Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jernej Černelč

Integracija odkrivanja storitev v sisteme upravljanja API-jev

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: prof. dr. Matjaž Branko Jurič

Ljubljana, 2018

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Analizirajte in opišite sisteme za upravljanje API-jev, arhitekturo tipičnih sistemov in načine registracije storitev. Proučite koncept API prehodov in opišite postopke preusmerjanja zahtevkov in druge koncepte delovanja, pri čemer razložite razliko med klasičnim in vsebniškim okoljem. Implementirajte praktični primer, v katerem prikažite potrebne korake za integracijo mehanizma odkrivanja storitev v obstoječ API prehod. Opišite postopek razvoja in evalvirajte rešitev.

IZJAVA O AVTORSTVU ZAKLJUČNEGA DELA

Spodaj podpisani/-a _____, vpisna številka _____, avtor/-ica pisnega zaključnega dela študija z naslovom:

_____

_____

IZJAVLJAM

1. da sem pisno zaključno delo študija izdelal samostojno pod mentorstvom _____ in somentorstvom _____;
2. da je tiskana oblika pisnega zaključnega dela študija istovetna elektronski obliki pisnega zaključnega dela študija;
3. da sem pridobil/-a vsa potrebna dovoljenja za uporabo podatkov in avtorskih del v pisnem zaključnem delu študija in jih v pisnem zaključnem delu študija jasno označil/-a;
4. da sem pri pripravi pisnega zaključnega dela študija ravnal/-a v skladu z etičnimi načeli in, kjer je to potrebno, za raziskavo pridobil/-a soglasje etične komisije;
5. soglašam, da se elektronska oblika pisnega zaključnega dela študija uporabi za preverjanje podobnosti vsebine z drugimi deli s programsko opremo za preverjanje podobnosti vsebine, ki je povezana s študijskim informacijskim sistemom članice;
6. da na UL neodplačno, neizključno, prostorsko in časovno neomejeno prenašam pravico shranitve avtorskega dela v elektronski obliki, pravico reproduciranja ter pravico dajanja pisnega zaključnega dela študija na voljo javnosti na svetovnem spletu preko Repozitorija UL;
7. dovoljujem objavo svojih osebnih podatkov, ki so navedeni v pisnem zaključnem delu študija in tej izjavi, skupaj z objavo pisnega zaključnega dela študija.

V/Na: _____
Datum: _____

Podpis študenta/-ke:

_____

Zahvaljujem se svojemu mentorju prof. dr. Matjažu Branku Juriču in as. Antonu Zvonku Gazvodi za strokovno in praktično svetovanje ob pisanju diplomske naloge. Še posebej bi se zahvalil mami, očetu, bratoma, prijateljem in Katarini za stalno podporo in vsem ostalim, ki ste mi vsa leta pomagali čez študij.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
API	Application Programming Interface	Aplikacijski programski vmesnik
DoS	Denial of Service	Zavrnitev storitve
GUI	Graphical User Interface	Grafični uporabniški vmesnik
REST	Representational State Transfer	Reprezentativni prenos stanja
SOAP	Simple Object Access Protocol	Protokol za preprost objektni dostop
XML	Extensible Markup Language	Razširljiv označevalni jezik
JSON	JavaScript Object Notation	JavaScript objektni zapis
YAML	Yet Another Markup Language	Še en označevalni jezik
SLA	Service Level Agreement	Sporazum o zagotavljanju storitev
IP	Internet Protocol	Internetni protokol
HTTP	HyperText Transfer Protocol	Protokol za prenos hiperteksta
NAT	Network Address Translation	Translacija omrežnih naslovov
CDI	Context Dependency Injection	Vstavljanje kontekstne odvisnosti

Povzetek

Naslov: Integracija odkrivanja storitev v sisteme upravljanja API-jev

Avtor: Jernej Černelč

Porast števila API-jev v sodobnem svetu je danes nezanemarljiv. Monolitne aplikacije so začele prevzemati arhitekturo mikrostoritev, kjer se je število neupravljanih API-jev zaradi medprocesne komunikacije še povečalo. V diplomskem delu so predstavljene rešitve za upravljanje API-jev mikrostoritev, tehnologija odkrivanja storitev in njena integracija v sisteme upravljanja API-jev. V nalogi predstavimo, kaj ti sistemi so, kdo jih potrebuje in zakaj se uporabljajo. Izpostavljene so ključne komponente in njihova nadgradnja z odkrivanjem storitev.

Ključne besede: mikrostoritve, API, API prehod, upravljanje APIj-ev, tehnologija odkrivanja storitev.

Abstract

Title: Integration of service discovery into API management systems

Author: Jernej Černelč

The growing number of APIs in the modern world is nowadays non-negligible. Transformation from legacy monolithic applications into microservice architecture (MSA) led to even bigger number of unmanaged APIs caused by interprocess communication. This thesis presents solutions to API management for microservices, the technology of service discovery and its integration into API management systems. We present what these systems are, who needs them and what are they used for, while exposing their core components and their service discovery upgrades.

Keywords: microservices, API, API Gateway, API management, service discovery.

Poglavje 1

Uvod

Spletni API-ji so v zadnjih nekaj letih postali izredno popularni, saj sodobne aplikacije težijo k širši medsebojni povezanosti in izpostavljenosti svetovnemu spletu. Prav tako drobljenje aplikacij v t.i. mikrostoritve povzroča še večje število posameznih procesov, ki svojo logiko ponavadi izpostavljajo preko API-jev. Porast števila teh je povzročila razvoj sistemov za upravljanje API-jev, ki rešujejo probleme analize, nadzora, varnosti in monetizacije API-jev na enem mestu.

Takšen sistem mora imeti za lažjo skalabilnost in odpornost proti odpovedi, šibko sklopljene komponente. Pri njegovem načrtovanju je zato ključnega pomena zasnova dobre arhitekture in načinov komunikacije med posameznimi komponentami, da te ne pripomorejo k dodatni, nepotrebno veliki obremenitvi sistema. Namen diplomske naloge je dati bralcu dovolj znanja za razumevanje delovanja takšnih sistemov, ter pojasniti, kako jih načrtovati in v njih vpeljevati nove funkcionalnosti z zavedanjem tako pozitivnih kot negativnih učinkov, ki jih bodo imele na celotni sistem. Končni cilj naloge predstavlja integracija tehnologije odkrivanja storitev v obstoječ sistem upravljanja API-jev in vzpostavitev testnega okolja razvitih komponent s pomočjo tehnologije vsebnikov ter na novem sistemu izvesti časovne analize.

Diplomsko delo lahko razdelimo na dva dela. V prvem so predstavljeni

glavni koncepti delovanja sistemov upravljanja API-jev, API prehodov in integracije tehnologije odkrivanja storitev v taškne sisteme. V drugem delu pa so ti koncepti prikazani na praktičnem primeru, kjer smo jih nekaj realizirali in na njem opravili časovno analizo.

Poglavje 2

Upravljanje API-jev

Potreba po upravljanju API-jev je v modernih arhitekturah računalniških sistemov vedno močnejša. V sledečem poglavju opisujemo te sisteme, osredotočimo pa se predvsem na obvladovanje in medsebojno komunikacijo s pomočjo upravljanja API-jev.

2.1 Upravljani elementi

Zasnovati poskušamo sistem, ki bo imel pregled nad čim večjo množico API-jev, ki jih implementirajo monolitne aplikacije ali posamezne komponente (mikrostoritve). Osnovna enota v našem sistemu je torej API, ki ga lahko izpostavlja ena ali več instanc storitve. Podpreti želimo sledenje njegovemu celotnemu življenjskemu ciklu (angl. lifecycle), tako oskrbi kot porabi. Na strani oskrbe lastnik API-ja skrbi za procese definiranja, oblikovanja, kreiranja, nameščanja, nadgrajevanja, varnosti in optimizacije. Ta cikel se prepleta s postopkom porabljanja API-ja, kjer razvijalci pridobivajo API-je in oblikujejo, gradijo ter zaganjajo svoje aplikacije [31].

2.1.1 Monolitne aplikacije

Monolitne aplikacije so zasnovane predvsem za zunanjo komunikacijo, kjer vsa poslovna logika ponavadi poteka v enem samem procesu. Kadar pride do

preobremenitve enega dela, moramo skalirati celotno aplikacijo in pred njo postaviti namestniški strežnik (angl. proxy server), ki izravnava obremenitve (angl. load balancer). To pripelje do neunčikovite porabe virov, saj skaliramo tudi nepotrebne dele aplikacije.

2.1.2 Mikrostoritve

Tehnologija mikrostoritev (angl. microservices) se je pojavila kot nasprotje monolitnim arhitekturam. Aplikacijo sestavljajo šibko sklopljene, logično ločene komponente, ki se ponavadi izpostavljajo kot posamezne storitve. Ker je za medsebojno komunikacijo lahko definiranih veliko API-jev, kaj kmalu naletimo na zmešnjavo in pojavi se potreba po skupni točki njihovega upravljanja. Takšna arhitektura je odpornejša pred odpovedjo celotne aplikacije, saj so posamezne komponente ločeni procesi, ki so lahko skalirani v več instanc. Pojavi se tudi problem sledenja omrežnim naslovom, saj se lahko v skalabilnih okoljih stalno spreminjajo. Mikrostoritve tako poznajo le logična imena drugih storitev, za pravilno usmerjanje, iskanje instanc in v primeru več instanc, izravnavo obremenitve (angl. load balancing), pa poskrbi sistem odkrivanja storitev oziroma v našem primeru, upravljanja API-jev. Razvijalcem in porabnikom API-ja sta tako prihranjena vpogled v omrežni del sistema in možnost odkrivanja drugih mikrostoritev. Posledično se lahko osredotočijo na sam razvoj poslovne logike.

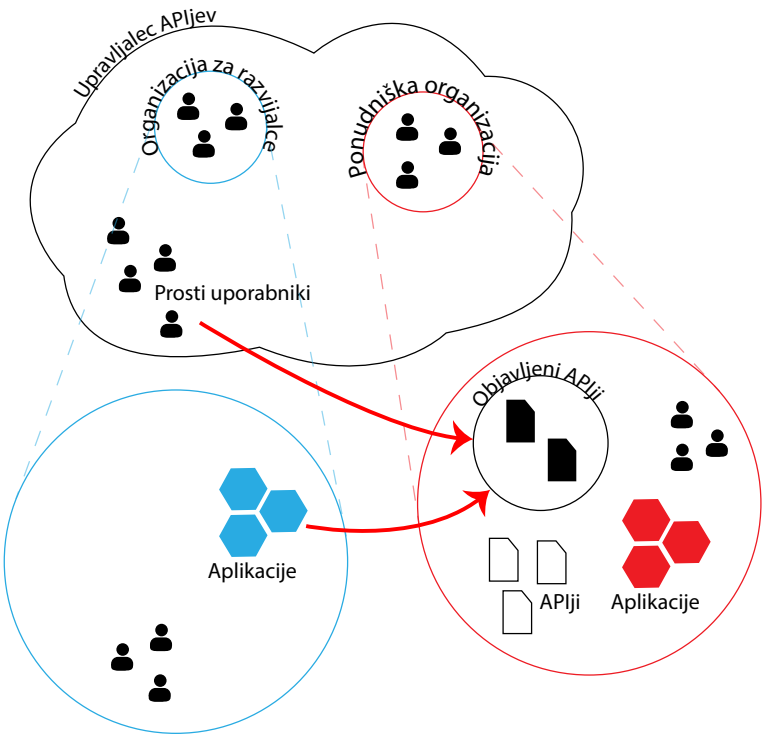
2.2 Arhitektura sistema

Implemetacija sistema upravljanja API-jev potrebuje vsaj dve ključni komponenti: upravljalca API-jev in API prehod. S procesom upravljanja API-jev se med njima stalno izmenjujejo podatki, ki skrbijo za konsistenco in sinhronizacijo komponent. Instance upravljalca API-jev in API prehoda lahko ponavadi poljubno skaliramo po več strežnikih v t.i. gručo (angl. cluster) in vpeljemo izravnavo obremenitve, kar pripomore k večji učinkovitosti in prepustnosti omrežja [32]. Komponenti imata v sistemu upravljanja API-jev

dodeljene določene vloge.

