebujemo eno komponento – strežnik, opcijsko pa lahko dodamo tudi CLI, s katerim lahko preko ukazne vrstice upravljamo z vsebino (slika 2). Kadar interaktivne poizvedbe in transformacije niso potrebne ali pa celo niso zaželene, lahko to komponento izpustimo in ob zagonu vmesnika podamo datoteko z definiranimi tabelami in tokovi podatkov, ki jih želimo ustvariti.

S svojimi funkcionalnostmi daje ksqlDB uporabniško izkušnjo podatkovne baze, ki omogoča povezovanje z drugimi sistemi, procesiranje podatkov in poizvedbe na "tabelah".

#### *3.4.1 ksqlDB v naši platformi*

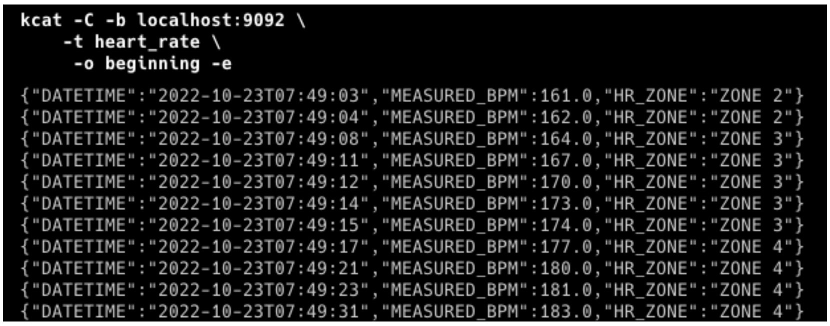
V naši platformi ksqlDB uporabljamo za sprotno filtriranje in obogatitev podatkov, ki prihajajo z merilnih naprav. Ker vsi podatki, ki jih izmeri določena naprava, pridejo v isto Kafka temo (Slika 3), jih s ksqlDB ločimo glede na tip meritve. V primeru je prikazano, kako meritve srčnega utripa ločimo od ostalih v svojo temo "heart_rate", vrednosti preimenujemo ("value" preimenujemo v "measured_bpm"), poleg tega pa glede na izmerjene vrednosti dodamo še podatek o coni srčnega utripa ("hr_zone"), na podlagi katere lahko analiziramo napor med dejavnostjo (slika 5).



```
ksql> select * from heart_rate limit 20;
+-----------------------------------+-----------------------------------+-----------------------------------+
|DATETIME                           |MEASURED_BPM                       |HR_ZONE                            |
+-----------------------------------+-----------------------------------+-----------------------------------+
|2022-10-23T07:49:03                |161.0                              |ZONE 2                             |
|2022-10-23T07:49:04                |162.0                              |ZONE 2                             |
|2022-10-23T07:49:08                |164.0                              |ZONE 3                             |
|2022-10-23T07:49:11                |167.0                              |ZONE 3                             |
|2022-10-23T07:49:12                |170.0                              |ZONE 3                             |
|2022-10-23T07:49:14                |173.0                              |ZONE 3                             |
|2022-10-23T07:49:15                |174.0                              |ZONE 3                             |
|2022-10-23T07:49:17                |177.0                              |ZONE 4                             |
|2022-10-23T07:49:21                |180.0                              |ZONE 4                             |
|2022-10-23T07:49:23                |181.0                              |ZONE 4                             |
|2022-10-23T07:49:31                |183.0                              |ZONE 4                             |
|2022-10-23T07:49:36                |184.0                              |ZONE 4                             |
|2022-10-23T07:49:43                |184.0                              |ZONE 4                             |
|2022-10-23T07:49:50                |185.0                              |ZONE 4                             |
|2022-10-23T07:49:51                |185.0                              |ZONE 4                             |
|2022-10-23T07:49:58                |185.0                              |ZONE 4                             |
|2022-10-23T07:50:06                |185.0                              |ZONE 4                             |
|2022-10-23T07:50:07                |186.0                              |ZONE 5                             |
|2022-10-23T07:50:14                |185.0                              |ZONE 4                             |
|2022-10-23T07:50:21                |186.0                              |ZONE 5                             |
Limit Reached
Query terminated
ksql>
```

**Slika 5: Primer poizvedbe s ksqlDB.**

Če se vrnemo na Kafko, lahko vidimo, da je zdaj na voljo tudi tema »heart_rate«, iz katere lahko beremo procesirane podatke (slika 6).



```
kcat -C -b localhost:9092 \
    -t heart_rate \
     -o beginning -e

{"DATETIME":"2022-10-23T07:49:03","MEASURED_BPM":161.0,"HR_ZONE":"ZONE 2"}
{"DATETIME":"2022-10-23T07:49:04","MEASURED_BPM":162.0,"HR_ZONE":"ZONE 2"}
{"DATETIME":"2022-10-23T07:49:08","MEASURED_BPM":164.0,"HR_ZONE":"ZONE 3"}
{"DATETIME":"2022-10-23T07:49:11","MEASURED_BPM":167.0,"HR_ZONE":"ZONE 3"}
{"DATETIME":"2022-10-23T07:49:12","MEASURED_BPM":170.0,"HR_ZONE":"ZONE 3"}
{"DATETIME":"2022-10-23T07:49:14","MEASURED_BPM":173.0,"HR_ZONE":"ZONE 3"}
{"DATETIME":"2022-10-23T07:49:15","MEASURED_BPM":174.0,"HR_ZONE":"ZONE 3"}
{"DATETIME":"2022-10-23T07:49:17","MEASURED_BPM":177.0,"HR_ZONE":"ZONE 4"}
{"DATETIME":"2022-10-23T07:49:21","MEASURED_BPM":180.0,"HR_ZONE":"ZONE 4"}
{"DATETIME":"2022-10-23T07:49:23","MEASURED_BPM":181.0,"HR_ZONE":"ZONE 4"}
{"DATETIME":"2022-10-23T07:49:31","MEASURED_BPM":183.0,"HR_ZONE":"ZONE 4"}
```

**Slika 6: Podatki, ki smo jih obdelali s ksqlDB so na voljo v novi Kafka temi "heart_rate".**

Poleg surovih podatkov lahko zdaj v Druid zajemamo tudi že procesirane podatke o srčem utripu in na njih delamo dodatne analize.

## 4 Zaključek

V prispevku smo na kratko predstavili osnutek podatkovne platforme za zajem in obdelavo biosenzoričnih podatkov v realnem času. Platforma podatke prejema iz različnih pametnih naprav (pametna ura, merilec srčnega utripa ...). Enotna vstopna točka za podatke je Kafka, ki deluje kot osrednji živčni sistem platforme. Podatke lahko nato "v letu" prečistimo, transformiramo in poenotimo s ksqlDB. Končne (pa tudi surove in neobdelane) podatke na koncu shranimo v podatkovno bazo Apache Druid, ki je optimizirana za hitro transakcijsko zapisovanje in že v osnovi kompatibilna s Kafko, po drugi strani pa je prilagojena za hitro branje in odziv na poizvedbe tudi na velikih količinah podatkov.

S predstavljeno rešitvijo lahko precej hitro pridemo do zametka platforme, ki v vseh fazah že uporablja ene od vodilnih gradnikov na tem področju. To je lahko osnova in model za izgradnjo prave produkcijske podatkovne platforme za delo s podatki v realnem času.

## Literatura

[1] https://kafka.apache.org/, Apache Kafka, obiskano 17. 7. 2023.

[2] https://ksqldb.io/, ksqlDB, obiskano 17. 7. 2023.

[3] https://druid.apache.org/, Apache Druid, obiskano 17. 7. 2023.

[4] https://zookeeper.apache.org/, Apache ZooKeeper, obiskano 18. 7. 2023.

[5] https://developer.confluent.io/learn/kraft/, KRaft: Apache Kafka without ZooKeeper, obiskano 18. 7. 2023.

[6] https://cwiki.apache.org/confluence/display/KAFKA/KIP-833%3A+Mark+KRaft+as+Production+Ready, KIP-833: Mark KRaft as Production Ready, obiskano 18. 7. 2023.

[7] https://druid.apache.org/druid-powered.html, Powered by Apache Druid, obiskano 19. 7. 2023.

[8] https://druid.apache.org/docs/latest/design/index.html, Introduction to Apache Druid, obiskano 19. 7. 2023.

[9] https://kafka.apache.org/documentation/streams/, Kafka Streams, obiskano 24. 7. 2023.

[10] https://docs.confluent.io/platform/current/connect/index.html, Kafka Connect, obiskano 24. 7. 2023.

# Platforma za napredno obdelavo merilnih podatkov za elektroenergetsko domeno

Andrej Bregar, Janez Mule, Franc Klauzner

Informatika, Informacijske storitve in inženiring, d.o.o., Maribor, Slovenija
andrej.bregar@informatika.si, janez.mule@informatika.si, franc.klauzner@informatika.si

V prispevku predstavimo napredno platformo za obdelavo merilnih podatkov (POMP), ki smo jo razvili za potrebe elektrodistribucijskih družb, katerim služi kot centralizirana baza za shranjevanje, validacijo in nadomeščanje 15-minutnih časovno odvisnih podatkov o števčnih meritvah. Opišemo večnivojsko arhitekturo platforme, ki temelji na sodobnih tehnologijah, kot so sporočilni sistem Apache Kafka, podatkovna baza TimescaleDB, mikrostoritveni vmesnik SmenDataBox ter storitveno vodilo Jurana. Platforma POMP združuje pet merilnih centrov in integrira več informacijskih sistemov, med katerimi izpostavimo rešitev Moj elektro, Centralni elektroenergetski portal Slovenije (CEEPS) in sistem eIS, ki so rezultat lastnega razvoja, pomenijo enotno vstopno točko za različne deležnike na elektroenergetskem trgu ter zagotavljajo mehanizme za poslovanje B2B in B2C. V članku podrobneje opišemo implementirane mehanizme za pridobivanje, prenos, validacijo in nadomeščanje merilnih podatkov. Umestimo jih v zakonodajni okvir in opredelimo pravila fleksibilnega obračuna omrežnine in električne energije, ki temeljijo na konceptih tarifnih blokov ter dogovorjene in presežne moči. Izpostavimo tudi ključne performančne podatke ter funkcionalne in nefunkcionalne značilnosti platforme POMP, ki potrjujejo, da je le-ta visoko zmogljiva, skalabilna in robustna.

**Ključne besede:**
merilni podatki
velepodatki
obračun omrežnine
tarifni blok
dogovorjena in presežna moč
Apache Kafka
TimeScaleDB
sistemske integracije

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.20

# 1 Uvod

Elektroenergetski sistem je izredno kompleksen. Kot pomemben del kritične infrastrukture države zajema veliko množico IT in OT integriranih naprav. Na teh napravah zbiramo podatke, ki so ključni za delovanje sistema, oskrbovanje odjemalcev z električno energijo in koordinacijo med deležniki na poslovnem, uporabniškem ter regulatornem nivoju. Zlasti so bistveni merilni podatki, ki pomenijo osnovo za obračun omrežnine in električne energije ter predstavljajo temeljne velepodatke sistema. Učinkovito zbiranje, shranjevanje in obdelovanje teh podatkov je velik tehnološki izziv. Še posebej se nivo zahtevnosti in kompleksnosti povečuje z uvedbo novih obračunskih pravil, ki vpeljujejo koncepte fleksibilnosti, tarifnih blokov, obračuna na osnovi 15-minutnih intervalnih vrednosti ter validacije in nadomeščanja nepopolnih podatkov. V podjetju Informatika smo zato vzpostavili napredno platformo za obdelavo merilnih podatkov (POMP), ki predstavlja centralizirano bazo za shranjevanje, validacijo in nadomeščanje časovno odvisnih (15-minutnih) podatkov. Do podatkov tako ni potrebno dostopati prek petih ločenih merilnih centrov distributerjev električne energije, temveč se iz teh centrov periodično potisnejo na osrednjo lokacijo. To omogoča naprednejšo podatkovno analitiko, boljše integracije in uvedbo novih poslovnih modelov, zlasti pa učinkovito prenovo obračunskega sistema, ki temelji na prehodu na nov tarifni sistem.

V nadaljevanju prispevka predstavimo platformo POMP. Opišemo njeno arhitekturo, uporabljene tehnologije, integracije z zunanjimi sistemi in tehnološke izzive, s katerimi smo se soočali. Platformo umestimo v zakonodajni okvir, ki določa pravila obračuna električne energije in omrežnine na osnovi 15-minutnih merilnih podatkov ter vpeljuje koncepte prožnosti elektroenergetskega trga. Dotaknemo se zelo pomembne problematike pridobivanja, pretoka, izmenjave, validacije in nadomeščanja merilnih podatkov, kar je ključno za delovanje sistema. V prispevku želimo izpostaviti kompleksnost platforme, njeno pomembnost za delovanje in tehnološki napredek slovenskega energetskega sistema ter nivo tehnološke zrelosti, ki ga je bilo potrebno doseči. Gre namreč za kompleksen sistem, ki povezuje množico podsistemov, aplikativnih sklopov in deležnikov na trgu. Platforma obdeluje veliko količino podatkov in na dnevnem nivoju izmenjuje okoli dva milijona sporočil (v okviru teh sporočil se prenaša dnevno preko 300 milijonov meritev iz različnih registrov na števcih), zaradi česar skušamo nakazati, da lahko predstavlja uspešen referenčni primer in dobro prakso, kako se na področju informacijskih tehnologij lotiti razvoja sodobnih skalabilnih in visoko zmogljivih rešitev za kompleksne poslovne domene in nacionalno pomembne sisteme kritične infrastrukture.

# 2 Zakonodajni okvir in uporabniške zahteve

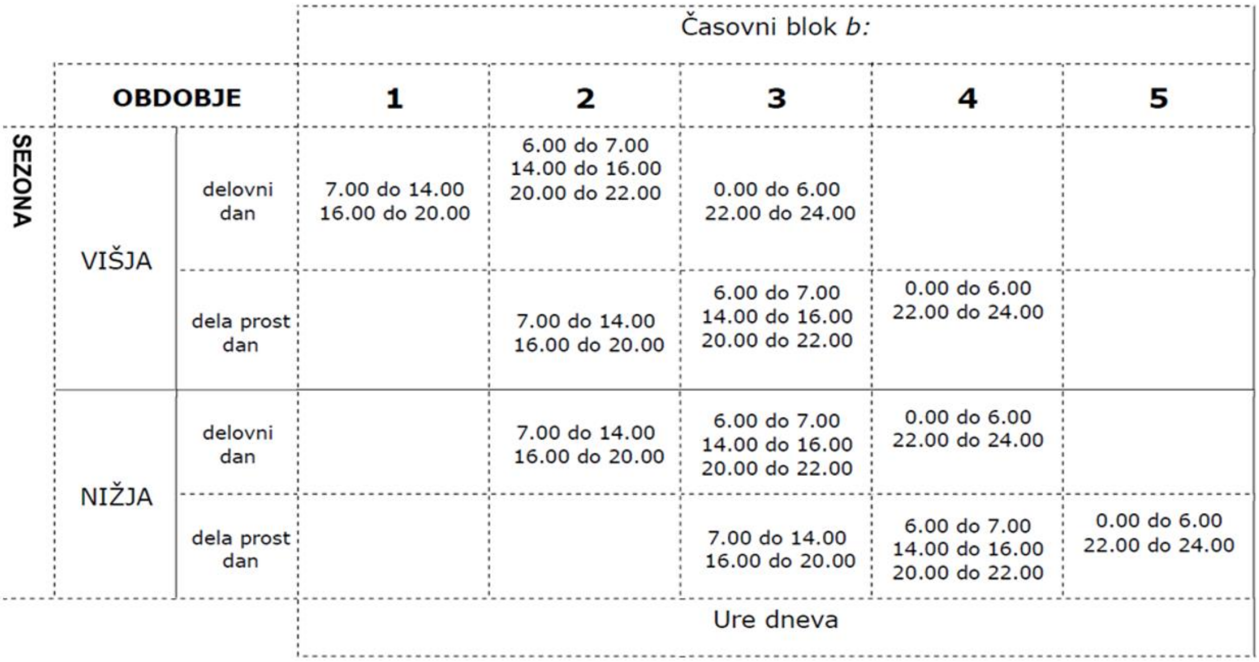
Dne 25. 11. 2022 je Agencija za energijo sprejela Akt o metodologiji za obračunavanje omrežnine za elektro operaterje [1]. Skladno s tem aktom bo stopil v veljavo nov tarifni sistem za obračunavanje omrežnine. Cilj novega tarifnega sistema je spodbuditi uporabnike, da s prilagajanjem odjema prispevajo k optimizaciji distribucijskega in tudi širše elektroenergetskega sistema. To lahko dosežemo z razbremenitvijo omrežja v času, ko je to najbolj obremenjeno z vidika porabe električne energije in moči.

Glavne značilnosti novega tarifnega sistema so:

- obračun, ki temelji na 15-minutnih meritvah,
- uvedba dveh sezon, višje med novembrom in februarjem ter nižje med marcem in oktobrom,
- uvedba pet časovnih blokov,
- razločevanje med dogovorjeno in presežno obračunsko močjo.

Spremembe, ki jih prinaša novi sistem, se nanašajo na vse uporabnike, tako na gospodinjstva kot na poslovne odjemalce, in bodo stopile v veljavo z dnem 1. 3. 2024.

Obračun omrežnine bo tudi v prihodnje temeljil na ločevanju glede na moč in energijo. Tarifne postavke omrežnine za obračunsko moč in energijo se bodo razlikovale glede na posamezen časovni blok. Ko bo omrežje najbolj obremenjeno, bodo najvišje, ko bo najmanj obremenjeno, pa najnižje. Porazdelitev časovnih blokov po sezonah, obdobjih in urah povzema slika 1 [1].



| | | Časovni blok *b*: | | | | |
|---|---|---|---|---|---|---|
| SEZONA | OBDOBJE | 1 | 2 | 3 | 4 | 5 |
| VIŠJA | delovni dan | 7.00 do 14.00<br>16.00 do 20.00 | 6.00 do 7.00<br>14.00 do 16.00<br>20.00 do 22.00 | 0.00 do 6.00<br>22.00 do 24.00 | | |
| | dela prost dan | | 7.00 do 14.00<br>16.00 do 20.00 | 6.00 do 7.00<br>14.00 do 16.00<br>20.00 do 22.00 | 0.00 do 6.00<br>22.00 do 24.00 | |
| NIŽJA | delovni dan | | 7.00 do 14.00<br>16.00 do 20.00 | 6.00 do 7.00<br>14.00 do 16.00<br>20.00 do 22.00 | 0.00 do 6.00<br>22.00 do 24.00 | |
| | dela prost dan | | | 7.00 do 14.00<br>16.00 do 20.00 | 6.00 do 7.00<br>14.00 do 16.00<br>20.00 do 22.00 | 0.00 do 6.00<br>22.00 do 24.00 |
| | | Ure dneva | | | | |

**Slika 1: Časovna porazdelitev časovnih blokov po sezonah, obdobjih in urah.**

Višja sezona bo trajala od novembra do februarja, nižja pa od marca do oktobra. Časovnih blokov bo pet, v posameznem dnevu pa lahko nastopijo trije različni časovni bloki. Najdražji bo časovni blok 1, ta se pojavi le v višji sezoni, najcenejša bo uporaba omrežja v časovnem bloku 5, ki nastopi le v nižji sezoni.

Cena omrežnine bo v večji meri odvisna od moči. V primerjavi z dosedanjo delitvijo stroškov omrežja gre torej za večjo stroškovno obremenitev tarifnih postavk za moč v primerjavi s tarifno postavko za energijo.

Nov tarifni sistem prinaša še dve novosti, dogovorjeno in presežno moč.

## 2.1 Dogovorjena moč

Dogovorjena moč pomeni vnaprej določeno obračunsko moč uporabnika sistema. Izhajala bo iz preteklih navad odjema električne energije in bo določena individualno za posamezno merilno mesto, vendar le tam, kjer so omogočene 15-minutne meritve. Za merilna mesta brez 15-minutnih meritev obračunsko moč predstavlja odstotek glede na priključno moč, ki je določen v Aktu [1].

Dogovorjeno obračunsko moč uporabnika sistema bo določil elektrooperater za posamezni časovni blok za prihodnje koledarsko leto in z njo seznanil uporabnika sistema z računom za omrežnino. Odjemalec bo lahko to vnaprej določeno dogovorjeno obračunsko moč spremenil glede na svojo pričakovano porabo. Če bo presodil, da lahko spremeni ali razporedi svojo porabo in uporablja omrežje v času manjše obremenitve omrežja, ko so tarifne postavke omrežnine nižje, bo neposredno vplival na svoje stroške omrežnine in si znižal mesečni račun za omrežnino. Sprememba dogovorjene moči je smiselna, če pretekli podatki, na podlagi katerih je bila izračunana dogovorjena moč, ne odražajo režima odjema v prihodnosti.

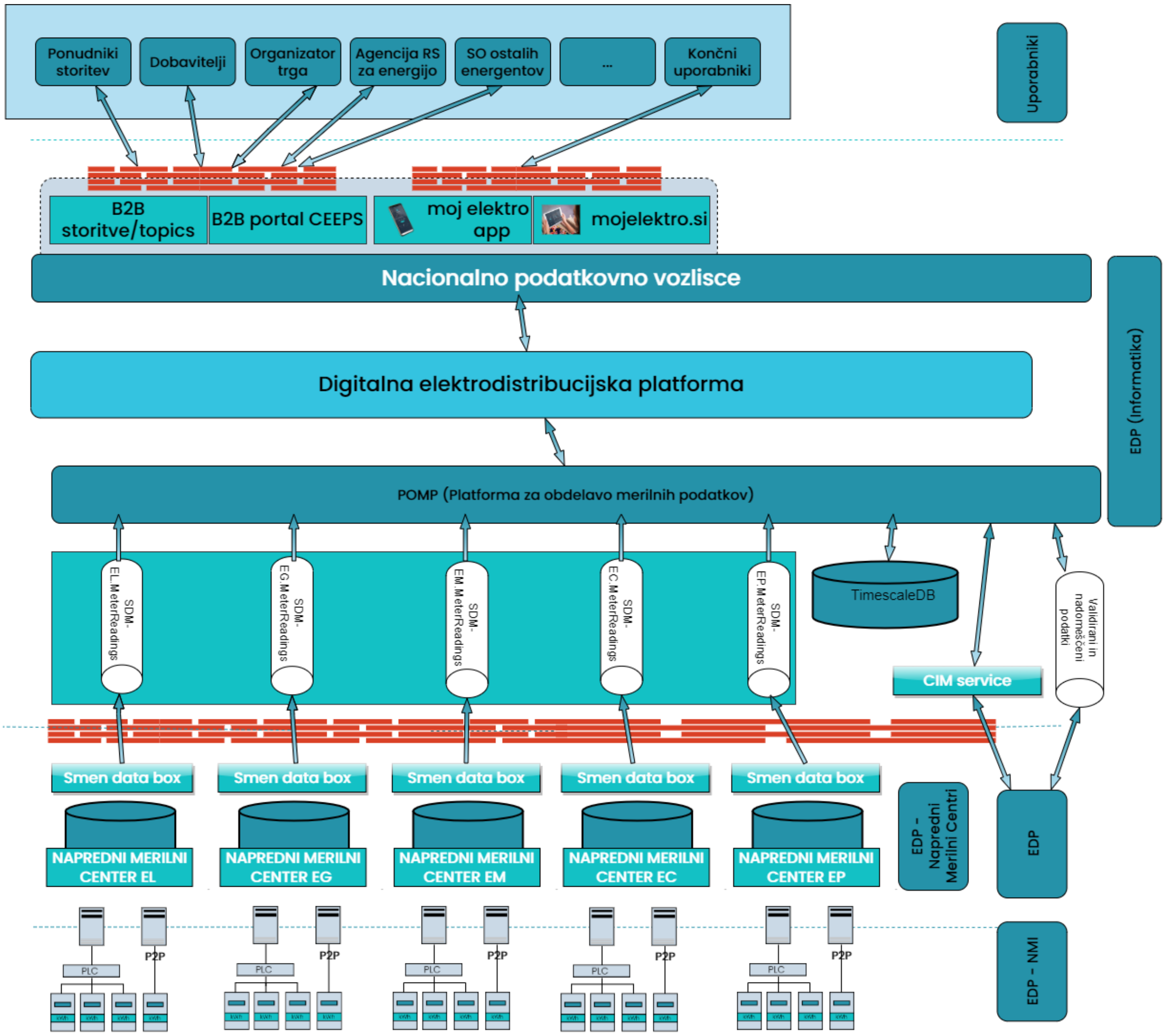
### 2.2 Presežna moč

Če dosežena moč (to je izmerjena moč odjema nad dogovorjeno obračunsko močjo) uporabnika sistema v posameznem časovnem bloku znotraj obračunskega meseca preseže dogovorjeno obračunsko moč, se uporabniku sistema za vsako 15-minutno prekoračitev obračuna omrežnina za presežno moč.

Zaradi obveznosti, ki jih nalagajo te zakonodajne zahteve, smo na Informatiki, d.o.o. sredi obsežnega projekta prenove obračunskega sistema. Ta projekt zajema števine pomembne vidike, med katerimi je ena izmed ključnih komponent platforma za obdelavo merilnih podatkov (POMP). Prav validirani 15-minutni podatki predstavljajo temelj novega obračunskega sistema.

## 3 Arhitektura platforme POMP

Več nivojsko arhitekturo platforme POMP prikazuje slika 2. Na najnižjem nivoju so merilni centri elektrodistribucijskih družb (Elektro Celje, Elektro Ljubljana, Elektro Maribor, Elektro Gorenjska in Elektro Primorska), ki pridobivajo merilne podatke iz merilnih naprav odjemalcev prek programabilnih logičnih kontrolnikov (PLC – Programmable Logic Controller) ali s protokoli P2P (Peer-to-Peer). Merilni podatki se nato po mehanizmu potiskanja (»push«) v rezinah prenesejo v centralno podatkovno bazo PostgreSQL TimescaleDB [2], ki je vzpostavljena na infrastrukturi Informatike.



Slika 2: Arhitektura platforme POMP.

### 3.1 Podatkovna baza TimescaleDB

TimescaleDB je odprtokodna podatkovna baza, ki je zasnovana posebej za učinkovito shranjevanje, upravljanje in analizo časovnih vrst podatkov, kakršni so tudi merilni podatki. Temelji na odprtokodni podatkovni bazi PostgreSQL, vendar dodaja razširjene funkcionalnosti za delo s časovnimi vrstami. Ena od ključnih prednosti TimescaleDB je možnost skaliranja in upravljanja z velikimi količinami podatkov. S kombinacijo PostgreSQL-ove zanesljivosti in razširjenih funkcionalnosti za delo s časovnimi vrstami TimescaleDB omogoča učinkovito shranjevanje in hitro izvajanje kompleksnih poizvedb nad časovnimi podatki. TimescaleDB uporablja particioniranje podatkov, kar omogoča deljenje podatkov v logične enote glede na določen časovni interval. To omogoča optimizacijo poizvedb in hitrejše iskanje podatkov glede na časovni indeks. Prav tako podpira stiskanje podatkov, kar zmanjšuje porabo prostora na disku. Poleg tega TimescaleDB ohranja skladnost s standardnim SQL-om, kar olajšuje prehod in uporabo. Integracija s široko paleto orodij in knjižnic, ki so na voljo za PostgreSQL, omogoča fleksibilnost in razširljivost pri razvoju aplikacij. Zaradi vseh naštetih prednosti smo se odločili, da bomo TimescaleDB uporabljali za centralno shranjevanje vseh merilnih podatkov na platformi POMP – gre za okoli 3 milijarde zapisov na mesec. Časovne tabele imajo dvodimenzionalno (»flat«) strukturo in vključujejo stolpce za stanja vseh merilnih registrov.

### 3.2 Pretočna platforma Apache Kafka

Prenos merilnih podatkov iz merilnih centrov v centralno bazo POMP se izvaja po »push« sistemu, in sicer na dnevnem nivoju. Za vsako merilno mesto se prenašajo časovno odvisni 15-minutni podatki o merjeni moči za štiri kvadrante, ki pokrivajo prejeto, oddano, delovno in jalovo energijo ter vse tarifne skupine. Posamezna časovna značka določa konec 15-minutnega intervala in odraža količino prejete ali oddane energije v tem intervalu.

Podatkovne strukture za prenos sporočil so standardizirane po modelu CIM (Common Information Model) oziroma IEC (International Electrotechnical Commission) 61968 [3] in IEC 61970 [4, 5], ki je temeljni referenčni model za področje energetike. Sporočila o meritvah so ovita s CIM ovojnico in jih potiskamo prek odprtokodne porazdeljene pretočne platforme Apache Kafka [6], ki smo jo izbrali iz naslednjih razlogov:

1. **Visoka zmogljivost in skalabilnost:** Kafka je zasnovana za obvladovanje velikega obsega podatkov in visoke stopnje pretoka. Omogoča hitro in učinkovito obdelavo podatkov ter je enostavno razširljiva in prilagodljiva platforma.
2. **Vzdržljivost in odpornost na napake:** Podatki se shranjujejo v shrambo (na disk), kar preprečuje izgubo podatkov v primeru napak ali izpadov sistema. Kafka omogoča tudi replikacijo sporočil med vozlišči, kar povečuje odpornost sistema na napake.
3. **Realnočasna obdelava podatkov:** Kafka je idealna platforma za obdelavo in posredovanje podatkov v realnem času. Omogoča hitro prenašanje podatkov med aplikacijami in sistemskimi komponentami skoraj brez zamika, kar je ključno za scenarije, ki zahtevajo takojšnjo odzivnost.
4. **Večopravilnost:** Kafka omogoča enostavno integracijo in povezovanje z drugimi orodji in platformami, kar bomo v prihodnosti izkoriščali tudi na drugih projektih. Kafka ima široko podporo in razvejano skupnost, kar pomeni, da obstaja veliko orodij, knjižnic in integracij, ki jih je možno uporabiti pri nadaljnjem razvoju in upravljanju sistema.
5. **Struktura sporočil:** Kafka je zasnovana na principu objavljaj-naročaj (publish-subscribe), kar omogoča enostavno upravljanje in usmerjanje sporočil med večimi odjemalci. To je še posebej koristno v našem okolju, kjer zahtevamo distribuirano obdelavo in pripravo podatkov ter različne tokove sporočil, ki se hranijo v ločenih temah (»topics«) za posamezne merilne centre.

V okviru projekta smo se odločili za uporabo platforme Confluent [7]. To je komercialna različica platforme Apache Kafka, ki nam je olajšala upravljanje, nadzor in integracije.

### 3.3 Vmesnik SmenDataBox

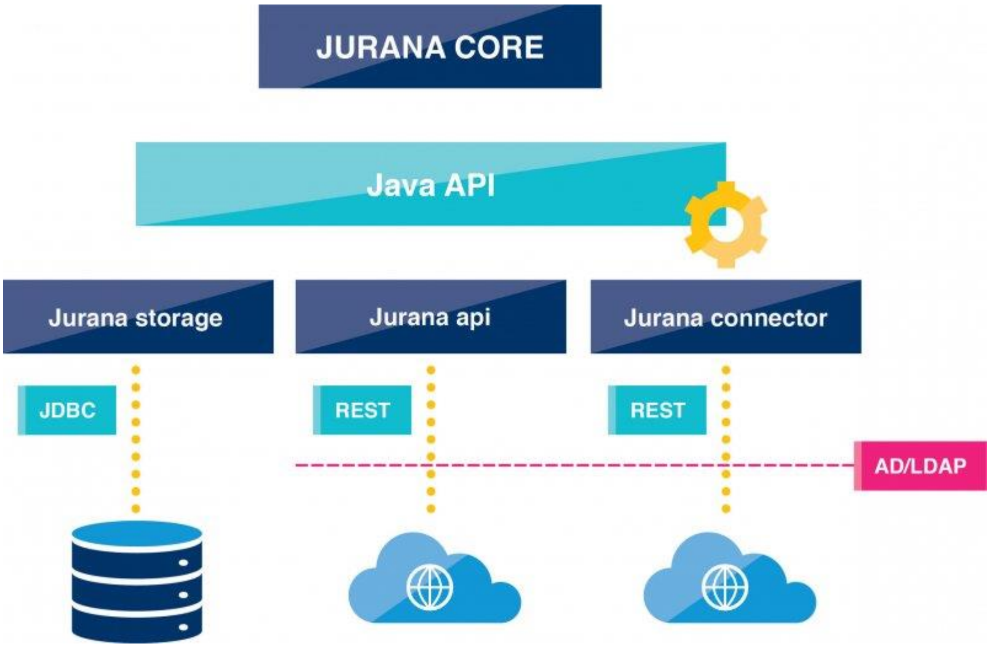
Vmesnik med merilnimi centri elektrodistribucijskih podjetij in Kafko je sistem SmenDataBox, ki temelji na mikrostoritveni arhitekturi ter tehnologijah .NET Core [8] in Vue.js [9]. SmenDataBox ovije podatke, ki pridejo v merilne centre, v CIM sporočila in jih odloži na platformo Kafka. Sporočilni tokovi se z namenom doseganja neodvisnosti prenašajo in hranijo v petih ločenih temah, tako da vsaka tema pripada merilnemu centru posamezne elektrodistribucijske družbe. Dodatne podrobnosti o sistemu SmenDataBox so predstavljene v četrtem poglavju prispevka.

### 3.4 Storitveno vodilo Jurana

Platforma podpira in implementira tudi integracije z več zunanjimi in zalednimi sistemi in aplikacijami. Namen teh integracij je omogočiti poslovanje in izmenjavo podatkov na nivojih B2B in B2C ter zagotoviti enotno vstopno točko in funkcionalnosti za različne deležnike na elektroenergetskem trgu. Ti deležniki so dobavitelji, ponudniki storitev, Agencija za energijo, organizator trga, uporabniki (referenti) v elektrodistribucijskih družbah, končni uporabniki (odjemalci in proizvajalci elektrike) in drugi. Posežejo lahko po različnih mehanizmih za izmenjavo energetskih podatkov ter sistemih in aplikacijah za dostop in obdelavo teh podatkov, ki jih podrobneje opisujemo v šestem poglavju prispevka.

Za integracije z zunanjimi sistemi smo na Informatiki razvili lastno napredno in zmogljivo storitveno vodilo Jurana. Le-to je zasnovano tako, da pospeši razvoj in nadgradnjo integracij ter jih naredi bolj fleksibilne. Gre za enotno ogrodje za podporo izvajanju integracijskih postopkov, ki omogoča dober pregled nad dogajanjem v procesih izvajanja integracijskih scenarijev. V ta namen smo razvili ločen sistem za spremljanje in ponovno proženje teh scenarijev – t.i. upravitelja integracij. V primeru izjem ali napak na ponornih sistemih lahko integracijske scenarije na zelo enostaven način tudi ponovimo. Vpeljava novih rešitev in povezovanje teh rešitev na obstoječe sisteme sta tako precej preprosta, predvsem pa je znatno olajšano vzdrževanje in sprotno odpravljanje napak v postopkih v produkcijskem okolju. Platforma je implementirana v programskem jeziku Java.

S platformo Jurana je integriranih že več kot 30 različnih informacijskih sistemov. Implementiranih je okoli 500 integracijskih scenarijev, na mesečnem nivoju pa se na B2B in B2C kanalih proži več kot 250.000 integracijskih postopkov.



**Slika 3: Večnivojska arhitektura storitvenega vodila Jurana.**

Večnivojska platformo Jurana sestoji iz več šibko sklopljenih modulov, ki jih prikazuje slika 3. Jurana Core je programska knjižnica, ki implementira algoritme za integracijo sistemov, upravljanje integracij in upravljenje z integracijskimi podatki, ki zajemajo tako integracijske podatkovne strukture kakor tudi metapodatke o stanju izvedbe integracij. Do jedra dostopamo z javanskim aplikacijskim programskim vmesnikom, ki omogoča široko uporabnost in neodvisnost od izvajalne/strojne platforme. Na ta vmesnik se povezujejo štirje moduli:

- Jurana Storage (modul za shranjevanje podatkov), ki z uporabo vmesnika JDBC (Java DataBase Connectivity) persistira in upravlja integracijske podatke v relacijski podatkovni bazi;
- Jurana API (aplikacijski programski vmesnik), ki ga različni integrirani informacijski sistemi, ki se izvajajo v oblaku ali v internih poslovnih okoljih, uporabijo, da lahko po standardu REST API (REpresentational State Transfer) izmenjujejo podatke in sinhronizirajo procesne storitve v sklopu integracijskih scenarijev;
- Jurana Connector (povezovalni modul), ki ga različni integrirani informacijski sistemi, ki se izvajajo v oblaku ali v internih poslovnih okoljih, uporabijo, da lahko po standardu REST API vzpostavijo in upravljajo medsebojne povezave;
- modul za upravljanje identitet, pooblastil in dostopov (AD – Active Directory, LDAP – Lightweight Directory Access Protocol), ki zagotavlja ključne varnostne mehanizme, t.j. celovitost, razpoložljivost in zaupnost podatkov.

# 4 Pridobivanje in prenos podatkov v platformo POMP

Prenos podatkov iz sistemov HES (Head End System) elektrodistribucijskih podjetij v platformo POMP poteka na dva načina:

- s sprotnim polnjenjem podatkov (push mehanizem) in
- s pridobivanjem zgodovinskih podatkov.

