Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Blaž Artač

Vpeljava mikrostoritev v Java EE aplikacije

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM
PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: prof. dr. Viljan Mahnič

Ljubljana, 2016

Rezultati diplomskega dela so intelektualna lastnina avtorja. Za objavljanje ali izkoriščanje rezultatov diplomskega dela je potrebno pisno soglasje avtorja, Fakultete za računalništvo in informatiko ter mentorja.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Vpeljava mikrostoritev v Java EE aplikacije

Tematika naloge:

Proučite koncept mikrostoritev ter analizirajte njihove prednosti in slabosti v primerjavi z monolitnimi aplikacijami. Predstavite arhitekturo mikrostoritev, obstoječa ogrodja in aplikacijske knjižnice za njihovo uporabo v Javi ter primere uspešne uvedbe mikrostoritev v svetu. Pridobljeno znanje uporabite na praktičnem primeru pretvorbe obstoječe monolitne aplikacije v sistem mikrostoritev.

Rad bi se zahvalil materi in očetu za vzpodbudo in podporo tekom mojega študija, podjetju Medius, da mi je omogočilo pisanje diplomske naloge in priskrbelo zanimivo temo ter mentorju prof. dr. Viljanu Mahniču za mentorstvo in nasvete tekom pisanja diplomske naloge.

Kazalo

Povzetek

Abstract

1 Uvod **1**

2 Monoliti in mikrostoritve **5**

2.1 Monoliti 6

2.2 Mikrostoritve 9

3 Arhitektura mikrostoritev **17**

3.1 Komunikacija med mikrostoritvami 17

3.1.1 Sinhrona 18

3.1.2 Asinhrona 19

3.2 Hranjenje podatkov 21

3.3 Arhitekturni vzorci 27

3.4 Zasnova celotnega sistema 32

3.4.1 Mikrostoritvena arhitektura z monolitnim jedrom . . . 32

3.4.2 Popolnoma mikrostoritvena arhitektura 34

4 Razvoj mikrostoritev **35**

4.1 Šasije 36

4.1.1 Spring Boot 37

4.1.2 Dropwizard 39

4.1.3 Wildfly Swarm 40

4.1.4 KumuluzEE . . . 40
4.2 Odkrivanje storitev . . . 41
4.2.1 Eureka . . . 45
4.2.2 ZooKeeper . . . 46
4.2.3 Consul . . . 47
4.3 Izenačevanje obremenitve . . . 48
4.3.1 Ribbon . . . 49
4.3.2 AWS Elastic Load Balancer . . . 50
4.4 Toleranca okvar (koncept odklopnika) . . . 50

5 Implementacija **53**
5.1 Kdaj je pravi čas za prehod? . . . 53
5.2 Uspešni primeri po svetu . . . 55
5.2.1 Primer mikrostoritev: Netflix . . . 55
5.2.2 Primer monolitne aplikacije: ETSY . . . 56
5.3 Primer preobrazbe dela obstoječega monolita v sistem mikrostoritev . . . 57
5.3.1 Obstoječa monolitna aplikacija . . . 58
5.3.2 Funkcionalna razgradnja dela monolitne aplikacije v mikrostoritve . . . 61
5.3.3 Tehnična implementacija . . . 68

6 Sklepne ugotovitve **73**

Literatura **83**

Seznam uporabljenih kratic

kratica	angleško	slovensko
ACID	Atomicity, Consistency, Isolation, Durability	(atomarnost, konsistentnost, izolacija, trajnost) lastnost baze podatkov, ki omogoča, da se vsaka transakcija v bazi izvede zanesljivo
Amazon S3	Amazon Simple Storage Service	Amazonova internetna shramba podatkov
API	Application Programming Interface	(aplikacijski programski vmesnik)
AWS	Amazon Web Services	(Amazonove spletne storitve) oblačna spletna storitev, ki ponuja rešitve za oblačno gostovanje
BASE	Basically Available, Soft-state, Eventual consistency	(osnovna razpoložljivost, mehko stanje, sočasno skladnost) lastnost baz podatkov, ki daje prednost razpoložljivosti v zameno za konsistentnost podatkov
CAP	Consistency, Availability, Partition tolerance	(konsistentnost, razpoložljivost, odpornost na delitve) Teorem, ki trdi, da porazdeljen sistem lahko zagotavlja največ dve omenjeni lastnosti hkrati

CDI	Context and Dependency Injection	(injiciranje odvisnosti) načrtovalski vzorec, ki dovoljuje izbiro komponente v času izvajanja programa
CLOB	Character Large OBject	polje v bazi podatkov, namenjeno shranjevanju velikih podatkov
CQRS	Command Query Responsibility Segregation	načrtovalski vzorec, ki predvideva uporabo različnih modelov za branje in pisanje v bazi podatkov
DDD	Domain-driven design	(domensko usmerjeno načrtovanje) je pristop k načrtovanju programske opreme na osnovi problemske domene in domenske logike
EAR	Enterprise Application Archive	(poslovna aplikacijska arhivska datoteka)
EC2	Elastic Compute Cloud	Amazonova storitev, ki ponuja prilagodljivo računsko moč v oblaku
EL	(Java) Expression Language	programski jezik za vključevanje programske logike v spletne strani JSF in JSP
EJB	Enterprise Java Bean	(strežniška javanska komponenta)
HTTP	HyperText Transfer Protocol	(hipertekstovni prenosni protokol) protokol za komunikacijo med odjemalcem in strežnikom na svetovnem spletu
IDE	Integrated Development Environment	(integrirano razvojno okolje)
IP	Internet Protocol	(internetni protokol)
IT	Information Technology	(informacijska tehnologija)
JAR	Java Archive	(javanska arhivska datoteka) datoteka, ki združuje vsebinsko povezane razrede java

Java EE	Java Enterprise Edition	(java, poslovna izdaja) javanska platforma namenjena delovanju v zahtevnejših informacijskih sistemih
Java SE	Java Software Edition	(java, programska izdaja)
JAX-RS	Java API for RESTful Services	komponenta java EE za razvoj spletnih storitev REST
JPA	Java Persistence API	komponenta java EE za abstrakcijo in komunikacijo s podatkovno bazo
JSF	JavaServer Faces	tehnologija znotraj java EE, ki skrbi za izris predlog v HTML glede na podatke (podobno JSP, vendar prilagojeno trendom v razvoju spletnih aplikacij)
JSON	Javascript Object Notation	(javascript objektna notacija) format podatkov za izmenjavo nestrukturiranih podatkov na spletu
JSON-P	JSON with Padding	knjižnica za izmenjavo in dostop do podatkov drugih spletne strani
JSP	Java Server Pages	tehnologija znotraj java EE, ki skrbi za izris predlog v HTML glede na podatke
JVM	Java Virtual Machine	(javanski navidezni stroj)
NoSQL	Not Only SQL	ne-relacijska baza podatkov (predvidoma ima BASE lastnosti)
PHP	Hypertext Preprocessor	programski jezik za razvoj strežniških spletnih aplikacij
REST	Representational State Transfer	(predstavitvena arhitektura za prenos podatkov) (sinhron) komunikacijski protokol za izmenjavo podatkov med klientom in strežnikom

RI	Reference Implementation	(referenčna implementacija) referenčna implementacija standardne java specifikacije, ki naj bi služila kot zgled ostalim implementacijam
RPC	Remote Procedure Call	protokol, ki omogoča odjemalcu oddaljen klic procedure ali metode, ki se izvede na strežniku
SOA	Service Oriented Architecture	(storitveno usmerjena arhitektura)
SQL	Structured Query Language	(strukturiran povpraševalni jezik) za delo z relacijskimi podatkovnimi bazami
SRP	Single Responsibility Principle	(princip enojne odgovornosti)
UV	Uporabniški vmesnik	
WAR	Web Application Archive	(spletna aplikacijska arhivska datoteka)
XML	EXtensible Markup Language	(razširljiv označevalni jezik) format podatkov za izmenjavo strukturiranih dokumentov v spletu

Povzetek

Naslov: Vpeljava mikrostoritev v Java EE aplikacije

Avtor: Blaž Artač

Zahtevnost (poslovnih) aplikacij se povečuje dnevno. Aplikacije morajo biti skalabilne, delovati na več platformah hkrati (splet, pametni telefoni ...), se povezovati in integrirati z zunanjimi storitvami, obdelovati velike količine podatkov v kratkem času, biti prilagojene za delovanje v oblaku ... Kljub novim izzivom pa se razvoja takih aplikacij še vedno lotevamo na monoliten način, ki postaja čedalje manj primeren za sodobno, hitro rastoče (oblačno) okolje. Kot odgovor na to so se pojavile mikrostoritve, ki obljubljajo rešitev, vendar hkrati skrivajo veliko pasti. V diplomski nalogi primerjamo oba načina razvoja aplikacij in pokažemo, kdaj je primerneje uporabiti enega in drugega. Podrobneje se usmerimo v razvoj mikrostoritev ter predstavimo koncepte in orodja, ki nam lahko pri tem pomagajo. Prikažemo različne načine za vpeljavo mikrostoritev v javanske aplikacije in na koncu enega izmed njih uporabimo za preoblikovanje obstoječe monolitne aplikacije v ekvivalentno rešitev z mikrostoritvami.

Ključne besede: mikrostoritve, java, mikrostoritvene šasije, skalabilnost, odkrivanje storitev, monolitne aplikacije, asinhrona in sinhrona komunikacija, samozadostni izvršljivi JAR.

Abstract

Title: Introducing microservices into Java EE applications

Author: Blaž Artač

Complexity of (enterprise) applications and software is increasing daily. Applications are required to be scalable, to operate simultaneously on different platforms (web, mobile . . .), to connect and integrate with external services, process large amounts of data in short time, to work in the cloud . . . Despite new challenges, the development of this kind of applications is still being resolved in a monolithic manner, which is becoming less and less suitable for modern, quickly growing (cloud) environment. Microservices try to address this challenges, but while providing certain solutions they also present new problems. In this thesis both styles are compared and it is shown when one is more appropriate for use than the other one. More specifically, focus is given on development of microservices and concepts and tools of trade, that can help along the way. Different ways of introducing microservices in Java applications are presented, according to application requirements, and one of them is used to transform existing Java monolithic application to microservices.

Keywords: microservices, Java, microservice chassis, scalability, service discovery, monolithic applications, synchronous and asynchronous communication, fat jar.

Poglavje 1

Uvod

Pojem mikrostoritve [31] je razmeroma nov, saj se je prvič pojavil pred 3 leti, vendar nas kot arhitektura računalniškega sistema spremlja že vsaj od začetka razvojev Unixa, ki temelji na arhitekturi storitev (npr. "grep" je primer posamezne (mikro)storitve v Unixu). Mikrostoritev opravlja samo eno nalogo, vendar jo opravi odlično. Kljub temu je do pred par leti le redko kdo razvijal poslovne rešitve v duhu mikrostoritev. Java svet sicer pozna koncept SOA (Service Oriented Architecture), ki pa se od mikrostoritev v marsičem razlikuje [27].

Razbitje java aplikacije na več mikrostoritev ima svoje prednosti in slabosti. Vsekakor to ni univerzalna rešitev za morebitne težave pri trenutno prevladujočem, monolitnem načinu razvoja (poslovnih) aplikacij. Med prednosti lahko štejemo skalabilnost, prilagodljivost, hitrost, ohlapno sklopljenost, hitrejši razvoj in neodvisnost uporabljene tehnologije (podatkovna baza, programski jezik ...) od ostalih mikrostoritev. Slabosti mikrostoritev so med drugim zahtevnejše upravljanje in testiranje, odvisnost od delovanja omrežja ter podvajanje kode in truda. V bolj zahtevnih poslovnih aplikacijah, ki med drugim zahtevajo revizijsko sled, se znajdemo še pred izzivom sledenja in beleženja (asinhronih) klicev med mikrostoritvami.

Spremeni se tudi način pakiranja projektov - do sedaj smo projekte monolitnih poslovnih aplikacij pakirali kot WAR (Web Application Archive)

in EAR (Enterprise Application Archive) ter jih namestili na aplikacijski strežnik (Jboss, Tomcat ...). S tem smo dobili velike datoteke (tudi 150 in več megabajtov), vendar vseh funkcionalnosti, ki nam jih je tako zapakiran projekt ponudil, ne potrebujemo vedno. Pri mikrostoritvah je v ospredje stopil samozadostni izvršljivi JAR (v angleščini poznamo pod imeni *self-sufficient executable JAR*, *uber jar* in *fat jar*) - v en JAR zapakiramo vse potrebno za zagon posamezne storitve (tako programske razrede kot vse odvisnosti, ki so potrebne za delovanje teh razredov) [46]. Zagon takšnih JARov je tudi do 10x hitrejši od običajne namestitve aplikacijski strežnik. S tem dodatno pohitrimo sam razvoj.

Izzivi pa niso le tehnološki. Razvoj mikrostoritev zahteva tudi boljšo komunikacijo znotraj podjetja. Pomembni so dogovori med razvijalci, ki določijo abstraktne vmesnike storitev z izpostavljenimi končnimi točkami, ki se ne smejo spreminjati v produkciji, saj lahko drugače podremo storitve, ki kličejo našo storitev. Odvisno od načina komunikacije med storitvami se mora novemu stilu prilagoditi celotno podjetje (kot primer vzemimo komunikacijo preko sporočilnih vrst, ki zahteva drugačno razmišljanje razvijalcev kot komunikacija preko protokola REST). Poleg tega mikrostoritve delujejo v omrežju naprav, kar predstavlja dodatne težave - odkrivanje storitev (angl. *service discovery*) [47], povezovanje med njimi, napake v omrežju, zahtevnost upravljanja . . . Posledično je potrebno razvijati z neuspehom v mislih (angl. *design for failure*) [31].