- **Upravljalniki API-jev** ponavadi ponujajo prijazen uporabniški grafični vmesnik (angl. GUI), ki omogoča preglednejše in hitrejše upravljanje API-jev. Tukaj se izvaja vsa poslovna logika, ki je povezana z vzdrževanjem, dokumentacijo in spremljanjem API-jev. Prav tako se hranijo podatki o uporabnikih, aplikacijah in organizacijah sistema.
- **API prehodi** skrbijo za uspešno posredovanje zahtevkov, namenjenih objavljenim API-jem, avtentikacijo in zbiranje podatkov za nadzor sistema. Natančneje jih opišemo v 3. poglavju.

2.2.1 Hierarhija uporabnikov sistema



Slika 2.1: Hierarhija sistema upravljanja API-jev.

S postavljeno infrastrukturo lahko uporabniki registrirajo in kličejo API-je. Uporabniki pripadajo bodisi eni ali več organizacijam, kjer skupno ali posamično upravljajo z API-ji ali aplikacijami, ki pripadajo organizacijam, kot je prikazano na sliki 2.1. Znotraj sistema upravljanja API-jev poznamo dve vrsti organizacij [32].

- **Ponudniške organizacije** so lastnice lastnih API-jev in koordinatorji njihovih načinov uporabe.
- **Organizacije za razvijalce** so samo lastnice aplikacij, ki uporabljajo izpostavljene API-je ponudniških organizacij.

2.3 Dokumentacija API-jev

API-ji so dobri le toliko, kot je dobra njihova dokumentacija. Ker se lahko vsak API razlikuje od drugih po načinu izpostavljanja končnih točk, je včasih težko intuitivno sklepati le iz njihove strukture, kaj te počnejo. Tako je danes dokumentacija nujna pri vsakemu API-ju. Dokumentacija lahko zajema primere uporabe, interaktivne vmesnike, kjer lahko klic na končne točke takoj izvedemo brez lastne kode ali drugih programov in jih preizkušamo z različnimi parametri. Ponavadi so prikazane tudi vse možne napake in struktura vrnjenih objektov [19].

2.4 Dodana vrednost sistemov upravljanja API-jev

API-ji imajo predpisane strukture in protokole, ki jih njihovi uporabniki morajo upoštevati. Najpogosteje uporabljena komunikacijska protokola sta REST in SOAP. Ti ponavadi sprejemajo in oddajajo strukture v obliki označevalnih jezikov (angl. markup languages), kot so XML, JSON in YAML [6]. Izpostaviti je potrebno tudi končne točke, ki jih API ponuja, ter jih dobro dokumentirati. Te lastnosti so ponavadi že sestavni del API-ja, potrebno

jih je samo še prenesti v upravljalni sistem. Doana vrednost v sistemu upravljanja API-jev so nadzorovanje, načini uporabe, objavljanje, omejitve in prehajanje zahtevkov.

2.4.1 Nadzorovanje

Ključna naloga sistemov upravljanja API-jev je spremljanje (angl. monitoring), ki nam omogoča tesno sledenje uporabniških izkušenj in učinkovitosti API-jev. Na podlagi zbranih metrik lahko ugotovimo, kateri API-ji se najpogosteje uporabljajo, in jih posledično skaliramo ter jim priredimo omejitve in načine uporabe. Prav tako lažje izsledimo napake in njihove izvore. Na API-jih z drugačnimi karakteristikami lahko spremljamo različne metrike, ki nam pokažejo učinkovitost delovanja posameznih API-jev. Te uporabniško definirane metrike imenujemo ključni kazalniki uspešnosti (angl. Key Performance Indicator – KPI) [12]. Orodje za vizualizacijo metrik ponavadi izpostavlja že GUI upravljalca API-jev, ki nam omogoča hiter pregled statističnih podatkov na istem sistemu. Nekaj pogosto spremljanih, osnovnih karakteristik je:

- število zahtevkov na časovno obdobje,
- povprečni odzivni čas,
- uspešnost zahtevkov,
- kršenje omejitev.

Manipulacija in zbiranje statističnih podatkov sta danes močno zaželeni orodji, ki organizacijam omogočata stalno izboljšavanje lastnih sistemov in rešitev na podlagi pridobljenih podatkov [4].

2.4.2 Načini uporabe

Načini uporabe (angl. plan) so dogovori med lastniško organizacijo in odjemalci API-ja. V njih so predpisane cene za dostop do določenega vira, ki se

lahko vežejo le na določene dostopne točke ali celotno storitev. Odjemalcu je preko sklenjene pogodbe (SLA) ponavadi zagotovljen konstanten in nemoten dostop do API-ja [12]. Če nad API-jem ni definiranega nobenega načina uporabe, je javno dostopen vsakemu, ki pozna objavljeni omrežni naslov končnih točk prehoda.

2.4.3 Objavljanje

API postane dostopen ciljnim uporabnikom šele z objavo na prehod. Ker je teh lahko več, ga ponavadi določimo in tako dobimo prirejene končne točke, preko katerih se pošiljajo zahtevki na objavljeni API. Podrobneje jih opišemo v 3. poglavju. Kadar želimo preprečiti dostop do API-ja, ga odjavimo.

2.4.4 Omejitve

Na objavljeni API se lahko postavijo določene omejitve, ki dodatno dirigirajo dostop do izbranega API-ja. Te so lahko avtentikacijske narave ali pa gre za omejevanje naslovov IP z belo in črno listo (angl. blacklist, whitelist policy), kjer lahko določamo iz katerih naslovov je oziroma ni možno dostopati do API-ja [33]. Časovno in količinsko omejevanje nam omogočata, da lažje nadzorujemo promet skozi naše omrežje in ga po želji prilagajamo. Izbrane omejitve se upoštevajo po objavi, ponavadi na prehodu, kamor se pošiljajo zahtevki.

2.4.5 Prehajanje zahtevkov

Po objavi API-ja na API prehod pridobimo prirejene končne točke, ki potekajo preko skupne enote, prehoda. Nad njimi se izvaja celoten nadzor in omejitve, definirane v upravljalniku API-jev in preslikovanje navideznih omrežnih naslovov prehoda v prave naslove končnih točk API-jev. Zahtevki se nato posredujejo preko prehoda nazaj odjemalcem, kjer se ponavadi beležijo metrike, ki nam pokažejo uporabne statistike API-ja.

2.5 Varnost

Vsak resnejši spletni sistem ima danes dobro zasnovano varnost. Čeprav jo večinoma ponujajo že komunikacijski protokoli, je dobra praksa implementirati še dodatna varnostna vrata, ki ustvarijo večplastno zaščito pred potencialnimi napadalci. Nadzor nad dostopanjem do izpostavljenih API-jev na podlagi avtentikacije in zaščita pred napadi sta bistveni funkcionalnosti, ki ju moramo implementirati za zagotavljanje varnosti. Omenjeni funkcionalnosti na kratko predstavimo v nadaljevanju.

2.5.1 Avtentikacija

API-ji so lahko javno dostopni ali zaklenjeni za naročnino, ki omejuje dostop do API-ja. Do javno dostopnih API-jev lahko dostopa vsakdo brez avtentikacijskega ključa, do zaklenjenih pa le tisti, ki jim je to odobril vzdrževalec ali lastnik API-ja. Odjemalec je lahko uporabnik ali aplikacija. Ta ob vsakem zahtevku poda svoj identifikacijski ključ preko katerega se identificira in prikaže svojo pravico oziroma nepravico za dostop do željenega API-ja. Ključi so zaradi dodatne varnosti in možnosti kraje ponavadi časovno omejeni, kjer rok in obnavljanje veljavnosti ključev največkrat specificirajo uporabljeni varnostni protokoli [3]. Kljub temu, je sistem varen le dokler uporabniki oziroma aplikacije varno skrivajo svoja uporabniška imena in gesla.

2.5.2 Napadi

Kraja ključev in posledično podatkov je usmerjena na posameznika ali posamezen API. Napad z zavrnitvijo storitve (angl. DoS attack) pa na celotni sistem, v našem primeru, prehod. Cilj DoS napadov je začasno preobremeniti sistem in onemogočiti dostop drugim. Napadalci to dosežejo z generiranjem ogromnega števila zahtevkov, ki jih pošiljajo na naš prehod in zapolnijo čakalne vrste. Čeprav se ti napadi ponavadi zaznajo in preprečijo že pri ponudnikih internetnih storitev, poznamo nekaj obrambnih mehanizmov, ki nikoli v celoti ne preprečijo vpliva napada [3]. Zato imamo pri upravljanju API-jev

kot vzdrževalec API-ja možnost omejevanja količine prometa skozi časovne enote, ki ga lahko uporabimo na posameznem uporabniku, aplikaciji, API-ju ali končni točki API-ja [12].

Poglavje 3

API prehodi

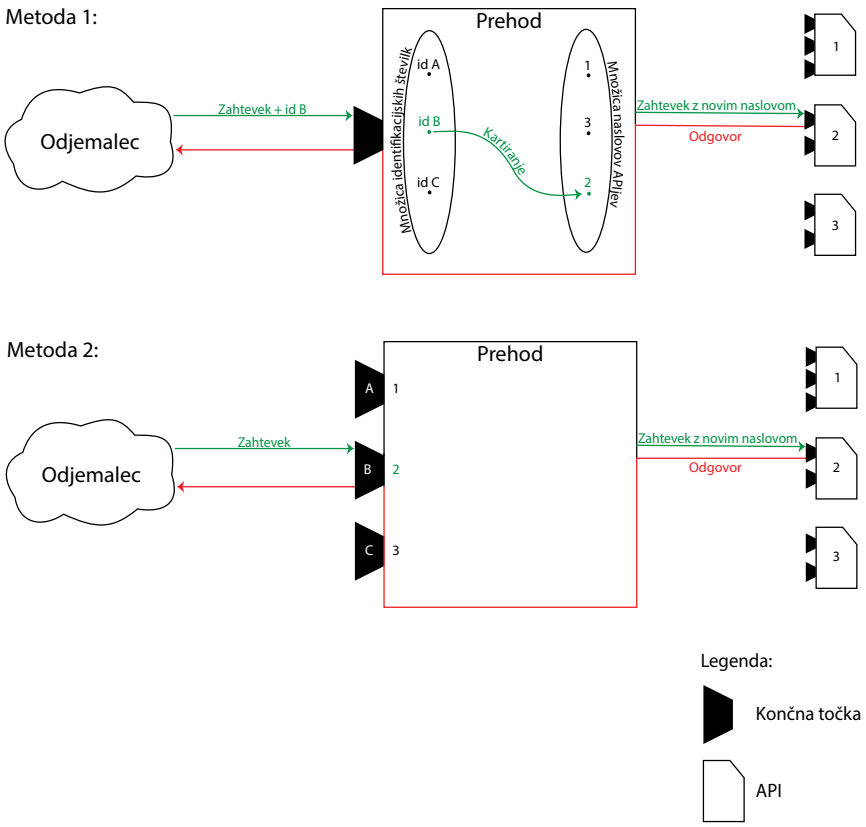
Vsak sistem upravljanja API-jev potrebuje vsaj eno instanco API prehoda. Skozi prehod poteka vsa komunikacija z zalednim sistemom strežnikov, kjer se nahajajo vsi API-ji in se sproti izvaja vsa poslovna logika, ki smo jo definirali v prejšnjem poglavju. V nadaljevanju predstavimo koncepte preslikovanja omrežnih naslovov, postavitev prehoda v sistem, njegove pridobitve in slabosti.

3.1 Postopek preusmerjanja zahtevkov

Glavna naloga API prehodov je ustrezno preusmerjanje zahtevkov od odjemalca k API-ju oziroma končni točki in nazaj. To lahko dosežemo na mnogo načinov, nekateri so bolj preprosti in počasnejši, nekateri so hitrejši in zahtevnejši za implementacijo. Naloga razvijalca pri zasnovi prehoda je izbrati najprimernejšega za našo rešitev. Dva preprostejša sta prikazana na sliki 3.1 [12].

- **Prvi način**, kjer vsakemu API-ju določimo unikatno identifikacijsko številko, ki jo v vsakem zahtevku namenjenemu prehodu dodamo v glavo, v kolikor se uporablja za prenos protokol HTTP [5]. To prehod prebere in preko kartiranja (angl. mapping) pridobi omrežni naslov API-ja za posredovanje zahtevka.

- **Drugi način**, kjer prehod za vsak zaledni registrirani API izpostavlja svojo končno točko z unikatno potjo. Ker so končne točke prehoda enolično povezane z omrežnimi naslovi API-jev, se zahtevki posredujejo na pravilni naslov.



Slika 3.1: Metodi posredovanja zahtevkov.