Sprotno polnjenje podatkov preko push mehanizma iz merilnih centrov deluje v novi različici sistema SmenDataBox preko platforme Kafka, kamor se sporočila vpisujejo v CIM strukturi. Push mehanizem deluje na način, da se podatek takoj, ko se vpiše v merilni center, posreduje tudi v Kafko. V platformi POMP imamo 2 aplikacijska strežnika Open Liberty povezana v grozd (cluster), na vsakem od njih pa obdeluje sporočila 20 delavcev. Povprečna obdelava sporočila traja 20 ms. Na dnevnem nivoju to pomeni cca. 2.000.000 sporočil za približno milijon merilnih mest, saj se prenašajo 15-minutni podatki in dnevna stanja v ločenih sporočilih.

Za pridobivanje zgodovinskih podatkov se je uporabila obstoječa funkcionalnost sistema SmenDataBox, ki omogoča asinhrono oddajo poizvedb za merilne podatke. Zahteve se vračajo preko protokola AMQP (Advanced Message Queuing Protocol) v MQ (Messaging Queue), od koder jih sistem POMP prevzema in obdeluje. V prvi fazi se je prenos merilnih podatkov vršil za obdobje celega leta 2022 in nato še za prvo polovico leta 2023. Velik izziv je bilo ustrezno načrtovanje obsega povpraševanj, saj obstoječi sistem deluje po principu FIFO (First In, First Out), uporablja pa se tudi za druge deležnike na trgu električne energije, ki morajo do določene ure prejeti sveže 15-minutne podatke. Načrtovanje se je zato izvajalo za vsak dan posebej, saj se je zaradi vseh aktivnosti v merilnih centrih morala količina prilagajati.

V tabeli 1 so prikazani podatki, ki se prenašajo v platformo POMP.

**Tabela 1: Podatki, ki se prenašajo v platformo POMP.**

| Naziv | Oznaka | Tip, smer | Perioda | Opis |
|---|---|---|---|---|
| Prejeta 15-minutna delovna energija | A+ | delovna prejem | 15 min | 15 minutna energija, A+, kWh |
| Oddana 15-minutna delovna energija | A- | delovna oddaja | 15 min | 15 minutna energija, A-, kWh |
| Prejeta 15-minutna jalova energija | R+ | jalova prejem | 15 min | 15 minutna energija, R+, kVArh |
| Oddana 15-minutna jalova energija | R- | jalova oddaja | 15 min | 15 minutna energija, R-, kVArh |
| Prejeta delovna energija ET | A+_T0 | delovna prejem ET | 24 h | 24 urno stanje, A+, kWh, T0 |
| Prejeta delovna energija VT | A+_T1 | delovna prejem VT | 24 h | 24 urno stanje, A+, kWh, T1 |
| Prejeta delovna energija MT | A+_T2 | delovna prejem MT | 24 h | 24 urno stanje, A+, kWh, T2 |
| Oddana delovna energija ET | A-_T0 | delovna oddaja ET | 24 h | 24 urno stanje, A-, kWh, T0 |
| Oddana delovna energija VT | A-_T1 | delovna oddaja VT | 24 h | 24 urno stanje, A-, kWh, T1 |
| Oddana delovna energija MT | A-_T1 | delovna oddaja MT | 24 h | 24 urno stanje, A-, kWh, T2 |
| Prejeta jalova energija ET | R+_T0 | jalova prejem ET | 24 h | 24 urno stanje, R+, kVArh, T0 |
| Prejeta jalova energija VT | R+_T1 | jalova prejem VT | 24 h | 24 urno stanje, R+, kVArh, T1 |
| Prejeta jalova energija MT | R+_T2 | jalova prejem MT | 24 h | 24 urno stanje, R+, kVArh, T2 |
| Oddana jalova energija ET | R-_T0 | jalova oddaja ET | 24 h | 24 urno stanje, R-, kVArh, T0 |
| Oddana jalova energija VT | R-_T1 | jalova oddaja VT | 24 h | 24 urno stanje, R-, kVArh, T1 |
| Oddana jalova energija MT | R-_T2 | jalova oddaja MT | 24 h | 24 urno stanje, R-, kVArh, T2 |

Vrednosti se vodijo posebej za 15-minutno periodo, za katero se shranjujejo količine energije, in posebej za dnevno periodo, kjer se shranjujejo števčna stanja. Za vsak odčitek se shranjuje tudi kvaliteta podatkov. V tabeli 2 so prikazani vsi možni statusi kvalitete podatkov. Določene statuse pridobimo že iz merilnih centrov, določene pa izvede mehanizem validacije in nadomeščanja v platformi POMP.

**Tabela 2: Statusi kvalitete podatkov.**

| Koda statusa odčitka | Opis statusa odčitka |
|---|---|
| | OK |
| 1.4.0 | Obračunski reset |
| 1.4.9 | Premik / sprememba časa |
| 1.4.1 | Preliv |
| 1.1.7 | Usodna napaka |
| 1.5.257 | Napačen podatek |
| 1.5.259 | Neprebran podatek |
| 1.2.32 | Izpad napetosti |
| 1.2.32 | Povrnitev napetosti |
| 3.10.1 | Sprememba vrednosti |
| 1.4.8 | Izbris pomnilnika |
| 1.4.6 | Sprememba parametrov v števcu |
| 1.8.0 | Uvoženi podatki |
| 1.4.131 | Podatki iz ročnih terminalov |
| 3.0.0 | Merjena vrednost |
| 3.8.1 | Končno ocenjena vrednost |
| 3.8.0 | Ocenjena vrednost |
| 3.5.259 | Manjkajoča vrednost |
| 3.7.3 | Zavrnjena vrednost |
| 3.10.0 | Začasna vrednost |

Trenutno statistiko podatkov v bazi sistema POMP povzema tabela 3. Podatki so za obdobje od 1. 1. 2022 do 1. 6. 2023. Potrebno je izpostaviti, da se po vklopljeni kompresiji poraba prostora zmanjša za več kot 90 %.

**Tabela 3: Statistika podatkov v bazi sistema POMP.**

| Vrsta odčitkov | Št. zapisov | Velikost podatkov [GB] |
|---|---|---|
| Stanja | 422.633.562 | 323 |
| 15 minutne meritve | 41.000.000.000 | 9.676 |

## 5 Validacija in nadomeščanje podatkov

Ključna funkcionalnost platforme POMP je, da poleg prenosa podatkov iz merilnih centrov na dnevnem nivoju omogoča tudi izvajanje validacije in nadomeščanja podatkov. Prav tako omogoča poizvedovanje po validiranih merilnih podatkih za potrebe interne analitike, izvozov, agregacij in drugih operacij. Dostop do validiranih in nadomeščenih podatkov je realiziran prek implementiranih vmesnikov. Na voljo je elektrodistribucijskim podjetjem, dobaviteljem za namen njihovega poslovanja in delovanja na elektroenergetskem trgu ter za potrebe pridobivanja podatkov iz sistema eIS in modula za obračun omrežnine. Metodologija validacije in nadomeščanja temelji na algoritmih, ki jih definirajo sistemska navodila, ter zajema postopke določitve upravičenosti nadomeščanja manjkajočih vrednosti, detekcije in čiščenja vrednosti izven območja značilk ter izračuna nadomestnih vrednosti.

Implementiran je tudi koncept dogovorjene moči, ki se za odjemalce s priključno močjo nad 43 kW za posamezni časovni blok določi na podlagi uporabnikovih doseženih 15-minutnih moči posameznega časovnega bloka v zadnjih 12 mesecih pred določitvijo dogovorjene obračunske moči. Za odjemalce s priključno močjo, enako ali manjšo od 43 kW, pa se dogovorjena obračunska moč določi za posamezen časovni blok od 1 do 4 kot povprečje treh konic posameznega časovnega bloka v obdobju zadnje višje sezone pred določitvijo dogovorjene obračunske moči.

Za izračun dogovorjenih moči se validacija in čiščenje podatkov izvajata po spodnjem algoritmu.

**Algoritem 1: Validacija in čiščenje podatkov za izračun dogovorjenih moči.**

V opazovanem obdobju, potrebnem za izračun dogovorjenih moči, se je izračunala delta dnevnih stanj in seštevek 15-minutnih podatkov (energija A+).

V kolikor je bila razlika med delto stanj in seštevki energij v okvirju tolerančne meje 2 %, so se meritve vzele v obdelavo za nadaljnje validacije.

V kolikor je razlika presegala tolerančno mejo, se je izvedla kontrola razlike med delto stanj in seštevki energij na dnevnem nivoju. Meritve za vse dni, kjer je razlika presegala tolerančno mejo 2 %, so se označile s statusom 3.7.3 – Zavrnjena vrednost.

Vsaka 1. meritev brez neustreznega statusa, ki je sledila zadnji meritvi s statusom 1.5.259, se je označila s statusom 3.7.3 – Zavrnjena vrednost.

Vse meritve, ki so vsebovale status 1.4.0 – Obračunski reset ali 1.4.9 – Premik / sprememba časa so se dodatno označile s statusom 3.7.3 – Zavrnjena vrednost.

Izločile so se naslednje meritve:

- Meritve, ki so vsebovale status 1.5.257 – Napačen podatek
- Meritve, ki so vsebovale status 1.5.259 – Neprebran podatek
- Meritve, ki so vsebovale status 3.10.1 – Sprememba vrednosti
- Meritve, ki so vsebovale status 3.7.3 – Zavrnjena vrednost
- Meritve, ki kjer je moč (P+ 15 min delovna moč, prejem iz omrežja) presegala 2-kratnik priključne moči

Izmed vseh ostalih meritev se je nato izračunal % kvalitete: št. ustreznih meritev / št. 15-minutnih intervalov opazovanega obdobja * 100.

Nato se je iz ustreznih meritev v vseh časovnih blokih poiskalo povprečje 3 konic (P+ 15 min delovna moč, prejem iz omrežja) (za MM s priključno močjo <= 43 kW) in najvišje konice posameznega časovnega bloka (za MM s priključno močjo > 43 kW), ki so bile nato osnova za izračun dogovorjene moči.

Dogovorjena moč se je nato izračunala po definiranih pravilih iz tabele 4.

**Tabela 4: Pravila za izračun dogovorjene moči.**

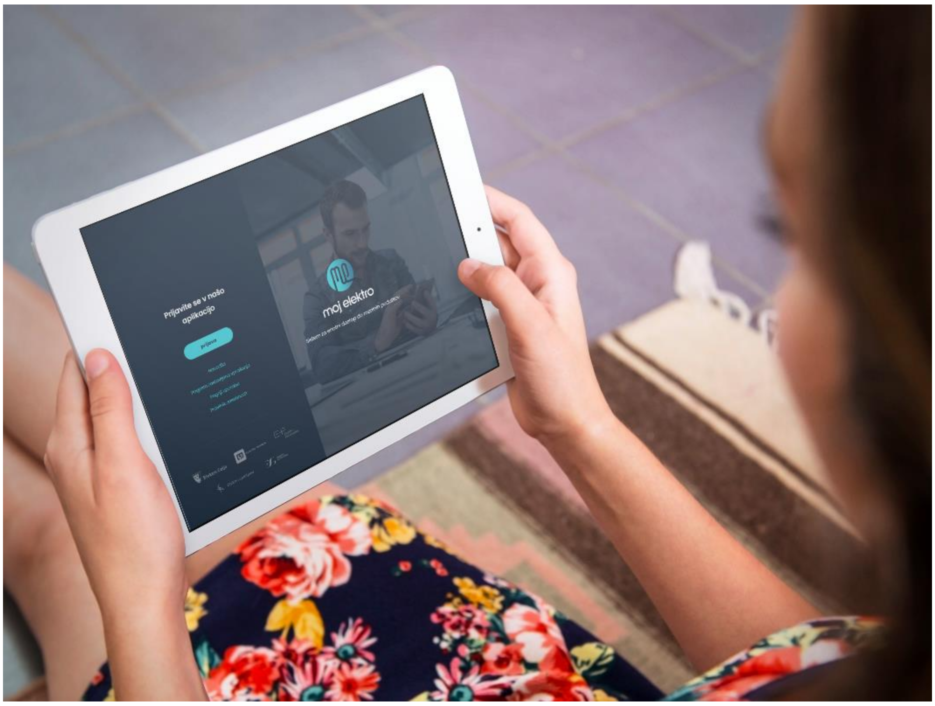
| Število faz | Priključna moč | Uporabljeno obdobje meritev | Izračun za posamezen blok |
|---|---|---|---|
| 1 | P <= 43 kW | 1. 11. preteklo leto do 28./29. 2. tekoče leto | – Povprečje treh konic posameznega bloka<br>– Dogovorjena moč za časovni blok b+1 mora biti večja ali enaka dogovorjeni moči za časovni blok b, v kolikor je vrednost manjša, se vzame izračunana vrednost predhodnega bloka<br>– Minimalno 31 % priključne moči<br>– Ne manj kot 2 kW<br>– Ne več kot priključna moč |
| 3 | P <= 17 kW | 1. 11. preteklo leto do 28./29. 2. tekoče leto | – Povprečje treh konic posameznega bloka<br>– Dogovorjena moč za časovni blok b+1 mora biti večja ali enaka dogovorjeni moči za časovni blok b, v kolikor je vrednost manjša, se vzame izračunana vrednost predhodnega bloka<br>– Minimalno 27 % priključne moči<br>– Ne manj kot 3,5 kW<br>– Ne več kot priključna moč |
| 3 | 17 kW < P <= 43 kW | 1. 11. preteklo leto do 28./29. 2. tekoče leto | – Povprečje treh konic posameznega bloka<br>– Dogovorjena moč za časovni blok b+1 mora biti večja ali enaka dogovorjeni moči za časovni blok b, v kolikor je vrednost manjša, se vzame izračunana vrednost predhodnega bloka<br>– Minimalno 34 % priključne moči<br>– Ne več kot priključna moč |
| 3 | P > 43 kW | 1. 5. preteklo leto do 30. 4. tekoče leto | – Maksimalna konica posameznega bloka<br>– Dogovorjena moč za časovni blok b+1 mora biti večja ali enaka dogovorjeni moči za časovni blok b, v kolikor je vrednost manjša, se vzame izračunana vrednost predhodnega bloka<br>– Minimalno 25 % priključne moči<br>– Ne več kot priključna moč |

## 6 Distribucija podatkov in integracija platforme POMP

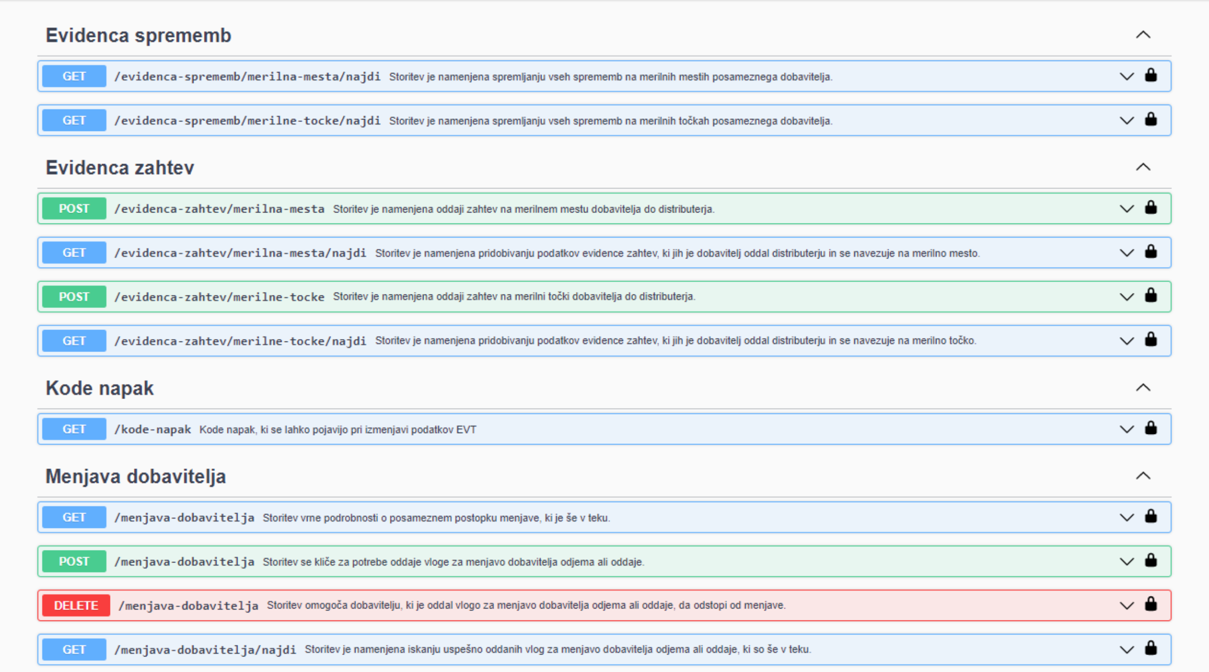
Platforma POMP je zasnovana tako, da omogoča integracijo z različnimi dodatnimi vmesniki, kar zagotavlja fleksibilnost in razširljivost sistema. Vključuje integracijo s tremi ključnimi sistemi: informacijskim sistemom eIS, portalom Moj elektro in portalom CEEPS (Centralni elektroenergetski portal Slovenije).

Informacijski sistem eIS je namenjen podpori obračuna električne energije in poslovnih procesov priključevanja odjemalcev v elektroenergetski domeni. S platformo POMP je povezan prek ustrezno razvitega vmesnika REST API (Representational State Transfer), ki omogoča komunikacijo med obema sistemoma. To vzpostavlja pretok podatkov in informacij med sistemoma POMP in eIS, kar zagotavlja nemoteno izvajanje obračunov in drugih poslovnih procesov.

Portal Moj elektro, ki je dostopen na naslovu https://mojelektro.si in je razviden s slike 4, ter portal CEEPS pa predstavljata enotno vstopno točko (EVT) za zunanje deležnike na elektroenergetskem trgu na nivoju uporabnikov in poslovnem nivoju. Portal Moj elektro omogoča B2C integracijo z odjemalci in proizvajalci električne energije, medtem ko je CEEPS namenjen B2B integraciji z regulatorji, agregatorji, dobavitelji in drugimi deležniki na trgu. Oba portala sta integrirana v platformo POMP prek ustrezno razvitih vmesnikov REST API, ki so ponazorjeni na sliki 5 in omogočajo dvosmerno komunikacijo med sistemoma. To je osnova za prenos podatkov, naročil in drugih informacij med sistemom POMP in portaloma Moj elektro ter CEEPS.



Slika 4: Portal Moj elektro[1].



Evidenca sprememb

GET `/evidenca-sprememb/merilna-mesta/najdi` Storitev je namenjena spremljanju vseh sprememb na merilnih mestih posameznega dobavitelja.

GET `/evidenca-sprememb/merilne-tocke/najdi` Storitev je namenjena spremljanju vseh sprememb na merilnih točkah posameznega dobavitelja.

Evidenca zahtev

POST `/evidenca-zahtev/merilna-mesta` Storitev je namenjena oddaji zahtev na merilnem mestu dobavitelja do distributerja.

GET `/evidenca-zahtev/merilna-mesta/najdi` Storitev je namenjena pridobivanju podatkov evidence zahtev, ki jih je dobavitelj oddal distributerju in se navezuje na merilno mesto.

POST `/evidenca-zahtev/merilne-tocke` Storitev je namenjena oddaji zahtev na merilni točki dobavitelja do distributerja.

GET `/evidenca-zahtev/merilne-tocke/najdi` Storitev je namenjena pridobivanju podatkov evidence zahtev, ki jih je dobavitelj oddal distributerju in se navezuje na merilno točko.

Kode napak

GET `/kode-napak` Kode napak, ki se lahko pojavijo pri izmenjavi podatkov EVT

Menjava dobavitelja

GET `/menjava-dobavitelja` Storitev vrne podrobnosti o posameznem postopku menjave, ki je še v teku.

POST `/menjava-dobavitelja` Storitev se kliče za potrebe oddaje vloge za menjavo dobavitelja odjema ali oddaje.

DELETE `/menjava-dobavitelja` Storitev omogoča dobavitelju, ki je oddal vlogo za menjavo dobavitelja odjema ali oddaje, da odstopi od menjave.

GET `/menjava-dobavitelja/najdi` Storitev je namenjena iskanju uspešno oddanih vlog za menjavo dobavitelja odjema ali oddaje, ki so še v teku.

Slika 5: REST storitve enotne vstopne točke.

---

[1] © Informatika d.o.o., slika izdelana in objavljena v skladu z licenco Envato Placeit Pty Ltd

Uporabniki, tako poslovni sistemi kot tudi končni uporabniki, lahko dostopajo do vmesnikov preko ustrezno zagotovljene avtorizacije. Končni uporabniki pridobijo dostop do API-jev z žetonom, ki je vezan na enotno avtorizacijo v portalu Moj elektro, ki je realizirana prek sistemov SI-PASS [10] in Rekono [11]. To zagotavlja varnost in nadzor dostopa do API-jev ter ohranja zasebnost in varnost uporabnikov.

Sistemi eIS, Moj elektro in CEEPS so razviti na platformi Java EE, kar omogoča uporabo širokega nabora programskih vmesnikov za aplikativne integracije. Poleg tega zagotavljajo tudi sodobne spletne uporabniške vmesnike za različne tipe končnih uporabnikov, vključno z zaposlenimi v elektrodistribucijskih družbah (informacijski sistem eIS), porabniki in proizvajalci električne energije (portal Moj elektro) ter poslovnimi subjekti (portal CEEPS). To omogoča enostavno uporabo in interakcijo s platformo ter ustvarja prijazno uporabniško izkušnjo za vse vključene strani.

Platforma POMP zagotavlja tudi vmesnike za dostop do podatkov za ostale zaledne sisteme elektrodistribucijskih podjetij. To pomeni, da so podatki, ki jih obdeluje (centralno validira in nadomešča) in zajema platforma POMP, preko vmesnikov na voljo drugim sistemom elektrodistribucijskih podjetij (npr. velepodatkovnim platformam), kjer se nadalje obdelujejo in uporabljajo za operativne in poslovne procese.

Povratna integracija podatkov tako omogoča celovit vpogled v stanje in dogajanje v elektrodistribucijskih podjetjih. Zaledni sistemi lahko spremljajo in analizirajo podatke, ki jih prejemajo iz platforme POMP, ter na njihovi podlagi izvajajo različne analize, poročila in ukrepe. S tem se omogoča boljše upravljanje, sprejemanje odločitev in izvajanje operativnih procesov v elektrodistribucijskih podjetjih.

Povratna integracija podatkov med platformo POMP in zalednimi sistemi elektrodistribucijskih podjetij je ključna za zagotavljanje celovitega in povezanega elektronskega ekosistema. S tem se dosežejo višja učinkovitost, zanesljivost in uspešnost pri upravljanju elektroenergetskega sistema ter izvajanju storitev za odjemalce in deležnike na elektroenergetskem trgu.

## 7 Zaključek

Vpeljani metodološki in tehnološki koncepti platforme POMP omogočajo, da lahko Informatika kot osrednji ponudnik IT storitev na energetskem trgu izvaja vse potrebne izračune (ki so npr. vezani na inicialno dogovorjeno moč) in skrbi za celoten obračun. Prav tako zagotavljajo visoko stopnjo digitalizacije poslovnih procesov, napredne integracijske mehanizme in vpeljavo poslovnih modelov prožnosti. Predvsem pa omogoča platforma POMP z uvedbo naprednih podatkovnih in pretočnih rešitev skalabilno obdelavo in hranjenje velepodatkov, ki jih lahko na neposreden in koristen način uporabimo tudi v številnih scenarijih na osnovi napredne podatkovne analitike in strojnega učenja.

## Literatura

[1] Akt o metodologiji za obračunavanje omrežnine za elektrooperaterje, Uradni list RS, št. 146/2022 z dne 25. 11. 2022, https://www.uradni-list.si/glasilo-uradni-list-rs/vsebina/2022-01-3624/akt-o-metodologiji-za-obracunavanje-omreznine-za-elektrooperaterje, obiskano 3. 7. 2023.

[2] https://www.timescale.com/, Informacije o podatkovni bazi TimeScaleDB, obiskano 5. 7. 2023.

[3] https://en.wikipedia.org/wiki/IEC_61968, Informacije o zbirki standardov CIM IEC 61968, obiskano 5. 7. 2023.

[4] https://en.wikipedia.org/wiki/IEC_61970, Informacije o zbirki standardov CIM IEC 61970, obiskano 21. 7. 2023.

[5] IEC 61970-301:2020+AMD1:2022 CSV Consolidated version, Energy management system application program interface (EMS-API) - Part 301: Common information model (CIM) base, International Electrotechnical Commission, https://webstore.iec.ch/publication/74467, obiskano 21. 7. 2023.

[6] https://kafka.apache.org/, Informacije o platformi Apache Kafka, obiskano 3. 7. 2023.

[7] https://www.confluent.io/, Informacije o platformi Confluent, obiskano 3. 7. 2023.

[8] https://en.wikipedia.org/wiki/.NET, Informacije o ogrodju .NET, obiskano 21. 7. 2023.

[9] https://en.wikipedia.org/wiki/Vue.js, Informacije o ogrodju Vue.js, obiskano 21. 7. 2023.

[10] https://www.si-trust.gov.si/sl/si-pass/, Spletna prijava in e-podpis SI-PASS, SI-TRUST, Državni center za storitve zaupanja, obiskano 22. 7. 2023.

[11] https://www.rekono.si/, Spletna stran rešitev Rekono, obiskano 22. 7. 2023.

# Nadgradnja podatkovnih jezer s pomočjo formata za shranjevanje Delta Lake

Martina Šestak, Muhamed Turkanović

Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko,
Maribor, Slovenija
martina.sestak@um.si, muhamed.turkanovic@um.si

Sodobni sistemi za upravljanje velepodatkov pogosto zbirajo (vele)podatke iz več različnih virov, pri čemer se podatki razlikujejo po stopnji strukturiranosti, količini, načinu in pogostosti zajemanja, itn. Ker arhitekture takšnih sistemov morejo hkrati zagotoviti paketno in pretočno obdelavo podatkov, so kot rešitev za učinkovito shranjevanje podatkov razvita podatkovna kolišča, ki kombinirajo možnosti podatkovnih skladišč in jezer. Ena pogosto uporabljanih rešitev za vzpostavitev podatkovnega kolišča na trgu je odprtokodna rešitev Delta Lake. V prispevku bomo podrobneje predstavili format Delta Lake ter analizirali glavne razlike v primerjavi z obstoječimi odprtimi formati za shranjevanje podatkov v podatkovnem kolišču, kot sta Apache Hudi in Iceberg. Lastnosti formata Delta Lake bomo tudi vključili v razpravo o novem vzorcu obdelave podatkov Delta ter ga bomo primerjali s trenutno razširjenima vzorcema za pretočno obdelavo podatkov Lambda in Kappa. Iz tehničnega vidika bomo uporabo formata Delta Lake prikazali na praktičnem primeru implementacije podatkovne platforme za integracijo agroživilskih podatkov. V primeru bomo prikazali, kako nadgraditi obstoječe podatkovno jezero s pomočjo formata Delta Lake. Podatke shranjene v podatkovnem jezeru bomo obdelali s pomočjo okvirja Spark, pri čemer bomo izpostavili prednosti in morebitne omejitve dela s formatom Delta Lake.

**Ključne besede:**
podatkovno kolišče
arhitektura velepodatkov
Delta Lake
pretočna obdelava podatkov
paketna obdelava podatkov
format za shranjevanje podatkov

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.21

# 1 Uvod

V svetu velikih podatkov in analitike velepodatkov nenehno iščejo inovativne rešitve za shranjevanje in upravljanje masovnih količin podatkov iz več različnih virov. Posledično so v zadnjih nekaj let predstavljene različne arhitekture s ciljem naslavljanja specifičnih izzivov pri upravljanju velepodatkov, kot so podatkovna skladišča in podatkovna jezera. Slednji dve arhitekturi pogosto najdemo v sodobnih velepodatkovnih sistemih, ki morejo bit zmožni ustrezno in učinkovito integrirati različno strukturirane podatke iz več virov, ki v ciljni sistem prihajajo kot paketi (angl. batches) ali podatkovni toki (angl. data streams), pri čemer je potrebno ohraniti tudi kakovost podatkov in razviti ustrezni pristop za njihovo analizo. V tem kontekstu so podatkovna skladišča predstavljena kot strukturirana rešitev za integracijo podatkov iz več podatkovnih baz, ki jih pogosto imajo organizacije, pri čemer se želi ohraniti konsistentnost oz. kakovost podatkov zaradi podatkovne analitike in podatkovno-vodenih odločitev na podlagi pripravljenih poročil. S pomočjo ETL (angl. Extract-Transform-Load) procesov, v katerih se surovi podatki podrobno počistijo in ustrezno transformirajo, se zagotovi, da so podatki shranjeni v podatkovno skladišče vedno strukturirani glede na predoločeno podatkovno shemo, kar močno olajša izvajanje poizvedb in analitičnih nalog. S druge strani pa so podatkovna jezera predstavljena ko rešitev v vlogi centralnega repozitorija surovih in različno strukturiranih podatkov iz različnih virov (npr. senzorji, dnevniške datoteke, satelitski posnetki, itn.). Shranjevanje podatkov v podatkovno jezero se izvaja po principu "shema ob branju" (angl. schema-on-read), kar pomeni, da podatkovna jezera ne zahtevajo, da so vhodni podatki pripravljeni na način, da ustrezajo predoločeni shemi. Posledično se podatki v podatkovno jezero shranjujejo v surovi obliki, kar močno izboljša učinkovitost pisanja v jezero. Shema se podatkom dodeli šele ob njihovem branju, ko določimo za kateri namen oz. poizvedbe jih potrebujemo. Zaradi tiste in drugih lastnosti so se podatkovna jezera pokazala kot ustrezna rešitev za izvajanje naprednih analiz podatkov, ki je hkrati tudi razširljiva v sodobnih infrastrukturah. Večletne izkušnje so pokazale, da izbira le ene arhitekture (podatkovno skladišče ali podatkovno jezero) ne zadostuje organizacijam, da bi popolnoma izpolnile njihove sodobne analitične potrebe in zahteve. Zaradi tega se v sodobnih informacijskih sistemov pogosto srečamo z dvostopenjsko arhitekturo [1], pri kateri se vsi podatki iz virov shranjujejo v podatkovnem jezeru in se potem le del teh podatkov hrani tudi v podatkovno skladišče v prečiščeni obliki. Implementacija takšne arhitekture prinaša možnost priprave poročil na podlagi strukturiranih podatkov v podatkovnem skladišču ter tudi analize s pomočjo strojnega učenja nad surovimi podatki shranjenimi v podatkovnem jezeru. Čeprav je takšna arhitektura razširljiva in zagotavlja kakovost podatkov, je tudi zahtevna za vzdrževanje, saj njena implementacija zahteva namestitev in nadzor več sistemskih komponent [2].

Kot prelomna zamisel se je v zadnjem desetletju pojavil pojem podatkovno kolišče (angl. data lakehouse), ki v osnovi združuje najboljše lastnosti podatkovnih jezer in podatkovnih skladišč znotraj le ene rešitve. Ta integrirani pristop podjetjem zagotavlja močno platformo za integracijo podatkov in napredno analitiko ter podpira sprejemanje odločitev, ki temeljijo na podatkih, kot nikoli prej. Na konceptualni ravni to pomeni, da se model podatkovnega skladišča uporabi nad podatkovnim jezerom, pri čemer ustrezne komponente skrbijo za podporo transakcijskega modela pri delu s podatki in uveljavljanje sheme pri shranjevanju podatkov v tabele zaradi zagotavljanja konsistentnosti in integritete podatkov. Ker je arhitektura podatkovnih kolišč precej nova, je praktičnih izkušenj z implementacijo ter potencialnimi prednosti in slabosti zelo malo. V članku bomo predstavili platformo Delta Lake, ki je ena izmed trenutno dostopnih tehnoloških rešitev za implementacijo podatkovnih kolišč. Osredotočili se bomo na nov format za shranjevanje Delta Lake, ki je predstavljen kot izboljšani format za shranjevanje namenjen ravno za podatkovna kolišča. V drugem delu članka bomo tudi prikazali način uporabe formata Delta Lake pri delu s podatki, ki so shranjeni v podatkovnem jezeru, ter kako se tisti format lahko uporabi pri nadgradnji obstoječega podatkovnega jezera zaradi izboljšave učinkovitosti upravljanja podatkov in izvajanja poizvedb nad podatki.

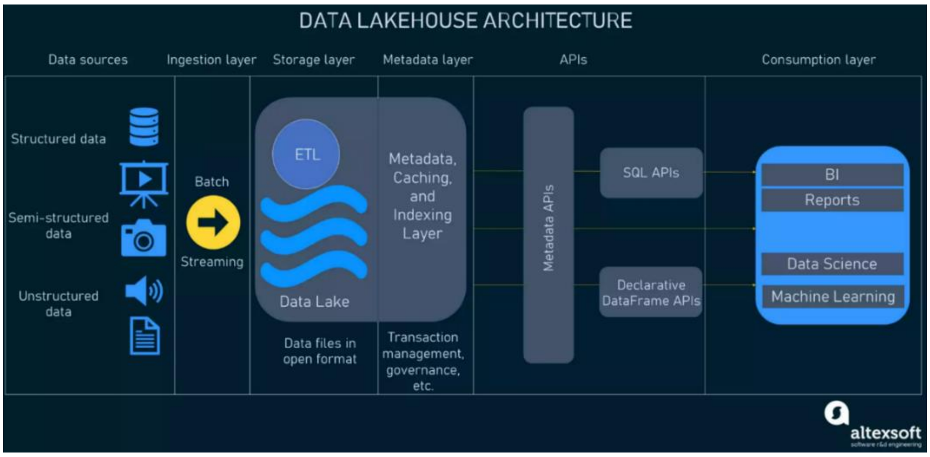
## 2 Podatkovno kolišče

Podatkovno kolišče predstavlja sodobno podatkovno arhitekturo, ki združuje prednosti podatkovnih jezer in podatkovnih skladišč v enovito platformo. Omogoča organizacijam shranjevanje surovih, strukturiranih in delno strukturiranih podatkov v njihovem izvornem formatu (kot pri podatkovnih jezerih), obenem pa podpira tradicionalno obdelavo strukturiranih podatkov in poizvedbe na podlagi jezika SQL (kot pri podatkovnih skladiščih). Rezultat je platforma, ki deluje kot edini vir resnice za vse vrste podatkov, ter omogoča brezhibno integracijo, obdelavo in analizo podatkov iz več virov.

V osnovi podatkovna kolišča kombinirajo nizke stroške in podporo za odprte formate shranjevanja podatkovnih jezer z visoko učinkovitostjo in transakcijskim pristopom k obdelavi podatkov podatkovnih skladišč [3, 4]. Pomembno je izpostaviti, da se takšni sistemi naslanjajo na nove formate za shranjevanje podatkov, kot so Delta Lake, Apache Hudi ali Apache Iceberg, ki so nastali kod nadgradnja tradicionalnih formatov za shranjevanje različno strukturiranih podatkov (npr. Avro, JSON), vendar s podporo za implementacijo transakcijskega modela ACID (angl. Atomicity, Consistency, Isolation, Durability) in drugih funkcionalnosti klasičnih podatkovnih baz, s katerimi se lahko zagotavlja konsistentnost podatkov. Na trgu se trenutno opaža trend nadgradnje podatkovnih jezer v podatkovna kolišča, saj slednja zagotavljajo višjo raven konsistentnosti podatkov, kot jo zagotavljajo podatkovna jezera, ki se v odsotnosti ustreznih mehanizmov dosti krat postopoma pretvorijo v podatkovna močvirja (angl. data swamp) [4].

Tako je arhitektura podatkovnega kolišča ponavadi sestavljena iz pet slojev (Slika 1), in sicer:

1. *Sloj zajema podatkov (angl. data ingestion layer)* – sloj odgovoren za zbiranje podatkov iz izvirnih sistemov v obliki paketov in/ali podatkovnih tokov;
2. *Sloj hrambe (angl. storage layer)* – sloj za nizkocenovno shranjevanje surovih in agregiranih podatkov kot objektov v določenem odprtem formatu, po navadi se kot rešitev na tem sloju uporablja podatkovno jezero;
3. *Sloj metapodatkov (angl. metadata layer)* – osrednji sloj podatkovnega kolišča, ki vsebuje poenoteni katalog z vsemi metapodatki o objektih shranjenih na sloju hrambe ter omogoča implementacijo transakcij po modelu ACID, upravljanja in uveljavljanja shem za agregirane podatke, predpomnjenja objektov iz jezera, indeksiranja za namen hitrejših poizvedb, itn.
4. *Sloj programskih vmesnikov (angl. API layer)* - sloj, ki vsebuje programske vmesnike (angl. Application Programming Interface, API) za uporabniški dostop do podatkov v kolišču za potrebe analiz;
5. *Sloj porabe (angl. consumption layer)* – sloj z različnimi orodji in aplikacijami za uporabo in vizualizacijo podatkov (npr. PowerBI).