Zaradi relativne mladosti mikrostoritev v java svetu na trgu ni veliko orodij, ki bi omogočala naprednejše upravljanje z mikrostoritvami, predvsem v produkciji. Tukaj so mišljene mikrostoritvene šasije, orodja za odkrivanje storitev, odpornost na izpade (angl. *fault tolerance*), nadzor in izenačevanje obremenitve (angl. *load balancing*) [13]. Rezultat tega je razmeroma majhno število aplikacij s kompleksno poslovno logiko, ki delujejo kot sistem mikrostoritev. Vendar lahko v bližnji prihodnosti pričakujemo porast in večjo zrelost potrebnih ogrodij in orodij.

Vidimo lahko, da je pred prehodom na mikrostoritve potreben teme-

ljit premislek. Največja napaka je brezglavo sledenje novim trendom, brez upoštevanja zgornjih (in ostalih) vidikov. V praksi najdemo priporočila, da se najprej razvije klasična, monolitna aplikacija, ki jo potem kasneje postopoma razbijemo na več mikrostoritev. Na ta način se izognemo prezgodnji optimizaciji (lahko da bo monolitska aplikacija povsem dovolj zmogljiva glede na potrebe) in imamo rezervno rešitev v primeru težav pri implementaciji mikrostoritev.

V nadaljevanju bomo podrobneje predstavili sam koncept mikrostoritev, argumentirali prednosti in slabosti, predstavili obstoječa ogrodja in aplikacijske knjižnice za mikrostoritve v Javi, omenili uspešne primere po svetu (Netflix) in preoblikovali del obstoječe monolitne aplikacije v sistem mikrostoritev. Dotaknili se bomo tudi arhitekture mikrostoritev in hranjenja ter izmenjave podatkov med njimi. Je pa področje mikrostoritev zelo obsežno in ga težko zajamemo že v daljši knjigi, kaj šele v diplomskem delu.

Poglavje 2

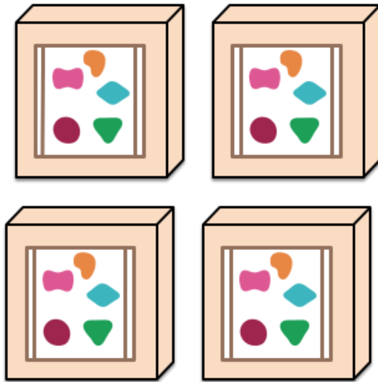
Monoliti in mikrostoritve

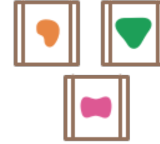
Zahteve in kompleksnost poslovnih aplikacij se povečuje dnevno. Aplikacije morajo biti skalabilne, neprestano razpoložljive, odporne na napake, delovati morajo na več platformah (prenosni in tablični računalniki, pametni telefoni ...), se povezovati z drugimi storitvami, obdelovati velike količine podatkov v kratkem času in še bi lahko naštevali. Posledično obstoječ, monoliten način razvoja javanskih aplikacij, ki je skalabilen samo do določene mere, postaja čedalje manj primeren za sodobno, hitro rastoče oblačno okolje.

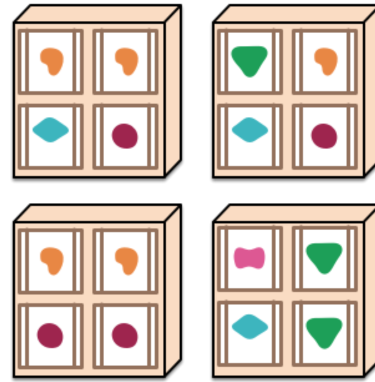
V zadnjih letih se je tako trend obrnil k mikrostoritvam. Velik del java sveta v zadnjem času govori o mikrostoritvah ter njihovih prednostih in slabostih. Nekateri jih hvalijo, drugi kritizirajo. Vendar bolj kot sam koncept je k porastu mikrostoritev veliko pripomoglo okolje, ki je zaradi visoke stopnje razvoja sprožilo potrebo po njih. Govor je predvsem o oblačnih rešitvah in ostalih zahtevah iz prejšnjega odstavka. Grobo primerjavo med mikrostoritvami in monoliti vidimo na sliki 2.1.

Je pa pred prehodom na mikrostoritve potreben temeljit premislek, saj veliko aplikacij ni primernih za tak korak. S prehodom na mikrostoritve se pojavi novo področje, ki zahteva našo pozornost - upravljanje, nadzorovanje in povezovanje mikrostoritev. Tako preložimo delo iz razvijalcev na sistemske inženirje [24].

Tekom tega poglavja bomo razložili, kaj so monoliti in kaj mikrostoritve







Slika 2.1: Na sliki vidimo razliko med monolitno (leva stran) in mikrostoritveno (desna stran) arhitekturo. Pri monolitu imamo začetno aplikacijo, ki jo ob povečanju prometa namestimo na dodatne strežniške instance. Pri mikrostoritvah pa imamo več gradnikov aplikacije in lahko vsakega posebej v poljubnem številu nameščamo na dodatne instance strežnikov ob povečanem prometu.

ter spoznali prednosti in slabosti obeh pristopov, kar nam bo kasneje pomagalo pri razumevanju različnih načinov komunikacije, ogrodij, vzorcev in orodij, ki se uporabljajo za razvoj mikrostoritev.

2.1 Monoliti

"*The design philosophy (*op. p.: *of a monolithic application) is that the application is responsible not just for a particular task, but can perform every step needed to complete a particular function.*" [14]

Monolitne aplikacije v Javi poznamo že od njenih začetkov. So de facto standard javanskega razvoja zadnjih 20 ali več let. V monolitni aplikaciji zapakiramo celoten projekt (uporabniški vmesnik, poslovna logika, dostop do baze, varnost ...) v en velik arhiv (WAR ali EAR), ki ga lahko namestimo na aplikacijski strežnik in se izvaja v svojem procesu ali instanci. Če želimo slediti arhitekturnim praksam, bomo monolit razdelili na več modulov in s tem logično razčlenili aplikacijo na več manjših delov. Tako dobimo modularen monolit, vendar se moramo zavedati, da je to logična oziroma poslovna modularnost in ne smemo razumeti posameznega modula kot mikrostoritve. Namen modularnosti je razčlenitev monolitne aplikacije v (poslovno) sorodne sklope, s čimer se poenostavi razvoj in razumevanje aplikacije. So pa še vedno vsi moduli znotraj istega projekta in se vsi skupaj izvajajo v enem procesu.

Če in ko se bodo pojavile potrebe po večji zmogljivosti aplikacije, jo bomo skalirali z dodajanjem novih strežniških instanc na katerih bo tekla aplikacija. Mogoče bomo ugotovili, da je povečana uporaba samo ene funkcionalnosti naše aplikacije, vendar druge rešitve ni - skalirati je potrebno celotno aplikacijo, četudi je preobremenjen samo njen najmanjši del.

Sčasoma bo naša monolitna aplikacija zrasla, dodale se bodo nove komponente, odstranile stare, razvijalci se bodo zamenjali (izguba znanja o aplikaciji), nabral se bo tehnični dolg in dobili bomo zapleten, velik in redkokomu razumljiv monoliten projekt. Od tega trenutka naprej se stroški vzdrževanja in nadaljnega razvoja projekta skokovito povečajo. Vsaka, četudi najmanjša, sprememba v programski kodi zahteva ponovno namestitev aplikacije na aplikacijski strežnik, kar že samo po sebi (brez projekta) vzame 20 ali več sekund, če pa imamo velik in kompleksen projekt, lahko traja tudi po 10 minut. Posledično produktivnost razvijalcev strmo pade. Tako za vsako spremembo, katero bi ob prvotnem razvoju aplikacije implementirali za ceno X, sedaj porabimo tudi do desetkrat več (primerjava na sliki 2.2).

Seveda to ni usoda vseh monolitnih projektov - veliko aplikacij ni nikoli potrebno (ekstremno) skalirati, prav tako tudi nadgradnje niso vedno

potrebne. Vendar če pride do tega trenutka, spoznamo, da bi imeli velike koristi od mikrostoritvene zasnove naše aplikacije.

Spodaj povzamemo nekatere prednosti in slabosti monolitnih aplikacij. Potrebno se je zavedati, da to ni končen in univerzalen seznam - razvijalska skupnost ima različne poglede na prednosti in slabosti tako monolitne kot mikrostoritvene arhitekture.

Prednosti:

- **lažje testiranje** - testiramo znotraj enega projekta v IDE, kar omogoča večji pregled in sledljivost [38]
- **lažje razhroščevanje** - sledenje klicem funkcij poteka znotraj IDE [38]
- **lažje zaznavanje napak** - vzemimo za primer spremembo imena metode (argumenti ostanejo isti). Z uporabo IDE lahko spremenimo ime metode in vse klice na to metodo v enem koraku. Če smo slučajno kje pozabili spremeniti ime, nas bo na to opozoril IDE, še pred prevajanjem izvorne kode
- **hitrejši (začetni) razvoj** - večina IDE okolij je prilagojena za razvoj monolitnih aplikacij [38]

Pomanjkljivosti:

- **tehnični dolg** - v kolikor ne skrbimo za programsko kodo aplikacije bo rasel tehnični dolg. Tehnični dolg se pojavi, ko namesto najboljše rešitve uporabimo najhitrejšo ali najlažjo [38]
- **kompleksnost aplikacije** - na določeni točki postane razvoj, testiranje in razhroščevanje zelo zapleteno (ni mogoče pričakovati, da bo en razvijalecrazumel celotno aplikacijo) [44]
- **dolgotrajno nameščanje na aplikacijski strežnik** - z rastjo monolita se podaljša čas, potreben za nameščanje na aplikacijski strežnik,

zaradi velikosti samega arhiva in njegovih odvisnosti (nekateri monoliti se zaganjajo tudi po 10 minut) [38, 44]

- **dolgoročna zaobljuba določeni tehnologiji** - vse tehnologije niso primerne za vse naloge. Kot primer vzemimo procesiranje signalov, ki je hitrejše v programskih jezikih Python ali C, medtem ko tega ne moremo izkoristiti, saj naša aplikacija temelji na java EE [38, 44]

2.2 Mikrostoritve

"*the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms*" (Fowler, 2014) [1]

V nasprotju z monolitnimi aplikacijami so mikrostoritve v svetu Jave precej nov koncept. Pojem mikrostoritve se je prvič pojavil približno 5 let nazaj [1], od takrat zanimanje za mikrostoritve stalno narašča. Je pa dejstvo, da to ni nov koncept v svetu IT. Prve primere najdemo že v UNIXu pred 40 leti, v Javi pa na začetku novega tisočletja, ko se je pojavila SOA. Mikrostoritve so povzele nekaj konceptov SOA, od nje se razlikujejo predvsem v načinu komunikacije (SOA uporablja poslovno podatkovno vodilo) in upravljanju (SOA je centralno upravljana).

Namesto podajanja standardne definicije mikrostoritev, ki je razvijalska skupnost še ni formirala, raje omenimo principe in koncepte, na katerih slonijo mikrostoritve (domensko usmerjeno načrtovanje, enojna odgovornost, ohlapna sklopljenost, poliglotno shranjevanje podatkov [29] ...). Brez upoštevanja naštetega bomo težko zgradili neodvisne in skalabilne mikrostoritve.

Prav tako je potrebno poudariti, da pridevnik mikro ni mišljen za število vrstic ali razredov v mikrostoritvi, temveč se osredotoča na funkcionalnost (odvisno od aplikacije lahko tudi poslovno logiko), saj mora mikrostoritev opravljati samo eno nalogo/funkcijo, vendar to odlično. Za boljšo predstavo

vzemimo dve mikrostoritvi: prvi podamo niz in nam vrne niz s povečanimi črkami. Drugi podamo sliko in jo izostri z uporabo posebnih algoritmov. Oba primera štejeta kot mikrostoritev, le da je prva, iz stališča programske kode, zelo kratka (primer v Javi se nahaja spodaj) medtem ko je druga kompleksnejša, saj procesiranje slike zaheva več razredov in odvisnosti do drugih knjižnic (primer izpuščen zaradi velikosti).

```
@Path("/")
public class PovecajVseCrke {
        @GET
        @Path("/povecajVseCrke/{niz}")
        @Produces(MediaType.TEXT_PLAIN)
        @Consumes(MediaType.TEXT_PLAIN)
        public String povecajVseCrke(@PathParam("niz")
                    String string) {
                return string.toUpperCase();
        }
}
```

Mikrostoritve posegajo tudi v organizacijsko strukturo podjetja ali obratno: organizacijska struktura podjetja se (bo) pozna(la) v strukturi mikrostoritev, kot pravi Conway (1968): "*organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations*" [25].

V monolitnih projektih je delitev razvijalcev na projektu večinoma odvisna od njihovega področja (uporabniški vmesnik, zaledni sistem, baza podatkov, testiranje ...) in glede na ta področja se tudi formirajo ekipe. Nasprotno je ob razvoju mikrostoritev priporočeno, da se za posamezno mikrostoritev formira ekipa, ki bo skrbela za celoten življenski cikel mikrostoritve: od razvoja, testiranja do zagona v produkciji in vzdrževanja. Velik zagovornik tega pristopa je podjetje Amazon, ki mu je tudi dalo ime: "You build it, you run it." [35]. V prednosti in slabosti takega pristopa se ne bomo podrobneje spuščali.