3.1.1 Pot zahtevka skozi prehod

Pot zahtevka se vedno začne pri odjemalcu, ki je lahko posamezen uporabnik ali aplikacija. Ta mora poznati način metode dostopa preko prehoda do

končne točke API-ja, ki smo jih opisali v podpoglavju 3.1, za pravilno generiranje omrežnega naslova zahtevka. Glede na izvazvajanje poslovne logike na prehodu lahko proces preusmerjanja razdelimo na sedem delov [22].

1. Sprejem zahtevka na končni točki prehoda.
2. Avtentikacija in preverjanje dovoljenj odjemalčevega dostopa.
3. Preverjanje načinov uporabe in omejitev API-ja.
4. Zabeleženje metrik zahtevka.
5. Pridobitev pravilnega omrežnega naslova gostitelja API-ja in oblikovanje nove poti.
6. Posredovanje zahtevka na nov omrežni naslov.
7. Sprejem odgovora in njegovo posredovanje nazaj odjemalcu.

Če kateri izmed korakov spodleti tehnično ali ne zadošča pogojem, se izvajanje v trenutku prekine in odjemalcu vrne ustrezno informativno opozorilo o napaki.

3.2 Vpliv API prehoda na sistem

Ali naš sistem potrebuje skupno točko API prehoda, je vprašanje, ki nima enoličnega odgovora. Odvisen je od scenarija in infrastrukture sistema. V tem podpoglavju opišemo nekaj prednosti in slabosti, ki jih uvedba tašnega prehoda prinese v naš sistem [11].

3.2.1 Prednosti

Glavna prednost je centraliziran nadzor nad avtentikacijo, dostopi in monetizacijo celotnega sistema, saj je prehod ločena enota, neodvisna od ostalih

komponent sistema. Kot prednost lahko izpostavimo tudi prosojnost zalednega sistema API-jev, saj prehod deluje kot namestniški strežnik, kjer odjemalec neposredno ne vidi zahtevanih API-jev, ki jih tako delno zakrijemo in zavarujemo pred zunanjim omrežjem. Podobno delujejo omrežja NAT [7], s poznavanjem katerih si lahko pomagamo pri zasnovi prehoda. Izpostavljenost API-jev zunanjemu omrežju posledično ni potrebna, kar pripomore k lažji in preprostejši konfiguraciji ter varnosti notranjega omrežja. Ker imamo skupno točko, lahko s kompozicijo API-jev učinkovito poenotimo nekatere klice. Namesto, da odjemalec naredi mnogo poizvedovanj na več končnih točk, mu API prehod pripravi eno kočno točko, ki združuje in njemu skrito naredi v ozadju več klicev za pridobitev vseh potrebnih podatkov [12].

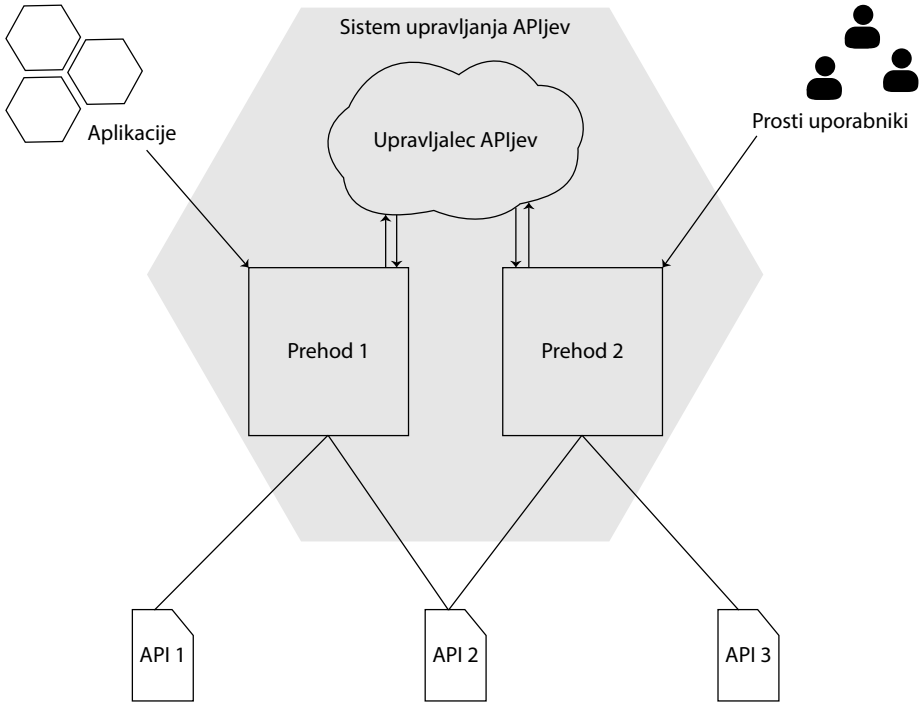
3.2.2 Slabosti

Kot centralna točka skozi katero poteka vsa komunikacija je prehod ozko grlo omrežja in kandidat za enotno točko odpovedi (angl. Single Point Of Failure – SPOF) [17]. Njegova odpoved je lahko kritična za delovanje sistema, kar ponavadi rešujemo s tehnologijo gručenja (angl. clustering), kjer na drugih strežnikih tečejo duplikati, ki ob primeru odpovedi prevzamejo njegovo vlogo. Z vpeljavo nove komponente porabimo dodaten čas za njen razvoj, postavitev v sistem in vzdrževanje. Prehod podaljša odzivni čas zahtevka od odjemalca proti API-ju in nazaj, kjer pazimo, da je ta čim manjši oziroma skoraj neopazen. Posledično se na poti zahtevka pojavi tudi dodaten skok (angl. hop).

3.3 Postavitev API prehoda v sistem

API prehod je, zaradi svoje vloge vstopne točke v sistemu, postavljen med izpostavljene API-je in odjemalce, kjer vsa vmesna komunikacija poteka preko prehoda samega, kot je prikazano na sliki 3.2. API je lahko povezan na več prehodov in na prehod je lahko vezanih več API-jev [12]. Ker je med upravljalecem in prehodi izmenjana velika količina podatkov, je priporočljivo, da

se nahajajo v neposredni bližini oziroma istem omrežju.



Slika 3.2: Arhitektura sistema upravljanja API-jev.

3.3.1 Uporaba API prehoda v arhitekturi mikrostoritev

Danes so razširjeni sistemi, ki se v celoti izvajajo v oblaku (angl. cloud), kjer lahko aplikacije poljubno skaliramo in jih naredimo relativno hitro dostopne po celotnem svetu. Vodilni ponudniki računalništva v oblaku (angl. cloud computing vendors) so Amazon, Google, HP, IBM in Microsoft [1]. V članku [10] iz leta 2013 je o storitvah v oblaku izpostavljen problem nekompatibilnosti in težkega povezovanja storitev, gostujočih na različnih oblačnih platformah. Za obvladovanje in upravljanje teh, je tako priročna uporaba API prehodov, ki po spletu razpršene storitve združuje preko enotnih točk.

Aplikacije v teh sistemih danes večinoma sledijo mikrostoritveni arhitekturi, kjer jih sestavljajo ena ali več mikrostoritev, kar posledično za medsebojno komunikacijo povzroči generiranje večje količine prometa. Ker se ta skozi čas neprestano spreminja, je zaželjena tudi drugačna zasnova API prehodov kot pri upravljanju API-jev monolitnih aplikacij. Za večjo učinkovitost in lažje sobivanje v istem okolju je priporočeno zasnovanje prehoda po arhitekturi mikrostoritev, saj lahko posamezne logične funkcinalnosti prehoda tako po potrebi skaliramo.

V sistemu mikrostoritev poznamo tri načine postavitve in delovanja API prehodov [20], ki so prikazani na sliki 3.3.

- **Centralizirani API prehodi** (angl. centralized API gateways) so skupna vstopna točka za notranje in zunanje API zahtevke. Takšno gručo prehodov lahko prosto horizontalno skaliramo, kar pomeni dodajanje novih instanc in nad njimi izvajamo za vse zahtevke izravnavanje obremenitve. Ker je takšna postavitev intuitivna (podobna monolitni arhitekturi) in enostavna za skaliranje, je tudi najbolje zastopana, vendar se zaradi mešanja notranjih in zunanjih klicev zmanjšata nadzor in varnost sistema.

- **Lastni API prehodi** (angl. private jet API gateways) so, kot nakazuje njihovo ime, lastni vsaki gruči instanc iste mikrostoritve. Posledično imamo veliko večji nadzor in varnost nad celotnim sistemom, kjer lahko vsak lastni API prehod skaliramo neodvisno od ostalih. Pridobimo možnost alociranja virov za posamezno mikrostoritev, kar pri centraliziranih API prehodih ni bilo možno, saj so se vse mikrostoritve sklicevale na isto gručo prehodov. Prav tako ločimo notranje in zunanje klice, vendar decentralizacija povzroči manjšo preglednost in težje obvladovanje odpovedi posameznih prehodov.

- **Priključni API prehodi** (angl. sidecar API gateways) se uporabljajo, kadar želimo imeti arhitekturo prepletanja storitev (angl. service-mesh). Prehod se v tem primeru priključi mikrostoritvi, ki lahko nato koristi

vse funkcinalnosti prehoda. Takšen pristop nam onemogoči neodvisno skaliranje prehoda od pripadajoče mikrostoritve, vendar se posledično izognemo dodatnemu skoku zahtevkov v zunanjem omrežju, ki se pojavi v zgoraj opisanih načinih postavitve API prehodov. V primeru več instanc je potrebna vpeljava vmesne enote, ki opravlja izravnavo obremenitve.

3.4 API prehod v okolju Kubernetes

Kubernetes je odprtokodno orodje za orkestracijo vsebnikov [35]. Omogoča preprosto in avtomatizirano zaganjanje, diagnostiko, skaliranje in upravljanje aplikacij v vsebnikih (angl. containers) na enem ali več gostiteljskih strežnikih. Na najnižjem nivoju imamo stroke (angl. pods). Ti enkapsulirajo enega oziroma mnogo vsebnikov, ki ponavadi predstavljajo neko skupno funkcionalnost. Vsak instanciran strok ima svoj lastni nabor vrat in naslov IP, kjer si ga notranji vsebniki delijo in drug drugega naslavljajo preko lokalnega gostitelja (angl. localhost). Naslavljanje drugih entitet poteka preko koordinacije skupnih mrežnih virov. Stroki se ob procesu skaliranja stalno zaganjajo in ugašajo, z njimi pa se spreminjajo tudi naslovi IP, ki se morda potrebujejo za klice v drugih strokih. Zato imamo stopnjo višje definirane storitve (angl. services) potrebne za grupiranje strokov v logične množice. Vsebovani stroki so ponavadi določeni preko lastnih izbirnih značk (angl. label selector), ki identificirajo instance repliciranih strokov. Ti so nato dosegljivi preko izpostavljenega stalnega naslova storitve, kjer se z izravnavo obremenitve med razpoložljivimi instancami izbere omrežni naslov stroka za posredovanje zahtevka. Kubernetes je idealno okolje za postavitev in skaliranje mikrostoritev API prehoda.

3.4.1 API prehod kot mikrostoritev

Glede na funkcinalnost lahko API prehod razdelimo na spodnje mikrostoritve [20].