**Slika 1. Konceptualni prikaz arhitekture podatkovnega kolišča**
Vir: [5].

Pri načrtovanju arhitekture podatkovnega kolišča je potrebno imeti v mislih, da more podatkovno kolišče biti zmožno obravnavati količino podatkov na ravni podatkovnih jezer, vendar z nizkimi stroški hrambe podatkov, ter podpirati možnost dostopa do podatkov s strani več uporabnikov preko različnih vmesnikov. Obenem pa more podatkovno kolišče zagotoviti izvedbo transakcij po modelu ACID zaradi zagotavljanja konsistentnosti podatkov, medtem ko sistem v celoti obdeluje podatke z visoko učinkovitostjo in je po potrebi razširljiv. Za arhitekturo podatkovnega kolišča je najbolj značilno, da sta sloja hrambe in obdelave oz. APIjev eksplicitno ločena (angl. decoupled) [5, 6], oz. naloge na sloju hrambe se po navadi izvajajo na ločenih komponentah (vozliščih) kot naloge na sloju obdelave, kar prinaša večjo razširljivost in omogoča vzporedno dostopanje in poizvedovanje po istih podatkov s strani različnih aplikacijah.

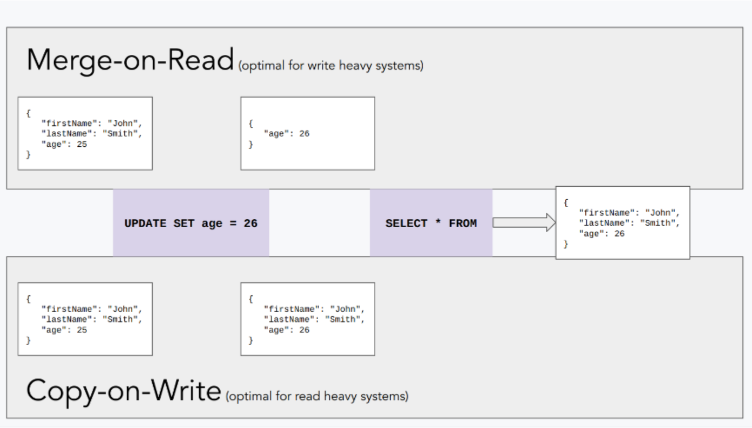
Trenutno se kot največja slabost podatkovnih kolišč izpostavlja nizka stopnja razvoja in praktičnih izkušenj, saj so podatkovna kolišča popolnoma nova tehnologija, ki se začela intenzivno razvijati šele leta 2021 [2]. Torej, nabor tehnoloških rešitev za implementacijo podatkovnih kolišč na trgu je precej omejen na le nekaj ponudnikov, kot sta Apache ali Databricks, in so prednosti vpeljave podatkovnih kolišč v različnih domenah še vedno nejasne oz., niso dovolj raziskane.

## 3 Formati za shranjevanje podatkov v podatkovnem kolišču

### 3.1 Apache Hudi

Apache Hudi (angl. Hadoop Updates, Deletes and Inserts) predstavlja nadgradnjo nad datotečnimi formati, kot sta Parquet in ORC, ki prinaša tudi možnost posodabljanja, brisanja, povrnitve prejšnjega stanja (angl. rollback), podporo za transakcije po modelu ACID ter verzioniranje podatkov shranjenih v tabele v podatkovnem kolišču. V primerjavi z osnovnimi formati v podatkovnem jezeru, ponuja možnost inkrementalnega shranjevanja podatkov v obliki paketov ter tudi podatkovnih tokov.

Podatki se shranjujejo v obliki dve vrsti tabel [6], in sicer tabeli tipa "kopiraj ob pisanju" (angl. copy on write, CoW) in "združi ob pisanju" (angl. merge on read, MoR) prikazanima na sliki 2. Izbira določene vrste tabele vpliva na način indeksiranja in fizičnega shranjevanja podatkov na datotečnemu sistemu (angl. Data File System, DFS). Pri tabeli "kopiraj ob pisanju" se podatki shranjujejo izključno v stolpčastem datotečnem formatu, kot je Parquet, in se potem ob vsaki operaciji pisanja preprosto posodablja različica datoteke, medtem ko se fizična datoteka prepiše. Posledično se poveča zakasnitev (angl. latency) pri vnašanju podatkov, ampak se izboljša čas izvajanja poizvedb nad podatki. Zaradi tega se uporaba tega tipa tabel priporoča v primerih z intenzivnimi operacijami branja podatkov. Namen tistega tipa tabel je izboljšati upravljanje tabel na način, da se posodabljanje podatkov izvaja na ravni datotek in ne celotne tabele, ter omogočiti inkrementalni zajem sprememb v podatkih, pri čemer se izognemo veliki količini manjših datotek za vsako spremembo tako, da spremembo zabeležimo v novi različici obstoječe datoteke.



```
{
   "firstName": "John",
   "lastName": "Smith",
   "age": 25
}
```

```
{
   "age": 26
}
```

```
UPDATE SET age = 26
```

```
SELECT * FROM
```

```
{
   "firstName": "John",
   "lastName": "Smith",
   "age": 26
}
```

```
{
   "firstName": "John",
   "lastName": "Smith",
   "age": 25
}
```

```
{
   "firstName": "John",
   "lastName": "Smith",
   "age": 26
}
```

**Slika 2: Primerjava posodabljanja podatkov v tabelah tipa MoR in CoW pri formatu Hudi**
Vir [7].

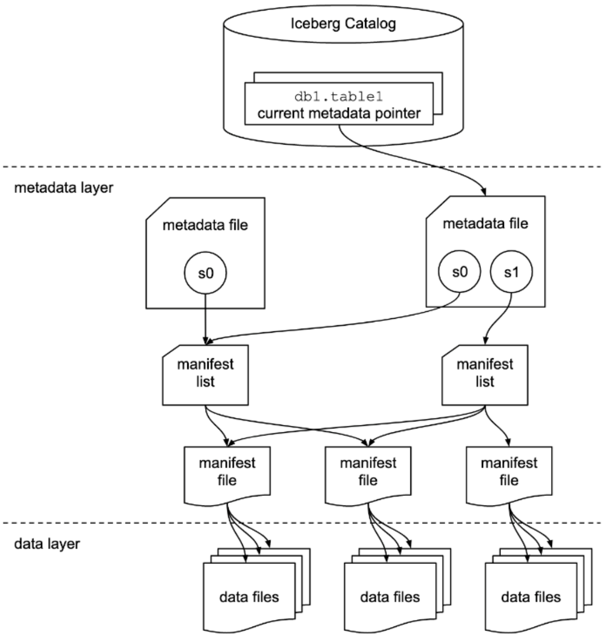
V tabeli tipa "združi ob branju" se uporablja kombinacija stolpčastega in datotečnega formata, ki temelji na vrsticah (angl. row-based), kot je Avro. Vsaka sprememba se beleži v delta datotekah, ki se kasneje (asinhrono) združijo in nastane nova različica stolpčaste datoteke. Iz stališča učinkovitosti, to pomeni, da se z uporabo tabele tipa "združi ob branju" znižajo zakesnitve pri vstavljanju novih podatkov v tabelo, hkrati pa se povečajo zakesnitve pri branju podatkov, ko je potrebno analizirati delta datoteke in združiti vse zapise v eno datoteko. Zaradi tega se uporaba tega tipa tabel priporoča v primerih z intenzivnimi operacijami pisanja podatkov. Namen tistega tipa tabele je doseči možnost približno realnočasovne obdelave podatkov neposredno nad datotečnim sistemom brez kopiranja datotek v zunanje sisteme.

## 3.2 Apache Iceberg

Apache Iceberg je visoko učinkoviti odprti format namenjen delu z velikimi analitičnimi tabelami (velikosti izražene v terabajtih in višje) v podatkovnih jezerih in kološčih, saj prinaša zanesljivost in enostavnost uporabe jezika SQL v poizvedovanju po tabelah. Zanesljivost dosreže z zagotavljanjem konsistentnosti podatkov po modelu ACID. Format shranjevanja je agnostičen do specifičnih datotečnih sistemov in se lahko uporablja sočasno z različnimi sodobnimi platformami za obdelavo podatkov, kot je Apache Spark. En izmed izzivov, ki ga naslavlja Iceberg, je možnost, da različni stroji za obdelavo podatkov lahko izvajajo poizvedbe sočasno in neodvisno, in sicer to pomeni, da format zagotavlja možnost obdelave podatkov v obliki paketov, podatkovnih tokov ali *ad hoc* poizvedb ter ponuja možnosti implementacije nalog za obdelavo v različnih programskih jezikih.

Podpira možnost sprememinjanja podatkovne sheme, tako da uporabniki po potrebi lahko dodajajo, posodabljajo ali odstranjujejo stolpce iz tabel. Posledično se vsi podatki verzionirajo in uporabniki lahko izvajajo poizvedbe tudi nad zgodovinskimi podatki za potrebe analiz.

Na ravni shranjevanja podatkov v tabele je Iceberg načrtovan kot abstraktni format ločen od fizične hrambe s pomočjo hrambe objektov. Vsa stanja tabel se beležijo kot metapodatki v Iceberg katalogu (Slika ), ki vključuje nesporedne kazalce za hitrejši dostop do tabele. Upravljanje in delo s Iceberg tabelami je omogočeno preko vmesnika Iceberg FileIO, s katerim se pospešuje interakcija med slojema hrambe in obdelave. Ob dodajanju ali branju podatkov iz Iceberg tabel, vmesnik FileIO po izvajanju ukaza *commit()* poskrbi, da se izvedejo ustrezne operacije pisanja/branja nad datotečnim sistemom ter shranijo metapodatki za to tabelo. Tukaj je razlika pri formatu Iceberg tista, da se podatkovne datoteke (angl. data file) hranijo v tabele in ne v mape na datotečnem sistemu, kot je postopek pri obstoječih formatih. Vsaka sprememba nad tabelo ustvari novo datoteko metapodatkov (angl. metadata file) in zameni prejšnje metapodatke, ki ponavadi vsebujejo posnetke (angl. snapshot) vnosov v tabeli, podatkovno shemo tabele in druge lastnosti. Znotraj tiste datoteke so posnetki podatkov shranjeni kot manifest datoteke (angl. manifest file), v katerih vsaka vrstica predstavlja podatkovno datoteko v tabeli in njezine metapodatke. Mehanizem posnetkov je ključen v primeru beleženja sprememb v viru podatkov, saj preprečuje prepisivanje metapodatkov za obstoječe podatkovne datoteke.

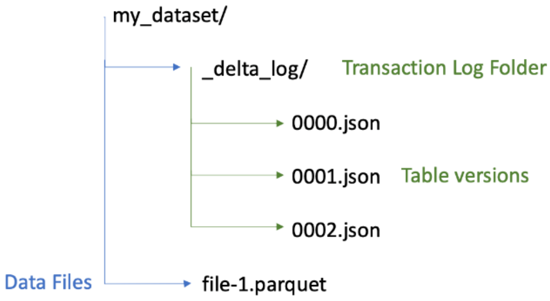


**Slika 3: Prikaz strukture shranjevanja podatkov v formatu Iceberg**
Vir: [8].

## 3.3 Delta Lake

Format Delta Lake predstavlja nadgradnjo datotečnega formata Parquet, pri katerem se podatki na fizični ravni shranjujejo kot zaporedne datoteke za posamezne stolpce v tabeli. Vključuje podporo za ACID transakcije, in sicer z najvišjo ravnijo izolacije, kar je močen mehanizem zagotavljanja konsistentnosti podatkov. Tako se v primeru neuspešne transakcije, samodejno sproži mehanizem beleženja rezultata transakcije in ustrezne akcije [10]. Dodatno prinaša izboljšanja v razširljivosti pri upravljanju metapodatkov, saj tudi tiste datoteke lahko hitro narastejo in morejo biti obdelane na učinkovit način. Tako kot prejšnji format, format Delta Lake tudi ponuja možnost časovnega prehajanja (angl. time travel), oz. dostopa in povrnitve na prejšnje verzije podatkov s pomočjo mehanizma podatkovnih posnetkov [9]. Podpira tudi možnost shranjevanja podatkovnih paketov in tokov ter omogoča spreminjanje podatkovne sheme tabel.

Vsaka tabela Delta Lake se shranjuje kot mapa, ki vsebuje datoteke z metapodatki shranjene v podmapo *_delta_log* in neposredne podatke shranjene v obliki Parquet datotek (slika 4). Vsaka sprememba nad tabelo se beleži kot transakcija in shranjuje kot datoteka v obliki JSON, s katerimi imamo možnost zgodovinskega pregleda sprememb nad tabelo.



**Slika 4: Prikaz shranjevanja podatkov v formatu Delta Lake**
Vir: [10].

### 3.4 Primerjava formatov za shranjevanje

Po pregledu literature in podrobni analizi dokumentacije vseh treh formatov bomo v tistem poglavju predstavili glavne razlike med formati Hudi, Iceberg in Delta Lake. Primerjavo smo naredili na podlagi lastnosti, za katere smo določili da značilno vplivajo na učinkovitost podatkovnega kolišča, ter na podlagi dodatnih lastnosti, ki so uporabljene za njihovo primerjanje v obstoječi literaturi. Kot je prikazano v Tabeli 1, vsa tri formata podpirajo izvedbo transakcij po modelu ACID in so v tem smislu nadgradnja obstoječih formatov, kot je Parquet, ki tiste podpore nimajo. Formata Iceberg in Delta Lake ponujajo ustrezne in zanesljive mehanizme za spreminjanje podatkovne sheme tabel, medtem ko je pri formatu Hudi tista funkcija v testni fazi in je trenutno dostopna le v primeru uporabe z okvirjem Apache Spark. Formati se predvsem razlikujejo v podpori za datotečne formate shranjevanja podatkov, saj Delta Lake predvsem temelji na uporabi formata Parquet, ki se v praksi izkazal kot naučinkovitejši format za poizvedovanje po podatkih. Druga dva formata zraven podpirata tudi ORC (angl. Optimized Row Columnar), ki je tudi stolpčasti format kot Parquet, vendar je bolj primeren v primeru potrebe po visoki zmogljivosti operacij pisanja.

Formata Hudi in Delta Lake sledita tabelaričnemu pristopu upravljanja metapodatkov za sledenje stanju tabel, medtem ko se pri formatu Iceberg še vedno srečamo z hierarhičnim pristopom. Posledično se pri izvajaju poizvedbe hierarhični pristop lahko negativno vpliva na čas izvajanja zaradi odvisnosti o hitrosti prehajanja po hierarhiji. S druge strani pa je pri tabelaričnemu pristopu tudi potrebno zagotoviti ustrezno upravljanje tabel metapodatkov, ki postopoma tudi narastejo in tako se lahko podaljša čas iskanja po tabeli brez prisotnosti ustreznih mehanizmov (npr. indeksi). Indeksi se ponavadi samodejno ali ročno ustvarijo nad medapodatki pri vseh treh formatih. Dodatno Delta Lake ponuja tudi možnost ustvarjanja t. i. *z-order* indeksov, pri kateri se med seboj povezani podatki shranijo v isto skupino datotek zaradi zmanjševanja količine datotek tekom iskanja. Trenutno formata Hudi in Delta Lake tudi zagotavljata višjo konsistentnost podatkov s pomočjo dodatnih integritetnih omejitev, kot so primarni ali tuji ključi. Kar se tiče strategije posodabljanja tabel, Delta Lake je edini format, ki zaenkrat ne podpira možnost implementacije strategije MoR, s katero bi lahko dosegli hitrejše izvajanje operacij pisanja, ampak vsi formati podpirajo strategijo CoW. Apache Hudi je predvsem namenjen shranjevanju podatkov le v datotečnih sistemih, ki temeljijo na okolju Hadoop (npr. HDFS, Hive), medtem ko sta druga dva formata razširila svojo podporo tudi na druga orodja in sisteme, kot so Spark, Trino, Flink, idr.

Lastnosti formatov tudi vplivajo na njihovo primarno nalogo ter ustrezne primere uporabe določenega formata v sodobnih sistemih. Namreč, čeprav vsa tri formata v določeni meri podpirajo nekatere lastnosti (npr. ACID, indeksiranje), formati se neprekinjeno razvijajo in ponujajo izboljšave v določenih segmentih, ki potem lahko predstavljajo prelomnico pri izbiri najbolj ustreznega formata. Tako se uporaba formata Hudi priporoča v primeru potrebe po realnočasovni obdelavi podatkov zaradi učinkovite strategije inkrementalne obdelave podatkov. Pri formatu Iceberg je poudarek bolj na učinkoviti izvedbi povpraševanj nad podatki in se zaradi tega priporoča njegova uporava v podatkovnih skladišč, oz. za analitične potrebe. V primerjavi z ostalima dvama formatoma, format Delta Lake zagotavlja največjo zanesljivost in konsistentnost podatkov in se priporoča njegova uporaba v implementaciji podatkovnih cevovodov, kjer obstaja potreba po paketni in realnočasovni obdelavi, ter v podatkovnih jezerih.

Jain in dr. [3] so v svojem članku predstavili rezultate primerjave učinkovitosti formatov Delta Lake, Apache Hudi in Iceberg. Pri uvažanju podatkov se format Hudi izkazal kot najpočasnejši ker ni optimiziran za paketni zajem podatkov, temveč za podatkovne toke in operacije *upsert* po določenem ključu. Format Delta Lake se izkazal kot najbolj učinkovit v primeru poizvedovanja po podatkih. V primerjavi s formatom Hudi je operacija branja hitrejša zaradi večje velikosti datoteke pri formatu Delta Lake. Namreč, običajno se podatki v Delta Lake tabeli hranijo v datoteke velikosti do 256 megabajtov (MB), medtem ko je ta velikost pri formatu Hudi omejena na 100 MB. Zaradi tega se pri formatu Hudi podatki shranijo v več datotek, ki jih je potrebno (de)kompresirati, kar potem podaljša čas iskanja po tabeli pri branju. V primerjavi s formatom Iceberg so razlike v učinkovitosti branja predvsem nastale zaradi uporabljane knjižnice za branje Parquet datotek s pomočjo okvirja Spark, kar je pri formatu Delta Lake neposredno integrirano in optimizirano. Dodatno je procesiranje metapodatkov tudi pokazalo močen vpliv na učinkovitost operacij v podatkovnem kolišču. Porazdeljeni pristop procesiranja pri formatih Hudi in Delta Lake je bolj primeren za analize velikih tabel v visoko razširljivih sistemih, medtem ko je pristop procesiranja na enem vozlišču pri formatu Iceberg zadostno le v primerih, ko so tabele metapodatkov manjše.

**Tabela 1: Primerjava formatov za shranjevanje podatkov v podatkovnem kolišču.**

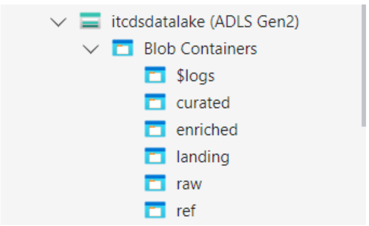
| | **Apache Hudi** | **Apache Iceberg** | **Delta Lake** |
|---|---|---|---|
| **Podpora za ACID transakcije** | ✓ | ✓ | ✓ |
| **Možnost spreminjanja sheme** | ? (dostopno edino za Spark) | ✓ | ✓ |
| **Podprti datotečni format** | Parquet, ORC | Parquet, ORC, Avro | Parquet |
| **Upravljanje metapodatkov** | Tabela metapodatkov + dnevniške datoteke transakcij | Hierarhične datoteke (manifesti) | Dnevniške datoteke transakcij + točke preverjanja |
| **Možnost uveljavljanja integritetnih omejitev nad tabelo** | ✓ | × | ✓ |
| **Podprti način posodabljanja** | Copy-on-Write<br>Merge-on-Read | Copy-on-Write<br>Merge-on-Read | Copy-on-Write |
| **Podpora za indekse** | ✓ (nad metapodatki) | ✓ (nad metapodatki) | ✓ (z-order indeksi) |
| **Načrtovanje strategije poizvedovanja** | Porazdeljeno (analiza tabele metapodatkov) | Eno vozlišče (analiza hierarhije manifestov) | Porazdeljeno (analiza tabele metapodatkov) |
| **Namenski datotečni sistem/stroj** | Hadoop | - | - |
| **Primarna naloga** | Učinkovita inkrementalna obdelava in indeksiranje | Visoka učinkovitost poizvedb | Visoka razširljivost in zanesljivost |
| **Priporočena uporaba** | Realnočasovna obdelava in analitika, podatkovna jezera | Podatkovna skladišča in analitika | Podatkovna jezera in cevovodi |
| **Ustanovitelj** | Uber | Netflix | Databricks |

# 4 Primer uporabe formata Delta Lake

V tistem poglavju bomo predstavili podrobnosti glede uporabe formata Delta Lake nad izgrajenim podatkovnim jezerom. Prikazali bomo podatkovni model podatkovnega jezera za izgradnjo platforme za deljenje agroživilskih podatkov in bomo predstavili način dela s podatki shranjenimi v formatu Delta Lake.

## 4.1 Opis primera in arhitektura podatkovnega jezera

Za pospešitev digitalizacije agroživilskega sektorja je potrebno izgraditi podatkovno platformo, ki bo omogočila deljenje agroživilskih podatkov s strani različnih deležnikov, kot so kmeti, senzorji, različne aplikacije dobaviteljev, itn. Ciljna platforma bo zbirala različno strukturirane podatke iz več različnih virov (npr. Blockchain omrežje, različni APIji, ročni vnos CSV datotek kmetov in pridelovalcev). Zbrani podatki se bodo potem po potrebi dodatno procesirali za namene različnih analiz ter implementacije APIjev za deljenje teh podatkov. Kot najbolj ustrezna arhitektura takšne platforme je izbrano podatkovno jezero zaradi možnosti shranjevanja velike količine podatkov v surovi obliki in možnosti za različne analize. Za procesiranje in transformacijo podatkov v ciljno obliko smo izbrali okvir Apache Spark. Sistemska infrastruktura je nameščena v okolju Azure, pri čemer kot rešitev za implementacijo podatkovnega jezera uporabljamo *Azure Data Lake Storage Gen2*. Gre za rešitev, kjer se podatki shranjujejo kot objekti (poimenovani kot angl. *blobs*) oz. datoteke upoštevajoč hierarhično strukturo map (poimenovane kot angl. *blob containers*), ki oblikujejo t. i. hierarhični imenski prostor (angl. hierarchical namespace) [12] (slika 5).



**Slika 5: Struktura map v podatkovnem jezeru.**

Znotraj podatkovnega jezera so podatki shranjeni v različne cone oz. mape glede na različno stopnjo obdelave (od surovih do agregiranih podatkov). Tako je podatkovno jezero sestavljeno iz naslednjih con:

1. *Landing* cona – vsebuje surove podatke zbrane iz podatkovnih virov in shranjene v izvirnem formatu (npr. JSON);
2. *Raw* cona – vsebuje podatke v surovi obliki, ki so grupirani po določenem ključu (npr. kmet, leto);
3. *Enriched* cona – vsebuje agregirane podatke iz individualnih podatkovnih zapisov oz. datotek (npr. skupno količino dobavljenega izdelka);
4. *Curated* cona – vsebuje agregirane podatke kot rezultate določenih analiz (npr. izguba pri proizvodnji izdelka);
5. *Ref* cona – vsebuje interne identifikatorje in metapodatke potrebne za sledenje podatkom.

Posamezne cone so dodatno porazdeljene v ustrezne podmape zaradi grupiranja podatkov oz. datotek po ustreznem ključu (npr. vse dobavnice v določenem letu za pridelovalce). Slednja hierarhična struktura je optimizirana glede na potrebe analiz s ciljem, da se znižajo stroški dostopa do podatkov. Podatki se med conami premikajo ob njihovi obdelavi, ki jo izvajamo s pomočjo okvirja Apache Spark in programske kode napisane v jeziku Python v smeri *landing-curated* con.

### 4.2 Uporaba formata Delta Lake

Kot rečeno se podatki v Azure podatkovnem jezeru po navadi shranjujejo kot datoteke oz. objekti, kar pomeni, da se pravzaprav pri delu s temi podatki (vnašanje, posodabljanje, brisanje, branje) uporabljajo akcije, ki smo jih vajeni pri delu z datotekami na datotečnem sistemu, kot sta dodajanje ali brisanje datotek v ustreznih mapah. Izkušnje in začetne poizvedbe so pokazale, da operacije nad datotekami prinašajo večje stroške (resursi in čas) pri obdelavi podatkov za namene analiz zaradi potrebe po hierarhičnem skeniranju vseh (pod)map. Dodatno pa želimo vpeljati mehanizme nadzora in zagotavljanja kakovosti končnih podatkov čez celotni cevovod. Zaradi teh omejitev želimo nadgraditi podatkovno jezero s formatom Delta Lake, s katerim bomo izboljšali učinkovitost obdelave in branja podatkov ter ohranili njihovo kakovost. Rezultat tega bo nadgradnja podatkovnega jezera v podatkovno kolišče.

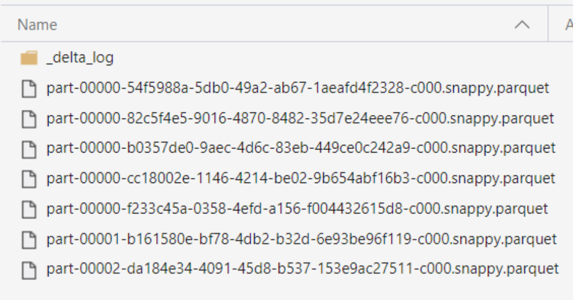
V okolju Azure je nadgradnjo podatkovnega jezera s podatkovnim koliščem možno izpeljati z vključitvijo storitve Azure Databricks, ki vključuje implementacijo arhitekture potrebne za učinkovito obdelavo podatkov shranjenih v različnih formatih (tudi Delta Lake) ter preprosto namestitev Spark gruče za izvajanje Spark poslov (angl. job), s katerimi obdelujemo podatke.

Pri migraciji vsakega podatkovnega zapisa (v obliki datoteke ali vrstice v Delta Lake tabeli) se izvaja določena transformacija implementirana v jeziku Python znotraj zasebne Spark skripte. V Sparku lahko dostopamo do metod za delo s formatom Delta Lake preko integrirane knjižnice *delta*, ki jo je potrebno vključiti na začetku skripte, kot je prikazano v izsečku programske kode na sliki 6. Najprej se povežemo na Azure Data Lake Storage Gen2 z ustreznimi uporabniškimi podatki in potem določimo povezavo do izvirne mape v podatkovnem jezeru, od kod želimo uvesti podatke (v tistem primeru je povezava shranjena v spremenljivki *landing_base_path*). Po uspešni povezavi uvezemo podatke kot Spark podatkovni okvir (angl. data frame) in nadaljujemo s potrebno transformacijo čez klicanje ustreznih metod za delo s Spark podatkovnim okvirjem. V tistem primeru želimo posamezne dobavnice (angl. delivery report) izdane pri pridelovalcih grupirati po pridelovalcu ter letu in mesecu ter ustvariti eno Delta Lake tabelo, kjer bo vsaka vrstica predstavljala eno dobavnico. Vsaka dobavnica, ki je prišla iz Blockchain omrežja v podatkovno jezero, vključuje podatke o pridelovalcu in količini določenih izdelkov, ki jih je tisti pridelovalec oz. kmet spravil na določeni dan. Postopek transformacije poteka tako, da iz dobavnic, ki so v surovi obliki zbrane kot JSON datoteke, izvlečemo informacijo o pridelovalcu, letu in mesecu ter shranimo končni podatkovni okvir v mapo *raw* s klicom metode *write()*, pri čemer kot format opredelimo *delta*.

```
from delta.tables import DeltaTable
from pyspark.sql.types import StructType, StructField, StringType, TimestampType
folder_content = dbutils.fs.ls(self.landing_base_path)
farmer_month_paths = {}

for file in folder_content:
       extracted_farmer_month = file.name
       for farmer_month in farmer_month_paths:
           if farmer_month == extracted_farmer_month:
              farmer_month_paths[extracted_farmer_month].append(file.path)
              break
       if not exists:
                farmer_month_paths[extracted_farmer_month] = [file.path]

df = self.spark.read.option("multiline", "true").json(farmer_month_paths[file_name])
… // regroup files to appropriate folders (e.g. /year/month)…
save_path = self._create_save_path(file_name)
df.write.mode("append").format("delta").save(save_path)
```

**Slika 6: Primer shranjevanja podatkov v tabelo Delta Lake.**

Rezultat je nova struktura map v *raw* coni v podatkovnem jezeru prikazana na sliki 7, pri čemer je tabela, ki predstavlja dobavnice, fizično shranjena kot mapa, ki vsebuje podatke o posameznih dobavnicah shranjene kot Parquet datoteke ter mapo *_delta_log* z zgodovinskim pregledom operacij oz. transakcij izvedenih nad tabelo.



_delta_log
part-00000-54f5988a-5db0-49a2-ab67-1aeafd4f2328-c000.snappy.parquet
part-00000-82c5f4e5-9016-4870-8482-35d7e24eee76-c000.snappy.parquet
part-00000-b0357de0-9aec-4d6c-83eb-449ce0c242a9-c000.snappy.parquet
part-00000-cc18002e-1146-4214-be02-9b654abf16b3-c000.snappy.parquet
part-00000-f233c45a-0358-4efd-a156-f004432615d8-c000.snappy.parquet
part-00001-b161580e-bf78-4db2-b32d-6e93be96f119-c000.snappy.parquet
part-00002-da184e34-4091-45d8-b537-153e9ac27511-c000.snappy.parquet

**Slika 7: Primer shranjene tabele dobavnic v formatu Delta Lake.**

Za prikaz podatkov v tabeli uporabljamo programsko kodo, in sicer podatke uvezemo v Spark podatkovni okvir iz lokacije tabele v podatkovnem jezeru, ki jo določimo s povezavo. Na sliki 8 je predstavljena programska koda za prikaz podatkov iz Delta Lake tabele v *enriched* coni, kjer so podatki o posameznih dobavnicah iz *raw* cone že agregirani na način, da se za vsako kombinacijo kmeta (določen z internim identifikatorjem), izdelka, leta in meseca posodablja skupna količina dobavljenega izdelka. Izseček programske kode za agregacijo podatkov je prikazan na sliki 9.

```
data_df= spark.read.load(povezava do Delta Lake tabele v jezeru)
display(data_df)
```

| | _id | crop | year | month | sumDelivery |
|---|---|---|---|---|---|
| 1 | 626436bd93d6ad0321c9f7ff | {"_id": "00005", "enota": "KG", "name": "BANANE KOL.", "CPA": "01.22.13"} | 2022 | 07 | 2142.08 |
| 2 | 626436bd93d6ad0321c9f7ff | {"_id": "00009", "enota": "KG", "name": "BRESKVE RUMENE ŠP.", "CPA": "01.24.26"} | 2022 | 07 | 3150.62 |
| 3 | 626436bd93d6ad0321c9f7ff | {"_id": "00008", "enota": "KG", "name": "BRESKVE RUM. IT.", "CPA": "01.24.25"} | 2022 | 07 | 3102.46 |
| 4 | 626436bd93d6ad0321c9f7ff | {"_id": "00003", "enota": "KG", "name": "AVOKADO", "CPA": "01.22.11"} | 2022 | 07 | 1585.1399999999999 |
| 5 | 626436bd93d6ad0321c9f7ff | {"_id": "00004", "enota": "KG", "name": "BANANE KOST.", "CPA": "01.22.12"} | 2022 | 07 | 2601.73 |
| 6 | 626436bd93d6ad0321c9f7ff | {"_id": "00006", "enota": "KG", "name": "EKO AMERIŠKE BOROVNICE", "CPA": "01.25.19"} | 2022 | 07 | 758.7 |
| 7 | 626436bd93d6ad0321c9f7ff | {" id": "00007", "enota": "KOS", "name": "BOROVNICE", "CPA": "01.25.19"} | 2022 | 07 | 1622 |

Slika 8: Primer programske kode za izpis podatkov iz Delta Lake tabele.

```
def calculate(df):
    df = (
       df.groupBy("_id", "crop", "year", "month")
         .agg(sum("quantity").alias(self.sum_alias))
       )
    return df
merged_df = union_all(*dfs)
new_farmer_total_df = calculate(merged_df)

save_path = self._create_save_path(self.file_name)
exists = DeltaTable.isDeltaTable(self.spark, save_path)
if exists:
   print("Delta table exists. Updating values in {}".format(save_path))
   deltaTable = DeltaTable.forPath(self.spark, save_path)
   deltaTable = deltaTable.alias("existing").merge(
       new_farmer_total_df.alias("updates"),
           "existing.crop = updates.crop AND existing._id = updates._id AND existing.year =
updates.year AND existing.month = updates.month", (set={self.sum_alias: col("updates." +
self.sum_alias)}).whenNotMatchedInsertAll().execute()

else:
   print("Delta table does not exist. Creating table with calculated values in
{}/{}".format(self.enriched_base_path,self.file_name))
   new_farmer_total_df.write.mode("overwrite").save(create_save_path(self.file_name))
```

Slika 9: Izsek programske kode za agregacijo in shranjevanje rezultatov poizvedbe nad Delta Lake tabelo.

## 5 Diskusija

Arhitektura podatkovnih kolišč predstavlja zamisel shranjevanja in obdelave podatkov znotraj le ene platforme, kjer se podatki hranijo v formatu, ki omogoča hiter dostop, razširljivost ter zagotavlja njihovo konsistentnost. Kod nadgradnja podatkovnih jezer, želijo podatkovna kolišča vpeljati določene funkcionalnosti podatkovnih baz in skladišč, ki jih podatkovna jezera privzeto ne podpirajo, vendar so potrebne pri zagotavljanju kakovosti podatkov za analitične potrebe. Vpeljava mehanizma transakcij in modela ACID iz podatkovnih skladišč in baz prinaša dodaten varnostni mehanizem za ta namen, saj na ta način podatkovno kolišče ni le rešitev, ki je visoko razširljiva in zmožna izvesti kompleksne analize nad veliko količino različno strukturiranih podatkov zbranih iz več virov (kar je primarno namen podatkovnih jezer), ampak je tudi centralni repozitorij konsistentnih podatkov z možnostjo vzpostavitve mehanizmov upravljanja podatkov (angl. data governance) v vsaki fazi podatkovnega cevovoda. Odsotnost nadzorovanja kakovosti podatkov je en izmed največjih izzivov pri vzpostavitvi podatkovnih jezer v podjetjih, od katerih pogosto nastanejo podatkovna močvirja, oz. rešitve, ki hranijo velike količine podatkov, ki niso ustrezno opisani z metapodatki in se posledično ne morejo uporabljati za nobene analize.

Nadgradnja podatkovnih jezer s formatom Delta Lake je ena izmed možnosti izgradnje učinkovitega sistema za upravljanje velepodatkov, ki je razširljiv in dovolj fleksibilen za kakršne koli trenutne in prihodne analitične potrebe. Kot je prikazano na praktičnem primeru, je postopek nadgradnje poenostavljen prav zaradi ločitve slojev hrambe in obdelave podatkov značilne za podatkovna kolišča. To pomeni, da se na sloju hrambe še vedno lahko kot rešitev uporablja podatkovno jezero (npr. Azure Data Lake Storage ali HDFS), vendar je potrebno le prilagoditi format shranjevanja (npr. Delta Lake). Tehnologija obdelave podatkov pa lahko ostane nespremenjena (npr. Spark), če je zaledni mehanizem za povpraševanja zmožen obdelati podatke shranjene v novem formatu. V tem primeru uporaba formata Delta Lake deluje kot dodaten sloj abstrakcije nad podatkovnim jezerom, ki uporabniku omogoča bolj prijazen in znani način dela s podatki shranjeni v tabele kot pri relacijskih bazah, čeprav so na fizični ravni le-ti še vedno shranjeni v hierarhično strukturo map in datotek v formatu Parquet na datotečnem sistemu. Možnost obdelave podatkov, pri čemer se vsaka sprememba nad tabelo izvaja kot transakcija in beleži v zaledni mapi, močno poveča kakovost podatkov in zanesljivost celotne rešitve. Dodatno je podpora za okvirje za obdelavo podatkov, kot je v našem primeru Spark, že dovolj stabilna in napredna ter vključuje potrebne metode za transformacijo in agregacijo podatkov za potrebe analiz. V našem primeru, kjer imamo obstoječe podatkovno jezero izgrajeno v okolju Azure se kot najbolj primerna varianta integracije formata Delta Lake izkazala vključitev platforme Azure Databricks. Z uporabo novega formata smo poenostavili način dela s podatki shranjeni v podatkovnem jezeru, saj več ni potrebno skenirati celotnih hierarhij map pri obdelavi podatkov shranjenih v datoteke in se podatkom dostopa kot vrsticam v tabeli. V ozadju pa je močen mehanizem dnevniškega beleženja transakcij, ki nam prinaša dodatne prednosti (npr. povrnitev na kakšno prejšnje stanje, zgodovinska analiza, itn.).

## Literatura

[1] ŠESTAK, Martina, TURKANOVIĆ, Muhamed "Pregled in analiza tehnoloških skladov za implementacijo sodobnih IT arhitektur velepodatkov", Uporabna informatika, letnik 31, številka 1, 2023, str. 27-44.

[2] ARMBRUST, Michael, et al. "Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics", Proceedings of the 11th Annual Conference on Innovative Data Systems Research (CIDR '21), številka 8, 2021.

[3] PARAS, Jain, et al. "Analyzing and Comparing Lakehouse Storage Systems", Proceedings of the 13th Annual Conference on Innovative Data Systems Research (CIDR '23), 2023.