Lastnosti:

- **domensko usmerjeno načrtovanje (DDD)** - pristop k načrtovanju programske opreme na osnovi problemske domene in domenske logike [38]
- **princip enojne odgovornosti (SRP)** - modul ali razred mora biti odgovoren za eno funkcionalnosti aplikacije in vsa ta funkcionalnost naj bo zajeta v danem modulu ali razredu [38]
- **jasno definirani vmesniki** (angl. *explicitly published interfaces*) - mikrostoritve med seboj komunicirajo preko jasno definiranih vmesnikov (pravil, pogodb, struktur), ki se ne smejo spreminjati oziroma morajo biti obratno združljivi (angl. *backward compatible*) [38]
- **decentralizirano upravljanje s podatki** - vsaka mikrostorite ima svojo bazo podatkov, ki načeloma ni (neposredno) dostopna drugim mikrostoritvam [1]
- **pametne vstopne točke in neumne povezave** - funkcionalnost se skriva v vmesnikih, medtem ko je povezava med mikrostoritvami le prenosna ali pa vsebuje osnovno usmerjanje [1]
- **preprosta komunikacija** - sinhrona komunikacija preko sinhronih komunikacijskih protokolov (REST . . .) ali asinhrona s pomočjo sporočilnih vrst [38]
- **odpornost na izpade** (angl. *designed for failure*) - programska koda in arhitektura mikrostoritev morata predvidevati začasne izpade v omrežju [1]
- **celostna rešitev** (angl. *full stack*) - posamezna mikrostoritev vsebuje vse od baze podatkov, poslovne logike do uporabniškega vmesnika

Prednosti:

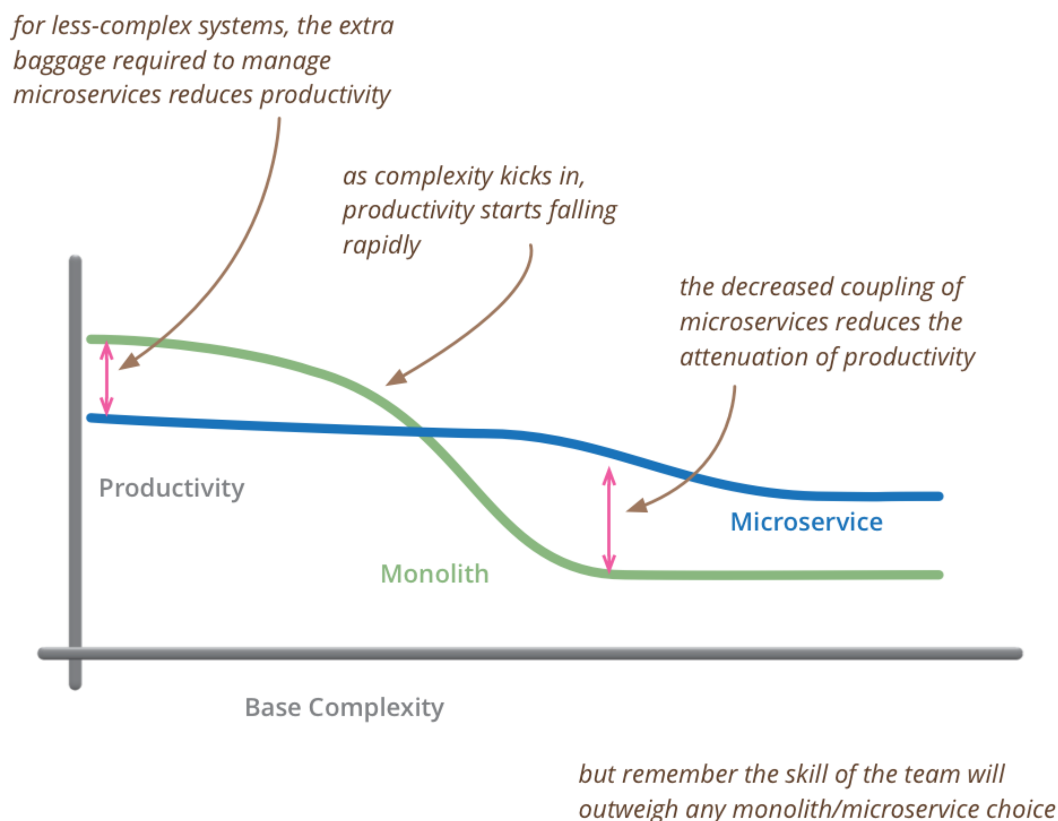
- **manjša problemska domena poveča razumevanje** - ker mikrostoritev zagotavlja samo eno funkcionalnost jo je lažje razumeti [38]
- **skalabilnost** (angl. *scalability*) - posamezne mikrostoritve lahko v realnem času skaliramo neodvisno od drugih, tako navzgor kot navzdol (ko se povečuje ali zmanjšuje obremenjenost) brez omejitev [38, 34]
- **neodvisno nameščanje na aplikacijski strežnik, nadgradnja in zamenjava** - sprememba v mikrostoritvi zahteva ponovno nameščanje samo te mikrostoritve, tudi zamenjava celotne mikrostoritve je preprostejša kot zamenjava dela monolita. Celotno nameščanje zaradi manjše skupne velikosti mikrostoritve poteka hitreje kot pri monolitu [38]
- **manjše ekipe** - izboljša se komunikacija, razumevanje delovanja in nalog mikrostoritve, poveča se odgovornost razvijalcev [44]
- **izolacija napak** - v primeru izpada ene mikrostoritve, bodo ostale še vedno delovale (v monolitni aplikaciji lahko napaka povzroči izpad celotne aplikacije) [38]
- **organiziranost okoli posameznih razmejenih poslovnih domen** - podobno kot princip enojne odgovornosti, mikrostoritev je zadolžena za posamezno poslovno domeno, posledično je lažje vzdrževanje in nadgradnja mikrostoritve [1]
- **ponovna uporaba** - posamezno mikrostoritev lahko kadarkoli uporabimo v drugi aplikaciji
- **(dolgoročno) nezavezujoča uporaba tehnologij** - za vsako mikrostoritev lahko uporabimo tehnologijo, ki je najbolj primerna za dan problem (čeprav so ostale mikrostoritve napisane v drugih tehnologijah), postopoma lahko nadgrajujemo knjižnice ali verzijo programskega jezika (npr. migriranje iz java SE 7 na java SE 8) [38, 34]
- **sprotna integracija in avtomatsko nameščanje na aplikacijski strežnik** (angl. *continuous integration and deployment*) - zaradi svoje

velikosti je mikrostoritve veliko lažje vključiti v proces sprotne integracije in avtomatskega nameščanja na aplikacijski strežnik [44]

Slabosti:

- **težje testiranje** - ob množici mikrostoritev se pokažejo težave z ustreznim (integracijskim) testiranjem (npr. ob testiranju posamezne mikrostoritve moramo prototipirati (angl. *mock*) klice na ostale mikrostoritve, saj želimo preveriti samo delovanje posamezne mikrostoritve, ki zato ne sme biti odvisna od razpoložljivosti drugih mikrostoritev) [56]
- **upravljanje mikrostoritev** - posledica porazdeljenosti in števila mikrostoritev je tudi težje upravljanje in nadzorovanje (če smo do sedaj upravljali tri projekte na aplikacijskih strežnikih, moramo sedaj upravljati več deset mikrostoritev) [56, 32]
- **zakasnitve v omrežju** - ker mikrostoritve komunicirajo preko omrežja, se soočamo z zakasnitvami (za primer vzamimo sinhron sistem, kjer klic na eno mikrostoritev traja okoli 100ms, vendar sproži verigo klicev na druge mikrostoritve, vsak klic doda okoli 100ms zakasnitve in na koncu lahko dobimo več sekund zakasnitve) [32]
- **sočasna skladnost** (angl. *eventual consistency*) - posledica uporabe ločenih baz podatkov za vsako mikrostoritev je začasno razhajanje med temi podatki, do njihove uskladitve oziroma osvežitve (ob registraciji uporabniškega računa uporabljamo mikrostoritev A, ki v svoji bazi naredi nov uporabniški račun, nato se hočemo nemudoma prijaviti, za kar skrbi mikrostoritev B, vendar B še ni procesiral informacij o novem uporabniškem računu od baze A, zato nam bo prijava na voljo šele čez nekaj sekund, ko se bodo podatki iz baze A prenesli v bazo B) [32]
- **podvojitev kode** - dve mikrostoritvi lahko uporabljata enak razred (npr. Jabolko), vendar, ker sta ločeni, moramo kopirati razred iz ene mikrostoritve v drugo (lahko bi sicer zapakirali razred Jabolko in ostale

skupne razrede v skupen JAR, vendar bi na tak način povečali sklopljenost med mikrostoritvama. Vsaka sprememba v skupnem JARu bi zahtevala ponovno nameščanje vseh mikrostoritev, ki uporabljajo skupni JAR, ali pa bi uporabljali starejše verzije skupnega JARa in tako povečali zapletenost sistema) [56]



Slika 2.2: Na sliki vidimo razliko med monolitno (leva stran) in mikrostoritveno (desna stran) arhitekturo. Pri monolitu imamo začetno aplikacijo, ki jo ob povečanju prometa namestimo na dodatne strežniške instance. Pri mikrostoritvah pa imamo več gradnikov aplikacije in lahko vsakega posebej v poljubnem številu nameščamo na dodatne instance strežnikov ob povečanem prometu.

Poglavje 3

Arhitektura mikrostoritev

3.1 Komunikacija med mikrostoritvami

V monolitnih aplikacijah nismo nikoli posvečali posebne pozornosti klicem funkcij in metod - njihova izvedba je bila samoumevna in hitra, znotraj enega procesa. Mikrostoritve pa tečejo vsaka v svojem procesu, ki se lahko nahaja na istem strežniku, v sosednji sobi ali celo na drugem koncu sveta. Za komunikacijo med njimi uporabljamo primerne komunikacijske protokole, vendar se moramo zavedati, da komunikacija preko omrežja ni vedno zanesljiva [26]. Zato moramo vzeti v zakup, da bo prihajalo do zakasnitev (manjših ali večjih) in da kakšen klic ne bo izveden, ker je prišlo do napake v omrežju. Tako se soočimo z novimi tveganji, ki niso neposredna posledica slabe programske kode ali arhitekture znotraj mikrostoritve, ampak omrežja, na katerega (načeloma) ne moremo vplivati.

Za komuniciranje med mikrostoritvami imamo na voljo dva načina komunikacije - sinhrono in asinhrono. Obe imata svoje prednosti in slabosti, načeloma pa se zaradi specifične narave mikrostoritev priporoča asinhrona komunikacija [40]. Je pa odvisno tudi od števnosti komunikacije - ena-proti-ena ali ena-proti-mnogo [47] (glej tabelo 3.1). Možna je tudi kombinacija obeh pristopov. Pojmi iz tabele so podrobneje razloženi v poglavju 3.1.2.

<table>
<tr><th></th><th>ena-proti-ena</th><th>ena-proti-mnogo</th></tr>
<tr><td>sinhrona komunikacija</td><td>zahteva/odgovor</td><td>–</td></tr>
<tr><td rowspan="2">asinhrona komunikacija</td><td>obvestilo</td><td>objavi/naroči</td></tr>
<tr><td>zahteva/asinhron odgovor</td><td>objavi/asinhron odgovor</td></tr>
</table>

Tabela 3.1: Priporočen način komunikacije med mikrostoritvami glede na števnost komunikacije

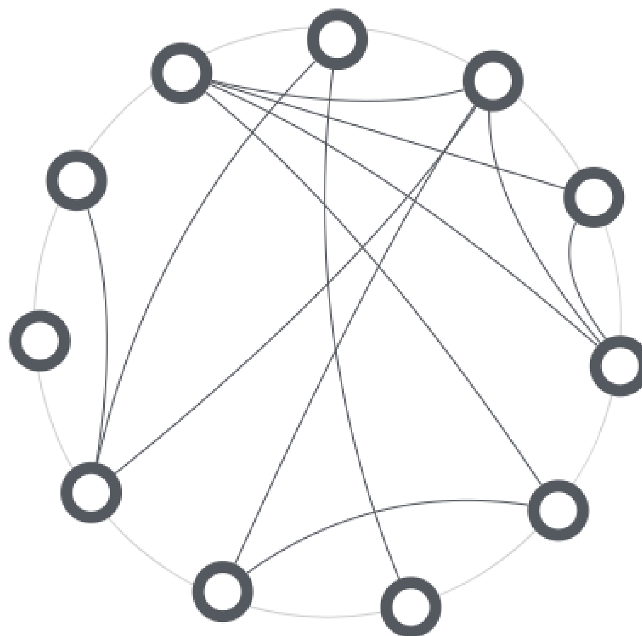
3.1.1 Sinhrona

Sinhrona komunikacija nam je znana že iz monolitnih aplikacij - klic metode in čakanje na njeno izvršitev (ter morebiten odgovor). Glavno pri sinhroni komunikaciji je, da ne nadaljujemo z izvajanjem programa dokler ne dobimo odgovora oziroma potrditve, da se je klicana metoda izvršila uspešno.

Dobre lastnosti tega pristopa so lažja implementacija (bolj poznan koncept) in razhroščevanje, konsistentnost informacij v sistemu ter potrditev izvršitve klicane storitve. Čeprav so te lastnosti zaželjene, nam lahko sinhronost v komunikaciji med mikrostoritvami povzroči več problemov kot koristi. Ob klicu druge storitve ni zagotovila, da ta storitev deluje/je dosegljiva. V tem primeru bo klicoča mikrostoritev zablokirala do trenutka, ko bo klicana storitev dosegljiva oziroma bo klicoča mikrostoritev dosegla časovno omejitev klica (v tem primeru nismo izvršili dane naloge in moramo (uporabniku) vrniti opozorilo o napaki). Tveganje za to se še poveča, ko imamo več zaporednih ali navzkrižnih klicev med storitvami, saj v tem primeru ena nedelujoča storitev posredno blokira vse ostale storitve, ki čakajo v verigi klicev. Ne smemo zanemariti tudi zakasnitev v omrežju, ki se v primeru več zaporednih ali navzkrižnih klicev med mikrostoritvami seštevajo (uporabniki dandanes pričakujejo hiter, skoraj takojšen odziv). Z direktnimi klici na mikrostoritve povzročimo tudi tesno sklopljenost sistema in odvisnost med posameznimi mikrostoritvami, kar lahko našo mikrostoritveno arhitekturo spremeni v porazdeljen monolit [40].

Uporaba sinhrone komunikacije je primernejša v nekompleksnih sistemih, kjer obstaja hierarhija klicev mikrostoritev (ni navzkrižnih klicev) in klic

posamezne mikrostoritve ne sproži (večje število) nadaljnih klicev na druge mikrostoritve, predvsem ne sme priti do navzkrižnega klicanja mikrostoritev. V nasprotnem primeru lahko pride do zapletenega sistema klicev (slika 3.1).