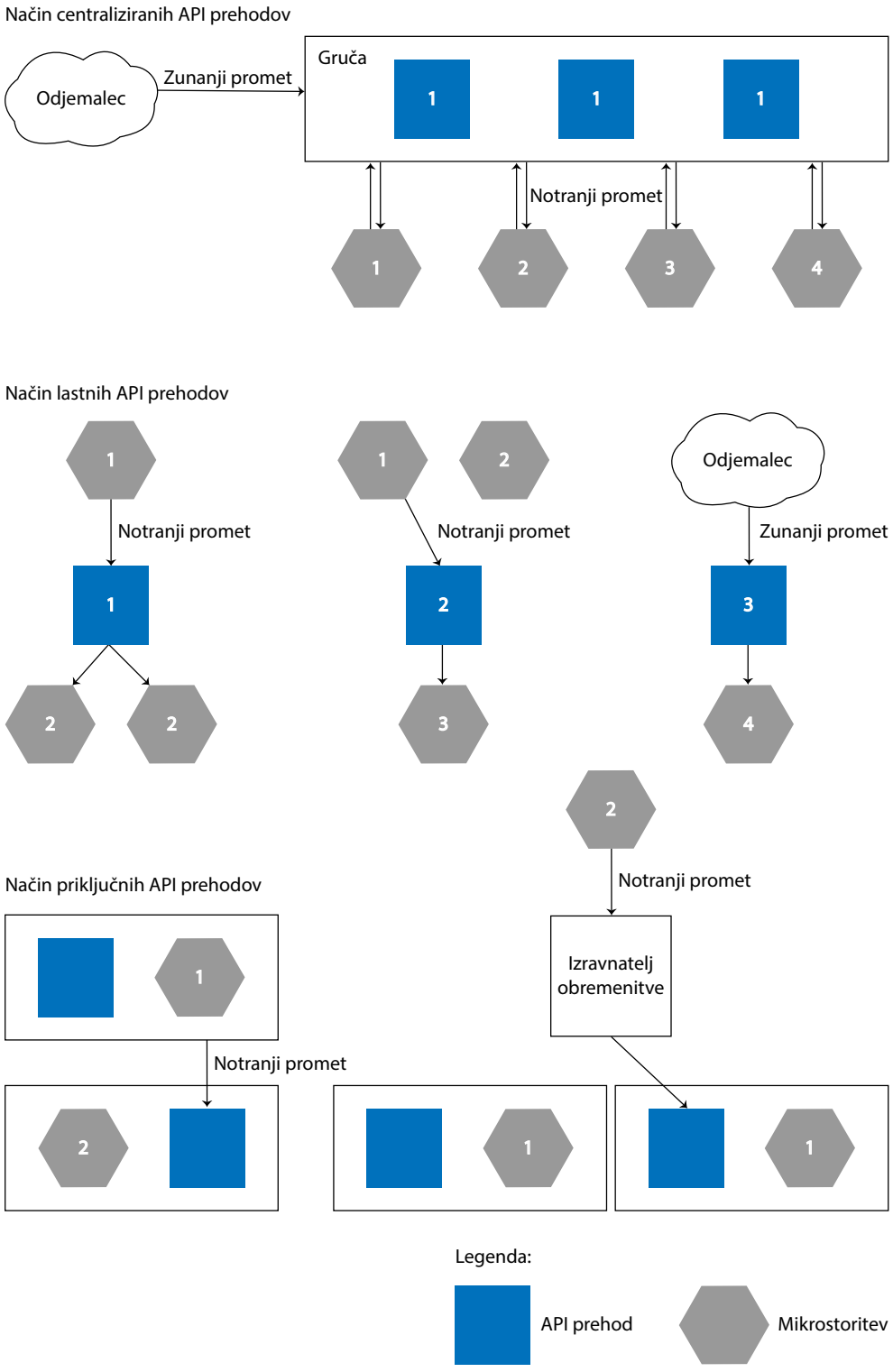
1. Dinamično odkrivanje omrežnih naslovov API-ja, transformacija in usmerjanje zahtevkov (nano API prehod)
2. Avtentikacija in varnost API-jev
3. Omejevanje pretoka do API-jev
4. Spremljanje in nadzor API-jev
5. Izravnavanje obremenitve
6. Predpomnjenje API zahtevkov
7. Monetizacija dostopa do API-jev
8. Kompozicija API-jev

Pri takšni porazdelitvi je za uspešno delovanje prehoda potrebna le prva mikrostoritev, ki se enkapsulira v strok in izpostavi zunanjemu svetu preko storitve. Za vsakega od preostalih se definira nov strok in se za medsebojno vidljivost preko storitev izpostavi znotraj okolja Kubernetes. Preko reprodukcijskega nadzornika (enota orodja Kubernetes, ki skrbi za porajanje in ugašanje strokov) se lahko nato z enim ukazom enostavno prilagaja število instanc posamezne mikrostoritve prehoda. Tako lahko postavimo visoko dostopen in skalabilen API prehod, ki ga je možno poljubno prilagajati glede na količino prometa. Tako optimalno izkoristimo računalniško moč in hkrati odjemalcem sistema ponudimo nemoteni dostop do API-jev s konstantno hitrostjo.

3.4.2 Nano API prehod

V razvojnih in testnih okoljih ali manjših, manj kompleksnih sistemih si ponavadi ne želimo integracije celotnega sitema upravljanja API-jev z vsemi funkcionalnostmi nadzorovanja, varnosti, omejevanja prepustnosti in še ostalih vidikov. Takrat se poslužimo t. i. nano oziroma lahkih (angl. lightweight) API prehodov [8]. Takšni prehodi se pogosto uporabljajo v komunikaciji med

mikrostoritvami, kar smo opisali v razdelku 3.3.1, ki oblikujejo neko celoto, kjer vsaki od njih zaupamo in poznamo količino generiranega prometa. Ker je njihov cilj le uspešno in čim hirejše posredovanje potrebnih informacij, ponavadi vsebujejo le logiko, potrebno za preslikovanje zahtevkov in odkrivanje omrežnih naslovov objavljenih storitev.



Slika 3.3: Načini postavitve API prehoda v sistem mikrostoritev.

Poglavje 4

Razširitev sistema upravljanja API-jev s tehnologijo odkrivanja storitev

V poglavju najprej predstavimo tehnologijo odkrivanja storitev, odkrivanje storitev na API prehodu in registracijo storitev v register storitev, pri čemer izpostavimo tudi prednosti in slabosti.

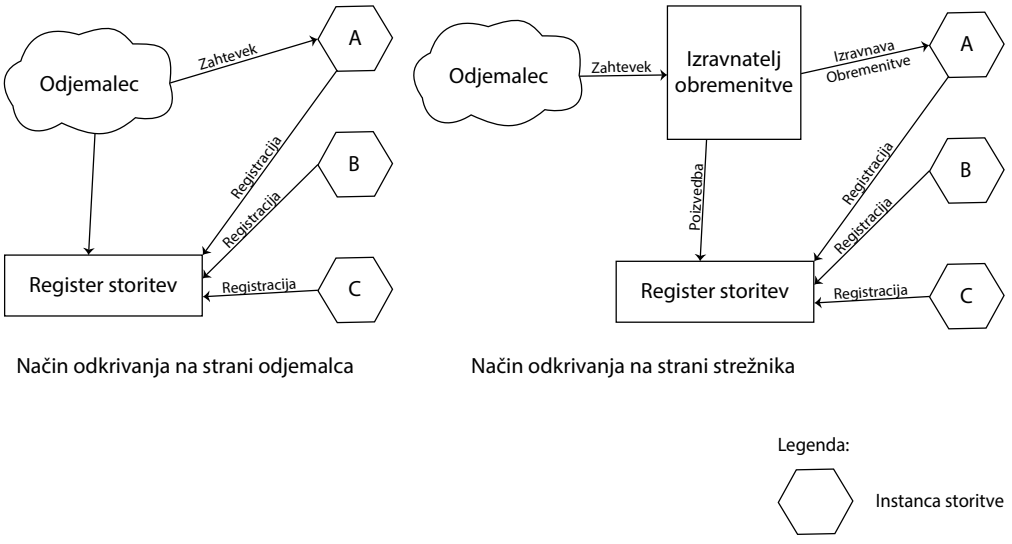
4.1 Tehnologija odkrivanja storitev

Odkrivanje storitev (angl. service discovery) je proces avtomatiziranega iskanja instanc ustrezne storitve za določeno opravilo [21]. Lokacija storitev na gostujočih strežnikih se včasih ni toliko spreminjala, zato potrebe po odkrivanju omrežnih naslovov strežnikov ni bilo. Pojavila pa se je z vpeljavo t.i. tehnologije mikrostoritev, kjer se storitve ponavadi s procesom skaliranja in gručenja dinamično razporejajo po več strežnikih. Da programerju ni potrebno ročno nastavljati novega omrežnega naslova ali popravljati konfiguracijskih datotek, za to poskrbi sistem, ki tem omrežnim naslovom sledi, jih posodablja in vrača našim procesom. V arhitekturi mikrostoritev pa iščemo omrežne naslove vseh instanc neke mikrostoritve.

Poznamo dva glavna načina odkrivanja storitev [16], ki sta prikazana na sliki 4.1.

- **Odkrivanje na strani odjemalca** (angl. client-side discovery), kjer je odjemalec neposredno vezan na register storitev in je odgovoren za pridobivanje fizičnih omrežnih naslovov instanc iskanih storitev ter njihovo izravnavo obremenitve.

- **Odkrivanje na strani strežnika** (angl. server-side discovery), kjer se odjemalec ne zaveda prisotnosti sistema odkrivanja storitev, oziroma mu je ta skrit. Strežnik, ki deluje kot izravnatelj obremenitve, sprejema zahtevke in preko poizvedbe na register storitev določi, na katero instanco jih posredovati.



Slika 4.1: Načina odkrivanja storitev.

4.1.1 Register storitev

Register storitev (angl. service registry) ima v sistemu odkrivanja storitev ključno vlogo hranjenja omrežnih naslovov obstoječih instanc storitev. Ker sta zaželjeni visoka razpoložljivost in ažuriranost, je takšna baza ponavadi postavljena na gruči strežnikov, ki uporablja za ohranjanje konsistentnosti replikcijski protokol. Možno je tudi predpomnjenje na odjemalčevi strani, vendar se omrežni naslovi v dinamičnih sistemih storitev hitro spreminjajo in je potrebno pogosto ažuriranje.

Za prijavo oziroma odjavo storitev se poslužujemo dveh načinov registracije v register storitev [16].

- **Način samoregistracije** (angl. self-registration pattern), kjer je storitev sama odgovorna za uspešno prijavo in odjavo v register storitev. Z metodo srčnega utripa (angl. heartbeat) ponavadi sporočajo svojo prisotnost in hkrati ažurirajo svojo lokacijo, če se ta spremeni.
- **Način registracije preko tretje osebe** (angl. third-party registration pattern), kjer je za uspešno prijavo in odjavo storitev odgovorna druga sistemska komponenta, imenovana registrator (angl. registrar). Ta lahko, preko metode izpraševanja (angl. polling) ali naročanja na dogodke, sledi spremembam naslovov instanc.

4.2 Odkrivanje storitev na API prehodu

API prehod mora poznati omrežni naslov in vrata (angl. port) vseh API-jev, ki so nanj registrirani. Da lažje podpremo sledenje dinamičnim omrežnim naslovom, uporabimo tehnologijo odkrivanja storitev na strani odjemalca, ki smo ga opisali v podpoglavju 4.1. Kot alternativa ali dodatek, lahko sistem podpira tudi statične omrežne naslove, ki jih ročno vnesemo v sistem. Prehod razširimo z ustreznim vmesnikom, ki implementira odkrivanje storitev in v funkcionalnost posredovanja zahtevkov dodamo dinamično pridobivanje naslovov, ki se lahko na prehodu predpomnijo.

4.2.1 Predpomnjenje storitev

Predpomnjenje na prehod doda podatkovno strukturo, kjer se lahko prepodmnijo omrežni naslovi pridobljeni iz registra storitev. Takšna struktura mora biti robustna in stalno ažurirana s spremembami stanja registra storitev. Predpomnjenje je uporabno, kadar se omrežni naslovi API-jev ne spreminjajo pogosto. Če bi se z vsakim zahtevkom spremenila tudi lokacija API-ja, bi se dodana vrednost predpomnjenja izničila. Ozko grlo sistema je ponavadi frekvenca osveževanja podatkovne strukture in ne njegova velikost in količina shranjenih podatkov.

4.2.2 Deložacijske strategije

Vemo, da podatkovne strukture nimajo neskončne kapacitete. Pomembno je torej tudi, katere zapise umaknemo oziroma izbrišemo iz strukture, kadar se ta zapolni. Da naredimo prostor za nove, lahko uporabimo eno izmed deložacijskih strategij [23] (angl. eviction strategies).

- **Najmanj nedavno uporabljen** (angl. Least Recently Used – LRU) element je izbran za izločitev, kadar ima najmanjšo vrednost časovnega žiga (angl. timestamp). Ta se določi ob postavitvi (angl. put) elementa v strukturo in pososdablja z njegovo pridobitvijo (angl. get) iz nje.

- **Najmanj pogosto uporabljen** (angl. Least Frequently Used – LFU) element je izbran za izločitev, kadar ima najnižjo vrednost števca uporab. Števec se nastavi ob postavitvi elementa v strukturo in se veča s številom pridobitev iz nje. Takšen algoritem ima veliko slabost, da se težje uveljavijo in obstojijo kasneje dodani elementi, čeprav se lahko ti uporabljajo pogosteje.

- **Prvi notri, prvi zunaj** (angl. First In First Out – FIFO) je strategija, ki element za izločitev izbere na podlagi vrstnega reda prihajanja v strukturo. Izbran je element, ki je v strukturo prišel pred vsemi

drugimi. Takšen algoritem je uporabljen, kadar predvidevamo, da je možnost uporabe prvega elementa v prihodnosti mala.