[4] ČUŠ, Blaž, GOLEC Darko "Data Lakehouse: Benefits in small and medium enterprises" Mednarodno inovativno poslovanje= Journal of Innovative Business and Management, letnik 14, številka 2, 2022, str. 1-10.

[5] https://www.altexsoft.com/blog/data-lakehouse/, Data Lakehouse: Concept, Key Features, and Architecture Layers, 2021, obiskano 24. 7. 2023.

[6] ERRAMI, Soukaina Ait, et al. "Spatial big data architecture: From Data Warehouses and Data Lakes to the LakeHouse", Journal of Parallel and Distributed Computing, številka 176, 2023, str. 70-79.

[7] https://hudi.apache.org/docs/table_types, Table & Query Types, obiskano 25. 7. 2023.

[8] https://bigdataboutique.com/blog/introduction-to-apache-hudi-c83367, Introduction to Apache Hudi, 2023, obiskano 26. 7. 2023.

[9] https://iceberg.apache.org/spec/, Iceberg Table Spec, obisako 26. 7. 2023.

[10] https://docs.delta.io/2.3.0/quick-start.html, Delta Lake Quickstart, obiskano 26. 7. 2023.

[11] https://www.ejztech.com/tech/project/2019/11/17/delta-lake, Delta Lake 101, 2019, obiskano 26. 7. 2023.

[12] https://learn.microsoft.com/en-us/azure/storage/blobs/data-lake-storage-introduction, Introduction to Azure Data Lake Storage Gen2, 2023, obiskano 27. 7. 2023.

# Vzpostavitev podatkovne baze iz pogorišča

Valter Miščič

A1 Slovenjia, Ljubljana, Slovenija
valter.miscic@a1si

Vse pogostejše in obsežnejše naravne in tudi drugim katastrofe spremljajo naše življenje v zadnjem obdobju. Obsežni požari, povodnji in potresi uničujejo naša mesta, rakete uničujejo infrastrukturo. Ogrožena so naša življenja, premoženje in vse, kar smo ustvarjali desetletja in kar nam je najvažnejše in najdražje. Če pogledamo na te stvari iz oči naših podjetij, so najpomembnejša stvar prav gotovo njeni podatki. Firma, ki zgubi svoje najpomembnejše podatke, praktično izgubi vse. Firme se morajo na takšne dogodke pripraviti in svoje podatke ustrezno zaščititi. S podvojevanjem računskih centrov vključno z vso strojno in programsko opremo ter podatki se lahko dokaj dobro zaščitimo. Žal, vse firme nimajo tega privilegija. Dodatni računski center seveda stane in pri denarju se največkrat ustavi. Velikokrat pa gre zgolj za nezavedanje ali podcenjevanje te problematike. Vsekakor je dobro, da imamo pripravljen načrt okrevanja, da mu vsi sledimo in da neprestano vadimo postopke okrevanja.

**Ključne besede:**
restavriranje po katastrofi
rezervni računski center
podatkovne tehnologije
podvojeni sistemi
administracija podatkovnih baz

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.22

# 1 Uvod

Vsem je jasno, da so danes podatki najpomembnejša stvar vsakega podjetja. Podatke čuvamo pred izgubo in zlorabo, skrivamo pred konkurenco. Za večino ljudi se zdi razpoložljivost podatkov samoumevna. Vedno pa obstaja možnost, da podatki niso razpoložljivi ali jih celo izgubimo. Problema se velikokrat zavemo šele ob izpadih informacijskih sistemov. Zgodi se, da so podatki začasno nedosegljivi: pride do okvare strojne opreme, napake v programski opremi ali do kake druge začasne težave, ki nam ne omogoča dostopa do podatkov. Take težave se večinoma rešijo v nekaj urah ali morda dneh. Smo kdaj pomislili, da lahko izgubimo vse, popolnoma vse: pogori nam računski center z vso opremo vred. Zgodi se katastrofa. Kaj bomo naredili v takem primeru. Imamo opremo, čas in znanje, da postavimo vse z ničle? Kaj, če za vedno ostanemo brez vseh podatkov. Podjetje zapremo, ljudi damo na cesto. Na take katastrofalne izpade se moramo pripraviti!

Sam sem administrator podatkovnih baz , zato bom problematiko opisal s stališča administratorja podatkovnih baz.

# 2 Naloge administratorjev

## 2.1 Najpomembnejša naloga administratorjev

Za vsako delovno mesto obstaja spisek del in nalog, ki jih mora opravljati. Te naloge so zapisane že ob razpisih za posamezno delovno mesto. Kot administratorju baz podatkov, mi je najpomembnejša naloga izvajanje arhiviranja podatkovnih baz in zmožnost njihovega restavriranja. To ne velja samo za administratorje baz, ampak velja na splošno za vse druge administratorje: sistemske inženirje, administratorje omrežja, aplikacij, itd.. Na vse ostale naloge se lahko pripravljamo več časa, se sproti učimo, dokumentiramo. Ko pa gre za restavriranje, se od nas pričakuje, da bomo hitri in učinkoviti. V takih trenutkih velja nekakšna napetost, živčnost in negotovost ne samo administratorjev, ampak tudi vseh, ki so od podatkov odvisni. Zato je pomembno, da imajo administratorji pogoje in izkušnje za uspešno restavriranje sistemov, za katere skrbijo.

## 2.2 Revizija dela administratorjev

Zaradi pomembnosti informatike se delo tistih, ki skrbijo zanjo, nenehno revidira. Do njih prihajajo najrazličnejši revizorji in ugotavljajo, če administratorji opravljajo svoje delo. Zanimivo je, da tudi revizije, ki se nanašajo na popolnoma specifične zadeve, vedno preverijo tudi delo informatikov. Zavedajo se, da predstavlja informatika veliko ranljivost in tveganje za podjetje. Med drugim se njihova vprašanja vedno dotaknejo tudi področja arhiviranja in restavriranja podatkovnih baz in vseh ostalih komponent informacijskega sistema. Vprašanja zgledajo nekako takole:

- Izvajate arhiviranje sistemov/podatkovnih baz/aplikacij?
- Beležite njihovo izvajanje?
- Izvajate monitoring nad arhiviranjem?
- Izvajate občasna restavriranja?

Vsak izmed administratorjev za svoje področje odgovori pritrdilno na vsa vprašanja. Večinoma je to tudi res. Vendar vsak administrator tudi pove, kakšni so predpogoji, da bo svoj del uspešno restavriral. Na primer, če hočem restavrirati podatkovno bazo, mora najprej sistemski inženir postaviti ali restavrirati strežnik, na katerega bom bazo restavriral. Administrator aplikacije pove, da mora za delovanje aplikacije najprej imeti dostop do restavrirane aplikacije. To je vsakomur jasno, vendar pa prav v teh odvisnostih včasih prihaja to zapletov. Računamo drug na drugega, nismo pa se nikoli popolnoma dogovorili, kaj kdo od koga vse pričakuje. Gre za pomembne malenkosti, ki se morda odkrijejo šele v primeru restavriranja.

Vsakršne revizije so koristne, čeprav jih informatiki ne maramo, ker lahko pokažejo na določene neustreznosti ali pa nas vsaj malo predramijo iz svoje samovšečnosti.

# 3 Dogodki s katastrofalnimi posledicami

## 3.1 Kakšne katastrofe se nam lahko zgodijo?

V zadnjem času smo na lokalni in svetovni ravni doživeli kar nekaj dogodkov, ki lahko vodijo v popolno katastrofo. Če jih naštejem samo nekaj:

- napad z raketami,
- poplave večjih razsežnosti,
- požari večjih razsežnosti,
- uničujoči potresi.

Na osebni ravni ti dogodki jemljejo naša življenja, našo imovino, uničujejo infrastrukturo. Za posameznika je to najbolj pretresljivo. Poleg tega lahko taki dogodki uničujejo tudi širšo infrastrukturo, cela mesta, tovarne. Za podjetja predstavlja najhujšo izgubo uničenje njenih dragocenih podatkov. Tega ne povrne nobena zavarovalnica in tega se ne da nikjer kupiti. Ne gre samo za izgubo podatkov. Podatke morda celo redno arhiviramo in bi jih lahko restavrirali. Pogubno je že, če ostanemo brez računskega centra.

Za postavitev novega centra, ki bi deloval v zadovoljivi funkcionalnosti bi potrebovali naslednje:

- pridobitev novega ustreznega prostora,
- nabava strojne opreme,
- vzpostavitev omrežja,
- priprava strežnikov,
- namestitev programske opreme in aplikacij,
- restavriranje baz,
- konfiguriranje aplikacij, baz, operacijskih sistemov.

Nepredstavljivo je, da bi nam pri današnjih kompleksnih arhitekturah to uspelo. Firme, ki se tega zavedajo, so na take dogodke bolje ali slabše pripravljene. Nekatere pa se ali sploh ne zavedajo ali pa zgolj niso pripravljene. Upam, da niste med njimi.

## 3.2 Kako se zaščititi pred takimi dogodki

Takšnih katastrofalnih dogodkov ne moremo predvideti in se jim ne moremo izogniti. Nanje pa smo lahko bolje ali slabše pripravljeni. Ko govorimo o uničenju celotnega računalniškega centra, imamo na razpolago različne scenarije okrevanja.

Najpogostejši in hkrati najslabši primer je, da se zanašamo zgolj na naše varnostne kopije. V tem primeru predvidevamo, da je verjetnost katastrofe majhna in bomo vzpostavili nov center z novo opremo in restavriranjem vseh komponent iz obstoječih varnostnih kopij. Vendar takšna rešitev sploh ni enostavna. Najprej je treba dobiti ustrezen prostor in nabaviti celotno novo opremo. Nabava nove opreme lahko traja mesece in tudi časovna stiska nam ne omogoča pogajanj za boljše cene.

Današnji informacijski sistemi so tako kompleksni, da je uresničitev te variante nepredstavljiva. Zamislimo si sistem s stotinami aplikacij in njihovimi povezavami do servisov, ki so povsod po svetu. Praktično nemogoče je, da bi firma to vzpostavila in preživela.

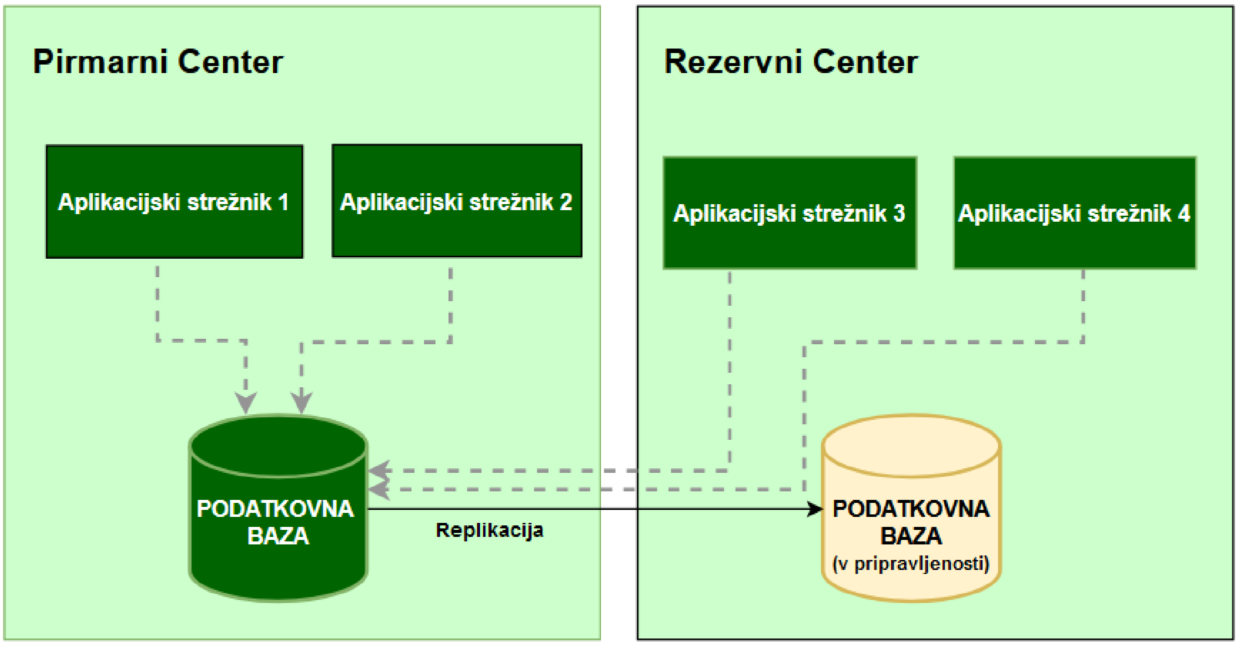
Drugi in tudi boljši pristop je, da poleg računskega centra (recimo mu primarni center) vzdržujemo še en t. i. rezervni center. Rezervni center mora biti opremljen s podobno opremo kot primarni center in živeti mora v sozvočju z njim. Imeti rezervni center z nekaj stare opreme nam ne bo dosti koristil. Morda smo pred leti res imeli takšne rezervne centre (zgolj zaradi boljšega občutka), vendar se je v zadnjem času z osveščanjem to drastično spremenilo. Rezervni centri so postali pravi rezervni centri in tudi v resnici zmorejo prenesti vse breme primarnih centov.

Pri rezervnih centrih je pomembna sama lokacija. Biti mora dovolj oddaljena od primarnega centa, da je ne bi zajela ista katastrofa kot primarni center. Sam prostor mora biti primeren za računski center. Biti mora ustrezno klimatiziran, na razpolago mora biti zadostno električno napajanje in podatkovno omrežje. Računalniško omrežje in požarno zidovje mora biti samostojno in neodvisno od primarnega centra. Strojna oprema mora biti zadostna in kompatibilna z opremo v primarnem centru.

Ko imamo strojno opremo v rezervnem centru postavljeno, je treba urediti še podatkovne baze in aplikacije. Rezervni center naj ne predstavlja samo prostor in opremo, ki čaka nekje v rezervi. Podatkovne baze naj bodo postavljene tako, da se na rezervni lokaciji vzdržujejo baze v pripravljenosti in jih kadarkoli lahko preklopimo iz primarne lokacije na rezervno in nazaj. Obstaja več načinov postavitev baz v pripravljenosti. Izberemo način, ki nam najbolj ustreza in nas zadovoljuje. Nekatere konfiguracije zahtevajo še dodatne licence, kar lahko dodatno podraži našo rešitev. Rezervni center predstavlja za podjetje praktično dvojni strošek za informatiko, zato ga gre izkoristiti.

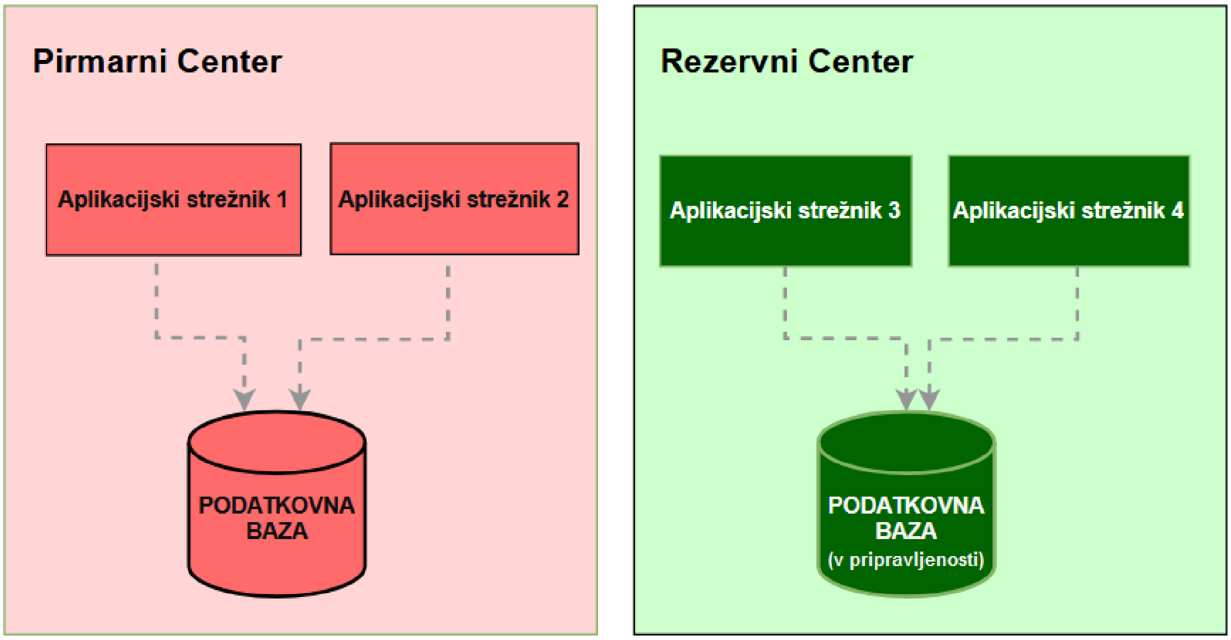
To je najlažje narediti z aplikacijskimi strežniki. Aplikacija, ki uporablja več aplikacijskih strežnikov, naj ima le te porazdeljene med obema centroma. Ob morebitnem izpadu primarnega centra, bodo aplikacijski strežniki na rezervni lokaciji že delovali le preklopili se bodo na bazo na rezervni lokaciji. Ker so aplikacijski strežniki na obeh lokacijah ves čas aktivni, se nam tudi ne more zgoditi, da bi ob nadgradnjah aplikacije pozabili na strežnike v pripravljenosti na rezervni lokaciji in bi ob morebitnem preklopu aplikacije le ta ne delovala pravilno.

Slika 1 prikazuje dobro postavljene aplikacije: oba centra sta aktivna, aplikacija deluje na dveh strežnikih na primarni lokaciji in na dveh strežnikih na rezervni lokaciji. Baza je aktivna na primarni strani, vse spremembe pa se sproti replicirajo na bazo v pripravljenosti na rezervni lokaciji.



**Slika 1: Dobra postavitev podatkovne baze in aplikacijskih strežnikov.**

Aplikacije morajo biti konfigurirane tako, da znajo delati s podatkovnimi bazami na primarni ali rezervni lokaciji brez intervencije administratorjev. Če bi morali vse aplikacije ročno preklapljati in konfigurirati v času aktiviranja rezervnega centra, bi nam to vzelo preveč časa. Aktivirati bi morali veliko ljudi, da bi vsak opravil svoje delo. V primeru izpada primarne lokacije se ali ročno ali samodejno aktivira podatkovna baza v pripravljenosti na rezervni lokaciji. Preživeli aplikacijski strežniki se namesto na primarno bazo povežejo z bazo, ki je bila do sedaj baza v pripravljenosti in je postala primarna baza.



**Slika 2: Ob izgubi primarnega centra, prevzame glavno vlogo rezervni center.**

Takšna arhitektura nam zagotavlja boljši spanec, s seboj pa prinaša tudi nekaj stranskih učinkov:

- stroški za računski center, opremo in licence se povečajo,
- več je dela,
- potrebujemo dodatni kader ali pa bolj obremeniti obstoječega.

Odločitev je vedno v naših rokah!

### 3.3 Kaj pa če nimamo vsega optimalno postavljenega

Morda imamo dobro postavljen rezervni center, vendar nimamo popolnoma vsega podvojenega. Razlogi za to so različni:

- Morda se nam zdi nepotrebno komplicirati z razvojnimi in testnimi okolji.
- Nekatere produkcijske aplikacije niso nujno potrebne za preživetje podjetja.
- Stare aplikacija ne omogočajo takšne arhitekture.
- Strošek podvojevanja aplikacijskih strežnikov za nekatere aplikacije je prevelik.
- Prepletenost in kompleksnost aplikacij je že sama po sebi nevzdržna in podvojevanje takega sistema je praktično nemogoče.

V takem primeru moramo stvari postaviti od začetka iz obstoječih arhivov. Seveda ni nujno, da nam bo to sploh uspelo. Predstavljajmo si aplikacijo, ki nam jo je v okviru pogodbe namestil prodajalec sam. Potreboval je nekaj mesecev, da je postavil strežnike, podatkovne baze, aplikacijo in poskrbel za vse potrebne konfiguracije in spravil aplikacijo v pogon. Zdaj naj bi sistemski inženirji, administratorji baz in aplikacij to postavili v enem dnevu. To bo zelo težko izvedljivo. Dandanes imajo podjetja na desetine takšnih aplikacij. Nihče jih ne obvlada v popolnosti. Dokumentacija je morda slaba ali je sploh ni. Morda celo podpore za nekatere aplikacije nimamo več, ker so firme ta produkt opustile ali so celo propadle.

Ne glede na vse, bi moralo imeti vsako podjetje vsaj načrt, kako okrevati po katastrofalnem dogodku. Ta načrt mora priti v zavest vseh vpletenih. Morajo ga upoštevati pri svojem delu in tudi izvajati občasne požarne vaje. Vsaka vaja pokaže določene pomanjkljivosti, ki jih je treba neprestano odpravljati.

Postavitev z ničle je zahtevna. Prvi korak je, da uredimo omrežje:

- pridobiti je potrebno IP naslove,
- urediti DNS in
- postaviti pravila na požarnem zidovju.

Tu se že lahko zatakne. Zamislimo si, da je primarni center uničen. Kje imamo dokumentacijo o propustih na požarnem zidovju? Jo sploh imamo? Jo imamo morda na kakšnem strežniku na uničeni primarni lokaciji? Odpiranje požarnega zidovja zna biti precej dolgotrajno, tudi če dokumentacijo imamo. V razvejanem okolju je morda treba urediti požarno zidovje tudi na drugih lokacijah - drugih firmah in celo državah, kar vse vpliva na čas postavitve novega okolja.

Tudi če na rezervni lokaciji nimamo že pripravljenih rezervnih strežnikov, je dobro imeti vsaj pripravljene rezervne IP naslove z ustreznimi propustnostmi na požarnem zidovju. To nam prihrani precej časa pri restavriranju strežnikov tako aplikacijskih kot baznih.

To je osnova, preden sploh začnemo z restavriranjem strežnikov in podatkovnih baz. Zdaj nastopijo sistemski inženirji, ki morajo postaviti aplikacijske in bazne strežnike. Dandanes strežniki niso več fizični stroji, ampak večinoma uporabljamo virtualizacijske mehanizme, kar nam precej poenostavi delo, vendar je tu prvo vprašanje, ki se postavi: koliko procesorjev je treba dodeliti strežniku, koliko pomnilnika, koliko diskovnega prostora? Pri malem številu strežnikov morda to celo poznamo, imamo dober občutek, vendar pri stotinah strežnikov je to nemogoče vedeti na pamet. Če dodelimo preveč resursov, jih bo morda zmanjkalo za druge, če jih dodelimo premalo, pa se morda aplikacije in baze ne bodo pobrale. Ker primarnega centra nimamo več, tam teh informacij ne moremo preveriti. Dobro je vse te podatke sproti pridobivati in jih hraniti nekje na varni lokaciji.

Sledi restavriranje naših strežnikov, podatkovnih baz in aplikacij. Tu pa najprej pade vprašanje: kje imamo arhive? Jih imamo zgolj v primarnem centru? V tem primeru bomo pri katastrofalnem dogodku izgubili tudi te. Vedno jih moramo imeti še na eni lokaciji. Najbolje je, da jih imamo tako na primarni kot rezervni lokaciji. Danes se za arhiviranje uporablja specializirano diskovno polje, ki omogoča samodejno preslikavo arhivov iz primarne lokacije na rezervno. V takem primeru postane restavriranje enostavnejše. Pri restavriranju baz je pomembno še, kje imamo katalog arhivov. Če smo ostali brez njega, bomo potrebovali malo več znanja in spretnosti, da bazo restavriramo. Zelo pomembno je nepréstano utrjevanje spretnosti restavriranja. Večje firme imajo zaposleno večje število administratorjev baz in se znotraj ekipe oblikuje skupina, ki je zadolžena zgolj za arhiviranje in restavriranje baz. V manjših firmah je administratorjev manj in morajo početi vse, kar je povezano z administracijo baz, vendar kot sem že prej zapisal, vsak administrator mora vedeti, da je restavriranje baz njegova najpomembnejša zadolžitev, ki jo mora neprestano utrjevati.

Ko sta podatkovna baza in aplikacija restavrirani, bodo morda potrebne še nastavitve določenih konfiguracij.

Največkrat se izkaže, da je potrebno postoriti še marsikaj. Aplikacija morda sploh ne deluje ali pa deluje le deloma.

Pri restavriranju ne gre vedno vse gladko. Včasih je treba postopek večkrat ponoviti. Pri tem se nabere nekaj nesnage. To je treba za seboj takoj počistiti. Urediti je treba dokumentacijo in ne smemo pozabiti, da je treba novo okolje takoj spet arhivirati.

### 3.4 Biti popolnoma nepripravljen

Niti pomisliti si ne upam, da v svoji organizaciji ne bi imel nobene možnosti takojšnjega preklopa baz oz. aplikacij na rezervni center. Dogodek, ki bi uničil primarno informacijsko infrastrukturo, bi bil poguben za celotno podjetje. Tega nikomur ne priporočam.

Danes obstajajo možnosti oblačnih storitev. Morda si umislimo rezervni center v oblaku vsaj za najpomembnejše informacijske sisteme, da ne ostanemo popolnoma brez vsega. Oblačne storitve niso vedno prava možnost - morda zaradi zakonske regulative, morda ker se bojimo za svoje podatke, da ne bi bili izpostavljeni zlorabam.

## 4 Zaključek

Verjetno si nismo nikoli zamišljali, da bi zaradi naravne katastrofe ali teroristične akcije ostali brez celotne informatike v podjetju. Vendar možnosti katastrofe obstajajo - o tem se lahko prepričamo skoraj vsak dan.

Pomembno je, da:

- se tega zavedamo,
- imamo pripravljeno okolje, da bomo imeli v primeru katastrofe kar najmanj dela za povrnite,
- neprestano utrjujemo svoje znanje,
- težimo k posodabljanju aplikacij in ukinjanju arhaičnih aplikacij brez podpore,
- poenostavljamo svoje sisteme in
- izvajamo ustrezne požarne vaje, da bomo pripravljeni, če pride do katastrofe.

# Hologrami kot nadgradnja inteligentnih rešitev

Ambrož Stropnik,[1] Boštjan Gorenšek,[1] Tomaž Smolčnik,[2] Primož Miheu[1]

[1] GoSMIT R&D d.o.o., Slovenj Gradec, Slovenija
ambroz.stropnik@gosmit.eu, bostjan.gorensek@gosmit.eu, primoz.miheu@gosmit.eu

[2] Šolski center Slovenj Gradec, Višja strokovna šola, Slovenj Gradec, Slovenija
tomaz.smolcnik@sc-sg.si

Prispevek oriše možnosti nadgradnje obstoječih inteligentnih rešitev s hologrami, ki dodajo inteligentnim rešitvam dodano vrednost. Dodana vrednost hologramov se odraža v vizualizaciji podatkov in interakciji z uporabnikom, ki uporabniško izkušnjo dvigne na čisto nov in uporabniku bolj naraven nivo. V prispevku se v okviru inteligentnih rešitev osredotočimo na pogovorne robote, ki so doživeli v zadnjih par mesecih pravi preporod. S tehničnega vidika predstavimo arhitekturo humanoidnega holograma, ki je integriran s pogovornim robotom, ter članek zaključimo z orisom pilotnega projekta in smernicami za nadaljnji razvoj.

**Ključne besede:**
hologrami
vizualizacija umetne inteligence
pogovorni roboti
umetna inteligenca
inteligentne rešitve

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.23

# 1 Uvod

Umetna inteligenca je v zadnjih nekaj letih močno napredovala. Prvi večji preboj umetne inteligence se je zgodil s prebojem globokih nevronskih mrež (angl. deep neural networks) in je omogočal reševanje zapletenih problemov (npr. prepoznava obrazov, zvoka itd…). Z uporabo globokih nevronskih mrež se je začel pravi preporod umetne inteligence na številnih področjih, ki se je iz leta v leto samo še stopnjeval (npr. avtonomna vozila, prevajalniki besedil iz enega jezika v drugega itd…). V zadnjih dveh letih je umetna inteligenca dosegla novi preboj na področju t.i. generativne umetne inteligence (angl. generative artificial intelligence). Slednja se osredotoča na ustvarjanje novih in izvirnih vsebin, kot so slike, zvoki, besedila in celo videoposnetki. Gre za uporabo algoritmov in strojnega učenja za generiranje avtonomnih in ustvarjalnih rezultatov. Tipični predstavnik praktične uporabe generativne umetne inteligence je pogovorni robot Chat GPT, ki temelji na jezikovnih modelih generativne umetne inteligence. Lansiranje omenjenega pogovornega robota v širšo uporabo, je močno pospešil uporabo različnih principov umetne inteligence na vseh področjih (šolstvo, šport, domača uporaba, uporaba v podjetjih). Še več, omenjeno orodje je podalo jasni signal, da bo uporaba principov umetne inteligence na vseh področjih v prihodnje nujno potrebna.

Na tej točki se v prvi vrsti pojavlja vprašanje kaj lahko vsemu temu tehnološkemu napredku umetne inteligence še sledi. Kar v danem trenutku manjka, je možnost ustrezne vizualizacije. Vizualizacija je trenutno sicer mogoča preko konvencionalnih naprav kot so mobilni telefoni, tablice, računalniki… Manjka pa pri vizualizaciji občutek stvarnosti in bolj realnega pogleda na podatke v 3D obliki. Na tej točki se zdi uporaba hologramov logični evolucijski korak tehnologije, saj uporaba hologramov v povezavi z umetno inteligenco ustvarja nove možnosti uporabe:

- Interakcija z uporabnikom: Hologramske projekcije lahko služijo kot vmesnik med uporabnikom in UI. Namesto tradicionalnih zaslonov in tipkovnic bi lahko hologramska projekcija predstavljala vizualno predstavitev UI, s katero bi uporabnik interaktiral z gestami ali glasovnimi ukazi. To bi omogočilo bolj intuitivno in naravno komunikacijo z umetno inteligenco.
- Vizualna predstavitev podatkov: UI lahko uporabijo hologramske projekcije za vizualizacijo podatkov. Namesto da preprosto prikažejo rezultate analiz in informacij na zaslonu, lahko hologrami ustvarijo tridimenzionalne predstavitve, ki omogočajo boljšo predstavo o podatkih in globlje razumevanje.
- Prikaz informacij v resničnem času: Hologrami lahko prikažejo informacije v realnem času na specifičnih lokacijah. S pomočjo senzorjev in UI se lahko hologrami uporabijo za prikaz informacij o okolici, kot so navigacijski podatki, turistične informacije ali podatki o izdelkih v trgovini.
- Izobraževanje in usposabljanje: Hologramske projekcije se lahko uporabijo za izobraževanje in usposabljanje na različnih področjih. Na primer, v medicini lahko hologrami prikažejo tridimenzionalne modele organov ali postopke kirurških posegov, kar omogoča boljšo predstavo in učenje za študente ali zdravnike.
- Ustvarjanje virtualnih asistentov: Hologramske projekcije lahko služijo kot vizualna podoba virtualnih asistentov, ki temeljijo na umetni inteligenci. Namesto uporabe samo glasovnega asistenta, lahko hologram predstavi virtualnega pomočnika, ki je videti kot resnična oseba in omogoča boljšo interakcijo (npr. v logistiki – paketomati, anketiranje ljudi, kultura – prikaz pisatelja in pogovor z njim).

Možnosti uporabe v procesu razvoja proizvoda:

- Vizualizacija konceptov: Hologramske projekcije lahko omogočajo vizualizacijo konceptov izdelkov v tridimenzionalnem prostoru. To omogoča razvijalcem in oblikovalcem, da bolje razumejo in ocenijo obliko, velikost, proporce in druge vidike izdelka, še preden se izdela fizičen prototip. S tem se lahko prihrani čas in stroški, saj se lahko zgodnje faze načrtovanja izvedejo hitreje in bolj učinkovito.
- Kreativno sodelovanje: Hologrami omogočajo več osebam, da hkrati sodelujejo pri razvoju izdelka, ne glede na to, kje se fizično nahajajo. S pomočjo hologramskih projekcij se lahko ustvari skupen virtualni prostor, v katerem lahko razvijalci, oblikovalci in drugi strokovnjaki delijo ideje, prikazujejo modele in se

interaktivno pogovarjajo o izdelku. To olajša sodelovanje na daljavo in omogoča hitro usklajevanje med različnimi ekipami.
- Preizkušanje funkcionalnosti: S pomočjo hologramov se lahko simulira delovanje izdelka na realističen način, še preden se izdela fizični prototip. To omogoča preizkušanje funkcionalnosti, gibanja delov, uporabniškega vmesnika in drugih vidikov izdelka v virtualnem okolju. S tem se lahko zaznajo morebitne napake ali izboljšave, ki jih je treba uvesti pred fizično izdelavo prototipa.
- Predstavitev izdelka interesentom: Hologrami se lahko uporabijo za impresivno predstavitev izdelka interesentom, investitorjem ali strankam. Namesto klasičnih 2D predstavitev ali prototipov lahko hologrami prikažejo realističen in privlačen prikaz izdelka v tridimenzionalnem prostoru. To lahko pomaga pri boljšem razumevanju in ocenjevanju izdelka s strani vpletenih.
- Virtualna prodajna izkušnja: S pomočjo hologramov se lahko ustvari virtualna prodajna izkušnja, v kateri lahko stranke raziskujejo izdelek in njegove funkcije na interaktiven način. To omogoča boljše vključevanje potencialnih strank, predstavitev različnih možnosti izdelka ter simulacijo uporabe in koristi izdelka.

Uporaba hologramov v povezavi z umetno inteligenco in različnimi napravami kot so video kamere in različni senzorji, so lahko izredno močno orodje in dajejo posledično veliko dodano vrednost.
V prispevku pokažemo oris možnosti uporabe tehnologije hologramov skupaj z orisom tehnične arhitekture uporabe hologramov z zalednimi sistemi.

## 2 Hologrami in hologramske rešitve

Hologrami in sam razvoj hologramov se je začel v letu 1947, kjer je britanski fizik Denis Gabor razvil matematični model holografije. Predstavljen model je bil izključno matematični model, ki je temeljil na principu interferometrije [1]. Naslednji pomembni mejnik je bil leta 1962, ko je sovjetski znanstvenik Jurij Denisyuk predstavil prvi delujoči hologram, ki je deloval s pomočjo bele svetlobe in fotografije. Leto kasneje je bil na področju hologramov narejen naslednji pomembni mejnik, ko je ameriški inženir Emmett Leith razvil prvi laserski hologram. Z uporabo laserske svetlobe je ustvaril hologram, ki je omogočal reprodukcijo tridimenzionalne podobe [3].

Od takrat se je holografska tehnologija še naprej razvijala, omogočala pa je napredke na področjih znanosti, zabave, umetnosti in komunikacije. Danes lahko holograme vidimo v različnih oblikah, kot so holografski zasloni, projekcije in interaktivne holografske sisteme.

Prva praktična uporaba holograma se je zgodila v 60. letih 20. stoletja. Leta 1968 je predvajanje holografskih podob postalo javno dostopno na Svetovni razstavi EXPO '68 v Montrealu, Kanada. Na tej razstavi je bilo prikazanih več hologramov, ki so pritegnili pozornost obiskovalcev in širše javnosti [4].

Med najbolj opaznimi primeri praktične uporabe hologramov na tej razstavi je bila holografska podoba zaporniške celice, ki jo je ustvaril umetnik Salvador Dalí. Ta hologram je bil priljubljena atrakcija in pomembna predstavitev holografije kot umetniškega medija [5].

Ta dogodek je pripomogel k širši ozaveščenosti o holografiji in spodbudil nadaljnji razvoj tehnologije. Po tem je sledilo več praktičnih uporab hologramov v različnih panogah, kot so znanost, medicina, zabava, oglaševanje in varnostna industrija.

Od takrat se je holografska tehnologija še naprej razvijala in izboljševala, kar je omogočilo številne inovacije in napredke na tem področju. Danes lahko holograme vidimo v različnih oblikah, kot so holografski koncerti, interaktivni prikazi, holografski zasloni in tridimenzionalne projekcije.

Iz tehničnega vidika, poznamo več različnih tipov hologramov med katerimi sta bolj znana tipa [6, 7, 8, 9]:

- Rotorski: prikaz holografske slike s pomočjo rotorja. Na rotorju so led diode, ki ustvarjajo svetlobo in s pomočjo vrtenja rotorja ustvarijo sliko v 3D obliki (glej sliko 1)

- Prizemski: prikaz holografske slike s pomočjo prizme (glej sliko 2). Ta tip holograma je znan tudi kot »pepperski-fenomen« in je vrsta holograma, ki ustvarja vizualno iluzijo slike v 3D obliki, ki se zdi, da plava ali se dviga nad površino. Bistvena slabost teh hologramov je njihova cena.

**Slika 1: Primer holograma, ki prikazuje sliko s pomočjo rotorja[1].**

**Slika 2: Primer holograma, ki prikazuje sliko s pomočjo prizme[2].**

Obstajajo tudi drugi različni tipi hologramov, ki pa so manj poznani in razširjeni kot omenjena dva tipa.