Slika 3.1: Slika prikazuje klicanje mikrostoritev med seboj v sinhronem sistemu mikrostoritev. Točke na krogu predstavljajo posamezne mikrostoritve, ki so navzkrižno povezane druga z drugo. Že na prvi pogled je opazna kompleksnost sistema. V primeru, da ena izmed teh mikrostoritev preneha delovati, je zaradi sinhronih klicev blokirana tudi večina ostalih storitev.

Mikrostoritve se lahko poslužijo različnih protokolov za sinhrono komunikacijo [47], med katerimi je najbolj poznan REST preko HTTP.

3.1.2 Asinhrona

Pri asinhroni komunikaciji mikrostoritev ne zahteva takojšnjega odgovora ali pa ga sploh ne zahteva. Na ta način se izognemo blokiranim mikrostoritvam in povečamo odpornost na zakasnitve v omrežju, saj mikrostoritev nadaljuje

svoje izvajanje nemudoma po klicu oziroma pošiljanju sporočila. Prav tako lažje komuniciramo z več storitvami naenkrat.

Asinhrona komunikacija zahteva drugačno arhitekturno in konceptualno zasnovo sistema ter programske kode. Mikrostoritve morajo biti zasnovane tako, da za nadaljevanje izvajanja ne čakajo na odgovor na klic oziroma sporočilo. S tem omejimo izpad posamezne mikrostoritve samo na to mikrostoritev in ne na celoten sistem (kot se zgodi v primeru sinhrone verige klicev).

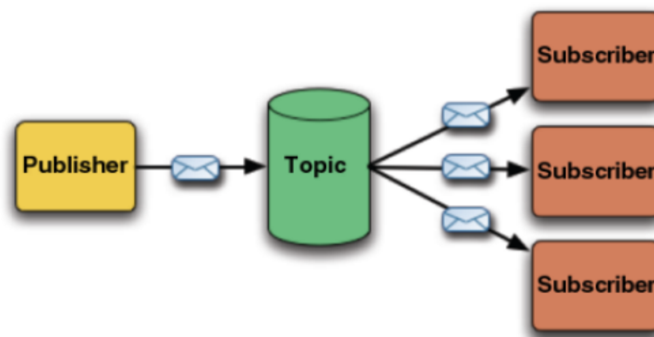
Prevladujejo trije asinhroni načini klicanja/obveščanja drugih storitev [47]:

- **obvestilo** (angl. *notification ali one-way request*) - obvestilo (angl. *notification ali one-way request*) - drugi storitvi pošljemo obvestilo in ne pričakujemo odgovor (v fizičnem svetu bi lahko to primerjali s pošiljanjem reklamnega materiala po pošti). Implementacija je preprosta, vendar z neposrednimi klici storitev povzročimo tesnejšo sklopljenost sistema.

- **zahteva/asinhron odgovor** (angl. *request/async response*) - klic druge storitve ne blokira izvajanje klicoče storitve, saj ta nadaljuje izvajanje in ne pričakuje takojšnjega odgovora. Ko odgovor prispe, ga klicoča funkcija obravnava. Kot pri obvestilu povzročimo tesnejšo sklopljenost sistema.

- **objavi/naroči** (angl. *publish-subscribe*) - mikrostoritev objavi sporočilo v sporočilno vrsto, na katero se lahko naročijo druge mikrostoritve in prejemajo vsa sporočila, ki so bila poslana v vrsto (slika 3.2). Ker se mikrostoritve ne zavedajo druga druge (komunicirajo le preko sporočilnih vrst) dobimo ohlapno sklopljen sistem. Možna je komunikacija z eno ali več mikrostoritvami naenkrat (prejeto sporočilo v sporočilno vrsto se kopira in pošlje vsem naročnikom naenkrat), prav tako lahko implementiramo asinhrone odgovore naročnikov. Dodatna lastnost sporočilnih vrst je hranjenje sporočil, ki niso bila dostavljena

naročnikom, in ponovno pošiljanje, ko je naročnik dosegljiv (s tem zagotovimo, da so vsa sporočila sčasoma obdelana).

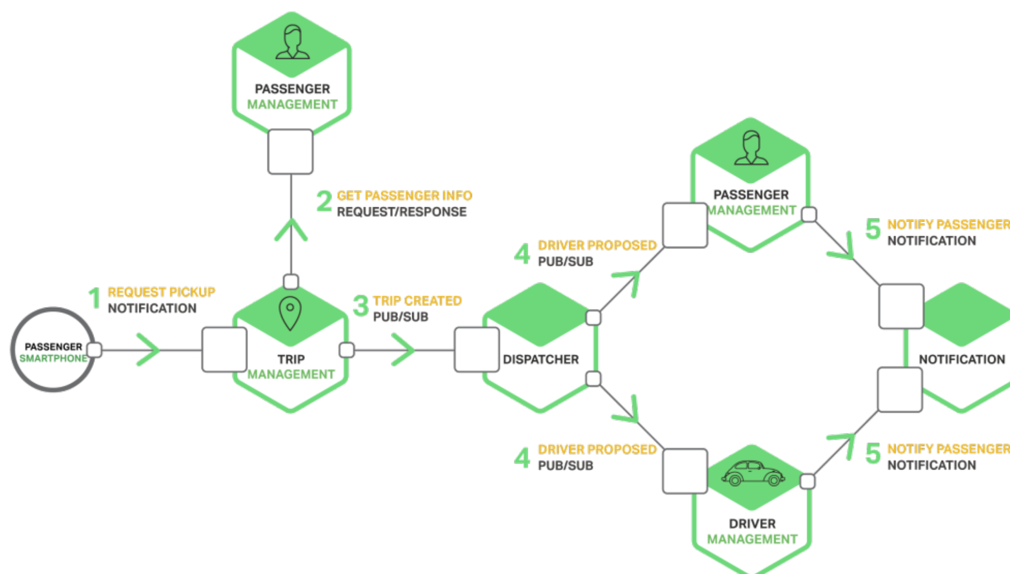


Slika 3.2: Primer komunikacije objavi/naroči preko sporočilne vrste (topic).

Komunikacija objavi/naroči je najbolj primeren način asinhrone komunikacije za mikrostoritve. V primeru kompleksnega sistema (primer na sliki 3.3) je priporočljivo, da večina komunikacije poteka na način objavi/naroči, z možnimi sinhronimi ali asinhronimi klici na robu sistema [47]. Posledica tega načina je sicer začasna nekonsistentnost podatkov v sistemu, kar pri nekaterih aplikacijah ni kritično, drugje pa je konsistentnost zahtevana in se moramo poslužiti ostalih načinov komunikacije, ki zagotavljajo konsistentnost podatkov (predvsem sinhrona komunikacija).

3.2 Hranjenje podatkov

Najbolj pogost način hranjena podatkov v monolitnih aplikacijah je uporaba enotne baze podatkov za celotno aplikacijo [1]. Tabele so med seboj povezane s primarnimi in sekundarnimi ključi, kar tudi izrazimo preko entitetnih razredov v aplikaciji. Razvijalec ima iz večine delov aplikacije dostop do celotne baze (v kodi lahko bere, spreminja in vstavlja po celotni bazi). Za monolitne aplikacije je taka hramba podatkov primerna in preprosta za implementacijo. Problem se zna pojaviti kasneje, ko je potrebno spremeniti tabelo v

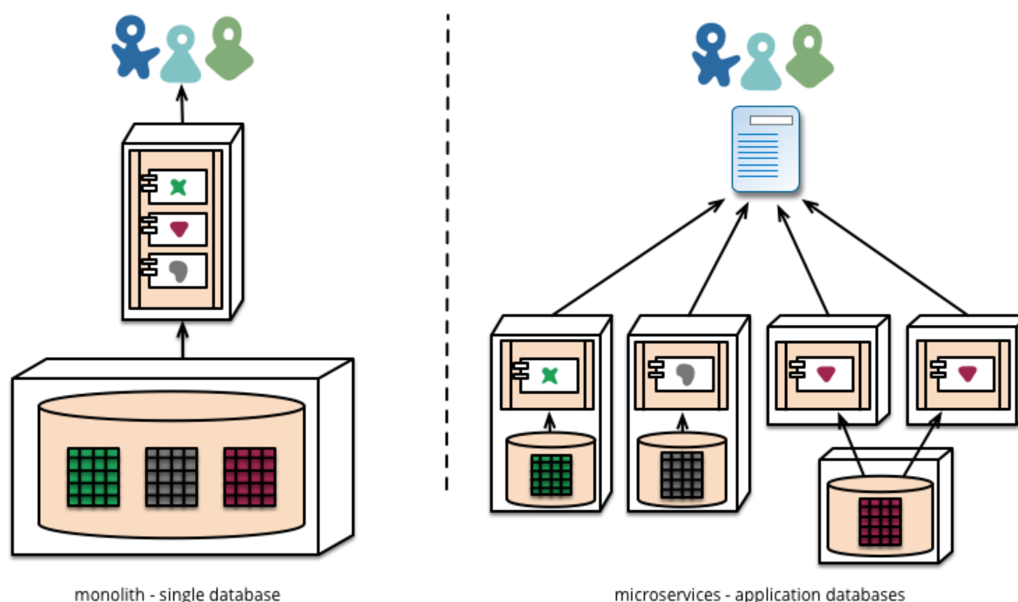


Slika 3.3: Na sliki vidimo sistem mikrostoritev, ki uporablja tako sinhrono kot asinhrono komunikacijo.

bazi (npr. dodati/odstraniti stolpec), ker ne vemo, na katere dele aplikacije bo ta sprememba vse vplivala. Še večja težava se pojavi pri migraciji baze na drugo tehnologijo, saj moramo prilagoditi vso logiko interakcije z bazo podatkov.

Za mikrostoritve, ki so po naravi ohlapno sklopljene med seboj, enotna baza podatkov ni primerna [50]. Tehnološko sicer ni nič narobe s tem, vendar iz vidika poslovne logike in priporočenih arhitekturnih praks ni zaželjeno, da imajo vse mikrostoritve, ki sestavljajo določeno aplikacijo, dostop do celotne baze. Posamezna mikrostoritev uporablja eno ali največ par tabel iz baze (če smo pravilno zasnovali celotno aplikacijo, seveda), zato naj bo sama odgovorna za ravnanje (predvsem spreminjanje) teh podatkov. Da povzamemo, vsaka mikrostoritev naj ima svojo bazo, za katero je odgovorna in jo lahko spreminja. V kolikor podatke iz baze posamezne mikrostoritve potrebujejo ostale mikrostoritve, naj mikrostoritev izpostavi API, preko katerega lahko

ostale mikrostoritve samo berejo (ne spreminjajo) podatkov. Spreminjanje podatkov je v domeni lastniške storitve. Vizualni prikaz razlike med bazo podatkov monolita in mikrostoritev na sliki 3.4.



Slika 3.4: Vizualna primerjava med hrambo podatkov v monolitni arhitekturi (levo) in mikrostoritveni arhitekturi (desno).

Zahtevo, da naj vsaka mikrostoritev spreminja le svoje podatke, lahko v primeru relacijskih baz uresničimo tudi z ločenimi shemami ali privatnimi tabelami znotraj ene baze podatkov. V kolikor zaklenemo posamezne tabele na določeno mikrostoritev, smo dejansko izpolnili zahteve ločenega shranjevanja podatkov. Moramo pa se zavedati, da zna priti do razkoraka pri skaliranju in bo del baze potreboval različno stopnjo skaliranja kot ostali deli. V tem primeru lahko že ob zasnovi arhitekture dodelimo isto bazo podatkov mikrostoritvam za katere pričakujemo enakomerno potrebo po skaliranju.

Prednosti ločenih baz vključujejo:

- **poliglotno shranjevanje podatkov** (angl. *polyglot persistence*) - posamezna vrsta baze podatkov ni primerne za vse vrste podatkov,

zato je smiselno implementirati različne vrste baz podatkov za različne vrste podatkov (npr. relacijske baze niso primerne za zapis socialnih grafov)

- **večjo avtonomijo mikrostoritev** - podatkovni model baze posamezne mikrostoritve lahko spreminjamo brez strahu, da bo koda zunaj dane mikrostoritve prenehala delovati (ker se nobena zunanja koda ne more in ne sme sklicevati na bazo podatkov dane mikrostoritve) in brez zamudnega dogovarjanja in usklajevanja sprememb z ostalimi oddelki in ekipami razvijalcev

- **večjo skalabilnost** - različne vrste baz omogočajo različne stopnje skalabilnosti. Poleg tega, če uporabljamo mikrostoritve in skupno bazo podatkov, smo pri skaliranju omejeni z zgornjo mejo skalabilnosti skupne baze podatkov (če imamo 5 mikrostoritev in moramo eno izmed teh ekstremno skalirati, bomo prisiljeni ekstremno skalirati tudi skupno bazo podatkov, dokler ne bomo dosegli njene omejitve, s tem da najverjetneje večina skupne baze ne potrebuje skaliranja)