Opisani algoritmi dolgoročno delujejo bolje, če iz strukture vzamejo naključni vzorec elementov in nad njim izvedejo izločanje. Izkaže se, da izločen element ob velikosti vzorca petnajstih elementov v devetindevetdesetih odstotkih pade v spodnjo četrtino po uporabljenosti vseh elementov [23].

4.2.3 Strategije predpomnjenja

Za hitro delovanje API prehoda je ključna pravilna izbira načina osveževanja podatkovne strukture, ki je lahko, zaradi hitrega spreminjanja registra storitev, netrivialna. Podatki v predpomnilni strukturi naj bi bili popolna kopija dela baze, kjer v našem primeru bazo predstavlja register storitev, hranjene entitete pa omrežni naslovi instanc storitev. Poznamo štiri pogosto uporabljene strategije sinhronizacije predpomnilne strukture z bazo entitet [13], kjer sta na sliki 4.2 prikazani dve najbolj zastopani.

- **Predpomnjenje na stran** (angl. Cache-aside) je strategija, kjer aplikacija ročno skrbi za zapise v predpomnilni strukturi in bazi entitet. Če zahtevane entitete v predpomnilni strukturi ni, se naredi poizvedba na bazo. Entiteta se nato za prihodnje zahteve na isto entiteto shrani v predpomnilno strukturo. Posodabljanje entitet poteka vzporedno na bazi in predpomnilni strukturi hkrati, kjer moramo zaradi konsistence podatkov proces izvesti kot trasakcijo. Metoda omogoča čisto sinhronizacijo podatkov v obeh objektih, vendar smo primorani izvesti zgoraj opisane korake za vsako metodo pridobivanja entitet. Rešitev tako ni primerna, če zapise v bazi spreminja tretja oseba. Posledično odpade monžnost za integracijo s tehnologijo odkrivanja storitev.

- **Branje skozi** (angl. Read-through) je strategija, kjer prepustimo sinhronizacijo z bazo predpomnilniškemu ponudniku (angl. cache provider), ki deluje kot predpomnilniška abstraktna plast med predpomnilniško strukturo in bazo. Če zahtevane entitete v predpomnilni

strukturi ni, se naredi poizvedba na predpomnilniškega ponudnika. V primeru, ko je ta še nima v evidenci, se naredi poizvedba na bazo. Entiteta se nato na abstraktni plasti stalno opazuje in neodvisno od zahtev posodablja, glede na spremembe te entitete v bazi. Prihodnje zahteve iz abstraktne plasti pridobijo ažurirano entiteto, brez poizvedbe na bazo. Metod za pridobitev entitet iz baze tako ni potrebno pisati za vsako entiteto posebej. Takšna strategija je primerna, kadar želimo entitete v bazi spreminjati preko tretje osebe.

- **Pisanje skozi** (angl. Write-through) je strategija, ki ima enako arhitekturno zasnovo kot strategija branja-skozi, vendar se zapisi entitet v bazi spreminjajo glede na spremembe v predpomnilni strukturi. Če želimo, lahko s transakcijami zagotovimo čisto sinhronizacijo podatkov.

- **Pisanje vzadaj** (angl. Write-behind) je strategija, kjer aplikacija spremembe entitet v predpomnilni strukturi periodično zapisuje (angl. flush) v bazo. Takšna strategija je primerna, kadar ne zahtevamo stalne konsistence podatkov.

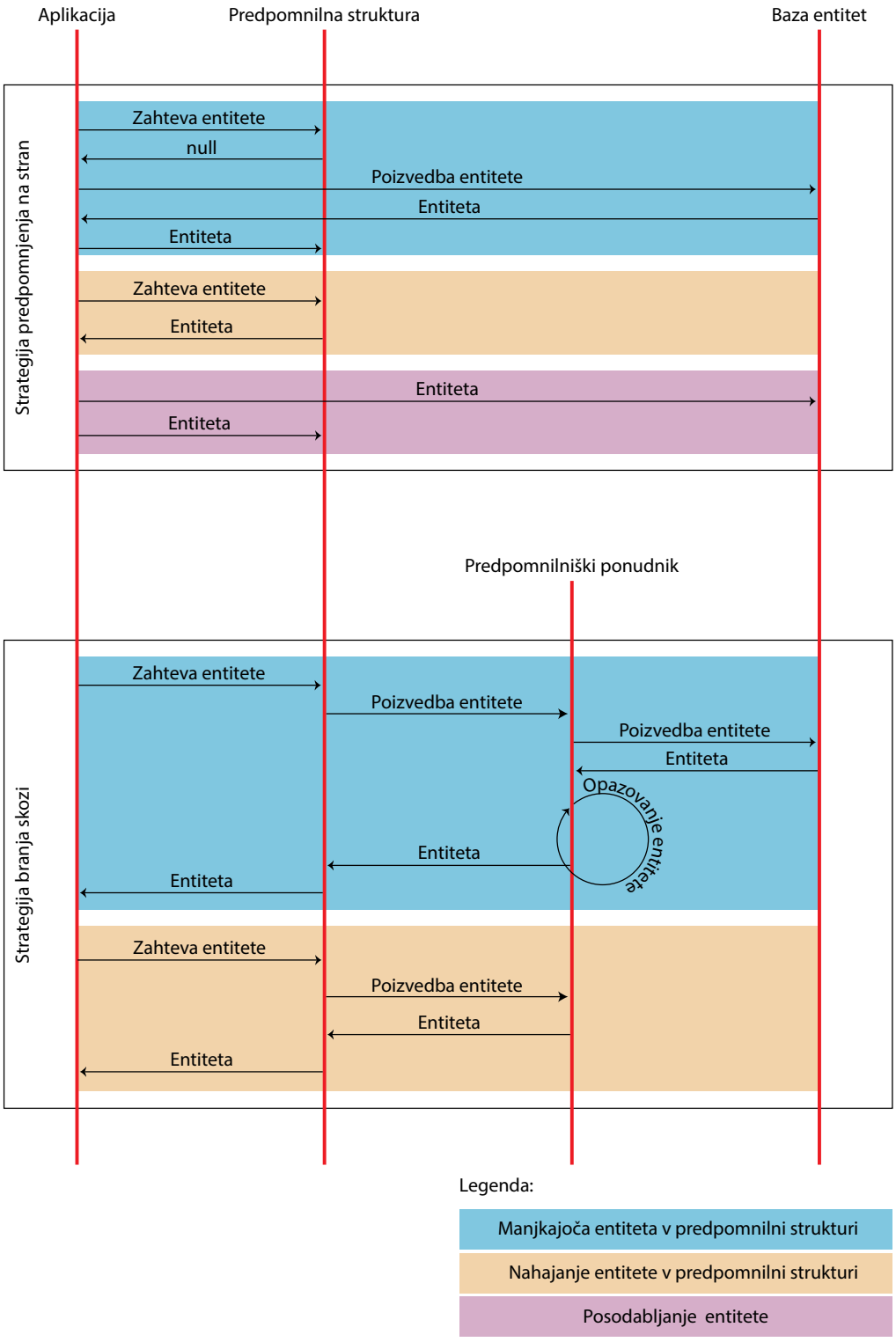
Kot je navedeno v zborniku [18] se izkaže, da lahko v praksi z dobro izbiro algoritma dosežemo petindevetdeset odstotno efektivnost zgornje meje učinkovitosti idealnega algoritma predpomnjenja.

4.3 Registracija storitev v register storitev

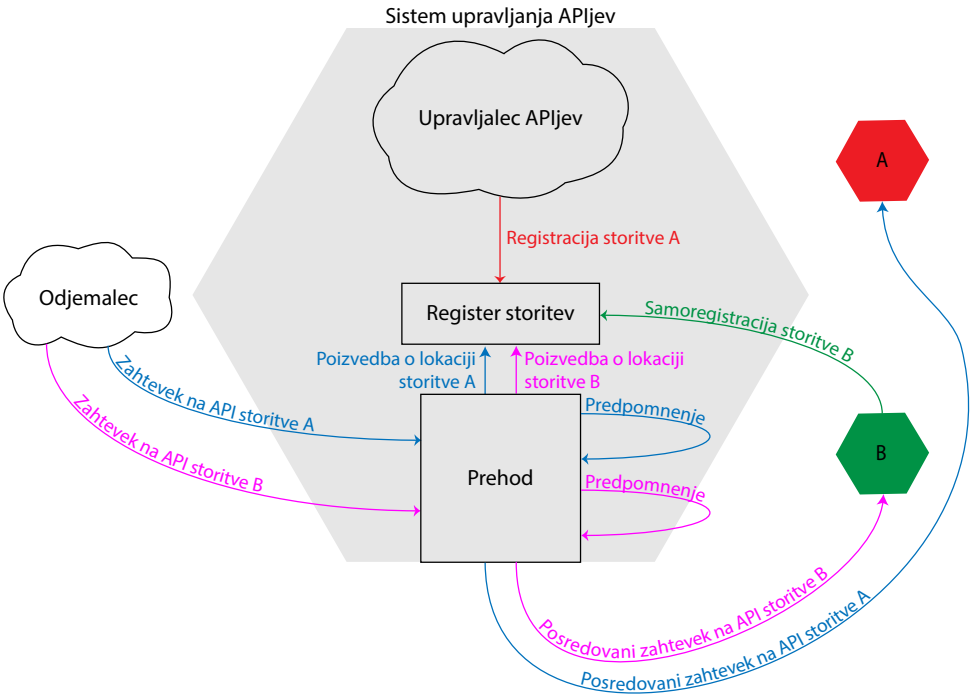
Storitve, ki jih želimo odkrivati v našem sistemu upravljanja API-jev, morajo biti registrirane v registeru storitev, na katerega se sklicuje API prehod. To lahko storijo preko registracije tretje osebe ali načina samoregistracije, ki smo ju opisali v razdelku 4.1.1. Predvsem moramo paziti, da ohranjamo skladno strukturo registra storitev, saj je od nje odvisna uspešnost odkrivanja storitev na strani prehoda. Primerni kandidati za vlogo registra storitev v sistemu upravljanja API-jev so lahko na primer ZooKeeper, Consul in etcd. ZooKeeper je projekt pod okriljem Apache Software Foundation, ki

ponuja distribuirano hrambo konfiguracijskih podatkov in je eden najpopularnejših tehnologij za odkrivanje storitev. Storitev se lahko vanj registrira z unikatnim imenom in nosi informacijo o njegovi lokaciji. Druga možnost je HashiCorp-ov projekt Consul, ki implementira distribuirano arhitekturo prepletanja storitev, preko katere lahko varno povezujemo in konfiguriramo storitve v različnih okoljih. Orodje etcd smo uporabili v naši rešitvi in ga bolj podrobno opisali v razdelku 5.1.5.

Komponenta, odgovorna za registracijo in odjavo v register storitev preko tretje osebe, je upravljalec API-jev, ki za to izpostavlja posebni končni točki. V primeru samoregistracije storitev se te v register storitev registrirajo in odjavijo same, kot je prikazano na sliki 4.3.



Slika 4.2: Stretegiji sinhronizacije predpomnilne strukture z bazo entitet.



Slika 4.3: Arhitektura sistema upravljanja API-jev z dodano komponento odkrivanja storitev.

Poglavje 5

Integracija tehnologije odkrivanja storitev v obstoječi sistem upravljanja API-jev

V poglavju opisujemo integracijo tehnologije odkrivanja storitev v obstoječi sistem upravljanja API-jev. Najprej predstavimo uporabljene tehnologije, nato pa implementacijo in integracijo rešitve.

5.1 Uporabljene tehnologije

Za glavni programski jezik smo izbrali Javo, saj vse uporabljene glavne komponente izbranega sistema temeljijo na platformi Java EE [26](Java Enterprise Edition). Ta ponuja okolje za izvajanje javanskih aplikacij in specificira orodja ter vmesnike za izgradnjo visoko razpoložljive, distribuirane in skalabilne poslovne programske opreme (angl. enterprise applications) velikega obsega.

5.1.1 Java

Java je objektno usmerjen programski jezik. Prvotno ga je razvil James Gotling v podjetju, ki ga je kasneje pod okrilje vzel Oracle. Jezik je prevzel

veliko sintakse iz programskih jezikov C in C++, vendar ni ohranil vseh nižje nivojskih funkcionalnosti, kakor je npr. direktno manipuliranje s kazalniki (angl. pointers). Nekateri ga poslednično razvrščajo med jezike primerne za učenje konceptov programiranja, saj je relativno varen in programerju skriva procese v strojni opremi računalnika. Javanska koda se ponavadi prevede v t.i. bitno kodo (angl. bytecode), ki je zasnovana za zagon na javanskem navideznem stroju (angl. Java Virtual Machine – JVM), kar je ena od prednosti jezika, saj lahko vse računalniške arhitekture z nameščeno javansko virtualko posledično zaganjajo javanske aplikacije, tudi če so te bile prevedene na drugačnih sistemih.