Hologrami kot takšni so medij s katerim lahko ustvarimo sliko v 3D obliki, ki za razliko od drugih prikazovalnikov (npr. TV itd…) daje vtis resničnosti. Z razvojem tehnologije na preostalih področjih (npr. različni senzorji gibanja itd..), ter v kombinaciji z ustrezno programsko opremo (npr. podprto s pomočjo umetne inteligence), pa hologrami doživljajo nov preporod na samem trgu.

---

[1] Hypervsn Solo L, vir: https://hypervsn.com/hypervsn-solo-combo

[2] © Adobe Stock, slika licencirana za potrebe podjetja GoSMIT R&D d.o.o.

# 3 Inteligentne rešitve

Inteligentne rešitve predstavljajo pomemben del napredka na področju umetne inteligence in tehnologij globokega učenja. Z združevanjem podatkov, algoritmov in naprednih analitičnih metod omogočajo reševanje kompleksnih problemov, napovedovanje trendov ter avtomatizacijo in optimizacijo različnih procesov.

Inteligentne rešitve so kompleksni informacijski sistemi, ki temeljijo na naprednih algoritmih, strojnem učenju in umetni inteligenci. Z uporabo velikih količin podatkov in analitičnih metod omogočajo samodejno učenje, prilagajanje ter izvajanje inteligentnih odločitev in akcij brez človeškega posredovanja.

Razvoj inteligentnih rešitev je bil v zadnjem času izjemno dinamičen. Napredek na področju algoritmov strojnega učenja, globokega učenja in obdelave naravnega jezika je omogočil boljše razumevanje podatkov, zaznavanje vzorcev, napovedovanje in avtomatizacijo nalog.

Inteligentne rešitve najdemo v številnih panogah in podjetjih. V industriji se uporabljajo za optimizacijo proizvodnih procesov, napovedovanje okvar strojev in izboljšanje učinkovitosti. V trgovini se uporabljajo za personalizacijo priporočil izdelkov in izboljšanje uporabniške izkušnje. V finančni industriji se uporabljajo za odkrivanje goljufij, napovedovanje tržnih trendov in izvajanje avtomatiziranih naložbenih strategij. Ti primeri le prikazujejo majhen del obsega in vpliva inteligentnih rešitev v industriji.

Obstaja več različnih tipov inteligentnih rešitev, med katerimi so bolj izpostavljeni sledeči [11, 12, 13]:

- Napredna analitika podatkov: Inteligentne rešitve za napredno analitiko podatkov omogočajo razumevanje, obdelavo in interpretacijo velikih količin podatkov za pridobivanje dragocenih vpogledov. To vključuje tehnike strojnega učenja, analitiko preslikav, napovedne modele in obdelavo naravnega jezika.
- Avtomatizacija procesov: Inteligentne rešitve za avtomatizacijo procesov uporabljajo algoritme umetne inteligence za izvajanje nalog, ki so bile prej izvajane ročno ali s človeškim posredovanjem. To vključuje avtomatizacijo ponavljajočih se nalog, odločanje na podlagi podatkov, samodejno generiranje poročil in druge avtomatizirane operacije.
- Pogovorni roboti: Inteligentne rešitve v obliki pogovornih robotov uporabljajo obdelavo naravnega jezika in algoritme strojnega učenja, da omogočajo interakcijo med človekom in strojem v naravnem jeziku. Pogovorni roboti se uporabljajo v podpornih storitvah, informacijskih centrih, trgovinah in drugih okoljih, kjer je potrebna interakcija z uporabniki.
- Robotika in avtonomni sistemi: Inteligentne rešitve imajo ključno vlogo pri razvoju robotike in avtonomnih sistemov. Roboti in avtonomni stroji, opremljeni s senzorji in naprednimi algoritmi, so postali sposobni samostojnega delovanja, navigacije in odločanja v dinamičnih okoljih. Slednje je vodilo do razvoja avtonomnih vozil, robotske kirurgije, brezpilotnih letalnikov in drugih naprednih sistemov.
- Računalniški vid: Razvoj inteligentnih rešitev na področju računalniškega vida je omogočil zaznavanje in analizo vizualnih podatkov. S pomočjo naprednih algoritmov za prepoznavanje slik in videoposnetkov se lahko identificirajo objekti, zaznavajo obrazni izrazi, prepoznajo vzorci in analizira vedenje. To se uporablja v področjih, kot so varnost, nadzor, medicina, avtonomna vozila in še več.

V članku se osredotočamo na inteligentne rešitve povezane s pogovornimi roboti.

## 3.1 Pogovorni roboti

Pogovorni roboti predstavljajo eno izmed bolj raziskovanih in inovativnih področij umetne inteligence. Z razvojem tehnologij globokega učenja in obdelave naravnega jezika so pogovorni roboti sposobni razumeti in komunicirati s človekom na način, ki je čedalje bolj naraven in učinkovit.

Začetki pogovornih robotov segajo daleč v preteklost, vendar pa se je njihov razvoj v zadnjih letih izjemno pospešil. Na začetku so bili pogovorni roboti preprosti in omejeni na pred nastavljene odgovore. S tehnološkim napredkom in vpeljavo umetne inteligence so se pogovorni roboti razvili v naprednejše sisteme, ki temeljijo na strojnem učenju in globokem učenju.

Med bolj prepoznavnimi pogovornimi roboti je Apple-ov Siri, ki je prvič predstavljen leta 2011. Od takrat so se pojavili številni drugi pogovorni roboti, kot so Amazon-ov Alexa, Google Assistant in Microsoftov Cortana. Ti pogovorni roboti so vgrajeni v pametne naprave kot so pametni telefoni, zvočniki, pametne televizije in domači asistenti [14].

Eden izmed ključnih mejnikov pri razvoju pogovornih robotov je bil dosežen napredek pri razumevanju naravnega jezika. S pomočjo naprednih modelov globokega učenja, kot so rekurentne nevronske mreže (RNN) in kasneje transformerji, so pogovorni roboti sposobni razumeti kompleksne govorne vzorce, prepoznati kontekst in razločevati namen uporabnikovega vprašanja [14, 15].

Poleg razumevanja naravnega jezika se je razvila tudi sposobnost generiranja naravnega jezika (angl. Generative Artificial Intelligence) in sicer z uporabo modelov, kot je model GPT (angl. Generative Pre-trained Transformer) [16], s katero so pogovorni roboti sposobni ustvarjati tekoče in relevantne odgovore ter izvajati pametne dialoge s človekom. Danes so poleg modela GPT znani še številni drugi jezikovni modeli, ki se uporabljajo pri implementacijah pogovornih robotov, npr.: BERT, RoBERT, Bard, LLaMA 2 (Meta) itd…

Z uporabo naprednih jezikovnih modelov pri implementaciji pogovornih robotov pa se odpirajo širše možnosti njihove uporabe. Tako je Gartner lansko leto objavil študijo v kateri ocenjuje, da bodo pogovorni roboti postali primarni kanal za pomoč strankam v naslednjih petih letih [17].

Ne glede na razvoj umetne inteligence in jezikovnih modelov, ki so v zadnjem času naredili preboj pri uporabi pogovornih robotov (npr. Chat GPT), pa jih v osnovi iz tehničnega vidika še vedno delimo na dva tipa pogovornih robotov in sicer: delovanje na osnovi pravil in delovanje na osnovi umetne inteligence [14].

Delovanje pogovornih robotov na osnovi pravil (angl. rule-based chatbots) temelji na vnaprej definiranih pravilih (scenarijih) oziroma odločitvenih drevesih [18]. Osnovna ideja teh pogovornih robotov je ta, da z vnaprej določenimi vprašanji in delnimi odgovori, pogovorni robot vodi uporabnika skozi pogovor in uporabnika pripelje do rešitve oz. posreduje podatke do ustrezne osebe. Uporabljena so lahko enostavna ali zelo zapletena pravila (scenariji). Bistveno pri tem tipu pogovornih robotov pa je to, da niso sposobni odgovarjati na vprašanja izven definiranih pravil in se ne učijo sproti skozi interakcije z ljudmi [18]. Obstaja kar nekaj ogrodij (angl. frameworks), ki omogočajo implementacijo takšnega tipa pogovornega robota: Google DialogFlow, Microsoft Bot Framework…

Drugi tip pogovornih robotov so pogovorni roboti, ki temeljijo na uporabi algoritmov umetne inteligence (angl. Artificial Intelligence Chatbots). V ta sklop pogovornih robotov spadajo pogovorni roboti, ki za svoje delovanje uporabljajo ali rekurentne nevronske mreže, ali napredne jezikovne modele kot je model GPT [19, 20].

## 4 Uporaba hologramov in inteligentnih rešitev iz tehničnega vidika

Hologrami so rešitve, ki so namenjene vizualizaciji podatkov in predstavitvi objekta v 3D obliki. Slednje omogoča drugačen vpogled na predstavljeni objekt kot pa to omogočajo konvencionalni prikazovalniki (npr. monitorji, televizorji itd…). Največja dodana vrednost je prikaz objekta v 3D obliki, ki daje uporabniku občutek resničnosti. Pri tem je potrebno upoštevati še nadgradljivost hologramov z različnimi senzorji (npr. senzorji gibanja…), ter kamerami (npr. za prepoznavo emocij) in mikrofoni, ter zvočniki, ki pa omogočajo interaktivno komunikacijo med hologramom in uporabnikom.

S takšnimi hologrami nadgrajene inteligentne rešitve močno pridobijo na uporabnosti in dodani vrednosti. V industriji, v procesu razvoja novega produkta se lahko razne simulacije novega produkta prikazujejo na hologramu, kjer je možno iz različnih kotov opazovati delovanje novega produkta še preden gre le ta v prototipno izdelavo.

Podobno velja tudi za vizualizacijo različnih podatkov in modelov (npr. molekul), ki jih je s klasičnimi prikazovalniki težje prikazovati in se lahko s hologrami lažje ter bolj nazorno prezentirajo uporabnikom.

Hologrami so na določen način logična nadgradnja pogovornih robotov. Slednji že zdaj dajejo vtis uporabniku pri pogovoru, da se uporabnik pogovarja z osebo. Uporaba holograma v kombinaciji s pogovornim robotom pa to komunikacijo dvigne na nov nivo. Z dobro implementiranim prikazom osebe na hologramu (t.i. avatar) in ustrezno obdelanim zvokom, lahko dosežemo zelo visoko stopnjo občutka resničnosti pri komunikaciji med uporabnikom in pogovornim robotom. Slednjo sploh dosežemo, če se hologram obogati s senzorji gibanja (npr. sledenju uporabnika), kamero za prepoznavo obraza in emocij uporabnika, s spreminjanjem barve in tona glasu med pogovorom.

Eden takšnih naprednih pogovornih robotov, imenovan »Vesna«, je bil razvit na Slovaškem in sicer za banko Erste Group Bank AG [9,10]. Pogovorni robot »Vesna« v obliki holograma svetuje strankam glede bančnih računov, plačilnih kartic in potencialnih investicij. Podprtih je več kot 150 različnih scenarijev s čimer so pokrita vsa finančna področja, ki so potencialno zanimiva za stranke. Iz tehnološkega vidika je pogovorni robot »Vesna« implementiran kot kombinacija različnih naprednih in sodobnih tehnologij kot je generativna umetna inteligenca, prepoznava zvoka (angl. voice recognition), prenos besedil v govor (angl. text to speech), ter zaznava čustev uporabnikov. Na ta način je omogočeno razumevanje strank in posledično kvalitetno podajanje odgovorov strankam. Na sliki spodaj je prikazan pogovorni robot »Vesna« s hologramom [9, 10].

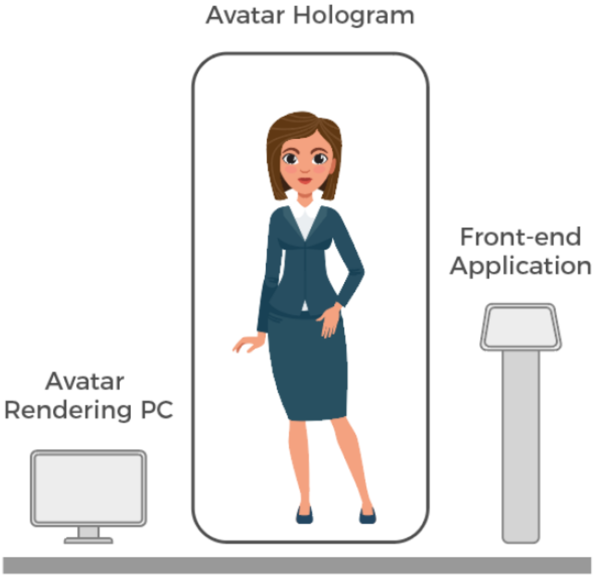
**Slika 3: Primer humanoidnega holograma**
Vir: [10].

### 4.1 Arhitektura sistema z uporabo hologramov

Integracija holograma je v praksi enostavna, saj gre za prikaz podatkov oziroma objekta na samem hologramu, ki pa se izvede na način prenosa slike preko HDMI. Seveda je to eden izmed bolj enostavnih načinov povezav in se kompleksnost povezave holograma s samim sistemom razlikuje glede na same potrebe in zahteve prikaza. Slednje je odvisno predvsem od tega kaj se zraven holograma od strojne opreme še zahteva (npr. senzorji, kamere itd…). Ker je teh možnosti enostavno preveč, se v članku osredotočamo na arhitekturo integracije holograma s pogovornim robotom.

Na sliki spodaj je prikazana shema sistema humanoidnega holograma iz katere je razvidno katere komponente strojne opreme se potrebujejo za implementacijo humanoidnega holograma. Za implementacijo so potrebne naslednje tri komponente:

- **Hologram**. Hologram prikazuje avatarja navidezne osebe.
- **Računalnik**. Računalnik preko katerega se avatar prikazuje na samem hologramu. Slednji je navadno nameščen za ohišjem holograma, da le ta ni viden končnemu uporabniku.
- **Prikazovalnik aplikacije**. Prikazovalnik aplikacije je tablica, ki uporabniku prikazuje pogovornega robota v obliki aplikacije. Slednji je pomemben predvsem pri robnih primerih uporabe, ko prepoznava govora uporabnika zaradi različnih razlogov (npr. nejasni govor) ni uspešna.

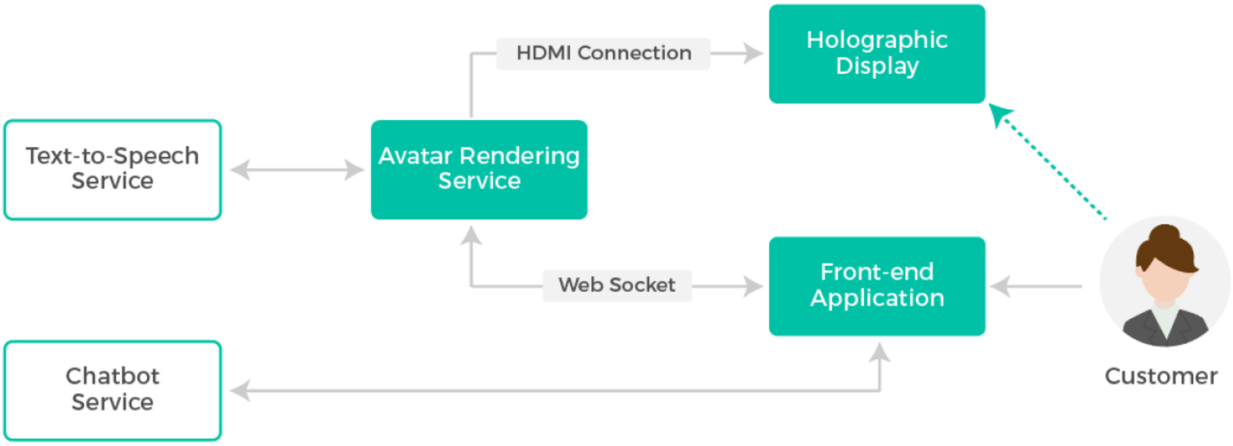


**Slika 4: Shema sistema holograma**
Vir: [9].

Iz tehničnega vidika arhitekturo sistema, ki omogoča postavitev sistema kot je prikazan na sliki 4, prikazuje slika 5 spodaj.

Osrednji del sistema predstavlja aplikacija, ki teče na računalniku preko katerega se na hologramu prikazuje slika osebe oziroma avatar. Ta aplikacija skrbi za ustrezni prikaz avatarja in za povezavo s storitvijo pretvorbe besedila v govor. Slednja je lahko tudi zunanja storitev (npr. Google Text-To-Speech, Microsoft Azure Cognitive Services…) ali pa je del lastne implementacije.

Aplikacija, ki teče na tablici je dejansko grafični vmesnik pogovornega robota (angl. Front-end application) in uporablja mikrofon tablice na kateri teče, ter posledično omogoča storitev preslikave zvoka v besedilo (angl. speech-to-text). Aplikacija je integrirana z osrednjo aplikacijo, ki je zadolžena za prikaz avatarja na hologramu in pretvorbo besedila v zvok. Pogovorni robot je lahko storitev, ki je plod lastne implementacije ali pa je to storitev, ki teče v oblaku – odvisno od zahtev projekta. Na tej točki je potrebno izpostaviti, da je v praksi možna tudi integracija pogovornega robota še z drugimi zalednimi sistemi, kar pa na shemi ni zajeto in je odvisno od same domene implementacije projekta.



**Slika 5: Arhitektura sistema.**

Sam nabor tehnologij (angl. technical stack) za implementacijo arhitekture sistema kot je prikazan na sliki 5, je lahko zelo različen in močno odvisen od zahtev projekta kamor spada tudi ustrezna izbira orodij in tehnologij. V kolikor gre za implementacijo enostavnejših primerov humanoidnih pogovornih robotov, je možno relativno enostavno uporabiti oblačne storitve, ki so že na voljo (npr. Google Cloud AI, Microsoft Azure…). V tem primeru je potrebno z ustreznim orodjem podpreti avatarja (npr. Unity ali Blender), ter implementirati aplikacijo za predstavitev avatarja na hologramu in aplikacijo, ki je grafični vmesnik pogovornega robota na tablici (npr. implementacija v programskem jeziku Flutter).

V kolikor so zahteve projekta bolj kompleksne, je potrebno za omenjene oblačne rešitve iskati ustrezna orodja.

## 4.2 Primer pilotnega projekta

V podjetju se lotevamo pilotnega projekta implementacije humanoidnega holograma v kombinaciji s pogovornim robotom, ki bo po videzu, glasu in znanju (znanju pogovornega robota) upodobil znanega pokojnega pisatelja Leopolda Suhodolčana. Ideja pilotnega projekta je pobližje se spoznati s tehnologijo hologramov na eni strani in zmožnosti tehnologije pogovornih robotov v kombinaciji s hologramom. Tu je izziv večplasten, kjer je potrebno zagotoviti sledeče:

- Na osnovi filmskega in slikovnega gradiva pisatelja, izdelati realistični 3D model pisatelja.
- Na osnovi zvočnih zapisov izluščiti način njegovega govora (frekvenco), ter različne tone in barve glasu (glede na situacijo pogovora).
- Implementacija pogovornega robota na osnovi GPT modela, ki bo naučen iz pisateljevih del, rokopisov in monografije.

Posebni izziv bo vse omenjene segmente povezati v smiselno in realistično celoto, kjer bo uporabljen Holographic Human Hologram podjetja HyperVSN, ter njihova storitev Digital Avatar, ki bo pogovornega robota ustrezno povezala s sliko na hologramu in zvoku, ki se bo predvajal sinhrono s sliko 3D pisatelja na hologramu. Pri implementaciji bo uporabljena arhitektura, ki je bila predstavljena v poglavju 4.1 na sliki 5.

Ideja je, da bo omenjen pilotni projekt za določeno obdobje na voljo v izbrani knjižnici na Koroškem, kjer bomo merili tudi odziv uporabnikov na implementirano rešitev. Slednje je poleg tehnološkega izziva pomemben segment, saj hologrami v kombinaciji s pogovornimi roboti v praksi pri končnih uporabnikih predstavljajo veliko neznanko.

## 5 Zaključek

V prispevku smo predstavili holograme in način integracije hologramov z inteligentnimi rešitvami na konkretnem primeru pogovornega robota. Hologrami lahko močno obogatijo uporabniško izkušnjo, ter inteligentnim rešitvam dodajo dodano vrednost v smislu predstavitve in interakcije z uporabnikom v tridimenzionalnem prostoru. Slednje se lahko odraža kot obogatitev pogovornega robota s prikazom avatarja na hologramu, pregled novega produkta v primeru razvoja produkta v proizvodnem procesu ali le kot vizualizacija podatkov. Smernice Gartnerja kažejo v vedno večjo uporabo hologramskih rešitev v kombinaciji z inteligentnimi rešitvami (npr. virtualni asistenti, virtualni prodajalci itd…). Predstavljajte si svet, kjer konference vključujejo 3D predstavitve udeležencev in gostujoče predavatelje, s katerimi lahko kljub oddaljenosti komunicirate večplastno in v tridimenzionalnem prostoru. Te rešitve so trenutno izjemno butične a bosta čas in trg v prihodnosti pokazala v kolikšni meri in kdaj se bodo napovedi množične uporabe tudi uresničile.

## Literatura

[1] Gabor Dennis »A new microscopic principle«, Nature, 161 (4098), 1946

[2] Denisyuk Yuri N, »On the reflection of optical properties o fan object in a wave field of light scattered by it«, Doklady Akademii Nauk SSSR, 1962

[3] Upatniek Juris, Leaonard Carl, »Diffraction efficiency of bleached photographically recorded intereference patterns«, Applied Optics 8, 85-89, 1969

[4] »The Nobel Prize in Physics 1971«, Nobelprize.org. Obiskano 21. 4 2023.

[5] A. Graube, »Advances in Bleaching Methods for Photographically Recorded Holograms«, Appl. Opt. 13, 2942-2946 (1974)

[6] »Holographic 3D Technology: From Sci-fi Fantasy to Engineering Reality«, International Year of Light 2015 – Blog (light2015blog.org), 28.4.2015, Archived from the original on 30.4.2017.

[7] »A Holography FAQ«, HoloWiki. 26.5.2023. Archived from the original on 6 November 2010. Retrieved 21 April 2012.

[8] Johnston, Sean F., »11 - Channeling Dreams«, Holograms: A Cultural History. Oxford University Press. ISBN 978-0191021381, 2015

[9] https://hypervsn.com/, HYPERVSN is a 3D Integrated Holographic System for advertising, digital signage, events, obiskano 29. 4. 2023

[10] https://hypervsn.com/blog/slovenska-sporitelna-introduced-holographic-assistant-in-banking-industry.html, Slovenská Sporiteľňa Brings the First Holographic Assistant to Banking, obiskano 16. 6. 2023

[11] Hu, X., Huang, M., & Zeng, A. Z. (2007). An intelligent solution system for a vehicle routing problem in urban distribution. International Journal of Innovative Computing, Information and Control, 3(1), 189-198.

[12] Gupta, A. K., & Kumar, S. (2022, December). Intelligent Solution Guidance Tool (ISGT). In 2022 IEEE International Conference for Women in Innovation, Technology & Entrepreneurship (ICWITE) (pp. 1-3). IEEE.

[13] Huang, X. L., Ma, X., & Hu, F. (2018). Machine learning and intelligent communications. Mobile Networks and Applications, 23, 68-70.

[14] Lokman, A. S., & Ameedeen, M. A. (2019). Modern chatbot systems: A technical review. In Proceedings of the Future Technologies Conference (FTC) 2018: Volume 2 (pp. 1012-1023). Springer International Publishing.

[15] Shawar, B. A., & Atwell, E. (2007, April). Different measurement metrics to evaluate a chatbot system. In Proceedings of the workshop on bridging the gap: Academic and industrial research in dialog technologies (pp. 89-96).

[16] Panda, S., & Kaur, N. (2023). Exploring the viability of ChatGPT as an alternative to traditional chatbot systems in library and information centers. Library Hi Tech News, 40(3), 22-25.

[17] https://www.gartner.com/en/newsroom/press-releases/2022-07-27-gartner-predicts-chatbots-will-become-a-primary-customer-service-channel-within-five-years, obiskano 26. 7. 2023

[18] Farah, J. C., Spaenlehauer, B., Ingram, S., Purohit, A. K., Holzer, A., & Gillet, D. (2023). Harnessing Rule-Based Chatbots to Support Teaching Python Programming Best Practices. In 26th International Conference on Interactive Collaborative Learning (ICL2023).

[19] Aggarwal, A., Tam, C. C., Wu, D., Li, X., & Qiao, S. (2023). Artificial Intelligence–Based Chatbots for Promoting Health Behavioral Changes: Systematic Review. Journal of Medical Internet Research, 25, e40789.

[20] Liu, L., & Duffy, V. G. (2023). Exploring the Future Development of Artificial Intelligence (AI) Applications in Chatbots: A Bibliometric Analysis. International Journal of Social Robotics, 15(5), 703-716.

# Uporaba programskega jezika Julia za namene podatkovne znanosti

Tadej Lahovnik, Grega Vrbančič, Iztok Fister ml., Vili Podgorelec

Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko,
Maribor, Slovenija
tadej.lahovnik1@um.si, grega.vrbancic@um.si,
iztok.fister1@um.si, vili.podporelec@um.si

Programski jezik Julia je v zadnjih letih pridobil veliko pozornosti zaradi svoje edinstvene kombinacije zmožnosti programiranja na visoki ravni, zmogljivosti in enostavne uporabe. Dandanes programski jezik spada med tri najpomembnejše jezike za namene podatkovne znanosti poleg jezika Python in R. Namen tega članka je zagotoviti pregled ključnih lastnosti jezika Julia in prikazati njegov potencial kot zmogljivega orodja za analizo podatkov, vizualizacijo in strojno učenje. V članku predstavimo temeljne koncepte programskega jezika Julia, vključno z njegovo sintakso, podatkovnimi strukturami in ekosistemom paketov. Dodatno se v članku posvetimo opisu bogatega nabora funkcij za manipulacijo in analizo podatkov, ki jih ponuja Julia. Poleg tega članek obravnava vizualno predstavitev podatkov z uporabo zmogljivih vizualizacijskih knjižnic programa Julia. Prikažemo, kako Julia omogoča ustvarjanje vizualno privlačnih in interaktivnih grafikonov, diagramov in nadzornih plošč, ki nam omogočajo učinkovito pridobivanje novih spoznanj, ki temeljijo na podatkih. V zaključku predstavimo še primerjavo med jeziki Julia, Python in R ter navedemo prednosti jezika Julia za podatkovne znanstvenike.

**Ključne besede:**
podatkovna znanost
strojno učenje
Julia
velepodatki
globoko učenje

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.24

# 1 Uvod

V zadnjih letih smo priča veliki rasti podatkov, ki nastajajo na različnih področjih, vse od različnih industrijskih okolij, sekvenciranja genetskih podatkov, zbiranja podatkov za analitiko in podobno [1]. Posledično se je zaradi rasti podatkov kot tudi njihove dostopnosti ter izzivov, ki jih prinašajo, močno razvilo področje podatkovne znanosti (angl. data science), kjer poskušajo podatkovni znanstveniki (angl. data scientists) in ostali deležniki izkoristiti moč odločanja na podlagi podatkov in s tem dobiti globlji vpogled v zakonitosti, ki so skrite v podatkih [2]. Zaradi tega se je v zadnjih letih razvilo tudi veliko orodij, ki olajšajo delo podatkovnim znanstvenikom ter ljudem, ki rokujejo z veliko količino podatkov. Poleg različnih komercialnih rešitev za namene spopadanja s problemi podatkovne znanosti in ostalih programskih paketov v že obstoječih programskih jezikih, so se pojavili tudi nekateri novi programski jeziki, ki so namenjeni delu s podatki, izvajanju numeričnih analiz, podatkovnemu rudarjenju, strojnemu učenju in vizualizaciji podatkov. Eden izmed njih je programski jezik Julia, ki je visokozmogljiv programski jezik posebej zasnovan za reševanje izzivov povezanih z umetno inteligenco, analitiko velikih količin podatkov in podatkovno znanost [3]. Programski jezik Julia, ki je nastal s skupnimi prizadevanji raznolike skupnosti raziskovalcev in razvijalcev, ponuja edinstveno kombinacijo preprostosti, hitrosti in prilagodljivosti, zato je idealna izbira za podatkovno-usmerjene aplikacije. Ena od ključnih gonilnih sil priljubljenosti programskega jezika Julia je njegova zmožnost učinkovitega izvajanja numeričnih in znanstvenih izračunov. S pomočjo prevajalnika JIT (just-in-time) Julia dosega zmogljivost, primerljivo s statično tipiziranimi jeziki, kot sta jezika C ali Fortran, hkrati pa zagotavlja enostavnost uporabe in dinamičnost, ki sta povezani z jeziki, ki uporabljajo tolmač. Ta prednost izhaja iz Juliinega inovativnega sistema tipov in večkratnega pošiljanja, ki omogočata učinkovito generiranje in optimizacijo kode. Na področju umetne inteligence in strojnega učenja se zmožnosti programskega jezika Julia kažejo z obsežnim ekosistemom paketov in knjižnic, ki so dostopne preko Juliinega repozitorija paketov[1]. Julia ponuja veliko orodij, ki raziskovalcem in praktikom omogočajo hitro izdelavo prototipov in uvajanje modelov umetne inteligence, ne glede na to, ali gre za gradnjo globokih nevronskih mrež, izvajanje algoritmov za obdelavo naravnega jezika ali raziskovanje najsodobnejših tehnik okrepitvenega učenja. Poleg tega se Julia brez težav povezuje s priljubljenimi ogrodji, kot sta TensorFlow in PyTorch, kar omogoča interoperabilnost in izkorišča prednosti različnih ekosistemov.

V tem strokovnem članku bomo predstavili osnovne značilnosti in zmogljivosti programskega jezika Julia, poudarili njegove prednosti in razpravljali o tem, kako se umešča v širši kontekst sodobnih metod umetne inteligence, analitike velikih količin podatkov, podatkovne znanosti in naraščajočih zahtev po razložljivi umetni inteligenci (angl. Explainable Artificial Intelligence, krajše XAI) [4].

Struktura članka je naslednja: v drugem poglavju so predstavljene glavne značilnosti programskega jezika Julia. V tretjem poglavju je prikazana uporaba programskega jezika Julia za namene podatkovne znanosti. V zadnjem četrtem poglavju je narejen povzetek članka.

---

[1] https://juliapackages.com

## 2 Programski jezik Julia

Julia je odprtokoden, visoko zmogljiv, ekspresiven in sodoben programski jezik za znanstveno računalništvo in obdelavo podatkov. Začetek razvoja programskega jezika Julia sega v leta 2009. Prva stabilna verzija (v1.0) je bila izdana leta 2018, najnovejša verzija (v1.9.2) pa julija 2023 [3], [5], [6]. Glavne značilnosti programskega jezika Julia so:

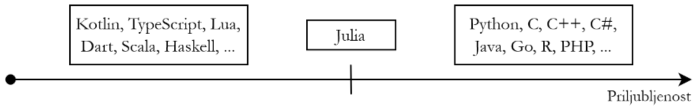
- večkratno pošiljanje (angl. multiple dispatch), kar pomeni posplošitev polimorfnega mehanizma, ki se uporablja v objektno-orientiranih programskih jezikih (OOP) [7],
- dinamično tipiziran jezik, kar pomeni, da tipov spremenljivk ni treba deklarirati v kodi. Poudariti je treba, da Julia ne omogoča statičnega preverjanja tipov, zaradi česar se lahko med izvajanjem pojavi t. i. »type error« [6],
- JIT prevajalnik z uporabo ogrodja LLVM[2], ki skupaj z zasnovo jezika Julii omogoča doseganje maksimalne zmogljivosti za numerično, tehnično in znanstveno računanje [6],
- vzporedno računanje (angl. parallel computing), ki uporabnikom omogoča prekinitev in nadaljevanje izračunov s popolnim nadzorom nad komunikacijo, brez potrebe po ročni interakciji z razporejevalnikom (angl. scheduler) operacijskega sistema [3].

V tabeli 2 je predstavljena osebna izkaznica programskega jezika Julia [3], [7], [8].

**Tabela 2: Osebna izkaznica programskega jezika Julia.**

| | |
|---|---|
| Nastanek jezika | 2009 |
| Prevajalnik | JIT z uporabo ogrodja LLVM |
| Nastal po vzoru sledečih programskih jezikov | Python, R, C, Lisp, Lua, … |
| Operacijski sistem | Deluje na vseh platformah |
| Končnice datotek | .jl |
| Spletna stran | https://julialang.org |
| GitHub repozitorij | https://github.com/JuliaLang/julia |

Priljubljenost programskega jezika dokazuje lestvica »Tiobe Index«, ki Julio uvršča na 24. mesto med vsemi programskimi jeziki. Na sliki 1 je prikazan odsek Tiobe lestvice in umestitev programskega jezika Julia.



**Slika 1: Delna Tiobe lestvica**
Vir: [9].

Programski jeziki Python, Java, PHP, Go in R so glede na priljubljenost prepričljivo pred Julio, a se ta uvršča pred Kotlin, TypeScript in Luo. Glavni razlog za zaostanek v priljubljenosti leži v nestabilnosti programskega jezika. Razvijalci neprenehoma pripravljajo izdaje, ki vključujejo manjše spremembe, le-te pa pogosto ne ponujajo združljivosti za nazaj (angl. backwards compatibility). Nekaj dodatnih razlogov za manjšo priljubljenost navaja spletni vir [10].

V nadaljevanju poglavja je predstavljenih nekaj preprostih in kratkih primerov uporabe programskega jezika Julia.

---

[2] low-level virtual machine

### 2.1 Uvodni program "Hello world"

Klasični uvodni program (t. i. »Hello World«) v programskem jeziku Julia zapišemo, kot prikazuje levi stolpec tabele 3. Če želimo, da se po besedilu izpiše še prazna vrstica, lahko namesto »print« uporabimo funkcijo »println«.

**Tabela 3: Hello World.**

```
'hello.jl'
print("Hello World")
```

```
>> julia hello.jl
>> Hello World
```

Programsko kodo po uspešni namestitvi Julie poženemo z ukazom »julia hello.jl«. V novi vrstici terminalnega okna se nato izpiše »Hello World«. S tem preprostim zgledom smo uspešno ustvarili svoj prvi program v programskem jeziku Julia.

### 2.2 Funkcije

V naslednjem primeru ustvarimo preprosto funkcijo (poimenovano »kvadriranje«). Funkcija prejme parameter »st« in ga pomnoži s seboj. Celotna implementacija funkcije »kvadriranje« je prikazana v levem stolpcu tabele 4. Po definiranju funkcije je treba blok kode zaključiti s ključno besedo »end«. Desni stolpec prikazuje poganjanje datoteke »kvadriranje.jl« in izpis rezultata funkcije v ukazni vrstici.

**Tabela 4: Funkcija kvadriranja.**

```
'kvadriranje.jl'
function kvadriranje(st)
   return st^2
end
print(kvadriranje(parse(Int64, ARGS[1])))
```

```
>> julia kvadriranje.jl 5
>> 25
```

V tem primeru je parameter funkcije podan kot vhodni parameter ob pogonu programa. Funkciji bi lahko parameter podali tudi neposredno v programski kodi. To dosežemo s spremembo zadnje vrstice kode, kjer zamenjamo parameter funkcije »kvadriranje« (npr. »print(kvadriranje(5))«). Kodo bi nato pognali z ukazom »julia kvadriranje.jl«.

### 2.3 Pogojni stavki

Pogojni stavki v Julii uporabljajo ključne besede »if«, »elseif« in »else«. Podobno kot funkcije je pogojne stavke treba zaključiti s ključno besedo »end«. Primer pogojnega stavka, ki preverja sodost in lihost števila, je prikazan v levem stolpcu tabele 5.

**Tabela 5: Pogojni stavki.**

```
'pogoj.jl'
function sodoLiho(st)
   if st%2 == 0
      print("Sodo")
   else
      print("Liho")
   end
end
sodoLiho(3)
```

```
>> julia pogoj.jl
>> Liho
```

Za preproste logične izjave lahko namesto bloka kode uporabimo ternarni operator (»?«). Del kode, ki sledi ternarnemu operatorju, se izvede, če je pogoj pred ternarnim operatorjem resničen, sicer se izvede del kode, ki sledi operatorju »:«. Tabela 6 prikazuje preverjanje sodosti in lihosti števila z uporabo ternarnega operatorja.

Tabela 6: Ternarni operator.

```
'pogoj.jl'
function sodoLiho(st)
   st%2 == 0 ? print("Sodo") : print("Liho")
end
sodoLiho(4)
```

```
>> julia pogoj.jl
>> Sodo
```

## 2.4 Zanke in funkcija »map«

Znotraj zanke lahko z uporabo »continue« preskočimo iteracijo. Z uporabo »break« se izvajanje zanke prekine. Podobno kot funkcije in pogojne stavke je zanke treba zaključiti s ključno besedo »end«. S pomočjo funkcije »map« lahko vrednostim v obstoječem seznamu priredimo nove vrednosti. V tem primeru vsa števila v seznamu kvadriramo in jih shranimo v nov seznam. V levem stolpcu tebele 7 sta prikazana primera uporabe zank in funkcije »map«.