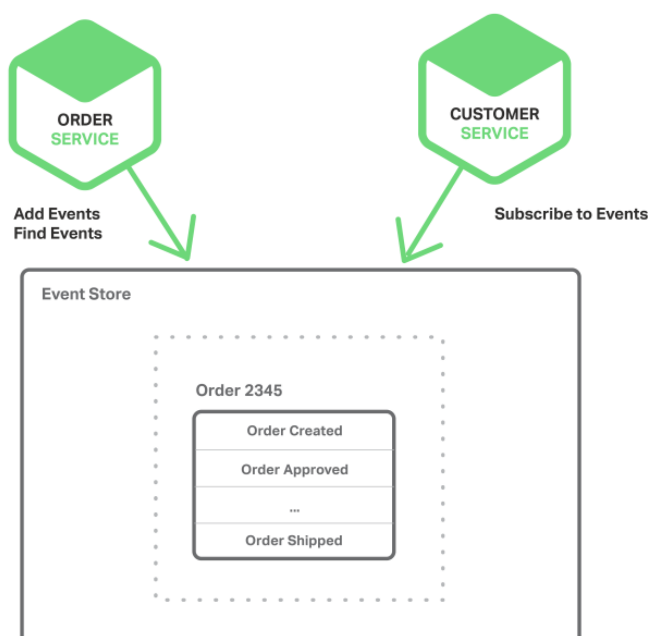
Seveda se je potrebno zavedati, kakor pri razbitju monolita na več mikrostoritev, da uporaba več ločenih in različnih baz podatkov poveča kompleksnost upravljanja le teh [47]. Pri razhroščevanju moramo sproti spremljati več baz podatkov ter jih znati upravljati, kar pogosto presega okvire posameznega razvijalca. V primeru, da izpostavimo API, preko katerega lahko druge storitve pridobijo podatke iz baze, povečamo sklopljenost storitev med seboj (tudi če uporabljamo asinhrone klice, mora klicoča storitev ob zagonu vedeti lokacijo klicane storitve). Poslovne transakcije, ki zajemajo več mikrostoritev in zagotavljajo konsistentnost podatkov v njihovih bazah podatkov so svojevrsten izziv. Vzporedno v (kompleksnejših) poslovnih aplikacijah poizvedbe v bazo niso omejene samo na posamezno tabelo, ampak združujejo podatke večih tabel (če bi uporabljali API za pridobivanje podatkov od mikrostoritev, bi izvedli par klicev preden bi pridobili vse podatke, kar nanese veliko večjo zakasnitev kakor združevanje tabel preko poizvedb v SQL bazi). Na koncu se

soočimo še s problemom repliciranja podatkov med več instancami iste baze, saj ob posodobitvi ene baze ta preko omrežja sporoči ostalim spremembo, kar lahko traja tudi sekundo ali več. Tudi če vse klice in repliciranje opravimo zelo hitro, ne moremo mimo dejstva, da se spopadamo s sočasno skladnostjo, ki je ne moremo preprečiti [32].

Iz omenjenega ni težko razbrati, da princip ACID [52], ki ga poznamo iz relacijskih baz, v porazdeljenem upravljanju podatkov ni možen. Izbrati moramo drugo pot, ki je poznana iz NoSQL baz in jo ponazarja BASE princip [52].

Zanimiva, vendar kompleksna rešitev (ki deluje po BASE principu) nekaterih prej omenjenih problemov je shranjevanje podatkov v obliki množice dogodkov [47], s katerimi lahko obnovimo trenutno stanje in tudi stanje objekta v katerikoli točki v času (garantirano zanesljiva revizijska sled). Dogodke shranjujemo v shrambo dogodkov (angl. *event store*), ki je hrbtenica dogodkovno vodenega upravljanja podatkov (angl. *event driven data management*). Shramba dogodkov (slika 3.5) omogoča naročanje na dogodke in brskanje po preteklih dogodkih preko APIja. Ob novem dogodku se ta zapiše v shrambo dogodkov, le ta pa posreduje dogodek naročenim mikrostoritvam, ki prejeti dogodek procesirajo in rezultat shranijo v svojo shrambo dogodkov v obliki novega dogodka. S tem osvežujemo podatke po celotnem sistemu še preden posamezna storitev zahteva nove podatke in se tako izognemo dolgotrajnim API klicem v trenutku, ko dejansko potrebujemo te podatke. Tak princip v angleščini imenujemo Event Sourcing [28]. S tem smo zagotovili sočasno skladnost in izvedbo poslovne transakcije po celotnem sistemu, vendar se še vedno soočamo s problemom poizvedb, ki zahtevajo podatke večih mikrostoritev.

To lahko rešimo z implementacijo ločene poizvedbe in zapisa (CQRS [50]). kjer ločimo poizvedbe in pisanje v bazo. Omejimo se samo na poizvedbe. Za kompleksnejše poizvedbe v bazo lahko ustvarimo posebne module, ki bojo več dogodkov različnih entitet združili v materializirane poglede (angl. *materialized view* - v relacijskih bazah je tak pogled rezultat vnaprej pripravljene



Slika 3.5: Primer shrambe dogodkov. Mikrostoritev "Order"ob kreiranju novega naročila kreira nov dogodek in ga pošlje v shrambo dogodkov. Mikrostoritev Čustomer"je obveščena o novih dogodkih (v tem primeru o kreiranju novega naročila) in bo kupca tako obvestila o uspešnem prejetju njegovega naročila, kasneje pa tudi o tem, kdaj je bilo naročilo odpravljeno, plačano ali prejeto.

poizvedbe na strani baze podatkov), ki bodo zapisani v NoSQL bazo. Mikrostoritve lahko nato preko APIjev omenjenih modulov izvajajo poizvedbe (potreben samo en API klic na modul, ker so podatki združeni v modulu). CQRS nam omogoča tudi različno skaliranje poizvedb in zapisa v bazo (nekatere aplikacije imajo neprimerno večje število poizvedb kot pisanja v bazo podatkov). Negativna stran dogodkovno vodenega upravljanja podatkov je odstopanje od tradicionalnih, razvijalski skupnosti znanih in razumljivih konceptov, ubadanje z nekonsistentnimi podatki ter zaznavanje podvojenih dogodkov.

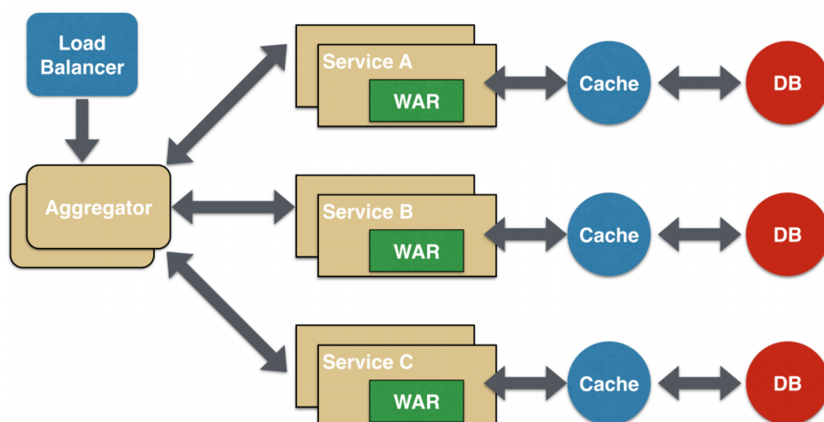
Spoznali smo, kako poteka shranjevanje in upravljanje podatkov v deželi mikrostoritev. Strinjamo se lahko, da, kot same mikrostoritve, tudi upravljanje podatkov dodatno oteži načrtovanje, implementacijo in vzdrževanje sistema mikrostoritev in je zato potreben resen premislek, preden se spustimo po tej poti. Vendar v kolikor se odločimo za mikrostoritve, ne bomo prišli daleč, če ne bomo uporabljali ločenih baz podatkov za posamezne mikrostoritve.

3.3 Arhitekturni vzorci

Do sedaj smo spoznali kaj so mikrostoritve, različne načine komunikacije med njimi in kako lahko shranjujejo podatke. Vse skupaj pa je potrebno povezati še v smiselno celoto. Mikrostoritve je potrebno pravilno povezovati med seboj, glede na naloge, ki naj bi jih opravljale, pri čemer nam lahko pomagajo različni vzorci [37], nekatere poznamo tudi iz drugih tehnoloških področij. Predstavljeni vzorci so mišljeni kot gradniki in navdih za sestavo celotnega sistema glede na dejanske zahteve in ne predstavljajo standardiziranih vzorcev (taki (še) ne obstajajo). Koncept izenačevalca obremenitev (angl. *load balancer*) bomo podrobneje spoznali kasneje, zaenkrat samo povejmo, da glede na intenzivnost prometa uravnoteženo usmerja klice na več instanc posamezne mikrostoritve.

- **Agregacija** (slika 3.6)

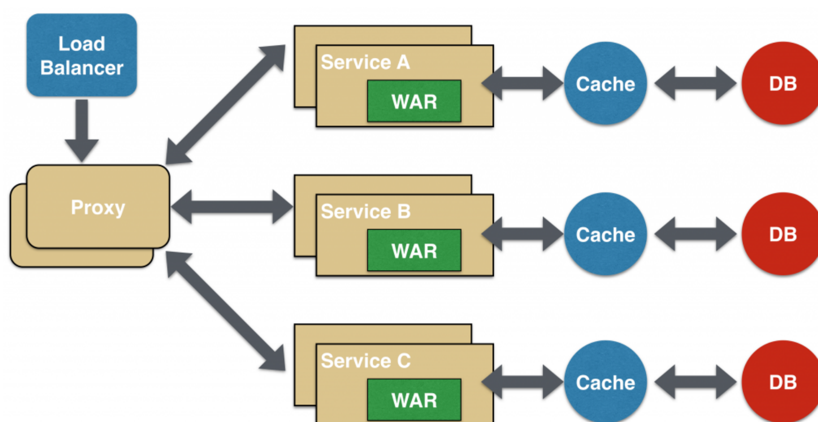
  Podatke iz več mikrostoritev združujemo (agregiramo) v agregatorski mikrostoritvi, lahko pa tudi direktno na strani klienta (npr. spletni vmesnik za pregled uporabniškega računa lahko neposredno združuje podatke od storitve registracije računa, košarice izdelkov, plačilne storitve ...). Komunikacija poteka preko sinhronih klicev, vsaka storitev ima svojo podatkovno bazo.



Slika 3.6: Primer agregacije mikrostoritev.

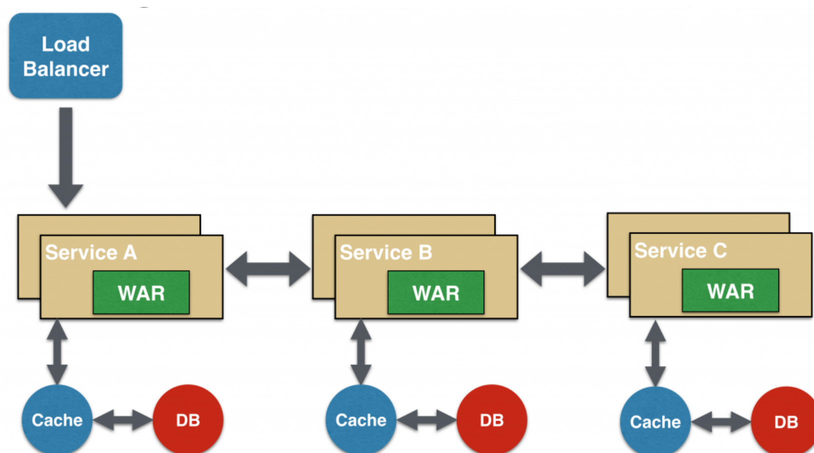
- **Posredniški strežnik** (angl. *proxy*) (slika 3.7)

  Preko posredniškega strežnika lahko, glede na podane podatke ob klicu, izbiramo, kam bomo posredovali klic. Za lažjo predstavo lahko to primerjamo s preoblaganjem metod - vedno kličemo isto funkcijo, a z drugimi parametri. Komunikacija poteka preko sinhronih klicev, vsaka storitev ima svojo podatkovno bazo.



Slika 3.7: Primer posredovanja klica preko posredniškega strežnika.

- **Veriženje mikrostoritev** (slika 3.8)

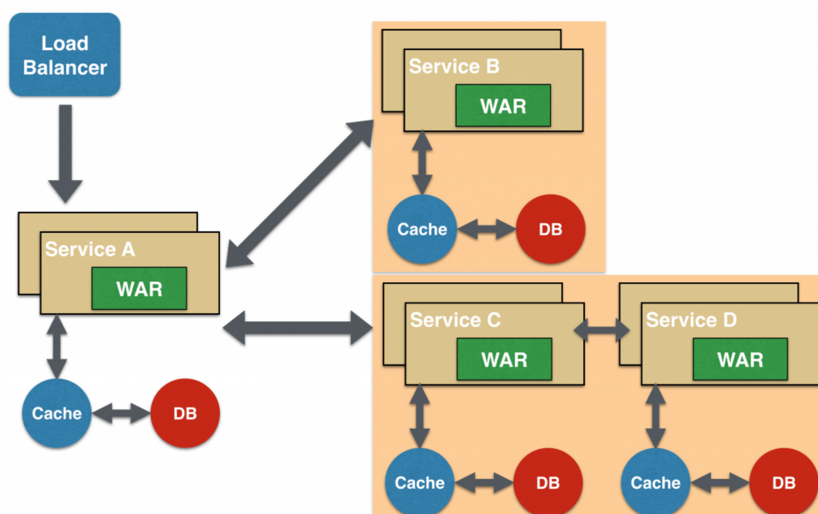


Slika 3.8: Primer veriženja mikrostoritev.

Klic na eno mikrostoritev sproži nadaljne klice od klicane storitve proti naslednji storitvi, ki lahko ponovno kliče neko drugo mikrostoritev. S tem zgradimo verigo klicev, kjer (lahko) vsaka storitev doda neko poslovno vrednost, vsi klici pa se na koncu združijo v enoten odgovor. Uporablja se sinhrona komunikacija, zato je potrebno biti pozoren, da veriga ne postane predolga, v nasprotnem primeru bo sistem blokiran dalj časa.