Leta 2018 je bil rangiran kot drugi najbolj zastopani programski jezik, takoj za programskim jezikom JavaScript [15]. S približno 9 milijoni aktivnih razvijalcev se je ustvarila velika skupnost, kar pripomore k lažjemu razvoju in pomoči novincem. Prav tako v Javi za večino računalniških tehnologij obstajajo razvite knjižnjice, ki jih lahko preko odvisnosti poljubno dodajamo obstoječim projektom.

Mi smo ga uporabili kot dobro poznan programski jezik za razvoj mikrostoritev. Kot so povedali programski inženirji v članku [14] iz leta 2018, sta bila Java in JVM testirana v produkcijskem okolju že več kot dvajset let. Večina programerjev ima posledično z njima večletne izkušnje, kar pripomore k hitremu in samozavestnemu razvoju novih mikrostoritev. Vendar pri lahkih mikrostoritvah s svojo težo izgubi nekaj dodane vrednosti, kjer prideta do izraza programska jezika Ruby in JavaScript s svojima ogrodjema Sinatra in NodeJS.

5.1.2 KumuluzEE

KumuluzEE je ogrodje za razvoj mikrostoritev na podlagi standardnih Java EE tehnologij [9]. Prav tako omogoča prenos obstoječih Java EE aplikacij v arhitekturo mikrostoritev. KumuluzEE mikrostoritve so lahke in narejene za enkapsulacijo v vsebnikih Docker. Razpolaga z moduli, kot so logiranje, konfiguracija, metrike, varnost, odkrivanje storitev in še več, ki pripomorejo

lažji integraciji v t.i cloud-native okolja. Za nas je bil predvsem zanimiv modul odkrivanja storitev, KumuluzEE Discovery.

5.1.3 KumuluzEE Discovery

KumuluzEE Discovery je modul ogrodja KumuluzEE, ki implementira tehnologijo odkrivanja storitev in omogoča registracijo, deregistracijo, odkrivanje in izravnavo obremenitve storitev na strani odjemalca. Za instanciranje uporablja CDI zrna (angl. CDI beans) in je namenjen postavitvi na ogrodje KumuluzEE, kjer služi kot samoregistrator storitev v register storitev. Vsebuje tudi funkcionalnost za odkrivanje ostalih storitev, ki so bile registrirane preko istega modula oziroma imajo enako strukturo zapisa v izbranem registru storitev. KumuluzEE trenutno sprejema dve različni tehnologiji v vlogi registra storitev, ki sta Consul in etcd [28]. Za našo implementacijo smo izbrali etcd, ki je bolje opisan v razdelku 5.2.1.

5.1.4 Orodje za upravljanje API-jev

S tehnologijo odkrivanja storitev smo nadgradili obstoječe orodje za upravljanje API-jev, ki smo ga razvijali v Laboratoriju za integracijo informacijskih sistemov na Fakulteti za računalništvo in informatiko Univerze v Ljubljani. Sestavljeno je iz treh komponent.

- **Upravljalec API-jev** deluje v našem sistemu kot jedro in centralna točka sistema upravljanja API-jev. Temelji na ogrodju KumuluzEE in izvaja vso poslovno logiko potrebno za dodajanje, odstranjevanje, spreminjanje, objavljanje in odjavljanje API-jev v sistemu. Preko odprtokodne rešitve Keycloak [27] izvaja avtentikacijo uporabnikov in preverja njihove pravice oziroma nepravice za dostop do zahtevanih funkcionalnosti. Poleg avtentikacijskega strežnika potrebujemo tudi povezavo na podatkovno bazo, ki je v našem primeru lahko instanca PostgresSQL ali Oracle relacijske baze.

- **Čelni spletni grafični vmesnik** na intuitiven način v spletnem brskalniku uporabniku ponudi vso preko RESTful API-ja izpostavljeno poslovno logiko upravljalca API-jev.

- **API prehod** izpostavlja končne točke preko katerih potekajo zahtevki na zaledne API-je. Izhodišče komponente je odprtokodni projekt Apiman [22], ki je bil prilagojen našemu sistemu. Ima neposredno povezavo na upravljalca API-jev, od koder sprejema ukaze kako ravnati z zahtevki na specifične API-je in vrača njihove metrike za statistično analizo sistema. Za hranjenje API-jev, načinov uporabe in omejitev se uporablja odprtokodna rešitev Elasticsearch [38], za potrebe avtentikacije pa ista instanca Keycloak-a kot pri upravljalcu API-jev.

5.1.5 Etcd

Etcd je odprtokodni distribuiran sistem za hrambo podatkov v obliki ključ-vrednost (angl. key-value store) [24], kjer imajo ključi drevesno strukturo običajnega datotečnega sistema (angl. directory tree structure), ki navigirajo do prepisanih vrednosti. Ime je pridobil po imenu direktorija "/etc", kjer se v Unix sistemih hranijo konfiguracijske datoteke, plus pripona "d", ki označuje distribucijo (ang. distributed). Ponavadi teče na gruči strežnikov. Zaradi uporabljenega konsenznega algoritma Raft [29] je priporočena uporaba lihega števila instanc. Njegova konsistenca in toleranca pred odpovedjo je primerna za shranjevanje naslovov storitev, ki jih potrebujemo v našem API prehodu. Za hranjenje podatkov v sistemu etcd smo uporabil KumuluzEE Discovery, ki ima sledečo strukturo zapisov [28].

- **Ključ**: `/environments/{okolje}/services/{ime_storitve}/{verzija_storitve}/instances/{avtomatsko_generiran_id_instance}/url`

- **Vrednost**: omrežni naslov storitve

Tako je vsaka storitev enolično določena preko imena, izvajalnega okolja (angl. runtime environment) in verzije, njene instance pa z unikatnim identifikacijskim številom, ki se samodejno generira ob zagonu storitve.

Primer pridobitve naslova instance storitve

Kot primer vzemimo storitev z imenom "testna-storitev", ki teče v razvijalskem okolju imenovanem "development-env". Recimo, da imamo aktivne tri instance v verziji "1.0" na omrežnih naslovih "127.0.0.1:3000", "127.0.0.1:3001" in "127.0.0.1:3003". Predpostavimo, da sta bila omrežna naslova prve in druge instance pred tem že enkrat pridobljena. Preko modula KumuluzEE Discovery naredimo poizvedbo na strežnik etcd s potjo:

`/environments/development-env/services/testna-storitev/1.0/`

Modul dobi vrnjene omrežne naslove instanc in preko algoritma Round-robin, ki opravlja funkcijo izravnave obremenitve, izbere nazadnje še neuporabljeno instanco. Klicana metoda za odkrivanje storitve vrne omrežni naslov tretje instance, "127.0.0.1:3003", na katerega lahko nato poljubno pošiljamo zahtevke.

5.1.6 Ostale tehnologije

Za celotno izpeljavo in gradnjo projektov smo uporabili tudi ostale tehnologije, med drugim Git [25] in Maven [36].

Git

Git je široko uporabljeno orodje za verzioniranje in sledenje projektom (angl. version control system). Beleži spremembe v datotekah in je primeren za ekipno delo, predvsem na razvoju izvorne kode programske opreme. Z njim lahko označimo katerikoli direktorij na našem računalniku kot t.i. repozitorij Git (angl. Git repository). Takšen lahko sledi spremembam v tem direktoriju čisto lokalno, vendar se ponavadi poveže na oddaljen repozitorij (angl. remote repository), kamor pošiljamo (angl. push) svoje spremembe in

tako prispevamo k skupnemu projektu. Sistem Git s pomočjo t.i. vej (angl. branches) podpira vzporedne tokove dela, ki so predvsem uporabni za razvoj novih funkcionalnosti, neodvisno od ostalih vej razvoja.

Maven

Maven je orodje, ki je bilo razvito in izdano pod okriljem Apache Software Foundation leta 2004. Uporablja se za avtomatizirano gradnjo predvsem javanskih projektov. Vsak Maven projekt mora vsebovati vsaj eno t.i. POM (angl. Project Object Model) datoteko, ki je napisana v formatu XML in vsebuje podrobnosti o projektu. V njej lahko definiramo odvisnosti, katerih ni potrebno imeti vnaprej nameščenih na izvajajočem računalniku, saj Maven poskrbi za njihovo namestitev iz oddaljenih repozitorijev, v kolikor jih najde lokalno. Maven projekte lahko gnezdimo in zgradimo t.i. hierarhični sistem Maven modulov v obliki drevesa.

5.2 Implementacija in integracija

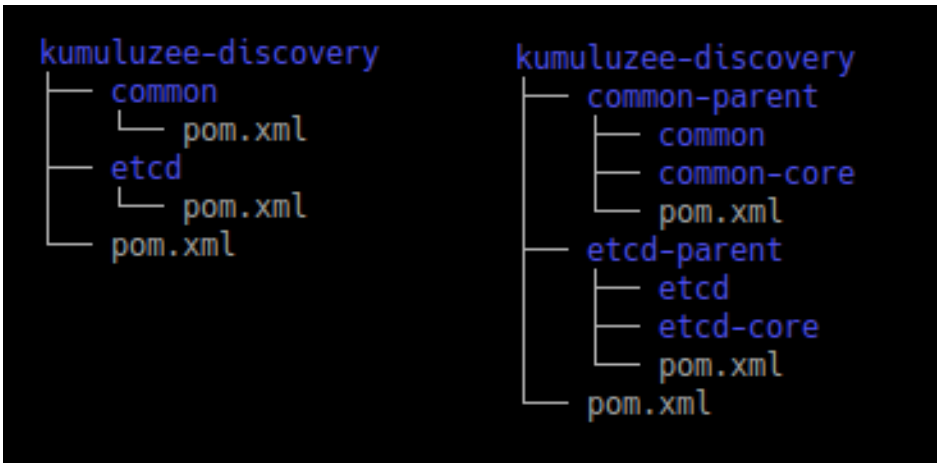
V prejšnjih poglavjih smo opisali koncepte in načine integracije tehnologije odkrivanja storitev v sisteme upravljanja API-jev. Za potrebe diplome smo z zgoraj opisanimi tehnologijami integracijo v obstoječi sistem tudi implementirali. Za uspešno delovanje odkrivanja storitev je bila potrebna integracija modula KumuluzEE Discovery v upravljalca API-jev, kjer smo morali omogočiti registracijo storitev preko tretje osebe in implementirati iskanje omrežnih naslovov že registriranih storitev.

5.2.1 Prilagoditev modula KumuluzEE Discovery

Ker API prehod uporabljenega orodja upravljanja API-jev ne podpira zrn CDI, smo zaradi nekompatibilnosti z njimi, morali v novi Git veji spremeniti notranjo arhitekturo modulov Maven celotnega projekta KumuluzEE Discovery. Paziti smo morali, da med procesom preurejanja nismo porušili

poslovne logike. Na sliki 5.1 vidimo levo na veji "master" originalno strukturo projekta, desno pa na veji "feature/nocdi" novo razporeditev Maven modulov.

Modula common in etcd smo oba ločili na dva dela, jedro (core) in del, ki podpira CDI zrna, in ju povezali preko starševskega modula. Vsa logika povezana z odkrivanjem storitev je ostala v jedrnih delih in je bila tako pripravljena za vključitev v druge projekte.



Slika 5.1: Sprememba struktre projekta KumuluzEE discovery.

Prilagojeno vejo smo nato zgradili preko orodja Maven in modul etcd-core kot odvisnost vključili na ustrezno mesto v projektih upravljalca API-jev in API prehoda.

5.2.2 Integracija v upravljalca API-jev

Ob uspešni vključitvi odvisnosti etcd-core v upravljalca API-jev smo morali najprej poskrbeti za pravilno instanciranje odjemalca etcd (angl. etcd client), ki bo izvajal klice na gručo etcd, v katero bo storitev registrirana. Za uspešno vzpostavitev tega upravljalec API-jev ob registraciji poda metodi naslednje parametre:

- uporabniško ime in geslo, ki preverjata veljavnost dostopa do gruče etcd, v kolikor ju zahtevamo,
- certifikat, ki preverja veljavnost dostopa do gruče etcd, če ga zahtevamo,
- omrežne naslove, kjer se nahajajo instance gruče etcd,
- par ostalih parametrov, ki določajo način povezave in strategijo ponavljanja neuspelih klicev.