Tabela 7: Zanke in funkcija »map«.

```
'zanka.jl'
for i in 1:10
   if i % 2 != 1
      continue
   end
   if i >= 8
      break
   end
   print(i, " ")
end
s = map(i -> i^2, [1, 2, 3, 4, 5])
print(s)
```

```
>> julia zanka.jl
>> 1 3 5 7
>> [1, 4, 9, 16, 25]
```

## 2.5 Seznami

Naslednji primer prikazuje uporabo seznamov (angl. array) v programskem jeziku Julia. Treba je poudariti, da se indeksiranje elementov v seznamu v Julii začne z 1 [11]. Tabela 8 prikazuje različne načine ustvarjanja seznamov. Seznam »s1« se ustvari skupaj s svojo vsebino, tip seznama pa se določi samodejno. Seznam »s2« je prazen in nima tipa, zaradi česar nekatere operacije nad seznami ne delujejo (npr. funkcija »push!« za dodajanje elementov). Seznam »s3« je prav tako prazen, vendar ima definiran tip. Sezname lahko ustvarimo tudi s pomočjo t. i. »list comprehensions« (npr. seznam »s4«), ki delujejo na podoben način kot funkcija map v prejšnjem primeru.

Tabela 8: Ustvarjanje seznamov.

```
julia> s1 = [1, 2, 3]
3-element Vector{Int64}: [1, 2, 3]
julia> s2 = []
0-element Vector{Any}: Any[]
julia> s3 = Int64[]
0-element Vector{Int64}
julia> s4 = [i^2 for i = 1:3]
3-element Vector{Int64}: [1, 4, 9]
```

Primer v tabeli 9 prikazuje uporabo t. i. »dequeue« funkcij nad seznami, ki jih ponuja programski jezik Julia. Med dequeue funkcije sodijo »push!«, »pop!«, »insert!«, »deleteat!«, »splice!«, »resize!«, »append!« in »prepend!«. Te omogočajo dodajanje in odstranjevanje elementov ter združevanje več seznamov v enega. Klicaj, ki sledi imenu funkcije, omogoča posodobitev prvega parametra, ki ga funkcija prejme.

**Tabela 9: »Dequeue« funkcije.**

```
julia> s = Any[1, 2, 3]
3-element Vector{Any}: Any[1, 2, 3]
julia> push!(s, "štiri")
4-element Vector{Any}: Any[1, 2, 3, "štiri"]
julia> pop!(s)
3-element Vector{Any}: Any[1, 2, 3]
julia> append!(s, ["štiri", "pet"])
5-element Vector{Any}: Any[1, 2, 3, "štiri", "pet"]
julia> deleteat!(s, 4)
4-element Vector{Any}: Any[1, 2, 3, "pet"]
```

### 2.6 Ekosistem paketov

Julia ponuja lastni upravitelj paketov, imenovan Pkg. Ta je za razliko od običajnih upraviteljev paketov, ki namestijo in upravljajo en globalni nabor paketov, zasnovan na podlagi "okolij"; neodvisnih naborov paketov, ki so lahko lokalni za posamezen projekt ali skupni in izbrani po imenu. Natančen nabor paketov in različic v okolju je zabeležen v manifestni datoteki, ki jo je mogoče preveriti v repozitoriju projekta in spremljati v sistemu za upravljanje z izvorno kodo (angl. version control), kar znatno izboljša ponovljivost projektov [12].

Upravitelj paketov Pkg omogoča enostavno nameščanje in posodabljanje paketov. Za namestitev novega paketa se uporablja ukaz »Pkg.add(»[ime paketa]«), za posodobitev paketov pa »Pkg.update()«. Nameščene pakete nato uporabimo v kodi z ukazoma »using [ime paketa]« in »import [ime paketa]«. Prvi način uvozi vse funkcije, ki jih paket ponuja, v trenutni imenski prostor, kar omogoči klicanje funkcije brez eksplicitne navedbe paketa. Pri drugem načinu je treba eksplicitno določiti, iz katerega paketa se funkcija kliče (»[ime paketa].funkcija()«). Ta način je predvsem uporaben, če se ime funkcije pojavi v več paketih [11].

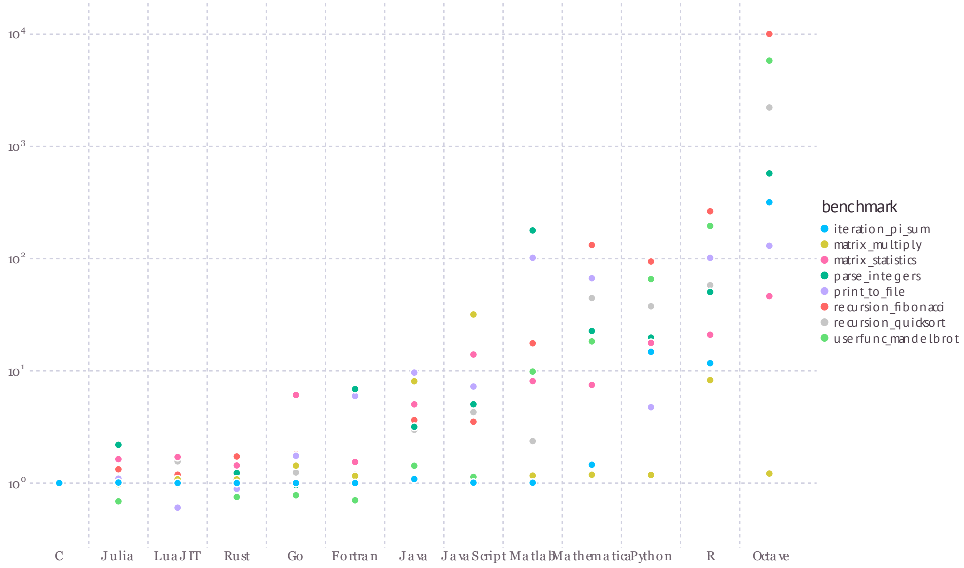
Bogat ekosistem paketov vključuje pakete za delo s podatki (npr. DataFrames.jl, CSV.jl), vizualizacijo podatkov (npr. Plots.jl, PlotlyJS.jl), strojno učenje (npr. TimeSeries.jl, Clustering.jl) in obdelavo slik (npr. Images.jl, ImageSegmentation.jl, ImageTransformations.jl) [13].

## 3 Programski jezik Julia in podatkovna znanost

O Pythonu se zadnja leta veliko govori. Glede na podatke indeksa Tiobe [9] je Python v letih 2018, 2020-2021 in v tem letu bil razglašen za programski jezik leta. Izjemno priljubljen je zlasti med podatkovnimi znanstveniki in strokovnjaki za strojno učenje ter se v veliki meri uporablja za umetno inteligenco.

Python je odprtokodni programski jezik; njegova preprostost in kratka krivulja učenja sta med ključnimi razlogi za njegovo priljubljenost. Rast Pythona je v zadnjem času skokovito narasla; vendar je Julia, razmeroma novejši jezik, v zadnjem času požel veliko zanimanja in iskanja še posebej v skupnosti podatkovnih znanstvenikov in ekspertov strojnega učenja. Ključen vzrok za vedno večje zanimanje v prej omenjeni skupnosti je hitrost izvajanja, ki je še kako pomembna pri časovno zahtevnih algoritmih strojnega učenja in obdelavi velikih količin podatkov.

Obstajajo različne primerjalne študije za primerjavo hitrosti Julie z drugimi programskimi jeziki. Prvi primer, ki ga lahko omenimo, je z uradne spletne strani jezika Julia [14]. V tem poskusu je osem algoritmov izvedenih v 12 različnih jezikih. Julia je bila med najuspešnejšimi v primerjavi s C, Lua in Rust. Presenetljivo je, da so najslabše rezultate v raziskavi dosegli jeziki, ki so jih najpogosteje uporabljali strokovnjaki s področja podatkovne znanosti in strojnega učenja: Matlab, Mathematica, Python, R in Octave v padajočem vrstnem redu (slika 2).



**Slika 2: Primerjava hitrosti izvajanja različnih algoritmov v različnih programskih jezikov.**
Vir: [14].

Avik Sengupta v knjigi Julia High Performance [15] primerja zmogljivost 10 programskih jezikov s C za izračun Mandelbrotove množice. Tudi v tem primeru je Julia med najuspešnejšimi z jeziki Fortran, Javascript in Lua. Po drugi strani pa so najmanj uspešni jeziki numeričnega računanja in podatkovne znanosti: Mathematica, Matlab, Python in R v padajočem vrstnem redu.

V nadaljevanju bomo predstavili ključne podobnosti oz. razlike iz vidika podatkovne znanosti med programskima jezikoma Julia in Python. Čeprav bo primerjava v splošnem med omenjenima programskima jezikoma, lahko te apliciramo tudi na programski jezik R, saj ta iz stališča izvajanja deluje podobno kot Python.

### 3.1 Branje podatkov

Prva naloga v vsakem delovnem postopku analize podatkov je preprosto branje podatkov, kar je treba nujno opraviti hitro in učinkovito, da se lahko začne bolj zanimivo delo. V številnih panogah in področjih je datotečni format CSV kralj za shranjevanje in izmenjavo tabelaričnih podatkov. Hitro in zanesljivo nalaganje datotek CSV je ključnega pomena, poleg tega pa mora biti dobro skalirano za različne velikosti datotek, vrste podatkov in oblike. V tem prispevku primerjamo zmogljivost branja 8 različnih naborov podatkov iz resničnega sveta s tremi različnimi razčlenjevalniki CSV: fread iz programa R, read_csv iz programa Pandas in CSV.jl iz programa Julia[3]. Vsak od njih je bil izbran kot »najboljši v svojem razredu« razčlenjevalnik (angl. parser) datotek CSV v programskih jezikih R, Python in Julia. Večnitnost je bistvenega pomena za doseganje največje zmogljivosti današnjih računalnikov, vendar je le en jezik (Julia) lahko dosledno in učinkovito uporabljal več jeder.

Vsa tri orodja imajo zanesljivo podporo za nalaganje najrazličnejših podatkovnih vrst s potencialno manjkajočimi vrednostmi, vendar samo knjižnici fread (R) in CSV.jl (Julia) podpirata večnitno nalaganje medtem ko Pandas podpira samo enonitno nalaganje datotek CSV. Knjižnica CSV.jl je posebna tudi zato, ker je edina, ki je v celoti implementirana v enakem programskem jeziku in ni implementirana v katerem izmed nižjenivojskih programskih jezikov (npr. C) ter zgolj ovita v programskem jeziku Python ali R. Pandas ima sicer nekoliko zmogljivejši razčlenjevalnik, ki temelji na jeziku Python, vendar je bistveno počasnejši in skoraj vse uporabe funkcije read_csv privzeto uporabljajo C implementacijo. Zato takšna primerjava ne predstavlja zgolj prikaza hitrosti nalaganja podatkov v programskem jeziku Julia, temveč nakazuje tudi na splošno zmogljivost programskega jezika.

---

[3] Opis eksperimenta skupaj z rezultati je na voljo na https://www.queryverse.org/benchmarks

Primerjava s stališča hitrosti izvajanja nalaganja podatkov kažejo, da je CSV.jl v programskem jeziku Julia 1,5- do 5-krat hitrejši od Pandas, tudi če je omejen na eno jedro; z omogočeno večnitnostjo je lahko več kot 20-krat hitrejši. R-jev fread je bolj konkurenčen, saj podpira večnitnost, vendar v številnih podatkovnih nizih še vedno zaostaja za Julio za 10-krat ali več. Za primerjalno analizo so bila uporabljena naslednja orodja: BenchmarkTools.jl za Julio, microbenchmark za R in timeit za Python.

## 3.2 Obdelava podatkov

Ko govorimo o obdelavi podatkov, imamo v mislih obdelavo strukturiranih podatkov, ki so navadno podani v obliki datotek CSV. V programskem jeziku Python je prva izbira za ta namen uporaba ene izmed najpopularnejših knjižnic; Pandas. Alternativa omenjeni knjižnici v programskem jeziku Julia je DataFrames.jl, ki je široko uporabljena knjižnica za obdelavo in analizo podatkov. Zagotavlja podobne funkcije kot Pandas v Pythonu in omogoča delo s strukturiranimi podatki v tabelarični obliki.

### *3.2.1 Ustvarjanje podatkovnih okvirjev*

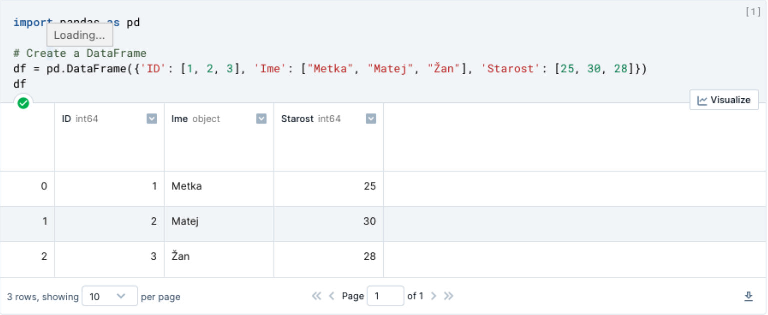
Za namen primerjave uporabe vzemimo ustvarjanje osnovnega podatkovnega okvirja (angl. DataFrame) v obeh programskih jezikih. Najprej zgradimo enostaven podatkovni okvir s tremi značilnicami: ID, ime in starost v programskem jeziku Julia (slika 3). V ta namen ustvarimo primerek razreda DataFrame, ki mu podamo vhodne parametre.

```
using DataFrames

df = DataFrame(ID=[1, 2, 3], Ime=["Metka", "Matej", "Žan"], Starost=[25, 30, 28])
```

```
3×3 DataFrame
 Row   ID      Ime      Starost
       Int64   String   Int64
   1       1   Metka         25
   2       2   Matej         30
   3       3   Žan           28
```

**Slika 3: Primer ustvarjanja podatkovnega okvirja v programskem jeziku Julia.**

Enako nato ponovimo tudi v programskem jeziku Python (slika 4). Kot je razvidno, je implementacija izgradnje enostavnega podatkovnega okvirja praktično enaka. Razlika je zgolj v načinu podajanja parametrov konstruktorju razreda DataFrame, končen rezultat pa je enak.



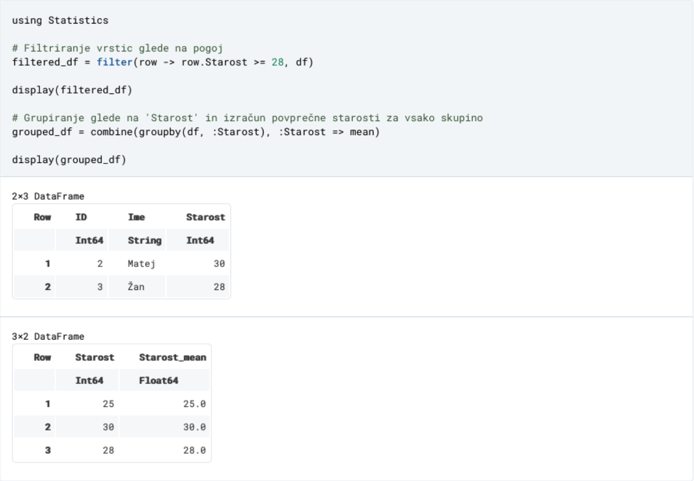
```
import pandas as pd

# Create a DataFrame
df = pd.DataFrame({'ID': [1, 2, 3], 'Ime': ["Metka", "Matej", "Žan"], 'Starost': [25, 30, 28]})
df
```

```
     ID int64   Ime object   Starost int64

0           1   Metka                   25
1           2   Matej                   30
2           3   Žan                     28

3 rows, showing 10 per page      Page 1 of 1
```

**Slika 4: Primer ustvarjanja podatkovnega okvirja v programske jeziku Python.**

### 3.2.2 *Manipulacija nad podatki*

V procesu obdelave podatkov je vsekakor ključna možnost manipulacije nad podatki, ki v splošnem vključuje operacije zapolnjevanja manjkajočih podatkov, transformiranje, preoblikovanje, filtriranje, grupiranje, agregiranje, itd. podatkov. Za namen prikaza uporabe bomo prikazali uporabo filtriranja in grupiranja ter izračuna povprečnih vrednosti. Slika 5 prikazuje uporabo prej omenjenih funkcij nad v sekciji 3.2.1 ustvarjenim podatkovnim okvirjem v programskem jeziku Julia.

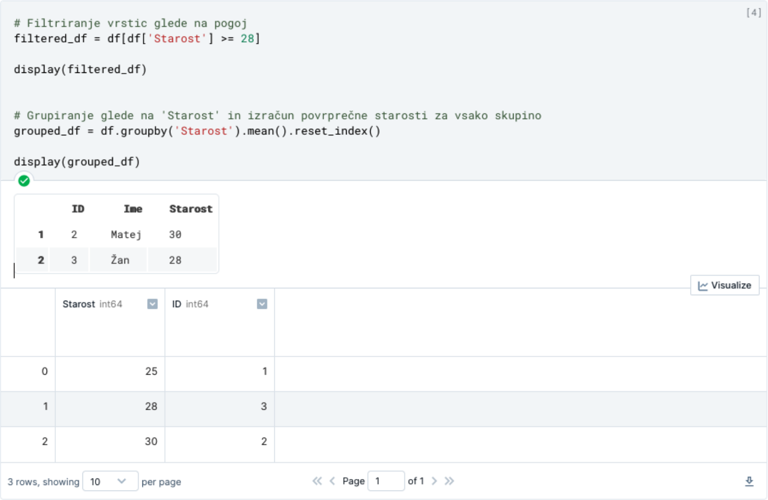


```
using Statistics

# Filtriranje vrstic glede na pogoj
filtered_df = filter(row -> row.Starost >= 28, df)

display(filtered_df)

# Grupiranje glede na 'Starost' in izračun povprečne starosti za vsako skupino
grouped_df = combine(groupby(df, :Starost), :Starost => mean)

display(grouped_df)
```

```
2×3 DataFrame
 Row   ID     Ime     Starost
       Int64  String  Int64
   1      2   Matej        30
   2      3   Žan          28

3×2 DataFrame
 Row   Starost  Starost_mean
       Int64    Float64
   1       25          25.0
   2       30          30.0
   3       28          28.0
```

**Slika 5: Uporaba osnovnih operacij filtriranja, grupiranja in izračuna povprečnih vrednosti nad podatkovnim okvirjem v programskem jeziku Julia.**

Implementacija enakega primera v programskem jeziku Python je razvidna na sliki 6, ki še enkrat več prikazuje, da je doseganje enakega cilja pri uporabi omenjenih knjižnic precej podobno oz. se razlikuje zgolj v osnovni sintaksi primerjanih programskih jezikov.



```python
# Filtriranje vrstic glede na pogoj
filtered_df = df[df['Starost'] >= 28]

display(filtered_df)


# Grupiranje glede na 'Starost' in izračun povrprečne starosti za vsako skupino
grouped_df = df.groupby('Starost').mean().reset_index()

display(grouped_df)
```

|   | ID | Ime | Starost |
|---|---|---|---|
| 1 | 2 | Matej | 30 |
| 2 | 3 | Žan | 28 |

|   | Starost int64 | ID int64 |
|---|---|---|
| 0 | 25 | 1 |
| 1 | 28 | 3 |
| 2 | 30 | 2 |

**Slika 6: Uporaba osnovnih operacij filtriranja, grupiranja in izračuna povprečnih vrednosti nad podatkovnim okvirjem v programskem jeziku Python.**

### *3.2.3 Hitrost izvajanja*

Za namen preverjanja oz. primerjave hitrosti izvajanja najbolj pogostih operacij nad podatkovnim okvirjem smo si zastavili enostaven eksperiment (slika 7). V vsakem izmed okolij smo ustvarili podatkovni okvir z milijon naključnimi vnosi števil med 20 in vključno 50. Nad ustvarjenim podatkovnim okvirjem smo nato izvedli testiranje hitrosti izvedbe za filtriranje in grupiranje ter izračun povprečne vrednosti starosti v vsaki izmed skupin. Dekorator @benchmark v programskem jeziku Julia omogoča samodejno beleženje časa izvajanja podanega ukaza oz. klica funkcije ter po koncu izvedbe tudi izpisa izmerjenih vrednosti.

```julia
using DataFrames
using BenchmarkTools

# Ustvarimo velik podatkovni okvir z naključnimi vrednostmi
df = DataFrame(ID=1:10^6, Starost=rand(20:50, 10^6))

# Filtriranje vrstic glede na podan pogoj
@benchmark filter(row -> row.Starost >= 40, df)
```

```
BenchmarkTools.Trial: 76 samples with 1 evaluation.
 Range (min … max):  59.928 ms … 87.199 ms  ┊ GC (min … max): 0.00% … 1.00%
 Time  (median):     64.829 ms              ┊ GC (median):    1.41%
 Time  (mean ± σ):   66.187 ms ±  5.101 ms  ┊ GC (mean ± σ):  1.27% ± 0.50%

  59.9 ms         Histogram: frequency by time        77.6 ms <

 Memory estimate: 38.75 MiB, allocs estimate: 1999514.
```

```julia
# Grupiranje in izračun povprevne starosti
@benchmark combine(groupby(df, :Starost), :Starost => mean)
```

```
BenchmarkTools.Trial: 1814 samples with 1 evaluation.
 Range (min … max):  1.882 ms … 8.136 ms  ┊ GC (min … max): 0.00% … 10.21%
 Time  (median):     2.177 ms             ┊ GC (median):    0.00%
 Time  (mean ± σ):   2.749 ms ± 1.089 ms  ┊ GC (mean ± σ):  7.57% ± 11.97%

  1.88 ms      Histogram: log(frequency) by time      6.16 ms <

 Memory estimate: 7.65 MiB, allocs estimate: 290.
```

**Slika 7: Prikaz hitrosti izvajanja operacij nad podatkovnim okvirjem v programskem jeziku Julia.**

Slika 8 prikazuje implementacijo enakega eksperimenta v programskem jeziku Python. Za merjenje časa izvajanja posamezne operacije nad podatkovnim okvirjem smo v tem primeru uporabili knjižnico timeit. Glede na izpisane rezultate izvedb obeh implementacij lahko ugotovimo, da je filtriranje v programskem jeziku Julia v povprečju trajalo 66 milisekund, medtem ko je enaka operacija v Pythonu trajala slabo sekundo. Še večja razlika, ki govori v korist programskemu jeziku Julia, se je pokazala pri grupiranju in izračunu povprečne vrednosti. Ta operacija se je v programskem jeziku Julia izvedla v manj kot 3 milisekundah, na drugi strani pa je Python za to potreboval 2 sekundi.

Glede na to, da že pri popolnoma preprostem primeru, kot je naš, lahko opazimo ogromno prednost programskega jezika Julia, je mogoče pričakovati, da se hitrost izvedbe še bolj izrazito opazi pri manipulaciji nad kompleksnejšimi podatkovnimi strukturami ter operacijami.

```
import pandas as pd
import numpy as np
import timeit

# Create a large DataFrame
df = pd.DataFrame({'ID': range(1, 10**6 + 1), 'Age': np.random.randint(20, 51, 10**6)})

# Filtering rows based on a condition
def filter_condition():
    return df[df['Age'] >= 40]

# Grouping and calculating the mean age
def group_and_mean():
    return df.groupby('Age').mean().reset_index()

# Benchmark filtering
filter_time = timeit.timeit(filter_condition, number=100)

# Benchmark grouping and mean calculation
group_time = timeit.timeit(group_and_mean, number=100)

print(f"Filtering time (Python): {filter_time:.5f} seconds")
print(f"Grouping and Mean calculation time (Python): {group_time:.5f} seconds")
```

```
Filtering time (Python): 0.98302 seconds
Grouping and Mean calculation time (Python): 2.02604 seconds
```

**Slika 8: Prikaz hitrosti izvajanja nad podatkovnim okvirjem v programskem jeziku Julia.**

### 3.3 Vizualiziranje podatkov

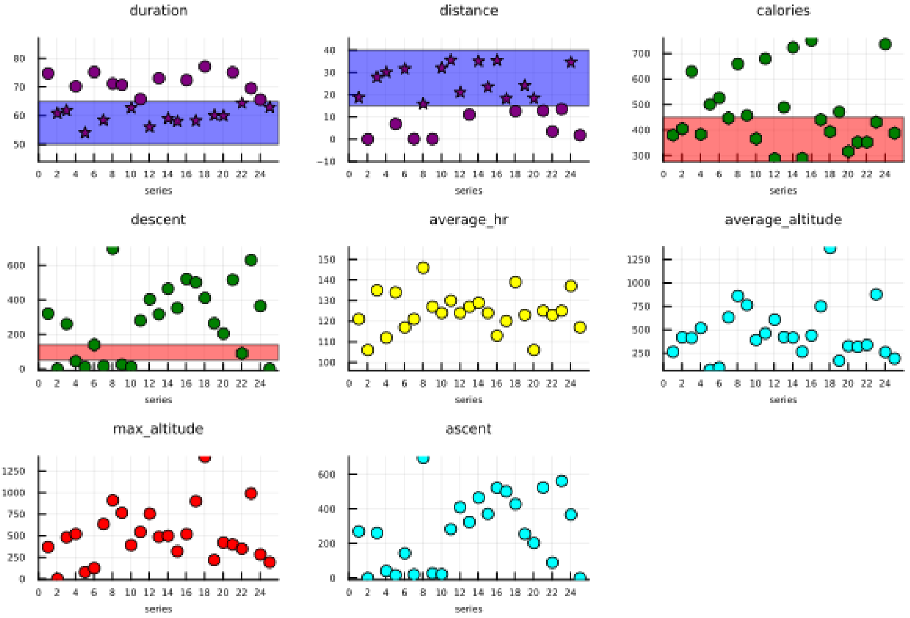
Vizualizacija podatkov je po navadi zadnji korak pri odkrivanju znanja iz podatkov oziroma analize podatkov. Namen vizualiziranja oziroma vizualizacije podatkov je grafična predstavitev informacij ter podatkov. Je učinkovita tehnika za posredovanje zapletenih podatkovnih vzorcev in vpogledov z vizualnimi elementi, kot so preglednice, grafi, diagrami in pismenke (angl. glyphs). Programski jezik Julia se je na področju vizualizacije podatkov izkazal kot odlično orodje za ustvarjanje enostavnih kot tudi kompleksnih ter interaktivnih vizualizacij.

Znotraj ekosistema programskega jezika Julia lahko najdemo različne pakete in knjižnice, posebej zasnovane za različne naloge vizualizacije podatkov. Najpomembnejša knjižnica je verjetno knjižnica Plots.jl, ki zagotavlja poenoten vmesnik za ustvarjanje številnih grafikonov z minimalno kodo. Plots.jl podpira več zaledij, kot so GR, Plotly in PyPlot, kar uporabnikom omogoča, da izberejo najprimernejši način vizualizacije za svoje potrebe. Primer uporabe knjižnice Plots.jl za vizualizacijo odkritega znanja iz asociacijskih pravil je prikazan na sliki 9.

Drugo zelo priljubljeno ogrodje za vizualizacijo podatkov v Julii je Makie.jl. Makie ponuja visoko zmogljive interaktivne grafe, ki so idealni za obdelavo velikih zbirk podatkov ter ustvarjanje vizualizacij v realnem času. Njegove interaktivne zmožnosti omogočajo nemoteno raziskovanje in manipuliranje s podatkovnimi točkami neposredno znotraj ploskve.

Poleg tega Julia podpira integracijo s priljubljenimi knjižnicami za izrisovanje iz drugih programskih jezikov, kot sta Matplotlib preko PyCall.jl [17] in Plotly preko PlotlyJS.jl knjižnic. To uporabnikom omogoča, da uporabijo obstoječe knjižnice za vizualizacijo in jih nemoteno združijo z zmogljivostmi analize podatkov, ki jih ponuja Julia.

Z izkoriščanjem zmogljivosti vizualizacije podatkov programskega jezika Julia in uporabo njegovega bogatega ekosistema ogrodij in knjižnic lahko uporabniki odkrijejo globlje vpoglede in svoje ugotovitve učinkovito sporočijo širšemu občinstvu.



**Slika 9: Uporaba knjižnice Plots.jl za vizualizacijo odkritega znanja iz asociacijskih pravil**
Vir: [16].

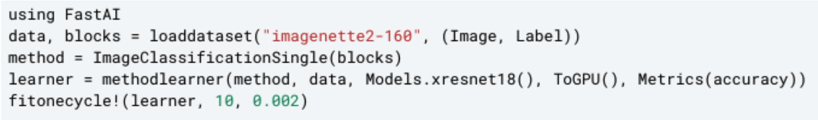
### 3.4 Strojno učenje

Ekosistem knjižnic za strojno učenje v programskem jeziku Julia iz leta v leto narašča. Ena ključnih knjižnic je Flux.jl, ki zagotavlja podporo za globoko učenje in nevronske mreže. Flux.jl [18] ponuja samodejno diferenciacijo, podporo za grafične procesorje in jedrnato sintakso, ki spominja na matematični zapis. Flux.jl nudi nizkonivojski način implementacije algoritmov strojnega učenja, ki sicer uporabniku nudi možnost neposrednega nadzora nad učenjem, po drugi strani pa je za začetnika morebiti prezahteven. Za enostavnejšo splošno uporabo je bila razvita knjižnica FastAI.jl [19], ki omogoča uporabo algoritmov strojnega učenja v le nekaj vrsticah programske kode (Slika 9: Primer izgradnje napovednega modela za klasifikacijo slik z uporabo XresNet arhitekture globoke nevronske mreže.

V splošnem sta tako Julia kot tudi Python povsem legitimni izbiri za uporabo pri strojnem učenju, vsaka s svojimi prednostmi. Julia se odlikuje pri visokozmogljivem računalništvu in numeričnih nalogah, zato je primerna za aplikacije, kjer je hitrost ključnega pomena. Po drugi strani pa je Python, prav zaradi njegove enostavnosti in nizke krivulje učenja, dobra izbira za različne profile uporabnikov, saj je kos različnim nalogam strojnega učenja.

Poleg omenjene knjižnice Flux.jl velja omeniti še knjižnico MLJ.jl [20], ki je celovita knjižnica za strojno učenje, katere namen je zagotoviti dosleden vmesnik za različne naloge strojnega učenja, vključno s predobdelavo, modeliranjem, vrednotenjem in cevovodi. MLJ.jl se dobro povezuje z drugimi paketi Julia in zagotavlja enoten način dela z različnimi modeli in vrstami podatkov. Prav tako velja omeniti še knjižnico Turing.jl [21] za verjetnostno programiranje, ki omogoča definiranje in ocenjevanje verjetnostnih modelov z uporabo visokonivojske sintakse. Podpira različne inferenčne algoritme, vključno z algoritmom MCMC (Markov Chain Monte Carlo), zato je primerna za bayesovsko analizo podatkov in verjetnostno modeliranje.

Nasproti rastočemu ekosistemu knjižnic za strojno učenje stoji Python kot uveljavljen programski jezik skupaj z naborom široko uporabljanih in uveljavljenih knjižnic. Slabost slednjega je seveda interpretativna narava Pythona, vendar knjižnice, kot so NumPy, TensorFlow in PyTorch, s svojo nizkonivojsko implementacijo to kar se da omilijo. Ne nazadnje, v kolikor je Tensorflow naša prva izbira, ko je govora o knjižnici za globoko učenje, je za slednjo na voljo tudi vmesnik za programski jezik Julia, ki omogoča domorodno uporabo knjižnice.



```
using FastAI
data, blocks = loaddataset("imagenette2-160", (Image, Label))
method = ImageClassificationSingle(blocks)
learner = methodlearner(method, data, Models.xresnet18(), ToGPU(), Metrics(accuracy))
fitonecycle!(learner, 10, 0.002)
```

**Slika 9: Primer izgradnje napovednega modela za klasifikacijo slik z uporabo XresNet arhitekture globoke nevronske mreže.**

V splošnem sta tako Julia kot tudi Python povsem legitimni izbiri za uporabo pri strojnem učenju, vsaka s svojimi prednostmi. Julia se odlikuje pri visokozmogljivem računalništvu in numeričnih nalogah, zato je primerna za aplikacije, kjer je hitrost ključnega pomena. Po drugi strani pa je Python, prav zaradi njegove enostavnosti in nizke krivulje učenja, dobra izbira za različne profile uporabnikov, saj je kos različnim nalogam strojnega učenja.

Poleg omenjene knjižnice Flux.jl velja omeniti še knjižnico MLJ.jl [20], ki je celovita knjižnica za strojno učenje, katere namen je zagotoviti dosleden vmesnik za različne naloge strojnega učenja, vključno s predobdelavo, modeliranjem, vrednotenjem in cevovodi. MLJ.jl se dobro povezuje z drugimi paketi Julia in zagotavlja enoten način dela z različnimi modeli in vrstami podatkov. Prav tako velja omeniti še knjižnico Turing.jl [21] za verjetnostno programiranje, ki omogoča definiranje in ocenjevanje verjetnostnih modelov z uporabo visokonivojske sintakse. Podpira različne inferenčne algoritme, vključno z algoritmom MCMC (Markov Chain Monte Carlo), zato je primerna za bayesovsko analizo podatkov in verjetnostno modeliranje.

Če ne želimo sprejemati kompromisov s stališča hitrosti izvajanja, nam je na voljo tudi možnost, da znotraj Julia ekosistema uporabljamo Pythonove knjižnice strojnega učenja. To je mogoče doseči z interoperabilnostjo med obema jezikoma. V ta namen je na voljo nekaj knjižnic, ki nam uporabo Python knjižnic v programskem jeziku Julia povsem poenostavijo. Prva izmed knjižnic, ki smo jo omenili že v poglavju o vizualizaciji, je PyCall, ki omogoča klicanje funkcij Pythona in uporabo knjižnic neposredno iz Julie. Preprost primer uporabe scikit-learn, priljubljene knjižnice strojnega učenja Python, iz programskega okolja Julie je prikazan na sliki 10.

.

```
using PyCall

# Vključitev knjižnice
@pyimport sklearn.linear_model as lm

# Kreiranje modela linearne regresije
model = lm.LinearRegression()

# Učenje modela
X = [[1.0, 2.0], [2.0, 3.0]]
y = [3.0, 5.0]
model.fit(X,y)

# Napovedovanje
model.predict(X)
```

```
2-element Vector{Float64}:
 3.0
 5.0
```

**Slika 10: Primer uporabe Python knjižnice scikit-learn znotraj programskega okolja Julia.**

Še en primer je knjižnica Conda.jl. Conda je sicer poznana kot upravitelj paketov in razvojnih okolij za Python. Knjižnica Conda.jl nam omogoča, da znotraj ekosistema Julie ustvarimo okolje Python in namestimo Python knjižnice. V povezavi s prej predstavljeno knjižnico PyCall.jl lahko enostavno upravljamo množico Python knjižnic in jih brez težav uporabljamo znotraj programskega okolja Julia.

Čeprav se ekosistem strojnega učenja v Julii še razvija, ima več zmogljivih paketov, kot so Flux.jl za globoko učenje, MLJ.jl za splošno strojno učenje in Turing.jl za verjetnostno programiranje. Te knjižnice lahko v Julii pokrivajo široko paleto nalog strojnega učenja.

Neobstoj ali nezrelost knjižnic tako nista več razloga za neuporabo programskega jezika Julia za namen strojnega učenja. Na koncu je izbira med uporabo knjižnic za strojno učenje v jeziku Python iz Julie ali uporabo izvornih knjižnic Julie odvisna od dejavnikov, kot so zapletenost projekta, zahteve glede zmogljivosti, razpoložljivost posebnih funkcij in strokovno znanje ekipe v obeh jezikih.

## 4 Zaključek

V tem prispevku smo na kratko predstavili programski jezik Julia in njegove značilnosti. Julia je jezik, namenjen znanstvenemu računalništvu in obdelavi podatkov. Odlikujeta ga JIT prevajalnik z uporabo ogrodja LLVM, ki omogoča doseganje maksimalne zmogljivosti za numerično, tehnično in znanstveno računanje, ter vzporedno računanje, zaradi česar ocenjujemo, da je jezik zelo primeren za uporabo na področju podatkovne znanosti in obdelave podatkov. Čeprav se ekosistem programskega jezika Julia še razvija, ima več zmogljivih knjižnic, kot so Flux.jl za globoko učenje, MLJ.jl za splošno strojno učenje in Turing.jl za verjetnostno programiranje. S temi knjižnicami lahko v Julii pokrijemo širok razpon nalog strojnega učenja. Morebitne težave glede zmanjšanega nabora knjižnic pa lahko naslovimo z uporabo knjižnic, kot sta Conda.jl in PyCall.jl, ki omogočata uporabo širokega nabora Pythonovih knjižnic ob minimalnem žrtvovanju zmogljivosti.

Zaključimo lahko, da je programski jezik Julia legitimna izbira za namen izvajanja podatkovne znanosti z zrelimi knjižnicami, ki pokrivajo ključen nabor nalog podatkovnih znanstvenikov. V primerjavi s programskim jezikom Python mu lahko očitamo malenkost strmejšo krivuljo učenja in manjši nabor virov za učenje. Omeniti še velja, da je Python poleg enostavnosti, še posebej mikaven z razširjenostjo izvajalnega okolja Jupyter, ki uporabniku omogoča uporabo brez namestitve na lasten sistem. Julia ima sicer na voljo enako izvajalno okolje, vendar večina največjih ponudnikov Julie še nima v naboru izvajalnih okolij, kar je lahko dodaten minus. Če potrebujemo izvajanje na ravni nizkonivojskih programskih jezikov, pa Julia, ne glede na omenjeno, predstavlja zelo primerno okolje s poznano sintakso široko uporabljanih interpreterskih jezikov.