- **Vejitev** (slika 3.9)

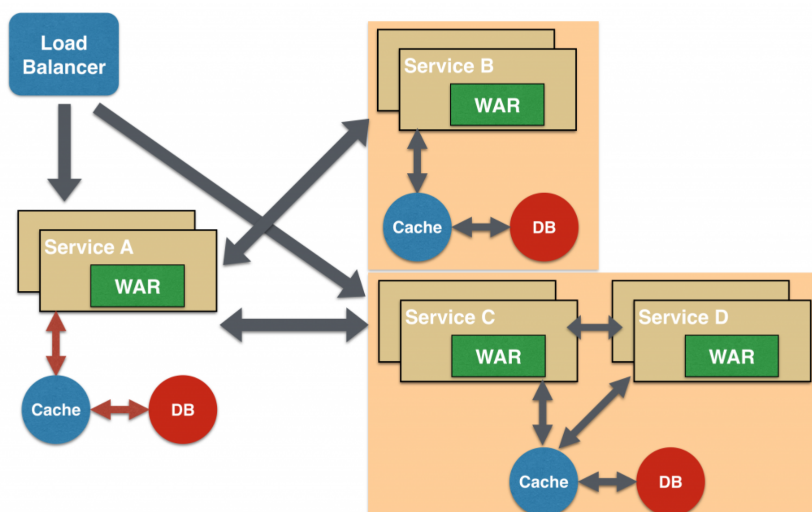
  Vzorec je mešanica med agregatorskim in posredniškim strežnikom, tudi uporabljamo ga lahko na oba načina. Storitev A se lahko glede na podatke odloči, da posreduje dva vzporedna klica (proti storitvama B in C), ali pa pokliče samo eno izmed vej. Komunikacija poteka preko sinhronih klicev, vsaka storitev ima svojo bazo.



Slika 3.9: Primer vejitve mikrostoritev.

- **Deljena baza podatkov** (slika 3.10)

  Čeprav je deljenje baze podatkov med več storitev odsvetovano, je v določenih primerih to potrebno, predvsem pri prenovi obstoječih mono-

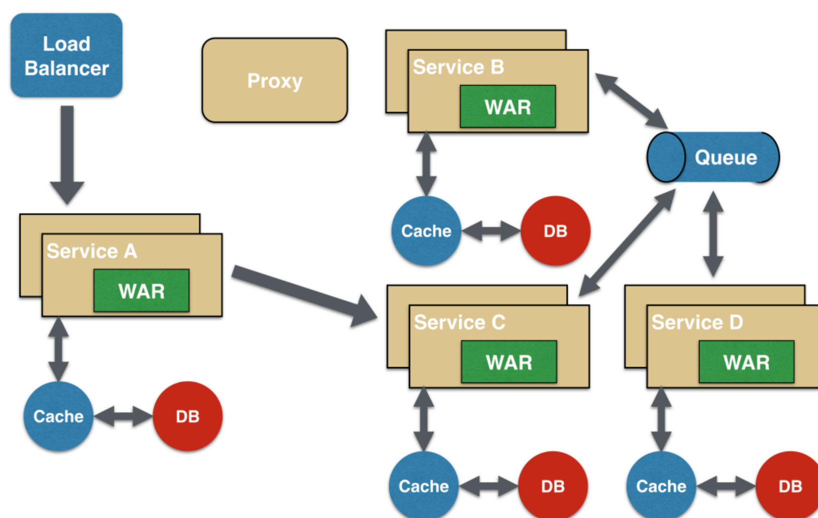


Slika 3.10: Primer deljene baze podatkov.

litnih aplikacij, kjer si ne moremo privoščiti nekonsistentnih podatkov. Bazo je v takih primerih najbolj primerno deliti med storitve, ki so tesneje sklopljene med seboj, storitve pa povezati v verigo.

- **Asinhrona komunikacija** (slika 3.11)

  Do sedaj smo spoznali le oblikovalske vzorce, ki so primernejši za sinhrono komunikacijo, ki zablokira celoten sistem do konca izvajanja posameznega klica. Za mikrostoritve je primernejša asinhrona komunikacija, npr. preko sporočilnih vrst, kot na sliki []. Na sliki vidimo, da storitev A (a)sinhrono kliče storitev C, ki storitvama B in D podatke posreduje preko sporočilne vrste in se s tem izogne blokiranju sistema. Možne so tudi kombinacije sinhrone in asinhrone komunikacije, na zgornjem primeru lahko uvedemo sinhrono komunikacijo med storitvama A in C.



Slika 3.11: Primer arhitekture mikrostoritev z asinhrono komunikacijo.

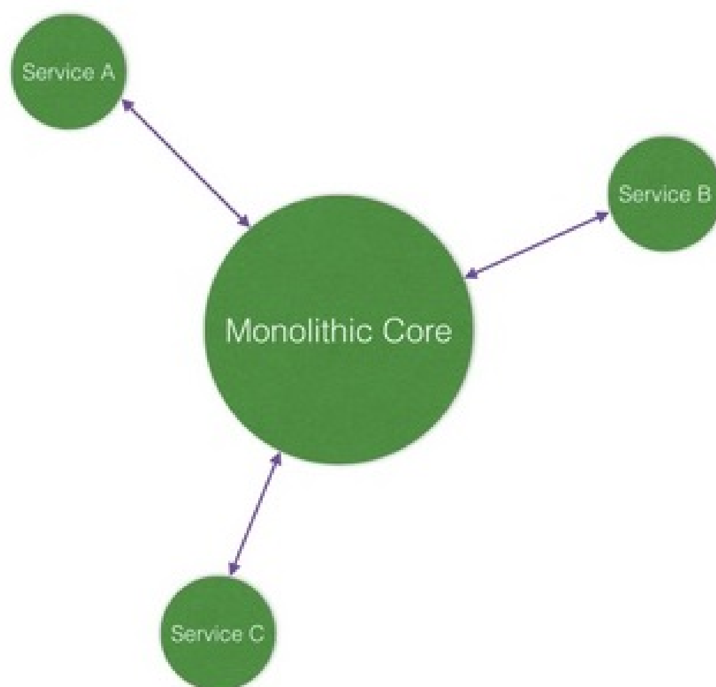
3.4 Zasnova celotnega sistema

V prejšnjem poglavju smo spoznali oblikovalske vzorce za povezovanje med mikrostoritvami. Vendar ti vzorci ne predstavljajo celotno arhitekturo sistema, saj jih lahko poljubno kombiniramo in tako zgradimo končen sistem. Na ta način lahko dobimo popolnoma mikrostoritveni sistem. Včasih pa potrebujemo samo določene lastnosti mikrostoritev in nočemo zaradi tega razbiti delujoč monolitni sistem. V takem primeru lahko obdržimo jedro, medtem ko nekatere dele monolitne aplikacije (če so razvijalci sledili principom domensko usmerjenega načrtovanja, je monolit modularen in ga je lažje razbiti) preoblikujemo v mikrostoritve. Prav tako se bomo ob postopnem prehodu iz monolitne aplikacije v mikrostoritveno na neki točki znašli med obema slogoma, kjer bomo imeli del aplikacije monolitne, del pa preoblikovan v mikrostoritve, ki jih bo klical monolitni del aplikacije. V nadaljevanju bomo spoznali oba omenjena pristopa ter njune prednosti in slabosti [43]. °

3.4.1 Mikrostoritvena arhitektura z monolitnim jedrom

Da bi dosegli nekatere lastnosti mikrostoritev (npr. skalabilnost), nam ni vedno potrebno razbiti celo monolitno aplikacijo na mikrostoritve. Lahko obdržimo del monolita kot jedro, ostale komponente pa preoblikujemo v mikrostoritve (slika 3.12). Večina poslovne logike ostane v monolitnem jedru, v mikrostoritve pa se preselijo komponente, ki jih je potrebno skalirati ali manj pomembne komponente, kot so obveščanje uporabnikov (preko e-maila, SMSov ...), naloge v ozadju ... Ker monolitno jedro uporablja svojo (največkrat skupno) bazo podatkov je najbolje, če lahko ob klicu mikrostoritve iz monolitnega jedra posredujemo vse potrebne podatke potrebne za njeno izvajanje in se tako izognemo grajenju dodatnih baz ali poizvedb na obstoječo skupno bazo podatkov.

Mikrostoritvena arhitektura z monolitnim jedrom je lahko naš končni cilj ali pa vmesna točka do popolnoma mikrostoritvene arhitekture. V prvem primeru je obstoječi monolit postal prevelik, težko ga je vzdrževati in je bila



Slika 3.12: Primer mikrostoritvene arhitekture z monolitnim jedrom.

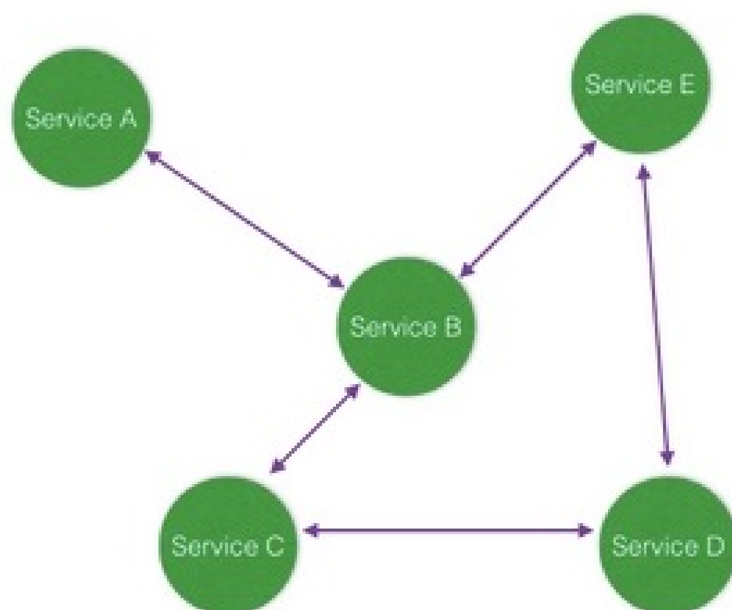
zato sprejeta odločitev, da se nekatere njegove komponente odvojijo. Lahko pa ga ohranimo nedotaknjenega in odvojimo samo nove komponente. V drugem primeru smo se odločili za postopen prehod iz monolitne aplikacije v mikrostoritve. Ta vmesni korak med monolitno in mikrostoritveno aplikacijo nam omogoča lažji prehod med obema konceptoma, saj je direkten prehod zelo težak [33]. Lahko najdemo tudi priporočila, da, čeprav je naš končni cilj sistem mikrostoritev, naj vseeno najprej zgradimo monolit - ker je hitreje in ob grajenju monolita lažje vidimo, kako bo sistem na koncu dejansko razdeljen na komponente, kakor pa da (slepo) predvidevamo na začetku projekta.

Prednost tega pristopa je, da lahko poberemo najboljše iz obeh svetov - imamo skupno bazo podatkov, manj vzdrževanja strežnikov in manj (kritične) komunikacije preko omrežja, na drugi strani pa pridobimo skalabilnost, možnost uporabe različnih tehnologij, izolacijo napak in drugo.

3.4.2 Popolnoma mikrostoritvena arhitektura

Ko nimamo nekega osrednjega dela aplikacije, ki izvaja večino poslovne logike, ampak je ta razdeljena med mikrostoritve, kjer vsaka opravi svojo nalogo, ostalo pa posredujemo naprej v obliki zahtevkov ali obveščanja, govorimo o popolnoma mikrostoritveni arhitekturi (slika 3.13). Mikrostoritve so med seboj povezane na različne načine, nekatere od njih smo spoznali v prejšnjem poglavju.

Prednosti in pomanjkljivosti popolnoma mikrostoritveno usmerjene arhitekture so identične tistim od mikrostoritev, ki smo jih spoznali že v podpoglavju 2.2 o mikrostoritvah in se zato ne bomo podrobneje spuščali v njih. Ponovimo pa, da je tak pristop zahteven in se je ponesrečil že veliko izkušenim razvijalskim ekipam [24].



Slika 3.13: Primer mikrostoritvene arhitekture z monolitnim jedrom.

Poglavje 4

Razvoj mikrostoritev

V prejšnjih poglavjih smo spoznali kaj so mikrostoritve, njihove prednosti in slabosti v primerjavi z bolj tradicionalnimi (monolitnimi) aplikacijami, načine medsebojne komunikacije in shranjevanja podatkov ter arhitekturne oblike sistemov mikrostoritev. Sedaj se bomo poglobili v tehnološki del razvoja mikrostoritev - spoznali bomo orodja, ki pohitrijo razvoj mikrostoritev in olajšajo upravljanje z njimi.

Najprej se bomo seznanili s šasijami - ogrodji, ki vsebujejo knjižnice in servlete za preprostejši in hitrejši razvoj mikrostoritev. Z njihovo pomočjo lahko mikrostoritev zaženemo preko konzole v kratkem času (pod 15 sekund) in v parih vrsticah izpostavimo osnovne REST končne točke.

Nato bomo spoznali, kako mikrostoritve dejansko vedo druga za drugo, kar je prvi predpogoj za (sinhrono) komunikacijo med njimi. Spoznali bomo odkrivanje storitev, katere so prednosti in slabosti na strani strežnika oziroma klienta ter katera orodja za odkrivanje storitev poznamo.

Sledila bo razlaga pojma izenačevalec obremenitve (angl. *load balancer*), ki smo ga spoznali že pri Arhitekturnih vzorcih (podpoglavje 3.3) in predstavitev primernih izenačevalcev obremenitve.

V primeru več zaporednih neuspelih klicev proti določeni mikrostoritvi ugotovimo, da je bolje, da do teh klicev nebi prišlo in bi se s tem izognili nepotrebnemu trošenju sistemskih virov. Spoznali se bomo s konceptom od-

klopnika (angl. *circuit breaker*) [22] in aktualnimi odklopniki za mikrostoritve

4.1 Šasije

Ob vzpostavljanju monolitnega projekta ni nič nenavadnega, če porabimo dan ali dva samo za vzpostavitev okolja. Potrebno je postaviti strežnik, konfigurirati logiranje, spremljanje sistema in ostalo. Ker se bo na monolitnem projektu delalo po več mesecev, dan ali dva na začetku za postavitev okolja ne pomenita veliko.