Metodo smo kot nov vir REST (angl. REST resource) izpostavili poleg virov objave in odjave API-jev. Vir sprejme med parametri poti identifikator API-ja, ki ga hočemo registrirati in opisane parametre za sklicevanje na pravilno gručo etcd. Preko identifikatorja iz baze pridobimo ustrezni API in iz njega izvlečemo potrebne identifikatorje storitve.

Tako imamo vse potrebne parametre za klic integrirane funkcije odvisnosti KumuluzEE Discovery, ki storitev registrira v register storitev. Ker se pri objavljanju API-ja njegovi identifikatorji za odkrivanje posredujejo API prehodu, je odkritje omrežnih naslovov na prehodu možno s trenutkom njegove registracije. Paziti moramo, da storitev registriramo na isto gručo etcd, na katero se sklicuje API prehod z objavljenim API-jem.

5.2.3 Integracija v API prehod

Kot pri upravljalcu smo preko Maven odvisnosti vključili modul etcd-core. API prehod smo zasnovali, da ohranja povezavo na enega odjemalca etcd, ki se vzpostavi ob instanciranju prehoda. Parametre za ustrezno povezavo na gručo etcd prebere iz okoljskih spremenljivk, ki so enake tistim v razdelku 5.2.2, pri postavitvi odjemalca etcd v upravljalcu API-jev.

Objekt, definiran v odvisnosti KumuluzEE Discovery, z atributom “etcd”, ki hrani prej vzpostavljenega odjemalca etcd, izpostavlja metodo za odkrivanje storitev. Ta sprejema naslednje parametre:

- ime storitve,
- izvajalno okolje storitve,
- verzijo storitve.

Po uspešno izvedeni poizvedbi na odjemalca etcd, če je storitev pravilno registrirana v pravi register storitev, ta vrne omrežni naslov instance, kot je opisano v primeru razdelka 5.1.5. Dobljeni omrežni naslov nato priredimo atributu "endpoint" objekta API, ki se kasneje uporabi za določitev končne destinacije, ciljnega API-ja.

Prej omenjeni objekt hrani tudi atribut "serviceInstances", predstavljajoč predpomnilniško strukturo, ki je deklarirana kot dvojno kartirana podatkovna struktura (angl. Map data structure). Ta hrani vrednosti v obliki ključ-vrednost, kar naredi preslikavo med etcd zapisom in zapisom v predpomnilniški strukturi trivialno. Zunanja kartirana podatkovna struktura označuje posamezno storitev, notranja pa posamezno instanco storitve. Primerna in tudi uporabljena strategija predpomnjenja je branje skozi, ki smo jo opisali v razdelku 4.2.3.

Ob vsakem prvem zahtevku na nek API se po prejetih omrežnih naslovih iz registra storitev naredi zapis v predpomnilniško strukturo, iz katere se pridobivajo naslovi storitve za prihodnje klice na isti API, kjer se na notranji kartirani podatkovni strukturi izvaja izravnavanje obremenitve instanc. Na nov zapis se prav tako postavi t.i. opazovanje ključa (angl. watch-key), ki spremlja spremembe v registru storitev in sproti ažurira pripadajoč zapis v predpomnilniški strukturi. Vsi nadaljnji zahtevki na storitev pridobijo omrežni naslov instanc iz te strukture, na kateri tudi poteka izravnava obremenitve.

Poglavje 6

Testno okolje in rezultati

V tem poglavju je predstavljena postavitev testnega okolja razširjenega sistema upravljanja API-jev, ki smo ga opisali v prejšnjem poglavju, na lokalnem računalniku s pomočjo vsebnikov Docker. Na tej postavitvi smo preučili časovni vpliv dodane tehnologije odkrivanja storitev na celotni sistem.

6.1 Testna orodja

Za potrebe testiranja smo uporabili nekaj orodij, ki so nam pri postavitvi testnega okolja našega sistema upravljanja API-jev olajšala delo.

6.1.1 Vsebniki Docker

Že od nekdaj se pojavljala potreba po unificiranem nameščanju programske kode in nastavljanju primernega izvajalnega okolja na različnih operacijskih sistemih. Ta problem so nekako reševale virtualne naprave (angl. virtual machines), vendar so bile te obremenjujoče za celoten strežnik, saj so vsebovale vse komponente operacijskega sistema, tudi tiste, ki jih sama aplikacija za uspešno delovanje ni potrebovala. Tako se je pojavila rešitev z lahkimi (angl. light-weight) vsebniki [30].

Docker je orodje, ki skrbi za instanciranje vsebnikov iz slik (angl. images). Glavna razlika med vsebniki Docker in ostalimi virtualnimi napravami

je, da si vsebniki Docker delijo jedro operacijskega sistema z gostiteljem. Kot je navedeno v članku [2] iz leta 2015, se lahko na povprečnem namiznem računalniku izvaja več sto vsebnikov, medtem ko vzporedno izvajanje par virtualnih naprav povzroča težave. Slike gradimo z datotekami Dockerfile, kjer z uporabo preproste sintakse definiramo, kako zgraditi sliko, kar je idealno za enkapsuliranje mikrostoritev v vsebnike Docker. Tehnologijo smo uporabili kot orodje za postavitev testnega okolja [2] našega sistema.

Docker-compose

Pojavitev mikrostoritev in njihova enkapsulacija v vsebnike ni spremenila le arhitekture storitev, ampak tudi okolje v katerem se izvajajo. Za lažje obvladovanje skupine vsebnikov, ki morda sestavljajo le eno aplikacijo, se je pojavil Docker-compose za hitrejšo postavitev v vseh okoljih, predvsem razvojnih. Docker-compose je orodje za definiranje in izvajanje več vsebnikov hkrati [37]. Ker uporablja vsebnike Docker, potrebujemo na našem sistemu nameščen Docker stroj (angl. Docker engine). Preko označevalnega jezika YAML napišemo konfiguracijsko datoteko, da vzpostavimo večje število poljubno konfiguriranih storitev z enim samim ukazom:

```
$ docker-compose -f {ime_konfiguracijske_datoteke} up
```

In ga nato tudi ukinemo z ukazom:

```
$ docker-compose -f {ime_konfiguracijske_datoteke} down
```

6.1.2 JMeter

JMeter je odprtokodno orodje pod okriljem Apache Software Foundation, napisano v programskem jeziku Java. Prvotno je bilo zasnovano za testiranje spletnih aplikacij, a se je uveljavilo tudi za merjenje zmogljivosti ostalih sistemov [34]. Abstraktno si ga lahko predstavljamo kot skupek večih brskalnikov, ki z generiranjem večje količine prometa simulira uporabnike na našem sistemu. Tako je primeren za testiranje robustnosti omrežij, strežnikov ali

gruč strežnikov. Meri lahko tudi splošne performance testiranih sistemov, kjer nam pridobljeni podatki dajo informacijo o potencialnih nevarnostih v primeru postavitve v produkcijsko okolje.

Medtem ko je orodje Postman (grafično orodje za testiranje RESTful API-jev) bolj primerno za testiranje pravilnega delovanja kočnih točk, je JMeter primeren za preverjanje, koliko prometa naš sistem za pričakovano delovanje še prenese. V našem primeru smo orodje uporabili za generacijo večje vzporedne količine zahtevkov na objavljeni API preko API prehoda.

6.2 Postavitev testnega okolja

Za potrebe testiranja smo morali definirati datoteke Dockerfile za upravljalca API-jev, API prehod, čelni spletni grafični vmesnik in storitev hranjenja strank, kjer so slike ostalih potrebnih vsebnikov že bile definirane. S pomočjo orodja Docker-compose smo v konfiguracijskih datotekah določili sistemske spremenljivke, ki so večinoma narekovale delovanje vsebovanih aplikacij ali povedale omrežni naslov ostalih storitev. Nato smo z ukazom "docker-compose up" pognali vsako izmed konfiguracijskih datotek in tako vzpostavili naslednje vsebnike storitev, opisanih v 5. poglavju.

- **Vsebnik upravljalca API-jev** je izhajal iz slike "openjdk:8u131-jdk-alpine", kjer smo morali priskrbeti primerne parametre za povezavo na vsebnik PostgresSQL in instanco Keycloak-a.

- **Vsebnik API prehoda** je izhajal iz slike "openjdk:8u131-jdk-alpine", kjer smo morali priskrbeti primerne parametre za povezavo na vsebnika etcd in Elasticsearch.

- **Vsebnik čelnega spletnega grafičnega vmesnika** je izhajal iz slike "digitallyseamless/nodejs-bower-grunt", kjer se je spletna stran servirala na ogrodju NodeJS preko orodja Grunt. Poskrbeti smo morali za primerne parametre za povezavo na vsebnik upravljalca API-jev.

- **Vsebnik etcd** je bil instanciran iz slike "quay.io/coreos/etcd:latest".
- **Vsebnik storiteve hranjenja strank** je izhajal iz slike "openjdk:8u131-jdk-alpine", kjer nam je služila kot testni RESTful API, na katerega smo pošiljali zahtevke.
- **Vsebnik JMeter** je bil instanciran iz slike "justb4/jmeter':latest", kjer nam je služil kot generator večjega števila zahtevkov.
- **Vsebnik Elasticsearch** je bil instanciran iz slike "elasticsearch:1.7.5", kjer smo ga uporabili kot podatkovno bazo vsebnika API prehoda.
- **Vsebnik PostgreSQL** je bil instanciran iz slike "postgres:10", kjer smo ga uporabili kot podatkovno bazo vsebnika upravljalca API-jev.

Ker so vsebniki omrežno izolirana okolja, smo jih postavili v skupno omrežje Docker (angl. Docker network), ki virtualizira umetno omrežje, preko katerega se lahko vsebniki med seboj sporazumevajo. Instanca avtentikacijske komponente (Keycloak) je bila strežena izven ustvarjenega, notranjega omrežja Docker. Prav tako smo definirali prostore Docker (angl. Docker volumes), saj smo želeli, da se podatki shranjeni v bazah PostgreSQL in Elasicsearch, ohranijo tudi ob brisanju omenjenih vsebnikov. Da smo lahko do spletišča, ki ga je serviral vsebnik čelnega spletnega grafičnega vmesnika, dostopali preko gostitelja, smo bili primorani na gostiteljskem omrežju izpostaviti vrata vsebnikov upravljalca API-jev in čelnega spletnega grafičnega vmesnika. Ker lahko do vsebnikov dostopamo le preko omrežnega naslova lokalnega gostitelja, smo na vsebniku čelnega spletnega grafičnega vmesnika nastavili omrežni naslov vsebnika upravljalca API-jev na lokalnega gostitelja.

Ob uspešni vzpostavitvi vseh komponent smo morali API storitev hranjenja strank preko vsebnika čelnega spletnega grafičnega vmesnika prijaviti v upravljalca API-jev in registrirati v vsebnik etcd. Nato smo ga še objavili na API prehod. Okolje je bilo tako pripravljeno na posredovanje klicev od vsebnika JMeter, preko vsebnika API prehoda, do vsebnika storitve hranjenja strank.

6.3 Časovna analiza

Ko smo preverili pravilno delovanje našega sistema, nas je zanimala časovna odzivnost in dodana latenca, ki jo povzroči uvedba tehnologije odkrivanja storitev. Ker je frekvenca registracij novih storitev v primerjavi z odkrivanjem storitev v sistemih upravljanja API-jev izredno mala, je razmerje vsote časovnih latenc, ki jih prineseta ti dve funkcionalnosti, zanemarljivo. Kljub temu da je registracija novih storitev kot posamezen proces časovno veliko požrešnejši, smo se osredotočili na največji skupni prispevek, ki ga prispeva odkrivanje storitev.