## Literatura

[1] S. Sagiroglu in D. Sinanc, „Big data: A review“, v 2013 international conference on collaboration technologies and systems (CTS), 2013, str. 42–47.

[2] V. Dhar, „Data science and prediction“, Commun ACM, let. 56, št. 12, str. 64–73, 2013.

[3] K. Gao, G. Mei, F. Piccialli, S. Cuomo, J. Tu, in Z. Huo, „Julia language in machine learning: Algorithms, applications, and open issues“, Comput Sci Rev, let. 37, str. 100254, 2020.

[4] A. B. Arrieta idr., „Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI“, Information fusion, let. 58, str. 82–115, 2020.

[5] „The Julia Programming Language“. https://julialang.org (pridobljeno 25. julij 2023).

[6] I. Balbaert, A. Sengupta, in M. Sherrington, Julia: High Performance Programming. Packt Publishing Ltd, 2016.

[7] „Julia (programming language)“. https://en.wikipedia.org/wiki/Julia_(programming_language) (pridobljeno 25. julij 2023).

[8] J. Bezanson, S. Karpinski, V. B. Shah, in A. Edelman, „Julia: A fast dynamic language for technical computing“, arXiv preprint arXiv:1209.5145, 2012.

[9] „TIOBE Index for July 2023“. https://www.tiobe.com/tiobe-index (pridobljeno 24. julij 2023).

[10] „Why is Julia not more popular?“ https://craftofcoding.wordpress.com/2020/09/24/why-is-julia-not-more-popular (pridobljeno 25. julij 2023).

[11] „Julia by Example“. https://juliabyexample.helpmanual.io (pridobljeno 27. julij 2023).

[12] „Pkg.jl“. https://pkgdocs.julialang.org/v1 (pridobljeno 27. julij 2023).

[13] E. Roesch idr., „Julia for biologists“, Nat Methods, let. 20, št. 5, str. 655–664, 2023.

[14] „Julia Micro-Benchmarks“. https://julialang.org/benchmarks (pridobljeno 27. julij 2023).

[15] A. Sengupta in A. Edelman, Julia High Performance: Optimizations, distributed computing, multithreading, and GPU programming with Julia 1.0 and beyond, 2nd Edition. Packt Publishing, 2019. [Na spletu]. Dostopno na: https://books.google.si/books?id=etacDwAAQBAJ

[16] I. Fister Jr, „NarmViz.jl". https://raw.githubusercontent.com/firefly-cpp/NarmViz.jl/main/.github/figures/Fig2.png (pridobljeno 2. avgust 2023).

[17] „PyCall.jl". https://github.com/JuliaPy/PyCall.jl (pridobljeno 3. avgust 2023).

[18] „Flux.jl". https://fluxml.ai/Flux.jl/stable (pridobljeno 3. avgust 2023).

[19] „FastAI.jl". https://github.com/FluxML/FastAI.jl (pridobljeno 3. avgust 2023).

[20] „MLJ.jl". https://github.com/alan-turing-institute/MLJ.jl (pridobljeno 3. avgust 2023).

[21] „Turing.jl". https://turing.ml (pridobljeno 3. avgust 2023).

# Vpeljava umetne inteligence v poslovno analitiko

Aleksej Miloševič, Gregor Pirš, Miha Pavlinek

Databox Inc., Ptuj, Slovenija
aleksej@databox.com, gregor.pirs@databox.com, miha.pavlinek@databox.com

Poslovni uporabniki se vsakodnevno srečujejo z ogromnim številom ključnih kazalnikov in povezanih metrik, preko katerih spremljajo uspešnost poslovanja. Pri tem pogosto analizirajo enormne količine podatkov ter v njih iščejo povezave, vzorce in trende ter iz tega oblikujejo znanje, s katerim skušajo pridobiti prednost pred konkurenco. Ker se s količino podatkov veča kompleksnost njihovega razumevanja, se podjetja običajno poslužujejo rešitev za poslovno analitiko, ki omogočajo spremljanje poslovanja preko preglednih plošč, poročil, alarmov, ipd. Prav takšno rešitev ponujamo v podjetju Databox. Z namenom, da podpremo vse štiri tipe poslovne analitike: deskriptivno, diagnostično, prediktivno in preskriptivno, smo v naše produkte vpeljali številne funkcionalnosti, ki jih poganjajo metode umetne inteligence. Z njimi surove podatke pretvarjamo v kakovostne informacije in znanja iz katerih uporabniki črpajo nove ugotovitve, s pomočjo katerih sprejemajo boljše poslovne odločitve. V prispevku bomo predstavili inteligentne rešitve kot so napovedovanje metrik, zaznavanje anomalij, korelacije ter funkcionalnosti, ki temeljijo na generativni umetni inteligenci. Naslovili bomo izzive pri zasnovi in razvoju tovrstnih rešitev ter ponudili vpogled v njihovo delovanje.

**Ključne besede:**
poslovna analitika
umetna inteligenca
napovedovanje
zaznavanje anomalij
korelacije
generativni modeli

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.25

## 1 Uvod

Poslovna analitika zajema veščine, tehnologije in prakse za preiskovanje in proučevanje preteklih poslovnih rezultatov z namenom pridobivanja vpogledov in usmerjanja poslovnega načrtovanja [1]. Na osnovi povezovanja podatkov iz različnih virov in uporabe statističnih metod se področje osredotoča predvsem na razvoj novih vpogledov, razumevanje poslovne uspešnosti in napovedovanje poslovnih dogodkov.

Za razliko od poslovne inteligence, ki je vezana na pregled in analizo preteklega poslovanja na osnovi predpriprave podatkov in poročil, se poslovna analitika ukvarja s poglobljeno analizo podatkov ter prediktivnimi modeli za podporo odločanju in razvoju poslovne strategije. Na primer, medtem ko poslovna inteligenca podjetju predstavlja trenuten profil idealne stranke, poslovna analitika ponuja vpogled v profil bodoče stranke.

V podjetju Databox razvijamo SaaS produkt za poslovno analitiko, ki je prilagojen zlasti manjšim in srednjim podjetjem, katera si težko privoščijo postavitev in vzdrževanje analitične infrastrukture ter analitika, ki bi pridobival in preiskoval podatke. Poleg zbiranja podatkov, upravljanja preglednih plošč ter spremljanja poročil in alarmov želimo našim odjemnikom ponuditi tudi dodatne vpoglede v poslovanje in jih voditi v sprejemanje boljših odločitev. V ta namen dodajamo nove funkcionalnosti, ki pretežno temeljijo na statističnih metodah in metodah umetne inteligence.

Pri tem se srečujemo s številnimi izzivi. Eden pomembnejših je generalizacija metod in parametrov za široko paleto podatkovnih zbirk. Medtem ko se podatkovni analitik v podjetju ukvarja s specifično zbirko podatkov nad katero v okviru razvoja vpogledov izvaja eksperimente in optimizira parametre modelov, naši sistemi uporabljajo generalne optimume, ki so prilagojeni da delujejo nad različnimi podatkovnimi zbirkami.

## 2 Štirje tipi poslovne analitike

Dolgoročni cilj naših produktov je celovita podpora podjetjem tako pri sprejemanju poslovnih odločitev kot pri poslovnem planiranju na podlagi podatkov. V ta namen želimo podpreti vse 4 tipe analitike, ki so podrobneje predstavljene v nadaljevanju [2].

**Opisna analitika** preučuje podatke z namenom izboljšanja razumevanja dogajanja v preteklosti. Povzema zgodovinske podatke in ponuja vpogled v vzorce, trende in ključne metrike. Gre za najbolj osnovno vrsto analitike, kjer se za potrebe analize običajno uporabljajo agregacijske funkcije in osnovni matematični koncepti. V to kategorijo sodijo rešitve poslovne inteligence s preglednimi ploščami, poročili in alarmi.

**Diagnostična analitika** je nadgradnja opisne analitike, ki se ukvarja z iskanjem povezav med preteklimi dogodki in odgovarja na vprašanje zakaj? Vključuje poglobljeno analizo podatkov za prepoznavanje temeljnih vzrokov določenega rezultata.

**Napovedna analitika** se na podlagi zgodovinskih podatkov in statističnih metod ukvarja z napovedovanjem prihodnjih dogodkov ali rezultatov. Z uporabo vzorcev in povezav v podatkih podaja verjetnosti dogodkov. Sem sodi napovedovanje časovnih vrst, napovedovanje osipa (ang. Churn Prediction), pa tudi ocenjevanje uporabnikove preference do posameznih izdelkov v priporočilnih sistemih.

**Preskriptivna analitika** podaja priporočila za specifična dejanja z namenom optimizacije rezultatov. Posredovana priporočila so oblikovana na podlagi napovednih modelov in poslovnih ciljev. Uporablja napredne algoritme, tehnike optimizacije in simulacije za ustvarjanje uporabnih vpogledov. Dober primer preskriptivne analitike so sistemi za točkovanje in rangiranje uporabnikov glede na njihova dejanja. Te uporabljajo zlasti prodajniki, ki izbirajo in nagovarjajo le tiste uporabnike, pri katerih je možnost za pretvorbo v plačljive stranke največja.

Poleg zgoraj naštetih se v zadnjem obdobju vse več omenja tudi t.i. obogatena analitika (ang. Augmented Analytics) [3]. Sem sodijo sistemi, ki s pomočjo strojnega učenja in obdelave naravnega jezika avtomatizirajo analitične procese, uporabniku pa na razumljiv način predstavijo rezultate in vpoglede. Takšen primer bi bil sistem za avtomatsko spremljanje metrik poslovanja ter redno obveščanje o spremembah, trendih in posebnostih v naravnem jeziku.

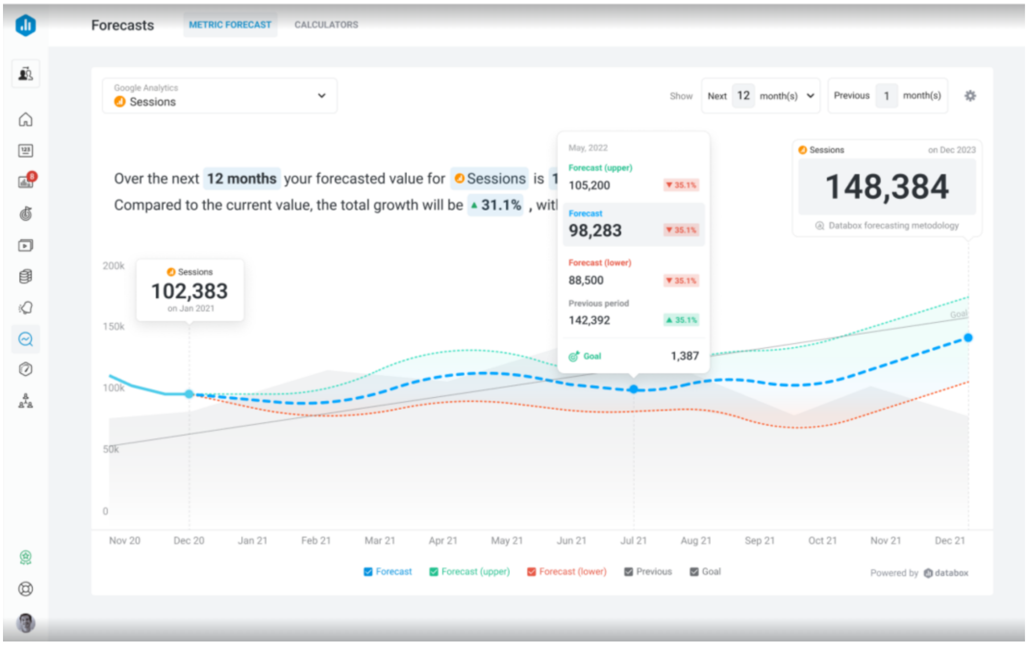
## 3 Nove funkcionalnosti v produktu za poslovno analitiko

### 3.1 Napovedovanje poslovnih metrik

Z napovedovanjem poslovnih metrik želimo našim uporabnikom ponuditi vpogled v prihodnost poslovanja. Pri tem uporabljamo modele za napovedovanje časovnih vrst iz odprtokodnega orodja Prophet [3]. Pri izbiri napovednih modelov smo se osredotočali predvsem na hitrost učenja in točnost. Prophet za napovedovanje uporablja preprostejše regresijske modele, ki v primerjavi z ostalimi bolj sofisticiranimi modeli zagotavljajo zelo hitre napovedi z visoko natančnostjo. Poleg tega ima sposobnost samodejnega prilagajanja parametrov, je odporen na manjkajoče podatke, v rezultatih pa poleg napovedi podaja tudi intervale zaupanja, s čimer je zelo primeren za napovedovanje raznolikih časovnih vrst. Slabost Propheta je njegova omejenost na univariatne časovne vrste, kar pomeni, da zna modele prilagajati zgolj nad eno časovno-odvisno spremenljivko.

Prophet pri svojem delovanju časovno vrsto razdeli na tri komponente: sestavljeni trendi, sezonskost in prazniki. Komponenta sestavljenih trendov je oblikovana s pomočjo sestave linearnih ali logističnih krivulj in samodejnim zaznavanjem spremembe trenda. Prav tako je samodejno tudi zaznavanje sezonskosti, medtem ko je potrebno praznike pred prilagajanjem modela posredovati manualno preko seznama.

V naši storitvi za napovedovanje poslovnih metrik dodatno pred učenjem modela iz podatkov porežemo prevelika odstopanja, ki se kažejo v obliki nenadnih skokov ali padcev. S tem se izognemo širokim intervalom zaupanja ter vplivu na zaznavanje sezonskosti ter tako povečujemo robustnost modela.



Google Analytics
Sessions

Show Next 12 month(s)
Previous 1 month(s)

Over the next 12 months your forecasted value for Sessions is 1
Compared to the current value, the total growth will be ▲ 31.1% , wit

**Slika 1: Funkcionalnost za napovedovanja poslovnih metrik.**

Kot omenjeno v uvodu, je bil glavni izziv razvoja storitve za napovedovanja generalizacija modelov za raznolike časovne vrste. To je bilo deloma rešeno z ogrodjem za evalvacijo napovednih modelov, preko katerega smo poganjali eksperimente ter preučevali njihove rezultate, deloma pa s segmentacijo časovnih vrst glede na njihove značilnosti ter prilagajanje parametrov .

Za naprej si želimo vpeljati multivariatne modele, ki bodo pri napovedovanju uporabljali znanje iz različnih povezanih metrik. Npr. za napovedovanje skupnega letnega prihodka v spletni trgovini bo algoritem zmožen poleg zgodovine prihodkov uporabiti tudi obisk na spletni strani, stopnjo konverzije ter vrednost povprečnega naročila. Glavni izziv pri takšnih modelih je vpeljava dodatnih podatkovnih cevovodov, bistveno daljši čas gradnje modela in posledično podražitev. Takšnih modelov ni mogoče pripravljati sproti, pač pa jih je potrebno graditi na zalogo.

## 3.2 Zaznavanje anomalij

Agilnost in sposobnost prilagajanja spremembam sta dve od ključnih lastnosti, ki izjemno uspešna podjetja dvigujejo nad povprečje. V tem smislu je prepoznavanje nenavadnih in nepredvidljivih dogodkov - oziroma odkrivanje anomalij - eno izmed osrednjih orodij. Pravočasno zaznavanje takšnih anomalij podjetjem omogoča hitro reakcijo z namenom zmanjševanja možne škode, hkrati pa izkoriščanje nastalih priložnosti. Za odkrivanje anomalij se uporabljajo modeli, ki vključujejo statistične metode in pristope strojnega učenja, med katerimi so tako klasične kot tudi metode, temelječe na nevronskih mrežah. Postopek rezultira z oceno anomalije v obsegu med -1 in 1, ki prikazuje stopnjo nenavadnosti posamezne podatkovne točke.

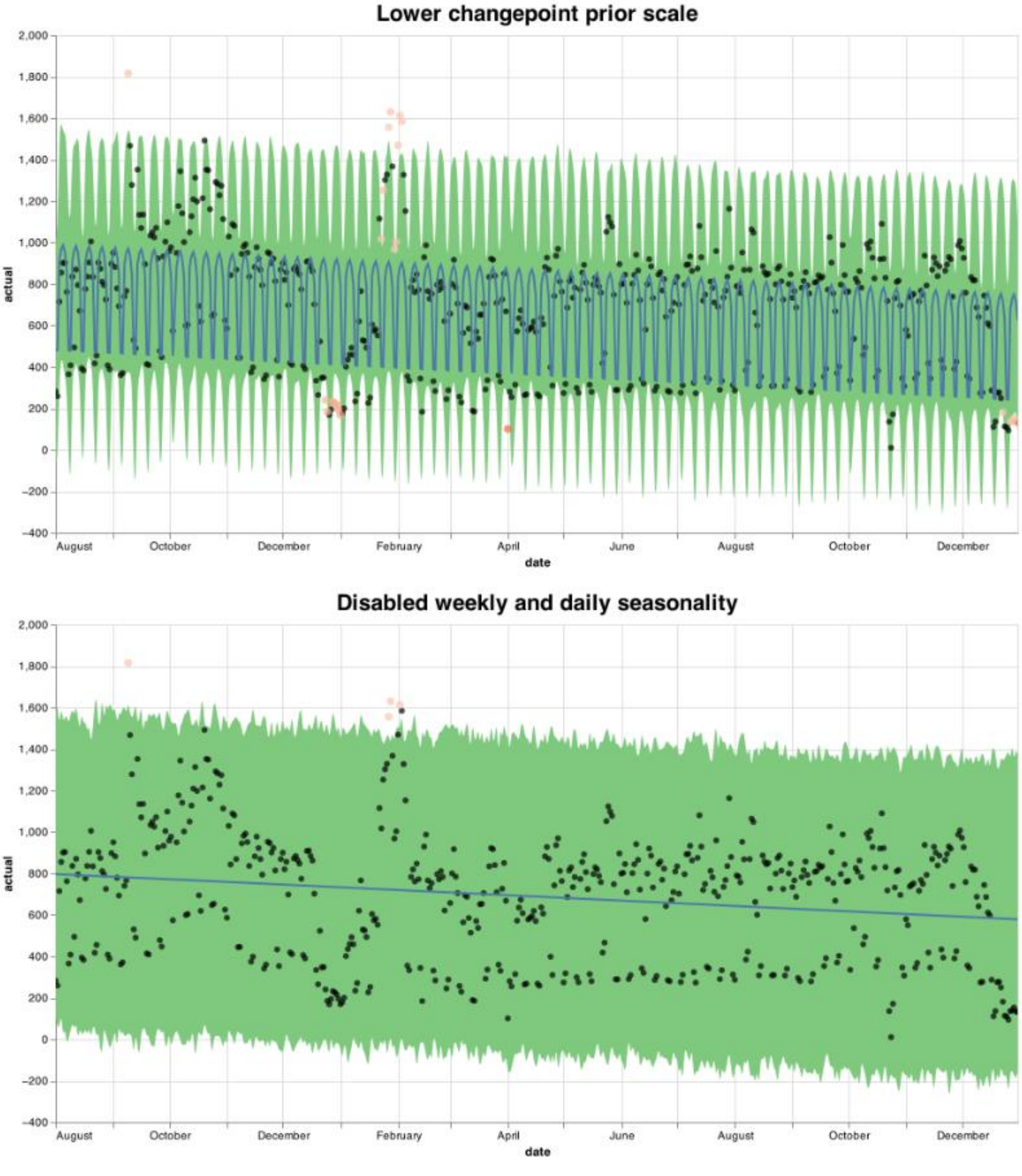
Obstajajo tri vrste anomalij: točkovne, kontekstualne in kolektivne [4]. Točkovne ali dogodkovne anomalije glede na vrednost bistveno odstopajo od ostalih točk v nizu. Za njih je značilno, da so globalne, saj so anomalne v primerjavi s poljubno normalno točko v časovni vrsti. Najpogosteje nastanejo zaradi napak pri zbiranju ali obdelavi podatkov, ali pa označujejo nenavaden ali nepričakovan dogodek. Kontekstualne anomalije v globalnem smislu posebej ne izstopajo, vendar so anomalne znotraj lokalnega konteksta. Naglo povečanje števila uporabnikov med božičem ali nenadno zmanjšanje prodaje na začetku leta se lahko smatra kot normalno obnašanje zaradi časa, v katerem se anomalija pojavi. Če pa prodaja skozi ta čas ostane relativno stabilna, lahko to interpretiramo kot kontekstualno anomalijo. Kolektivne anomalije predstavljajo skupino. Kolektivne anomalije predstavljajo več zaporednih anomalij znotraj niza.Vzrok za vse tipe anomalij je potrebno določiti empirično.

Pri razvijanju sistema za zaznavanje anomalij v okviru produkta Databox je ključnega pomena izbira modela, ki omogoča hitro identifikacijo anomalij ter zmožnost obdelave obsežnih časovnih nizov. Zaradi velikega števila različnih poslovnih časovnih nizov je robustnost modela, ki ne potrebuje neprestanega človeškega nadzora, še posebej pomembna. Za zaznavanje anomalij smo, podobno kot za napovedovanje časovnih nizov, uporabili Facebookov model Prophet. Ta model se dobro prilagaja trendom in sezonskim vzorcem v podatkih, zato ga je mogoče uporabiti ne le za napovedovanje prihodnosti, temveč tudi za "napovedovanje" sedanjosti. Če dejanska vrednost preseže interval zaupanja napovedi, lahko to kaže na anomalijo. Ker model v napoved vključuje element sezonskosti, je mogoče zaznavanje tako točkovnih kot kontekstualnih in kolektivnih anomalij. Interpretacije posameznih anomalnih pojavov je odgovornost uporabnika.

Vključitev zaznavanja anomalij kot funkcionalnost platforme je posebej zanimivo za Databoxovo ciljno publiko majhnih in srednje velikih podjetij, ki pogosto nimajo finančnih in strokovnih kapacitet za vzpostavitev lastnih sistemov. Sistem namreč lahko omogoča zaznavanje anomalij za več kot 80 različnih integracij, med drugimi tudi za tiste, ki v lastni platformi zaznavanja anomalij ne podpirajo. Najbolj zaželen primer uporabe zaznavanje anomalij je gotovo avtomatiziranje alarmov, kar omogoča strankam, da se lahko v najkrajšem možnem času odzovejo na izredni pojav.

Glavni izziv pri zaznavanju anomalij predstavlja heterogena narava časovnih vrst. Kar je za eno metriko in uporabnika anomalija, to ne velja za drugega uporabnika. Z nastavljanjem hiperparametrov modela je mogoče spreminjati občutljivost zaznavanja anomalij. To omogoča razvoj sistema za povratne informacije, ki uporabniku omogoča spreminjanje občutljivosti modela glede na lastne potrebe.

Na sliki 2 je prikazana primerjava rezultata procesa zaznavanja anomalij na primeru poslovne časovne vrste z dnevno granulacijo in izrazito tedensko sezonskostjo. Na zgornjem grafu je zaznavanje sezonskosti vklopljena, na spodnjem pa je izklopljena. Zaradi prisotnosti sezonskosti so na zgornjem grafu vidne razen točkovnih anomalij tudi kontekstualne anomalije, na spodnjem pa zgolj točkovne.

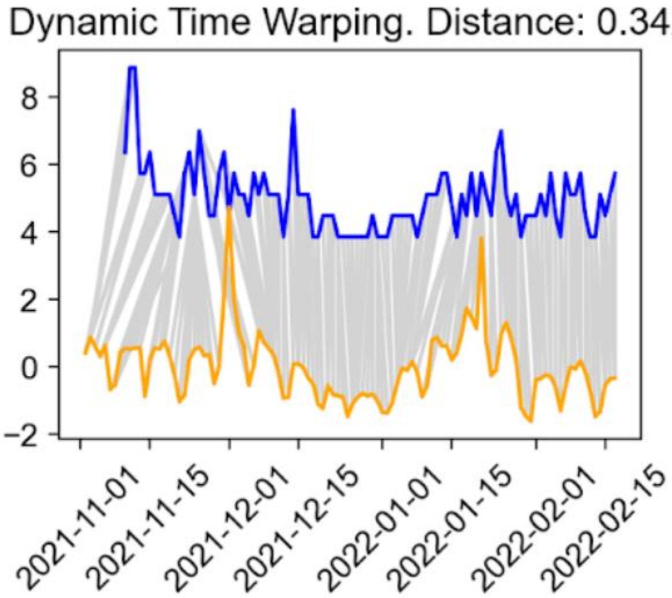


**Slika 2: Primerjava rezultata procesa zaznavanja anomalij.**

### 3.3 Korelacije med poslovnimi metrikami

Da bi podprli tudi diagnostično analitiko, trenutno razvijamo funkcionalnost za predstavitev korelacij med poslovnimi metrikami. Cilj je, da za uporabnika identificiramo povezane metrike in jih ustrezno vizualiziramo, kar bo uporabnikom omogočilo vpogled v relacije med metrikami. Osnovna zahteva funkcionalnosti je v njeni intrepetabilnosti, kar pomeni, da morajo biti povezane metrike smiselno predstavljene ter uporabniku razložljive. V ta namen smo pred pričetkom razvoja v okviru raziskovalnega projekta analizirali različne mere podobnosti za

časovne vrste ter preučevali njihovo časovno zahtevnost, obravnavanje manjkajočih vrednosti ter interpretabilnost.
Mere podobnosti, ki smo jih izbrali in jih lahko uporabimo v produktu so Pearsonov koeficient korelacije, navzkrižne korelacije (ang. Cross-correlation) ter metoda dinamičnega časovnega sledenja (ang. Dynamic Time Warping - DTW). Vsaka med njimi prinaša izzive, zlasti ko gre za interpretabilnost rezultatov. Npr. metoda DTW zahteva kompleksne vizualizacije, vendar lahko ponudi tudi najbolj nepričakovane vpoglede.



**Slika 3: Predstavite korelacij z metodo dinamičnega časovnega sledenja.**

Korelacij pa ne želimo podpirati zgolj v okviru vizualizacij, pač pa želimo na njihovi osnovi uporabnika tudi obveščati, npr. kadar se korelirane metrike pričnejo oddaljevati, ali pa ga usmerjati na korelirane metrike npr. pri zaznavanju anomalij. Končni cilj je identifikacija ključnih dejavnikov upešnosti poslovanja. Npr. ob vpogledu, da prodaja v poslovalnici korelira s tedensko sezonskostjo in je največja ob sobotah v času tik pred koncem delovnega časa, lahko poslovanje izboljšamo s prilagoditvijo delovnega časa.

### 3.4 Generativna umetna inteligenca

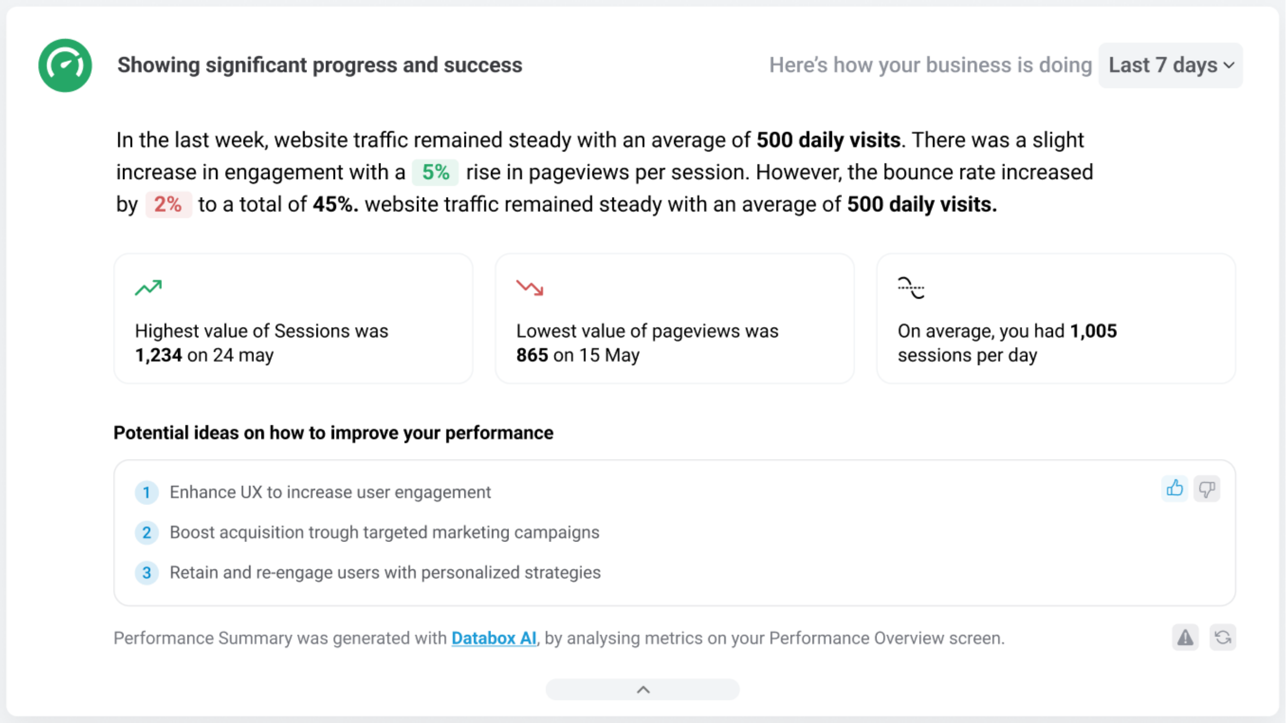
Generativna umetna inteligenca se je v preteklem letu izkazala kot prelomna tehnologija, ki korenito spreminja naš način življenja in dela. Generiranje nove vsebine se je izkazalo za uporabno ne samo v kreativnih disciplinah, kot so umetnost, glasba in ustvarjalna besedila, ampak tudi v poslovni analitiki. Ta tehnologija Databoxu omogoča prehod iz vloge orodja za poročanje in vizualizacijo podatkov v vlogo aktivnega asistenta pri sprejemanju odločitev.

V kombinaciji z obsežnimi agregiranimi podatki, generativna umetna inteligenca omogoča integracijo funkcionalnosti, ki z dodajanjem prilagojenih informacij obogatijo obstoječe podatke. Eden izmed ključnih primerov uporabe v našem produktu je generiranje celovitih pisnih poročil, ki poudarjajo ključne ugotovitve. S pomočjo generativne umetne inteligence lahko ustvarimo pripovedi, ki izpostavijo najpomembnejše informacije in tako olajšajo razumevanje kompleksnih podatkov širokemu krogu uporabnikov.

Z namenom, da bi to omogočili, smo razvili mikrostoritev, ki ponuja funkcionalnosti generativne umetne inteligence drugim komponentam Databoxovega storitvenega ekosistema. Storitev prejme podatke in metapodatke poslovnih metrik, nato pa vrne povzetek, trend in priporočila, ki temeljijo na podatkih in rezultatih drugih storitev, podprtih z umetno inteligenco. Povzetek vključuje deskriptivni in analitični opis podatkov, trend pa je binarna vrednost, ki zbere trend vseh podanih podatkov v en rezultat. Priporočila so rezultat preskriptivne analize generativnega modela in so predstavljena v obliki alinej. Odjemalci storitve lahko prilagajajo parametre, kot so največje število znakov in število priporočil.

Uporabljamo generativni modelt GPT4 [6], katerega razvija raziskovalni laboratorij podjetja OpenAI. Zanj smo se odločili na podlagi pozitivnih izkušenj med testiranjem. Zahtevnost našega produkta namreč presega zgolj proizvodnjo golega besedila. Generativni model mora besedilo obogatiti s HTML značkami, ki besedilo dopolnjujejo z dodatno semantiko, kot je na primer uporaba zelene barve za naraščajoče in rdeče barve za padajoče metrike. Da bi bilo branje rezultatov modela enostavnejše, je dodatna zahteva, da model vrne rezultate skladno z JSON shemo, ki je podana v vhodnih podatkih modela.

Razvoj aplikacij, podprtih z velikimi jezikovnimi modeli predstavlja poseben izziv. Razen pravilne integracije v že obstoječe sisteme je potrebno pri snovanju rešitve upoštevati skalabilnost, odzivnost in upravljanje s stroški. Posebno težavo pri razvoju in testiranju rešitve je predstavljala omejitev dostopa do modela ChatGPT4. Zaradi velikega povpraševanja po modelu smo bili precej časa na čakalni listi za dostop. Posledično smo aplikacijski del rešitve začeli razvijati preden smo dobili dostop do ChatGPT4. Rešitve smo prototipirali s spletnim vmesnikom, ki ga gosti OpenAI. Razen tega smo se spopadali z najboljšim načinom priprave in posredovanja JSON sheme tako, da jo bo model uporabil za formatiranje izhoda. Kot rešitev smo uporabili "tagging" funkcionalnost knjižnice LangChain[7], katera se je v svetu razvoja aplikacij na osnovi velikih jezikovnih modelov izkazala kot nepogrešljivo.



In the last week, website traffic remained steady with an average of **500 daily visits**. There was a slight increase in engagement with a 5% rise in pageviews per session. However, the bounce rate increased by 2% to a total of **45%.** website traffic remained steady with an average of **500 daily visits.**

Highest value of Sessions was **1,234** on 24 may

Lowest value of pageviews was **865** on 15 May

On average, you had **1,005** sessions per day

**Potential ideas on how to improve your performance**

1 Enhance UX to increase user engagement
2 Boost acquisition trough targeted marketing campaigns
3 Retain and re-engage users with personalized strategies

Performance Summary was generated with Databox AI, by analysing metrics on your Performance Overview screen.

**Slika 4: Primer generiranega opisa večih metrik.**

## 4 Zaključek

Živimo v vznemirljivem času, ko se umetna inteligenca razvija s svetlobno hitrostjo. Tehnološki vzpon na tem področju so v zadnjem obdobju dodatno pospešili veliki jezikovni modeli ter povezane tehnologije kot je npr. ogrodje Langchain. Podjetja, ki se tega tehnološkega napredka zavedajo in tehnologijo ustrezno izkoriščajo, bodisi z neposredno vpeljavo teh tehnologij v procese dela, bodisi z uporabo naprednih orodij za poslovno analitiko, so nedvomno učinkovitejša in si ustvarjajo konkurenčno prednost.

V prispevku smo predstavili funkcionalnosti, preko katerih tudi sami vpeljujemo metode umetne inteligence v naš osnovni produkt za poslovno analitiko. Z napovedovanjem poslovnih metrik, zaznavanjem anomalij, iskanjem korelacij med metrikami ter funkcionalnostmi na osnovi generativne umetne inteligence želimo podpreti vse tipe poslovne analitike.

Predstavili smo tudi pomembnejše izzive, s katerimi smo se srečevali med razvojem. Pri tem izpostavljamo izzive zaradi uporabe generaliziranih metod in parametrov na različnih podatkovnih zbirkah, poseben izziv pa predstavlja tudi zavedanje, da obnašanje tovrstnih sistemov definirajo predvsem podatki in ne zgolj nekakšno zaporedje funkcij, česar smo vajeni iz tradicionalnih sistemov s popolnoma predvidljivimi izhodi.

## Literatura

[1] https://en.wikipedia.org/wiki/Business_analytics, obiskano 21. 7. 2023.

[2] https://www.analyticsinsight.net/four-types-of-business-analytics-to-know/, MEHTA Anushka, "Four Types of Business Analytics to Know", 2017, obiskano 21. 7. 2023.

[3] https://en.wikipedia.org/wiki/Augmented_Analytics, obiskano 27. 7. 2023.

[4] TAYLOR Sean J, LETHAM Benjamin "Forecasting at scale", PeerJ Preprints, 2017. https://facebook.github.io/prophet/, obiskano 27. 7. 2023.

[5] CHANDOLA Varun., BANERJEE Arindam, KUMAR Vipin, Anomaly detection: A survey, ACM Computing Surveys, letnik 41, številka 2 str. 1-58, 2009

[6] OpenAI, GPT4 Technical report, 2023

[7] https://www.langchain.com, obiskano 26.7.2023

# Podatkovni inženiring – vzpon, razvoj in prihodnost

Jure Jeraj, Anžej Curk

Result d.o.o., Ljubljana, Slovenija
jure.jeraj@result.si, anzej.curk@result.si

Podatkovni inženiring je doživel neverjeten vzpon in razvoj v zadnjih letih, postajajoč ključen gradnik sodobnega poslovanja in tehnološkega napredka. Ta kompetenca se osredotoča na oblikovanje, razvoj in vzdrževanje robustnih podatkovnih arhitektur, ki omogočajo učinkovito obdelavo, shranjevanje in analizo velikih količin podatkov. Je odgovor na eksponentno rast količine podatkov, ki jih proizvajajo organizacije in uporabniki po vsem svetu. Je privedel do nastanka številnih tehnologij in orodij, ki omogočajo boljše obvladovanje podatkov. To vključuje napredne podatkovne baze, porazdeljene sisteme za obdelavo podatkov, orodja za integracijo in transformacijo podatkov ter platforme za orkestracijo in upravljanje podatkovnih tokov. Prav tako so se razvile metodologije in prakse, ki omogočajo avtomatizacijo in standardizacijo procesov podatkovnega inženiringa ter zagotavljajo doslednost in zanesljivost. Podatkovni inženiring postaja pomembne člen vsake moderne organizacije ali družbe, ki so prepoznale možnost inovativnih rešitev na podlagi moči podatkov.