Na drugi strani so mikrostoritve manj obsežne, vsaka skrbi samo za eno funkcionalnost, zato ima vsaka aplikacija večje število različnih mikrostoritev. Vzpostavljanje vsake mikrostoritve več kot uro bi bila potrata časa in bi močno vplivala na produktivnost ekipe. V ta namen so se razvile mikrostrukturne šasije (angl. *microservice chassis* [48]) oziroma ogrodja za hitrejši in preprostejši razvoj mikrostoritev. Z njihovo pomočjo lahko v parih minutah naredimo in zaženemo novo mikrostoritev (izvzemši programiranje poslovne logike) in s tem močno skrajšamo in poenotimo razvoj.

Šasije težijo k čim manjši končni velikosti, imajo vgrajene servlete, prirejene knjižnice za hitrejši razvoj (npr. REST knjižnica) in kompatibilnost s servlet okoljem. Ponujajo podporo odkrivanju storitev, integracijo z izenačevalci obremenitve ter odklopniki, izdelavo samozadostnega izvršljivega JARa in so kompatibilne z različnimi orodji za upravljanje z odvisnostmi in prevajanjem kode (kot so Maven, Gradle, Ivy in podobni).

Izbira šasije je pogojena tudi s tehnologijo, ki jo želimo uporabiti oziroma jo uporablja aplikacija, ki jo moramo prenoviti. Za primer vzemimo aplikacijo, ki uporablja Spring knjižnico - uporaba java EE šasije (npr. KumuluzEE) tu ne bo mogoča. To tehnično imenujemo zaklep (angl. *lock-in), saj kasneje tekom izdelave aplikacije ne bo možno zamenjati šasijo brez obsežnega programiranja.* Medtem pa lahko šasije, ki podpirajo iste knjižnice (npr. KumuluzEE in Wildfly Swarm podpirata java EE) zamenjamo druga z drugo v razmeroma kratkem času (odvisno tudi od stopnje podpore java

EE posamezne knjižnice v tem konkretnem primeru).

Pogledali in primerjali bomo štiri šasije, ki so ta trenutek najbolj razširjene in podprte na tržišču - Spring Boot, Dropwizard, Wildfly Swarm in KumuluzEE. Hiter pregled se nahaja v tabeli 4.1, v nadaljevanju pa si še podrobneje oglejmo vsako šasijo.

4.1.1 Spring Boot

Spring boot [16] prihaja iz znane družine Spring produktov, ki lajšajo vsakdanja opravila (java) razvijalcev. Pojavil se je v Spring 4.0 verziji leta 2014. V celoti se zanaša na Spring tehnologijo, k uporabi katere spodbuja tudi razvijalca. Je torej mnenjsko ogrodje, ki sledi konvenciji nad konfiguracijo (predvidene privzete lastnosti in nastavitve naj bi zadoščale večini primerov) [16]. Postavitev in zagon osnovne mikrostoritve z eno REST končno točko nam vzame okoli 5 minut.

V tabeli 4.1 lahko vidimo, katere knjižnice uporablja Spring Boot za posamezne naloge. Privzeti servlet je Tomcat, ponuja pa tudi Jetty in Undertow. Za REST privzeto uporablja Spring, vendar pa podpira tudi JAX-RS (in posledično njegove implementacije, npr. Jersey). Za procesiranje JSON objektov imamo na voljo kar nekaj knjižnic, prav tako za beleženje, medtem ko se pri metrikah in preverjanju zdravja mikrostoritev zanaša na lastne Spring knjižnice. Za injiciranje odvisnosti uporablja lastno ogrodje, možnosti za uporabo druge knjižnice (npr. Google Guice) ali java EE nimamo. Tudi pri operacijami z bazami podatkov nam omogoča različne knjižnice, privzeta pa je Spring Data.

Odlična je tudi podpora odkrivanju storitev, saj lahko preko anotacij in deklariranja odvisnosti v kratkem času podpremo Eureko, ZooKeeper in Consul. Za Eureko je potrebno le anotirati pravi razred v programski kodi in Spring Boot jo bo samodejno zagnal ob svojem zagonu [15]. ZooKeper in Consul sta podprta v obliki modulov, ki ju dodamo kot odvisnost v aplikacijo ter vstavimo pravo anotacijo na pravo mesto v kodi. Je pa pri zadnjih dveh potrebna ročna namestitev in konfiguracija, saj sta modula namenjena le

Tabela 4.1: Primerjava podpore mikrostoritvenih šasij tehnologijam za pomoč pri razvoju mikrostoritev

	Spring Boot	**Dropwizard**	**Wildfly Swarm**	**KumuluzEE**
samozadostni izvršljivi JAR	da	da	da	da
servlet HTTP	Tomcat (privzeto), Jetty, Undertow	Jetty	Undertow	Jetty
REST	Spring MVC, Jersey, Apache CXF	Jersey	JAX-RS	JAX-RS (Jersey)
JSON	Jackson	Jackson	JSON-P, Swagger	JSON-P
beleženje	Log4J, Logback, Commons Logging	Logback	Logging, Logstash	–
interakcija z bazo podatkov	Spring Data	Hibernate, JDBI	Hibernate	EclipseLink
injiciranje odvisnosti	Spring	privzeto ne (možno z Google Guice)	Weld	Weld
odkrivanje storitev	Eureka, Consul, ZooKeper	ZooKeper (preko Netflix Curator)	Consul, JGroups	–
upravljanje odvisnosti in prevajanje kode	Maven, Gradle	Maven, Gradle	Maven, Gradle	Maven

komunikaciji z že delujočo instanco ZooKeperja oziroma Consula.

Odločitev za uporabo Spring Boot šasije v svojem projektu je primarno pogojena z uporabo Spring ogrodja - v kolikor je celoten projekt narejen v Springu, je močno priporočena uporaba Spring Boota, saj je prilagojen in namenjen Spring ogrodju [41]. Kljub temu nam Spring Boot pušča določeno mero svobode, saj poleg privzetih Spring ogrodij omogoča uporabo tudi drugih popularnih ogrodij in knjižnic. Spring je nasploh zelo razširjen in podprt s strani skupnosti, veliko zunanjih knjižnic ponuja vsaj določeno mero integracije s Springom, kar naredi Spring in Spring Boot primerno ogrodje za implementacijo poslovno zahtevnejših mikrostoritev (ki potrebujejo injiciranje odvisnosti, podporo transakcijam ...).

4.1.2 Dropwizard

Dropwizard [1] je najstarejše ogrodje izmed obravnavanih, njegovi začetki segajo v leto 2011. Združuje razširjene in stabilne java knjižnice v zmogljivo orodje za kreiranje REST aplikacij (oziroma v našem primeru mikrostoritev). Preferira konvencijo čez konfiguracijo in je mnenjsko ogrodje.

Glavna prednost Dropwizarda je izjemno hitra vzpostavitev projekta, kar dodatno poveča produktivnost, vendar nam ne ponuja veliko izbire pri orodjih in knjižnicah. Za servlet uporablja Jetty, Jersey za REST, Jackson za JSON, beleženje izvaja s knjižnico Logback, shranjevanje in branje podatkov pa poteka preko Hibernatea ali JDBI. Transkacij in injiciranja odvisnosti privzeto niti ne podpira, zanesti se moramo na zunanje integracije (npr. Google Guice).

Izbira omenjenih orodij in knjižnic temelji na njihovi dolgoletni uporabi v java svetu, podporni razvijalski skupnosti in stabilnosti, tako da iz tega vidika ne moremo oporekati izbiri. Vendar v realnosti velikokrat naletimo na (podedovane) projekte ali pa ekipo razvijalcev, ki je navajena uporabe drugih knjižnic, in tukaj nam zaklenjenost oziroma neprilagodljivost Dropwizarda onemogoča njegovo uporabo.

4.1.3 Wildfly Swarm

Wildfly Swarm [2] je produkt zelo razširjenega aplikacijskega strežnika Wildfly (predhodno poznan pod imenom Jboss). Na trg je prišel maja 2015 in je razmeroma mlado in (še) nerazširjeno ogrodje. Swarm je dejansko razgrajen aplikacijski strežnik Wildfly, iz katerega izberemo potrebne podsisteme in jih vključimo v naš Swarm projekt. S tem se izognemo trošenju virov na nepotrebnih podsistemih.

Swarm podpira standardno Javo EE, kar pomeni, da lahko uporabimo celotno moč java EE v naši mikrostoritvi in smo tako dobili java EE alternativo Spring Boot ogrodju. Poleg tega nam omogoča uporabo preverjenih Wildfly nadzornih in upraviteljskih podsistemov, ki pripomorejo k večji produktivnosti razvijalcev. Prav tako Swarm omogoča hiter prehod iz aplikacije za aplikacijski strežnik Wildfly v samozagonski JAR oziroma mikrostortev. Wildfly podsistemi se v Swarmu imenujejo frakcije.

Podpira veliko standardnih java EE knjižnic, katerim so vsakodnevno dodane nove, med drugim Weld za injiciranje odvisnosti, Undertow kot servlet, standarden JAX-RS za REST in druge. Ko se bo podpora Swarma še dodatno izboljšala, lahko pričakujemo močno orodje za kreiranje java EE mikrostoritev z vsemi lastnostmi standardnih aplikacij na aplikacijskih strežnikih.

4.1.4 KumuluzEE

KumuluzEE [3] je rezultat sodelovanja med diplomantom in profesorjem Fakultete za računalništvi in informatiko v Ljubljani tekom študijskega leta 2014/2015. Tako kot Wildfly Swarm je tudi KumuluzEE namenjen standardni Javi EE, kmalu po izidu prve verzije je prejel nagrado "2015 java Duke's Choice Award Winnder" [18] (za primerjavo, nekateri izmed prejšnjih dobitnikov nagrade so Hadoop, ogrodje za porazdeljene sisteme in Jenkins, orodje za sprotno integracijo).

Trenutno KumuluzEE aplikacije tečejo na Jetty servletu, ki je bil izbran zaradi zmogljivosti in velikosti, načrtovana je tudi podpora preostalih popu-

larnih servletov. V podpori preostalih knjižnic in ogrodij zaostaja za Swarmom, trenutno podpira Servlet 3.1 (Jetty), WebSocket 1.1 (Jetty), JSP 2.3 (Jetty Apache Jasper), EL 3.0 (RI UEL), CDI 1.2 (RI Weld), JPA 2.1 (RI EclipseLink), JAX-RS 2.0 (RI Jersey), JSF 2.2 (RI Mojarra), Bean Validation 1.1, (RI Hibernate validator), JSON-P 1.0 (RI JSONP). Na prvi pogled izgleda veliko, vendar pri Swarmu, Spring Bootu in Dropwizardu nismo naštevali vseh podprtih knjižnic in ogrodij, zaradi velikega števila le teh.

KumuluzEE je trenutno eden izmed redkih konkurentov Swarmu, ki so dosegli primerno stopnjo zrelosti. Prednost pred Swarmom je ravno neodvisnost, saj Swarm spada pod Wildfly in ga tako naredi bolj primernega za aplikacije, ki tečejo na Wildfly aplikacijskih strežnikih ali za razvijalce, ki imajo izkušnje z Wildflyom. Vstopni prag za KumuluzEE je tako le znanje razvoja java EE aplikacij, prav tako stremi k minimiziranju potrebne konfiguracije in cilja na oblačno gostovanje. Uporaba v produkciji je (trenutno) odsvetovana.

4.2 Odkrivanje storitev

Kakor smo omenili, je ena glavnih prednosti mikrostoritev možnost skaliranja. Ob povečanem prometu zaženemo več instanc posamezne storitve med katere se porazdelijo zahtevki. Vendar v primeru sinhrone komunikacije, samo zagon nove instance ni dovolj; poznati moramo tudi IP naslov in vrata te storitve, da ji lahko pošljemo zahtevek, saj so naslovi dodeljeni dinamično ob kreiranju novih instanc in jih ne moremo zapeči v konfiguracijske datoteke. Pri tem nam lahko pomaga metoda odkrivanja storitev, ki shrani potrebne informacije (IP naslov, vrata, poverilnice za preverjanje pristnosti, protokole in ostale podatke) instanc vseh mikrostoritev v registru storitev (angl. *service registry*) [49] in jih po potrebi posreduje mikrostoritvam.

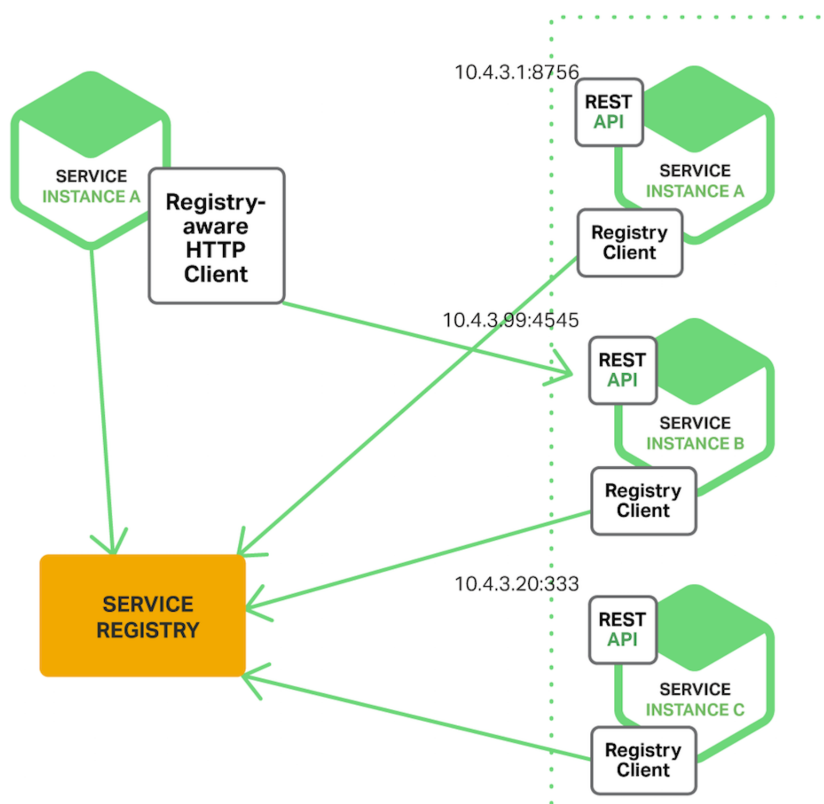
Register storitev je pravzaprav še ena izmed storitev v našem sistemu. Zaradi svoje naloge je njeno delovanje ključno za delovanje sistema. V primeru nedelovanja ostale storitve ne morejo komunicirati med seboj, razen če

si naslove drugih storitev začasno shranjujejo v predpomnilnik. Vendar tudi te naslovi sčasoma postanejo zastareli. Odkrivanje storitev lahko poteka na klientu (angl. *client-side discovery pattern* [47]) ali na strežniku (angl. *server-side discovery pattern* [47]).