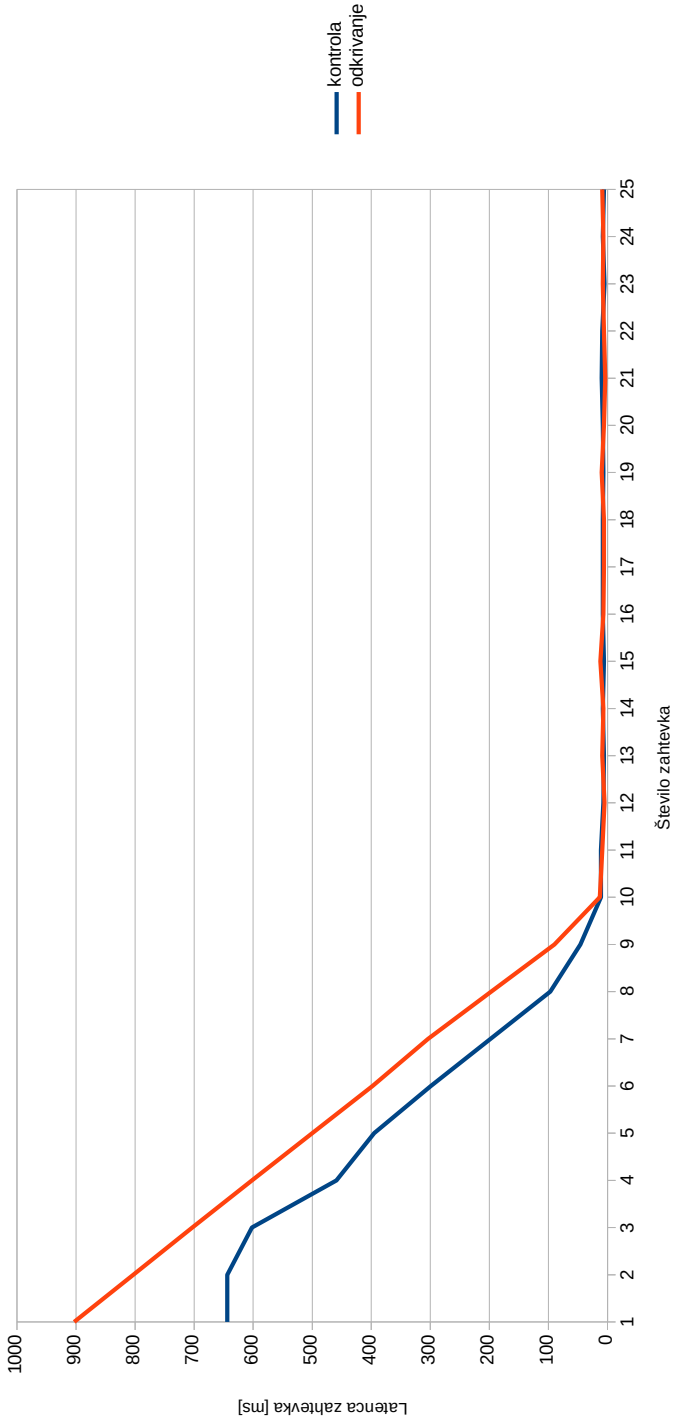
6.3.1 Odkrivanje storitev

Za potrebe testiranaja smo pripravili dve različici API prehoda. Ena je imela omogočeno odkrivanje storitev, druga pa je služila kot kontrola in je že imela določen fiksni omrežni naslov API-ja. Z orodjem JMeter smo v obeh primerih vzporedno, iz stotih niti pošiljali zahtevke z uvajalnim časom desetih sekund na vsebnik storitve hranjenja strank preko objavljenega API-ja in opazovali njihove latence. Iz grafa na sliki 6.1 je razvidno, da v zameno za boljšo uporabniško izkušnjo, žrtvujemo nekaj več milisekund časa, ki je potreben za poizvedbo na register storitev. Vidimo tudi, da se po nekaj prejetih zahtevkih latenca ustali (nizka je zaradi sobivanja vsebnikov na istem gostitelju), kar je posledica predpomnjenja podatkov iz vsebnika Elasticsearch in registra storitev na vsebniku API prehoda. Iz kontrole je razvidno, da tudi povezava na podatkovno bazo (vsebnik Elasticsearch) prispeva velik delež k skupni latenci.

6.4 Ugotovitve

Vpliv odkrivanja storitev lahko zmanjšamo, če register storitev postavimo bližje API prehodu. Najmanjšo latenco dosežemo, kadar sobivata na lokalnem gostitelju (v našem primeru, čeprav na istem računalniku, zaradi

vsebnikov in virtualizacije omrežja to ni bilo doseženo). Naša rešitev je tako primerna za uporabo, saj doda majhno latenco, a pomembno izboljša delovanje sistema upravljanja API-jev.



Slika 6.1: Graf primerjave latence pri uporabi tehnologije odkrivanja storitev

Poglavje 7

Zaključek

V nalogi smo uspešno zasnovali in implementirali integracijo tehnologije odkrivanja storitev v obstoječi sistem upravljanja API-jev. Implementacijo smo nato enkapsulirali v vsebnike Docker in jih preko orodja docker-compose postavili v testno okolje na lokalnem računalniku. Nad nadgrajenim sistemom smo izvedli obremenitvene teste in analizirali dodatno latenco, ki jo povzroči dodana tehnologija.

API prehodi kot ključni akterji posredovanja zahtevkov med odjemalci in končnimi točkami, lahko logiko posredovanja in preslikovanja implementirajo na različne načine, kjer smo v tej nalogi spoznali dva pogosto uporabljena. Srečali smo se s tehnologijo odkrivanja storitev, njenimi različicami in načini delovanja, kjer se je integracija v sistem upravljanja API-jev pojavila kot okvirna vsebina diplomskega dela. Izkazalo se je, da je nepredvidljivost lokacije omrežnih storitev lahko velika težava, kadar jih morajo klicati zunanji odjemalci. Tako smo na API prehodu implementirali odkrivanje storitev na skupno gručo etcd, ki je delovala kot skladišče omrežnih naslovov objavljenih storitev. Na koncu smo preko testnega okolja prišli do ugotovitev, da s takšnim sistemom pridobimo na robustnosti in prilagodljivosti sistema, vendar v zameno žrtvujemo nekaj časa. Naša naloga je najti zdravo razmerje med pozitivnimi in negativnimi učinki dodanih funkcinalnosti.

Razvoj in raziskava področja upravljanja API-jev sta nam odprli nove

poti za delo na nadaljnjih projektih usmerjenih v razvoj API-jev in sorodnih projektov oziroma raziskav. Zadnje čase je predvsem popularno področje mikro in nano API prehodov, ki smo jih opisali v podpoglavju 3.4. Njihove razširitve z dodatnimi tehnologijami bi zaradi potrebe po ohranjanju lahkosti in strnjenosti bile prava preizkušnja za prihodnji projekt.

Literatura

[1] M. Berry. Major cloud computing vendors. Dosegljivo: `http://www.itmanagerdaily.com/cloud-computing-vendors/`. [Dostopano: 12. 9. 2018].

[2] C. Boettiger. An introduction to docker for reproducible research. *SIGOPS Oper. Syst. Rev.*, vol. 49(1):str. 71–79, januar 2015.

[3] A. Mathuria C. Boyd. *Protocols for Authentication and Key Establishment*. Springer, 2003 edition, 2008.

[4] R. S. Pressman D. F. Rico. *ROI of Software Process Improvement: Metrics for Project Managers and Software Engineers*. J. Ross Publishing, Inc., 2004.

[5] B. Totty D. Gourley. *HTTP: The Definitive Guide*. O'Reilly Media, 1 edition, 2002.

[6] R. Daigneau. *Service Design Patterns: Fundamental Design Solutions for SOAP WSDL and RESTful Web Services*. Addison Wesley Professional, 1 edition, 2011.

[7] P. Francis. Network address translation (nat). *ACM SIGCOMM Computer Communication Review*, vol. 45:str. 50–50, april 2015.

[8] A. Helland. Using azure functions as a lightweight api gateway. Dosegljivo: `https://blogs.msdn.microsoft.com/azuredev/2017/`

03/14/using-azure-functions-as-a-lightweight-api-gateway/, 2017. [Dostopano: 12. 9. 2018].

[9] M. B. Jurič. Kumuluzee. Dosegljivo: https://github.com/kumuluz/kumuluzee/wiki, 2017. [Dostopano: 14. 9. 2018].

[10] J. L. Oliveira L. A. Bastião Silva, C. Costa. A common api for delivering services over multi-vendor cloud resources. *Journal of Systems and Software*, vol. 86(9):str. 2309–2317, 2013.

[11] M. Macero. *Learn Microservices with Spring Boot: A Practical Approach to RESTful Services using RabbitMQ, Eureka, Ribbon, Zuul and Cucumber*. Apress, 1 edition, 2017.

[12] U. Marovt. Zasnova skalabilnega in visoko dostopnega prehoda za upravljanje programskih vmesnikov, 2016.

[13] V. Mihalcea. A beginner's guide to cache synchronization strategies. Dosegljivo: https://vladmihalcea.com/a-beginners-guide-to-cache-synchronization-strategies/, 2018. [Dostopano: 13. 9. 2018].

[14] G. Motroc. Microservices in java: Yay or nay? Dosegljivo: https://jaxenter.com/devops-speakers-interview-microservices-141183.html, 2018. [Dostopano: 13. 9. 2018].

[15] S. O'Grady. The redmonk programming language rankings: June 2018. Dosegljivo: https://redmonk.com/sogrady/2018/08/10/language-rankings-6-18/, 2018. [Dostopano: 13. 9. 2018].

[16] C. Richardson. Service discovery in a microservices architecture. Dosegljivo: https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/, 2015. [Dostopano: 3. 9. 2018].

[17] M. Rouse. single point of failure (spof). Dosegljivo: https://searchdatacenter.techtarget.com/definition/Single-point-of-failure-SPOF, 2012. [Dostopano: 30. 8. 2018].

[18] S. Longo T. Jacobs. A study of caching strategies for web service discovery. In *2015 IEEE International Conference on Web Services*, pages str. 464–471, junij 2015.

[19] M. P. Robillard W. Maalej. Patterns of knowledge in api reference documentation. *IEEE Transactions on Software Engineering*, vol. 39:str. 1264–1282, september 2013.

[20] L. Warusawithana. Major cloud computing vendors. Dosegljivo: https://medium.com/@lakwarus/micro-api-gateway-58cce43f2d7d, 2018. [Dostopano: 12. 9. 2018].

[21] W. A. Halang X. Wang. *Discovery and Selection of Semantic Web Services*. Studies in Computational Intelligence 453. Springer-Verlag Berlin Heidelberg, 1 edition, 2013.

[22] Apiman. Dosegljivo: http://www.apiman.io/latest/. [Dostopano: 30. 8. 2018].

[23] Cache eviction algorithms. Dosegljivo: http://www.ehcache.org/documentation/2.7/apis/cache-eviction-algorithms.html. [Dostopano: 13. 9. 2018].

[24] Etcd overview. Dosegljivo: https://coreos.com/etcd/. [Dostopano: 7. 9. 2018].

[25] Git. Dosegljivo: https://git-scm.com/. [Dostopano: 14. 9. 2018].

[26] Java ee. Dosegljivo: https://www.oracle.com/technetwork/java/javaee/overview/index.html. [Dostopano: 7. 9. 2018].

[27] Keycloak. Dosegljivo: https://www.keycloak.org/. [Dostopano: 7. 9. 2018].

[28] Kumuluzee discovery. Dosegljivo: https://github.com/kumuluz/kumuluzee-discovery. [Dostopano: 7. 9. 2018].

[29] The raft consensus algorithm. Dosegljivo: https://raft.github.io/. [Dostopano: 7. 9. 2018].

[30] What is docker? Dosegljivo: https://opensource.com/resources/what-docker. [Dostopano: 8. 9. 2018].

[31] The api management playbook understanding solutions for api management. Dosegljivo: https://www.ca.com/content/dam/ca/us/files/ebook/the-api-management-playbook.pdf, 2015. [Dostopano: 11. 9. 2018].

[32] Api management concepts. Dosegljivo: https://www.ibm.com/support/knowledgecenter/en/SSWHYP_4.0.0/com.ibm.apimgmt.overview.doc/overview_apimgmt_about.html, 2016. [Dostopano: 28. 8. 2018].

[33] Api management concepts. Dosegljivo: https://blog.finjan.com/blacklisting-vs-whitelisting-understanding-the-security-benefits-of-each/, 2017. [Dostopano: 29. 8. 2018].

[34] Apache jmeter. Dosegljivo: https://jmeter.apache.org/, 2018. [Dostopano: 14. 9. 2018].

[35] Kubernetes. Dosegljivo: https://kubernetes.io/, 2018. [Dostopano: 11. 9. 2018].

[36] Maven. Dosegljivo: https://maven.apache.org/, 2018. [Dostopano: 14. 9. 2018].

[37] Overview of docker compose. Dosegljivo: https://docs.docker.com/compose/overview/, 2018. [Dostopano: 8. 9. 2018].

[38] The raft consensus algorithm. Dosegljivo: https://www.elastic.co/, 2018. [Dostopano: 7. 9. 2018].