**Ključne besede:**
podatkovni inženiring
velepodatki
upravljanje s podatki
procesiranje v realnem času
nOps

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.26

# 1 Uvod

Danes se podatkovni inženiring jemlje kot kritična komponenta modernih podatkovnih struktur. Te so posledica ekstremne rasti [1] količine podatkov v zadnjih letih, predvsem kot posledica vseh mobilnih naprav, interneta stvari, socialnih omrežij, oblačnega procesiranja ter tudi zaradi naglega razvoja poslovne inteligence v preteklem desetletju. Podatkov pa ni samo več, prihajajo hitreje in se pojavljajo v manj strukturiranih oblikah. Klasične podatkovne baze in klasični ETL procesi tega enostavno ne zmorejo več, zato je bilo potrebno poseči k zahtevnejšim rešitvam, te pa so za seboj potegnile tudi povsem novo kompetenco, in sicer podatkovni inženiring.

Podatkovni inženiring se tako povezuje s pojavitvijo velepodatkov. Velepodatki so na preprost način opredeljeni kot tiste zbirke podatkov, ki jih ni mogoče obdelovati s klasičnimi in tradicionalnimi metodami [2]. In če ni možno delati tradicionalno, so potrebne nove veščine. In tako se pojavi podatkovni inženiring kot ga opredeljujemo danes.

Sicer pa ne smemo spregledati, da se podatkovni inženiring sramežljivo pojavlja že od devetdesetih let dvajsetega stoletja z uveljavljanje podatkovnih zbirk in njihove analize. A vseeno pravi razmah se je zgodil s pojavitvijo velikih tehnoloških korporacij kot so Facebook, AirBnb, Amazon, Apple, Netflix… Njihov poslovni model v osnovi temelji na produktivizaciji in monetizaciji podatkov. In kjer so podatki osnovno sredstvo obstoja, je tudi inženirska skrb za podatke na najvišji prioriteti vodstva.

V povezavi z zgoraj omenjenimi korporacijami je objavljen zanimiv blog članek [3] Maxime Beauchemina, ki pravi, da se je leta 2011 pridružil Facebooku kot inženir poslovne analitike. Ko ga je leta 2013 zapustil je bil podatkovni inženir, pri tem pa ni niti napredoval niti ni zamenjal delovnega mesta. Enostavno je Facebook v tem času ugotovil, da je delo, ki ga je Maxime opravljal, presega običajno delo inženirja poslovne analitike. Zato so začeli razvijati nove veščine in tako ustvarili novo kompetenco.

Za zanimivost, podobno pot sva opravila tudi avtorja tega članka. Sredi prejšnjega desetletja sva se pridružila ekipi mednarodnega podjetja kot ETL razvijalca, nekaj let kasneje pa ugotavljava, da se lahko opiševa kot podatkovna inženirja. A postala sva pozorna še na en element – če sva pred leti delala skoraj enake zadeve, zdaj opravljava različne zadeve in se niti ne moreva več primerjati med seboj. To nakazuje, da je podatkovni inženiring postala zelo široka vloga in se lahko tudi znotraj nje pojavljajo določene specializacije. To širino bova poskusila na kratko predstaviti v tem prispevku, v smeri povzetka, kako se je ta kompetenca razvijala v preteklih letih.

Osredotočila se bova predvsem na elemente s katerimi sva imela izkušnje in jih dobro poznava: to pa povezala s ključnimi prelomnicami in mejniki, ki jih je zelo dobro povzel že pred nama tudi Tobias Macey v blog prispevku o vzponu podatkovnega inženiringa [4].

# 2 Apache License 2.0

Ob pogledu nazaj na vse ključne komponente, ki so pripeljali do današnjega stanja, lahko ugotovimo, da se je veliko inicialnega razvoja zgodilo v omenjenih velikih korporacijah. Pri tem pa je ključno, da so te korporacije zelo hitro svoje rešitve objavile kot odprtokodne, skoraj vedno pod licenco Apache License 2.0.

Apache License 2.0 je ena najbolj razširjenih odprtih licenc za programske rešitve. Prvič je bila objavljena s strani Apache Software Foundation, organizacije, ki spodbuja razvoj odprte programske opreme. Pred ustanovitvijo Apache Software Foundation leta 1999 je bila skupina razvijalcev, znana kot Apache Group, ključna pri ustvarjanju projekta Apache HTTP, ki je postavil temelje za številne druge odprtokodne projekte. Danes ta licenca omogoča svobodno uporabo, distribucijo in spreminjanje odprtokodne programske opreme, kar je prispevalo k razcvetu odprtih tehnologij po vsem svetu.

Te rešitve so tako močne in razširjene, da veliki ponudniki oblačnih rešitev, pa tudi komercialnih komponent uporabljajo kot osnovo za svoje storitve. S to licenco imajo namreč prost dostop do kakovostnih in preverjenih odprtih tehnologij, ki so že bile razvite in preizkušene v skupnosti. Nato nadgradijo te osnovne odprtokodne rešitve z lastnimi funkcijami, prilagoditvami in izboljšavami, ki ustrezajo njihovim specifičnim potrebam in zahtevam trga. S tem pristopom lahko ponudniki oblačnih storitev hitro razvijajo in ponujajo inovativne in konkurenčne rešitve, ki so hkrati stabilne in zanesljive za njihove stranke.

Zaradi tega imajo podatkovni inženirji še nekaj manj opaznih prednosti. Skupnost za podporo posameznih komponent je mnogo večja, saj je v veliko osnovnih primerih že dovolj podpora za odprtokodno rešitev in nato lahko rešuješ tudi primere na oblačni storitvi. Konkretni primer je lahko AWS Athena (storitev za ad hoc SQL poizvedbe nad podatkovnim jezerom), ki izhaja iz odprtokodne rešitve PrestoDB (tudi objavljen pod Apache License 2.0). Za samo sintakso in specifike AWS Athene je tako povsem dovolj, da uporabljamo dokumentacijo PrestoDB-ja.

Prednost pa je tudi v tem, da so visokonivojske arhitekture med seboj primerljive, predvsem na vseh večjih ponudnikih oblačnih storitve, kar pomeni, da se lahko inženirji hitreje prilagodijo na drugega ponudnika. Možna pa je tudi kombinacija originalnih odprtokodnih rešitev in lastnih komercialnih storitev teh ponudnikov, kar močno poveča fleksibilnost razvoja.

## 3 Apache Hadoop – velike količine podatkov

Apache Hadoop je prvo uveljavljeno ogrodje oz. zbirka orodij za obdelavo velikih količin podatkov. Razvijati sta ga začela Doug Cutting in Mike Caferalla leta 2004 v času porasta spletnih iskalnikov; glavni namen je bil narediti možnost porazdeljene obdelave podatkov. Doug Cutting se je leta 2006 pridružil Apache Software Foundation, s tem pa se je Hadoop prenesel v skupnost odprtokodnih projektov, posledično je doživel nagel razvoj na področju obdelave velike količine podatkov.

Temeljna prednost Apache Hadoopa je njegova sposobnost obvladovanja velikih količin podatkov na porazdeljen način preko ključne paradigme, imenovane MapReduce. Ta metoda razčleni velike naloge na manjše, ki jih nato porazdeli na več vozlišč v grozdu. Vsako vozlišče neodvisno izvaja naloge, nato pa rezultate združi v končni izhod. To omogoča izjemno paralelno obdelavo, ki poveča učinkovitost in zmogljivost sistema.

Druga ključna komponenta Apache Hadoopa je distribuiran sistem za shranjevanje podatkov Hadoop Distributed File System (HDFS). HDFS omogoča shranjevanje podatkov na več vozliščih v grozdu, kar zagotavlja vzdržljivost podatkov in visoko razpoložljivost. Zaradi te zasnove je Hadoop odporen na izpad posameznih vozlišč in omogoča nemoteno delovanje tudi pri okvarah strojne opreme.

Apache Hadoop je tako postal (in pravzaprav še vedno ostaja) osrednji del ekosistema za obdelavo podatkov v velikih količinah. Je odprtokoden in ima še vedno zelo veliko skupnost, ki ga uspešno razvija. Mogoče ni več prva izbira za končne uporabnike, je pa svojima konceptoma MapReduce in HDFS uporaben v drugih modernejših ogrodjih.

## 4 Apache Kafka – podatki v realnem času

Naslednji izziv pri velepodatkih je bila hitrost generiranje podatkov in potreba po upravljanju z njimi praktično takoj. Rešitev se je ponudila leta 2011, ko je Linkedin svojo interno rešitev naredil odprtokodno in jo predal Apache Software Foundationu.

Apache Kafka je danes praktično nepogrešljiva v ekosistemu obdelave podatkov v realnem času. Ena ključna prednost Apache Kafka je njegova sposobnost obvladovanja ogromnih tokov podatkov na visoki ravni zmogljivosti in nizki latentnosti. Deluje na principu porazdeljenega sistema, kjer se podatki shranjujejo v podatkovnih vrečah, imenovanih teme (ang. topics), in so nato poslani na različne porabnike (ang. consumers), ki jih obdelujejo skladno s svojimi potrebami. Ta arhitektura omogoča večkratno branje podatkov in preprosto horizontalno razširjanje sistema, kar zagotavlja visoko razpoložljivost in vzdržljivost.

Apache Kafka se pogosto uporablja kot osrednji gradnik v arhitekturi sistema za obdelavo podatkov v realnem času, kjer se podatki neprekinjeno zajemajo iz različnih virov, kot so senzorji, strežniki, aplikacije in druge naprave. Ti podatki se nato prenašajo v realnem času prek Kafka tokov, kar omogoča hitro obdelavo, analizo in posredovanje na druge ciljne sisteme ali aplikacije.
Poleg tega se Kafka odlično integrira z drugimi odprtokodnimi orodji za obdelavo podatkov, kot sta Apache Spark in Apache Hadoop (in vsemi komercialnimi izpeljankaj kot so npr. Cluodera, Databricks ipd.).

## 5 Podatkovna jezera in oblačne storitve

Prejšnja dva elementa omogočata hitrejšo obdelavo ogromne količine strukturiranih in tudi nestrukturiranih podatkov. Vpeljuje programerske tehnike v obdelavo podatkov, ki pa odstopajo od metod klasičnih relacijskih baz kot so npr. Oracle, IBM DB2 ali MS SQL Server. Enostavno je osnovna arhitekturna zasnova relacijskih baz namenjena za zagotavljanje (poenostavljeno) transakcijske konsistentnosti podatkov pri kompleksnih aplikativnih sistemih, ki temeljijo na digitalizaciji in avtomatizaciji procesov. Medtem ko zgoraj omenjeni koncepti so osredotočeni na upravljanje in obdelavo podatkov samih.

Če se spomnimo, ena izmed ključnih komponent Hadoopa je HDFS, ki je v bistvu prilagojen datotečni sistem. To pa pravzaprav pripelje do tega, da se pri obdelavi velepodatkov začenjamo ukvarjati tudi s samimi koncepti in sistemi za shranjevanje (s čimer se pri relacijskih bazah pravzaprav ne ukvarja niti bazni administrator). Zato je bilo to nekaj novega in to je tudi osnova za nastanek izraza podatkovno jezero. Ta se je začel uveljavljati šele po letu 2011, torej kar 5 let kasneje od prve verzija Hadoopa in HDFS-ja.

Podatkovna jezera so tako predvsem shrambe v katero lahko zelo hitro zapisujemo strukturirane, polstrukturirane in nestrukturirane podatke. Pri podatkovnih jezerih lahko podatke zapisujemo izredno hitro, ker se ne preverja konsistenca zapisov. Tak sistem omogoča boljšo skalabilnost, hkrati je tudi cenejši, ker ima manj kompleksnejših elementov. Te namreč nadomeščamo z ogrodji in rešitvami kot sta Hadoop, Kafka ….

Z že do zdaj povedanim lahko razumemo, da je z vsemi temi komponentami prišlo do situacije, ko je mnogo več elementov odvisnih od same arhitekturnih rešitev. S tem pa se je odprla novo področje, in sicer upravljanje skalabilnosti. Z možnostjo nadzora posameznih komponent se lahko bolje izrablja procesorska, strežniška in infrastrukturna moč. Potrebe po tem pa se lahko močno spreminjajo čez dan, čez teden, lahko tudi čez leto.

Ker so organizacije vedno bolj odvisne od tega procesiranja, so morale zagotavljati računalniško in mrežno infrastrukturo za pokrivanje največjih špic delovanja. Te pa se pojavljajo le občasno, kar pripelje do situacije, da je bila večino časa pripravljena infrastruktura neizkoriščena.

To pa je en od razlogov, da so se zelo razširile oblačne storitve, specifično tudi na področju podatkovnega inženiringa. Možnost skalabilnosti na manjši časovni enoti (lahko tudi na nivoju posamezne ure ali dneva) je izredno pomembna, hkrati pa tudi lastnost, ki jo ponudniki ponujajo na način »plačaj kolikor uporabljaš«.

Pri tem pa se je treba zavedati, da za vsako področje (npr. sistemska administracija, bazna administracija, varnost…) dobite storitev najvišje kvalitete. Glavni razlog, da je vse to možno, je lastnost, da je so mejni primeri oz. špice v zahtevah redke in se lahko lažje porazdelijo med tisoče različnih organizacij.

Smatramo, da je vzpon modernih podatkovnih struktur in oblačnih storitev povezan in da bi verjetno težko uspela en brez drugega. Podatkovna jezera in oblačne storitve so tako danes enako ključni gradnik v sodobni obdelavi podatkov kot predhodno omenjene komponente.

## 6 Orkestracija sistemov in podatkovni katalog

Verjetno imamo v vsakem sistemu skrite junake. Avtorja smatrava, da so to v našem primeru sistemi za orkestracijo opravil in sistemi za nadzor na izvajanje teh opravil ter dokumentiranje vseh meta podatkov na kateremkoli nivoju naše arhitekture.

Sodobne arhitekture na področju upravljanja s podatki so sestavljene iz različnih komponent. Tudi na oblačnih storitvah imamo veliko število storitev (npr. AWS ima skupno več kot 300 različnih storitev), s tem, da imamo vedno možnost uporabljati tudi odprtokodne rešitve. Ni pa niti nenavadno, da se uporabljajo storitve različnih ponudnikov oblačnih storitev.

Število različnih storitev in heterogenost povečujeta možnost, da se pri procesiranju podatkov kje zalomi. Izraz, ki se je uveljavil za nadzor, pregled in urejanje tega področja je orkestracija opravil. Sam izraz nakazuje na usklajeno delovanje orkestra. Vsak glasbenik orkestra mora točno vedeti kaj igra v katerem delu nastopa je, kaj se je in kaj se bo zgodilo. Večji kot je orkester, zahtevnejše je. Tako se povečuje vloga dirigenta. Povsem enako je v modernih podatkovnih rešitvah, saj imamo veliko komponent in (mikro) rešitev, ki jih moramo pravilno povezati v celoto. Vlogo dirigenta prevzemajo namenska orodja in rešitve za orkestracijo.

Trenutno zelo popularno orodje je Apache Airflow. Sledi vsem trendom iz področja, predvsem pa je izredno povezljiv s preostalimi rešitvami fundacije Apache. Povezljivost je pa ključna za dobro orkestracijo. V povezavi s spodobnim uporabniškim vmesnikom nam daje to možnost, da imamo lahko z enim (neodvisnim) orodjem nadzor nad vsemi akcijami in opravili pri našem upravljanju s podatki. Na ta način lahko vidimo zelo zemljevid naših opravil, poročila o izvajanju, ozka grla ali pa identificirati težave še predno bi jih opazili končni uporabniki.

Orkestracija je podatkovnim inženirjem še blizu; pravzaprav je samo močna nadgradnja nekdanjih »cron« opravil ali podobnih rešitev. Na drugi strani pa se zdi, da je podatkovni katalog nekaj, kar je vedno prvi kandidat, da se ga preskoči.

V razdrobljenem in heterogenem sistemu je pomembno, da imamo centraliziran sistem za preglednost in usklajenost podatkov oz. meta podatkov. V večjih organizacijah so procesi ločeni med posameznimi oddelki ali enotami, istočasno pa so oddelki, ki sprejemajo odločitve na podlagi podatkov, ki niso pod njihovim lastništvom. To so navadno večje organizacije, kjer je zaradi velikosti tudi komunikacija med enotami otežena zaradi široke organizacijske strukture.

V teh primerih je podatkovni katalog edina stična točka, da se lahko posamezne enote sporazumevajo med seboj kaj točno pomeni posamezen podatek. Ali pa kakšno je poslovno pravilo za posamezen kazalnik. Velja pripomniti, da je zelo dobra lastnost podatkovnih analitikov in znanstvenikov, da se vedno znajdejo in najdejo podatke, za svoje delo. A včasih samo iz podatkov samih ni možno razbrati, kaj predstavljajo. Posledično pomanjkljivo ali ceno napačno razumevanje lahko pripelje tudi do netočnih, zavajajočih ali celo kritično napačnih analiz, poročil ali ukrepov. Najbolj klasični primeri – za vsakega, ki je delal pri mednarodnih korporacijah – so časovni pasovi, valute in merske enote. Namreč hitrost 40 ni enaka v Nemčiji ali v Veliki Britaniji. A brez dodatnih informacij ni mogoče predvideti, ali so podatki v sistemu že pretvorjeni v enotno mersko enoto ali ne.

Namen podatkovnega kataloga je, da poveže sorodne podatke. In tudi da pove lastnosti posameznega podatka, predvsem če je časovno, geografsko ali kako drugače povezan.

Najina izkušnja je, da ravno za področje podatkovnega kataloge je še zelo malo res učinkovitih orodij, predvsem odprtokodnih. Trenutno je verjetno najbolj napredna odprtokodna rešitev DataHub, ki je tudi objavljen pod licenco Apache License 2.0.

## 7 Evolucija poslovne analitike v podatkovno analitiko

Uvodoma smo podali primer, kako se je podatkovni inženiring razvil iz poslovne analitike. Zanimivo je pogledati kakšen je po vseh teh spremembah današnji položaj poslovne analitike. Predvsem bi radi poudarili, da imamo tu malo smole s slovenskimi prevodi. Pri nas se je namreč – mogoče celo preveč – uveljavil izraz poslovna analitika; dobesedni prevod bi namreč bil poslovna inteligenca (ang. Business intelligence). To omenjamo, ker se danes mednarodno več uporablja podatkovni analitik (ang. Data Analytic). Ni pa samo preimenovana vloga, tudi vrline so se bolj specializirale na zmožnost iskanja, razumevanje, priprave in obdelave podatkov.

Praktično to pomeni nadaljnjo delitev med tehničnimi in poslovnimi vlogami. Izzivi poslovnih priložnosti se ves čas premikajo k skrajnostim; če je bilo pred 10 leti še izziv, kako pregledati vse lastne podatke, je danes izziv kako vpeljati inteligentne metode ne le nad lastnimi podatki, temveč v kombinaciji s komercialnimi, odprtimi in pravzaprav vsemi možnimi podatki.

V današnjem svetu se pa odpira dodatna vrlina interpretacije podatkov. Če smo včasih opredelili strokovnjaka za poslovno inteligenco kot most med tehničnim in poslovnim svetom, lahko danes ugotovimo, da je ta most vrlina posameznika, da razume podatke in jih zna smiselno predstaviti poslovnim uporabnikom in odločevalcem. To vlogo danes lahko opravljajo tako napredni poslovni analitiki kot razumevajoči podatkovni analitiki. Ta vez je vedno obstajala in bo tudi v prihodnje obstajala; spremembe so predvsem del zahtevanih specializacij kot odgovor razvoja, ki smo do zdaj že prikazali.

## 8 nOps

Ne moremo se izogniti vzporednicam med inženirjem za razvoj programske opreme in podatkovnim inženirjem. Pravzaprav so vzporednice zelo močne; podatkovni inženiring je nekje na stopnji kot je bil razvoj programske opreme npr. pred 15, 20 leti. Zato ne čudi, da se nekatere aktivnosti razvijajo po zelo podobnih poteh. Eno izmed področij je nOps, ki je nadpomenka raznim ostalim Ops sistemom, kot je npr. verjetno najbolj znan DevOps. Posledično se na področju podatkov poskuša vzpostaviti izraz DataOps.

Sami smo mnenja, da se ta izraz ne bo uveljavil, predvsem iz nekaj razlogov.

Prvi je, da tehnično gledano ni tako zelo drugačen od DevOps. Podatkovni inženiring je namreč le specializirana veja razvoja programske opreme, ki je specializirana za področje podatkov. Zato so tudi operacije za razvoj teh rešitev podobne operacijam za razvoj programske opreme. Iz tega vidika je DataOps pravzaprav zelo soroden kot DevOps.

Drugi razlog pa je, da so se same operacije nad kvaliteto podatkov, orkestracijo in lastnostmi že razvile v okviru lastnosti celovitega upravljanja s podatki (ang. Data Governance) in je velika možnost, da se to področje ne bo razumelo kot del DataOps. S tem pa potem

Obstaja pa še tretji razlog, kjer se pa težje razume vpliv na potencialni razvoj izraza DataOps. Prej in bolj se je namreč že uveljavilo področje MLOps, ki pokriva operacije nad strojnim učenje. To pa obsega tudi delo s podatki, saj brez urejenih in smiselnih podatkov ni metod strojnega učenja. Avtorja tega članka, kljub temu, da se identificirava kot podatkovna inženirja, bova težko sprejela izraz DataOps, ker smatrava, da se ga uveljavlja malo na silo, po vzoru ostalih uveljavljenih Ops zadev.

A ne glede na najino mnenje, aktivna operativna aktivnost na področju podatkov – kvaliteta, orkestracija in dokumentacija – je, je bila in bo ena izmed ključnih elementov upravljanja podatkov.

## 9 Prihodnost (podatkovnega) inženirstva

Kljub temu, da je napovedovanje prihodnjih trendov nehvaležno, pa vseeno lahko podava nekaj najinih razmišljanj, o prihodnosti podatkovnega inženirstva.

Iz prispevka sledi, da so se večje spremembe začele nekje od leta 2005. V naslednjih 10 letih se je izvedlo veliko nadgradenj in rešitev, ki so zavzele prazen prostor na področju upravljanja z velepodatki. Tega so omogočile evolucije programske opreme in infrastrukturne zmožnosti, predvsem oblačne storitve. V tem obdobju so se pripravili odgovori na enormen količine podatkov, procesiranje v realnem času in upravljanje tudi z nestrukturirani podatki. S tem smo zadostili potrebam velepodatkov, ki so osnovno opredeljeni s 3V definicijo (volumen [ang. volume], hitrost [ang. velocity] in raznolikost [ang. variaty]).

V prihodnosti ne pričakujemo takih strateških sprememb, kot so se zgodile v zadnjih 15 letih. Se bo pa še vedno razvijala in sledila trendom predvsem iz vidika združevanja še več podatkov ter poudarka na varnosti in sistemom za upravljanje s podatki. Verjetno bo veliko vzporednic s trendi razvoja programske opreme. Ogrodja, rešitve in knjižnice se bodo ves čas posodabljale, prilagajale na nove okoliščine in sledila ostalim potrebam po podatkih. Razvoj zagotovo ne bo izostal, a zdi se, da je trenutno predvsem čas za ustrezne za maksimizacijo izrabe tehnologije na strani poslovnega sveta.

Se pa nakazuje zanimiv trend, ki se dogaja v zadnjih letih - razvoj vloge inženirja strojnega učenja (ang. ML engineer) oz. na splošno vzpon inženirstva. Trenutno je namreč val navdušenja nad umetno inteligenco iz vseh področij. Lahko je to strojno učenje, lahko je to generativna umetna inteligenca, lahko kaj drugega.

Vse to navdušenje se bo preneslo do porasta večje količine modelov za reševanje najrazličnejših izzivov. S tem pa se bo pojavila težava vpeljati te rešitve v vsakdanjo, produkcijsko uporabo, najverjetneje na podlagi velepodatkov in obdelavo v realnem času.

Ocenjujemo, da se bo v bližnji prihodnost razvilo veliko različnih inženirskih smeri, a v nekem trenutku se bo ta raznolikost začela združevati in upamo, da se bo usmerilo v pot, da bomo razumeli, da imamo le ene inženirstvo z različnimi specializacijami. Ker konec koncev, veliko je skupnih elementov med inženirjev programske opreme, podatkovnim inženirjem ali pa inženirjem strojnega učenja.

## 10 Sklepna misel

V uvodnih mislih tega članka sva avtorja podala najino izkušnjo prehoda iz ETL razvijalca v podatkovna inženirja ter ugotovitve o raznolikosti te vrline. V prispevku sva želela prikazati obširen razvoj področja na katerem delujeva. Desetletje ali dve nazaj je bil IT sektor osredotočen na aplikacije – digitalno podporo procesom in njihovo avtomatizacijo. Ekosistem se je delil na transakcijsko (OLTP) in analitično (OLAP) področje, kjer pa je bil analitičen svet podrejen transakcijskem.

Nato pa so se začele vzpenjati korporacije, ki so se zavedale moči podatkov in so svoje poslovne modele ustvarile na podlagi inovativne izrabe podatkov. Kar naenkrat se je zgodilo, da organizacija ne sloni na zgolj na transakcijskih sistemih, temveč da so tudi podatki pomemben del generiranja prihodkov.

Zato so te korporacije začele usmerjati razvoj rešitev v smeri izrabe vseh podatkov – tudi nestrukturiranih - v najkrajšem možnem času. S tem so kreirale ekosistem velepodatkov, hkrati pa ponudile rešitve za njihovo procesiranje. Ena izmed pozitivnih posledic pa je tudi vpeljava novih vrlin, ki so združila kot vloga podatkovnega inženirstva.

Podatkovni inženiring je zdaj že uveljavljena in prepoznana vloga. Namenjena je organizacijam, ki se zavedajo, da morajo za svojo odpornost uporabljati več podatkov, tudi odprte in komercialne. In da morajo podatke procesirati hitreje od konkurence.

Za zaključek pa moramo biti vseeno tudi realni – podatkovni inženirji ne bodo nikoli zvezde organizacij verjetno niti IT ekip. Bodo pa ključni člen za uspešno produkcijsko vpeljavo metod umetne inteligence v organizacije in skupnosti.

## Literatura

[1] https://www.gartner.com/en/documents/3898487, Gartner Magic Quadrant for Data Management Solutions for Analytics, obiskano 25. 7. 2023

[2] JERAJ Jure, NERED Urška, NIKOLOSKI Stevanče "Velepodatki – 5V-jev v praktičnih primerih", Uporabna informatika. Letnik 31, Številka 1 (maj 2023), str. 4 – 14.

[3] https://medium.com/free-code-camp/the-rise-of-the-data-engineer-91be18f1e603, The Rise of the Data Engineer, obiskano 25. 7. 2023

[4] https://www.dataengineeringpodcast.com/six-year-retrospective-episode-361, Reflecting On The Past 6 Years Of Data Engineering, obiskano 25. 7. 2023

# OTS 2023 Sodobne informacijske tehnologije in storitve

## Zbornik šestindvajsete konference

Luka Pavlič, Tina Beranič, Marjan Heričko (ur.)

Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko,
Maribor, Slovenija
luka.pavlic@um.si, tina.beranic@um.si, marjan.hericko@um.si

V zborniku šestindvajsete konference OTS 2023 so objavljeni prispevki strokovnjakov s področja informatike, v katerih so predstavljena nova spoznanja in trendi razvoja, vpeljave, prilagajanja ter upravljanja informacijskih rešitev, kot tudi konkretni uspešni pristopi in dobre prakse. Prispevki naslavljajo področja sodobnih arhitekturnih izzivov, klasične, generativne in globoke umetne inteligence, sodobnih spletnih ali mobilnih uporabniških vmesnikov, kot tudi tradicionalnih, brezstrežniških in decentraliziranih zalednih sistemov v oblaku. Tematike prispevkov obsegajo tudi zagotavljanje ustreznega skalabilnega okolja zanje ter avtomatizacijo testiranja, merjenje kakovosti in dostavo s proaktivnim naslavljanjem najpogostejših kibernetskih napadov. Rdečo nit prispevkov predstavljajo podatkovne tehnologije, ki so zastopane v obliki klasičnih podatkovnih baz, podatkovnih jezer ter učinkovitega zbiranja, obdelave in vizualizacije velepodatkov. Prispevki tako še naprej omogočajo boljšo povezanost IT strokovnjakov, informatikov, arhitektov in razvijalcev naprednih IT rešitev in storitev, kot tudi akademske sfere in gospodarstva.

**Ključne besede:**
informatika
informacijske tehnologije
programsko inženirstvo
informacijski sistemi
informacijske rešitve
digitalna preobrazba
razvoj mobilnih in spletnih rešitev
arhitekture v oblaku
podatkovne tehnologije
poslovna inteligenca
umetna inteligenca in strojno učenje
obdelava velepodatkov in podatkovnih tokov
metode agilnega razvoja
tehnologije veriženja blokov
kibernetska varnost

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023

## GENERALNI POKROVITELJ

inova iT

## POKROVITELJI

bintegra
Svetovanje in IT rešitve

COMTRADE GAMING

GOSMIT

impol
Aluminium Industry

INEA

NFORMACIJSKE STORITVE IN INŽENIRING, D.O.O.
INFORMATIKA

MIKROpis®

.msg
PLAUT

nChain

niceHash

NOVUMRGI

OPERATO
member of ELES Group

RAZVOJNI CENTER
IKTS
ŽALEC

irC
RC IRC Celje, d.o.o. / Ltd.

Telekom Slovenije

tridens
technology

triglav

## MEDIJSKI POKROVITELJ

Računalniške novice
www.racunalniske-novice.com

**bintegra»**

Ponujamo odlično programsko opremo in IT rešitve.

**Kadarkoli, kjerkoli in na katerikoli napravi.**

## Napredne integracijske rešitve

Agilna integracija, ki združuje integracijske tehnologije, agilne tehnike razvoja in platforme v oblaku za izboljšanje hitrosti in varnosti razvoja programske opreme, je ključna podlaga za uspešno digitalno preobrazbo.
Zgradite, uporabite in podprite zunanje in notranje API-je za pospešitev razvoja novih storitev.

Pomagamo vam izbrati pravo tehnologijo, integrirati aplikacije, podatke in informacije za avtomatizacijo procesov in izboljšanje uporabniške izkušnje.

## Inovativni uporabniški portali

Skupaj z vami učinkovito rešujemo vaše izzive na naslednjih področjih:

**STRANKE »** Omogočite svojim strankam, da sami rešijo svoje težave brez pomoči zastopnika v živo. Ponudite jim inovativni samopostrežni portal. Predstavite ustreznejšo vsebino s personalizacijo za vsako skupino strank.

**PRIHODEK »** Povečajte spletni prihodek s trgovino B2B ali B2C. Ustvarite trajnostne konkurenčne prednosti in izkoristite nove priložnosti za prihodke.

**INTERNI PROCESI »** Posodobite svoj intranet in s tem povečajte produktivnost za zaposlene v pisarni, na daljavo, zmanjšajte preobremenitev z informacijami in preklapljanje konteksta, izboljšajte izkušnje zaposlenih,
ohranite institucionalno znanje in hitro vključite nove zaposlene.

## Bihub – 360° upravljanje komunikacije s strankami

Socialni mediji so eden najhitreje rastočih kanalov, ki jih potrošniki uporabljajo za interakcijo z blagovnimi znamkami.

Z rešitvijo **Bihub** se lahko pogovarjate s strankami preko e-pošte in družbenih medijev na enem mestu, lahko pa celotno komunikacijo dodelite tudi drugim kolegom ali ekipam.
Poenostavite lahko upravljanje družbenih medijev in ga uspešno integrirate v svoj CRM. Poskrbite, da ne boste zamudili nobenega vprašanja, potencialne stranke ali pritožbe strank.

www.bintegra.com info@bintegra.com

COMTRADE
GAMING

# About us

We are an independent software provider, delivering open gaming platforms and professional services to both the online and land-based gaming sectors with more than 15 years of experience.

Comtrade Gaming's strengths are the development of technology solutions based on industry standards and legislative requirements.

We are dedicated to fostering a supportive and inclusive workplace culture that values and respects the diverse backgrounds, perspectives, and talents of our employees, empowering them to contribute, collaborate, and grow professionally.

**iCore**

iGaming Platform

**sCore**

Gaming Management System

**gCore**

Game development and distribution platform

Visit our career page:

comtradegaming.com

Connected mHealth

connectedmhealth.com

# Virtualna rehabilitacija s pomočjo mHealth tehnologije

## Prihodnost rehabilitacije – fizioterapija pacientov na daljavo

Specialisti s področja fizikalne medicine in rehabilitacije so skupaj s strokovnjaki za informacijsko tehnologijo združili moči in ustvarili celovito orodje za izvajanje preventive in rehabilitacije na daljavo. Rešitev se aktivno uporabljaja v mnogih klinikah in rehabilitacijskih centrih po svetu.

2022
AWARDS

Nagrada na AACVPR (American Association of Cardiovascular and Pulmonary Rehabilitation)

# msg Plaut UAP

## Več kot 25 let zanesljiv in inovativen partner zavarovalnic

### msg.UAP - PRODUKT PRIHODNOSTI

- Platforma za celovito digitalno nadgradnjo upravljanja zavarovalnih portfeljev
- Enostranska spletna aplikacija
- Visoka stopnja prilagodljivosti
- 100% pripravljena na oblak
- Delovanje v realnem času
- Brez visokoletečih izrazov - zgolj tehnologija, ki deluje

### KDO SMO

msg life odateam je že več kot 25 let zanesljiv in inovativen partner zavarovalnic. Naše rešitve podpirajo vsako izmed naših strank z omogočanjem po meri prilagojene racionalizacije in avtomatizacije poslovnih procesov, fleksibilnega razvoja izdelkov in optimalne učinkovitosti. Kot del mednarodne organizacije msg Group se lahko ozremo na 40 let izkušenj na področju zavarovalniške industrije. Takšna dediščina s seboj prinaša razumevanje, da je napredek rezultat neprestanega premikanja mej inovativnosti.

### OBSEG STORITEV

- Razvoj celostnih informacijskih sistemov za zavarovalnice in njihove agente
- Razvoj mobilnih aplikacij
- Rešitve v oblačnih storitvah v mikrostoritveni arhitekturi z uporabo vsebnikov Docker
- Digitalizacija procesov
- Orodje in specializiranost za migracijo podatkov
- Celostne DevOps storitve

### NAŠA VIZIJA

Naš pristop predstavlja edinstvena kombinacija znanja na področju zavarovalništva in tehnološke ekspertize, ki temeljita na konceptu ustvarjanja okolja usmerjenega v uporabnika. Zavezani smo, da bomo v celoti razumeli procese vašega podjetja in potrebe zaposlenih. Naša agilna metodologija dela nam omogoča prilagajanje na vedno znova pojavljajoče se spremembe na tržišču in nove projektne situacije. Naš uspeh je posledica dragocenih partnerstev s poudarkom na plodnem in dolgoročnem sodelovanju.
Strankam prepuščamo svobodo odločanja pri tem, kar je najpomembnejše – njihova prihodnost.

### KONTAKT

www.msg-plaut-uap.si

Andrej Kline
Direktor msg Plaut UAP d.o.o.
andrej.kline@msg-life.com
+ 386 2 2356200


ENTERPRISE JAVA
APPLICATION DEVELOPMENT


AGILE SOFTWARE
DEVELOPMENT


INTERNATIONAL
PROJECTS


IT
SOLUTIONS
FOR
INSURANCE
BUSINESS


CLOUD SERVICES
FOR INSURERS


Programske rešitve
Razvoj programske opreme
Poslovna informatika
Vzdrževanje in svetovanje
Informacijska varnost


360°
5G
IOT
CO2
P

tridens
technology

Tridens je mariborsko IT podjetje, ki je v digitalnem prostoru prisotno že več kot 15 let.

V podjetju se ukvarjamo z razvojom lastnih produktov:

Distance (less than 1 km away)
The British Museum

Location
Great Russell St,
London WC1B 3DG, UK

Naša vizija je biti prepoznani kot zaupanja vreden partner za inovativne programske rešitve po celem svetu.

## Zaupajo nam:

ABB zeev DOW JONES cisco telemach

## Zaposlujemo!

Poglej prosta delovna mesta na naši spletni strani:
https://tridenstechnology.com/careers/
ali poskeniraj QR kodo.

Zavarovanje
kibernetske
zaščite
za podjetja.

# Poskrbite za varnost na spletu pri vašem poslovanju.

**triglav**

Vse bo v redu.
triglav.si

## GENERALNI POKROVITELJ

inova iT

## POKROVITELJI

bintegra
Svetovanje in IT rešitve

COMTRADE GAMING

GOSMIT

impol
Aluminium Industry

INEA

INFORMACIJSKE STORITVE IN INŽENIRING, D.O.O.
INFORMATIKA

MIKROpis

msg
PLAUT

nChain

niceHASH

NOVUMRGI

OPERATO
member of ELES Group

RAZVOJNI CENTER
IKTS
ŽALEC

irC
RC IRC Celje, d.o.o. / Ltd.

Telekom Slovenije

tridens
technology

triglav

## MEDIJSKI POKROVITELJ

Računalniške novice
www.racunalniske-novice.com

INŠTITUT ZA INFORMATIKO

Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko

IoT

Varnost

Kriptografija

Web 3

Podatkovne tehnologije

Umetna inteligenca

Poslovna analitika