Pri odkrivanju storitev na strani klienta le ta najprej kontaktira register storitev, ki mu poda informacije o instancah posamezne mikrostoritve, ki so trenutno na voljo. Klient nato na podlagi prejetih informacij in lastnih potreb izbere, katero instanco bo kontaktiral - izenačevanje obremenitve tako poteka na strani klienta in se lahko prilagodi zahtevam aplikacije. Primer lahko vidimo na sliki 4.1. Pri tem načinu odkrivanja storitev imamo samo en kritični del, register storitev, za katerega moramo zagotavljati dostopnost. Slabost odkrivanja storitev na strani klienta je tesnejša sklopljenost z registrom storitev; v vsaki tehnologiji moramo posebej implementirati logiko za komunikacijo z registrom storitev in tudi za izenačevanje obremenitev.

Drugi omenjeni pristop, odkrivanje storitev na strani strežnika, postavi izenačevalca obremenitve med klienta in register storitev (slika 4.2). Klient pošilja zahtevke izenačevalcu obremenitev, ki nato od registra storitev pridobi informacije o instancah mikrostoritve in se odloči, kateri bo posredoval zahtevek. Klient tako ni sklopljen z registrom storitev, saj komunicira le z izenačevalcem obremenitve. Glavna slabost tega pristopa je nova komponenta (izenačevalec obremenitve), za katero moramo zagotavljati dostopnost. Poleg tega imamo več klicev v omrežju kot v primeru odkrivanja na strani klienta.

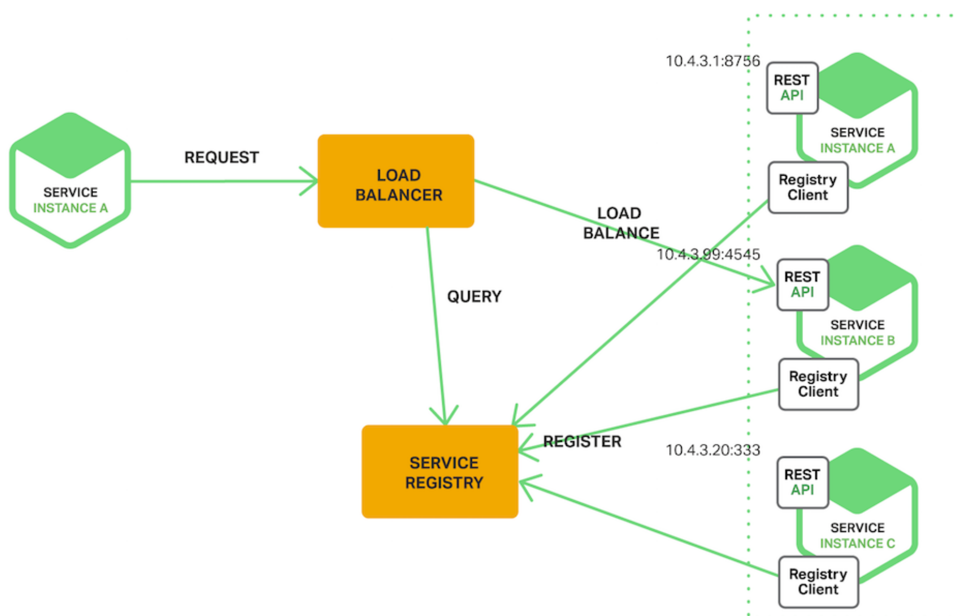
Ko se nova instanca mikrostoritve zažene, se lahko registrira v storitev za odkrivanje storitev na dva načina. V prvem načinu mikrostoritev sama skrbi za obveščanje registra storitev o svojem delovanju (samoregistracija, angl. *self registration* [47]). Ob zagonu se mora registrirati in podati svoj naslov. V primeru, da se mikrostoritev ugasne, pred tem obvesti register storitev in ta jo odstrani iz svojega repozitorija naslovov. Tekom delovanja se mora mikrostoritev redno javljati registru storitev, v nasprotnem primeru jo po določenem času neaktivnosti odstrani iz svojega repozitorija. Redno



Slika 4.1: Primer odkrivanja storitev na strani klienta.

javljanje lahko vsebuje podrobnejše opise stanja (zaganjanje, dosegljiv, preobremenjen, zaustavitev in drugi). Glavna pomanjkljivost samoregistracije je sklopljenost mikrostoritve z registrom storitev, saj mora mikrostoritev poznati njegov naslov. Poleg tega moramo implementirati logiko za komunikacijo z registrom storitev v vsaki tehnologiji posebej (npr. če imamo eno mikrostoritev v Javi in eno v Cju). Menjava orodja za odkrivanje storitev pomeni tudi spremembe v kodi vsake mikrostoritve.

V drugem načinu odkrivanja storitev se mikrostoritev neposredno ne zaveda orodja za odkrivanje storitev. Nalogo registracije in sledenja storitvam prevzame tretje orodje, registrar storitev (angl. *service registrar*) [47]. Registrar sledi spremembam s pregledovanjem okolja, v katerem tečejo instance



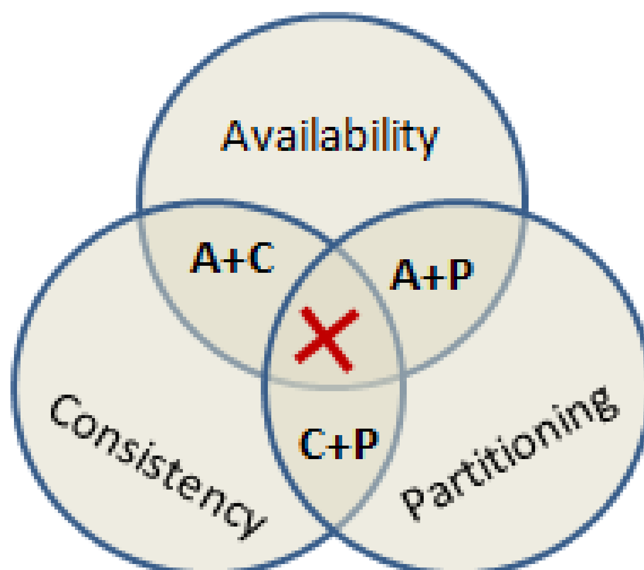
Slika 4.2: Primer odkrivanja storitev na strežniški strani.

mikrostoritev, ali prejemanjem dogodkov, ki jih sprožijo nove instance ob zagonu. Ob spremembi posodobi register storitev; registrira nove storitve ali odstrani nedelujoče. Kakor pri odkrivanju storitev na strani strežnika je tudi tu glavna prednost ohlapna sklopljenost mikrostoritve in registra storitev, medtem ko se ponovno soočimo s problemom vzdrževanje dodatne kritične komponente v našem sistemu.

Kakor smo omenili na začetku, je odkrivanje storitev predvsem potrebno v primeru sinhrone komunikacije, saj so instancam mikrostoritev dodeljeni dinamični IP naslovi oziroma vrata in ne moremo vnaprej vedeti, na katerih naslovih se bo nahajala določena storitev. V nasprotnem primeru pri asinhroni komunikaciji komuniciramo samo s sporočilno vrsto, ki redko, če sploh kdaj, spremeni svoj naslov. Zato lahko uporabimo konfiguracijske datoteke, v katere zapišemo potrebne podatke za komunikacijo s sporočilno vrsto in se izognemo potrebi po odkrivanju storitev.

V nadaljevanju si bomo pogledali najbolj priljubljena orodja za odkri-

vanje storitev na trgu in jih primerjali med seboj. Ugotovili bomo, da ni enotnega in najboljšega orodja, ki bi rešil vse izzive, vendar se moramo prilagajati glede na lastnosti in potrebe aplikacije, ki jo razvijamo. Eno izmed meril za primerjavo je CAP teorem (slika 4.3) [36], ki zatrjuje, da je v porazdeljenem sistemu nemogoče sočasno zagotoviti konsistentnost podatkov (angl. *consistency*), dostopnost (angl. *availability*) in odpornost na delitve omrežja (angl. *partition tolerance*). Večinoma porazdeljen sistem zagotavlja dve izmed teh lastnosti, odvisno od potreb. V nadaljevanju se bomo spoznali z dvema CP in enim AP registrom storitev, med katerimi bomo kasneje izbirali.



Slika 4.3: Slika CAP teorema, ki trdi, da v distribuiranem sistemu ni moč zagotoviti dostopnost, konsistentnost in odpornost na izpade hkrati.

4.2.1 Eureka

Eureka [4] je odprtokodno orodje za odkrivanje storitev, ki ga razvija Netflix in je namenjen interni komunikaciji mikrostoritev znotraj sistema. Spisana je

v Javi in je najbolj primerna za uporabo s storitvami, ki bazirajo na JVMju, poleg tega pa je prilagojena za Amazonov AWS oblak. Od CAP teorema zagotavlja AP. Eureka se deli na strežnik in klienta, uvrščamo jo v orodja za odkrivanje storitev na strani klienta.

Strežnik shranjuje lokacijo mikrostoritev, za komunikacijo uporablja REST API. Vsakih 30 sekund se mora storitev javiti Eureki, kar pomeni, da še deluje. V nasprotnem primeru po približno 90 sekundah neaktivnosti Eureka odstrani instanco mikrostoritve iz svojega repozitorija. Med delitvijo omrežja (zaradi izpadov) je še vedno omogočena registracija in odkrivanje storitev, ko se omrežje ponovno združi se zbrani podatki združijo in uskladijo med vsemi Eureka strežniki (P v CAP) [42].

Klient skrbi za registracijo storitve in ga vključimo v programsko kodo. Njegova naloga je tudi izenačevanje obremenitve (primarno uporablja round-robin izenačevanje) in predpomnenje informacij o lokacijah vseh ostalih mikrostoritev. S tem predpomnenjem Eureka zagotavlja dostopnost storitev (A v CAP), vendar bo klient informacije uporabil šele takrat, ko ne bo dostopen noben Eureka strežnik (takrat bodo informacije v predpomnilniku najboljše možne v danem trenutku).

4.2.2 ZooKeeper

ZooKeeper [5], ki ga razvija Apache, je veliko več kot le orodje za odkrivanje storitev, primarno je namenjen kot visoko zmogljiv koordinator med porazdeljenimi aplikacijami (omogoča vzdrževanje konfiguracijskih datotek, ponuja porazdeljeno sinhronizacijo med sistemi ...). Mi se bomo primarno usmerili v njegovo funkcionalnost kot register storitev.

ZooKeper zagotavlja CP v CAP teoremu, kar pomeni, da se bo ob delitvi omrežja manjši del omrežja ugasnil in bo nedostopen tistim klientom, ki ga uporabljajo. V preostalem delu omrežja pa je zagotovljena konsistentnost podatkov, saj bo ta del v trenutku po delitvi edini delujoč del sistema (lahko bi celo rekli, da je to celoten sistem, ker je drugi del ugasnil) in bojo vse informacije prihajale v ta del sistema [42].

Registracijo storitve moramo v kodi implementirati sami ali pa lahko uporabimo katero izmed knjižnic, ki so namenjene lažji implementaciji ZooKeeperja kot registra storitev (npr. Apache Curator). Sam ZooKeeper je tako kot Eureka napisan v Javi. Pri odločitvi za ZooKeeper je glavno, ali želimo konsistentnost podatkov in smo za to pripravljeni žrtvovati dostopnost do podatkov.

4.2.3 Consul

Consul [6], podobno kot ZooKeeper, omogoča več kot le registriranje storitev. Primarno je podatkovno skladišče, ki vsebuje register storitev in napredno ključ/vrednost shrambo (angl. *key/value store*) ter omogoča napredno preverjanje zdravja storitev. Consul izpostavlja API, preko katerega lahko storitve najdejo ostale storitve, komunikacija poteka po principu vsak z vsakim preko HTTP ali DNS. V kolikor se uporabi DNS, Consul vrne naslov samo ene instance storitve, ki jo izbere na podlagi preprostega naključnega algoritma (preprosto izenačevanje obremenitve). V kolikor uporabljamo HTTP, nam Consul vrne vse instance posamezne mikrostoritve in moramo sami izbrati med njimi. Kakor ZooKeeper tudi Consul preferira konsistentnost podatkov, torej je CP sistem (kljub temu ima možnost, da v primeru delitve sistema manjši del odgovarja na povpraševanja, vendar ni možno registrirati novih storitev).

Glavna prednost Consula je avtomatsko iskanje in registracija storitev, kar pomeni, da nam v sami programski kodi mikrostoritev ni potrebno implementirati registracijo storitve [19]. Consul pozna agente, ki jih lahko konfiguriramo kot strežnika ali klienta. Klienta namestimo na sistem, v katerem tečejo mikrostoritve, kjer bo sam zaznaval nove instance, preverjal njihovo zdravje in posredoval klice med mikrostoritvami in strežniki. Pri preverjanju zdravja mikrostoritev so pri Consulu še posebej poskrbeli za skalabilnost, saj namesto direktne komunikacije s centralnim strežnikom (kjer je za vsako mikrostoritev potrebna odprta povezava) uporablja Gossip protokol [12].

4.3 Izenačevanje obremenitve

Za uvod se postavimo v situacijo, ko imamo zagnano eno instanco mikrostoritve A in 20 instanc mikrostoritve B, ki morajo na neki točki poklicati A. Hitro ugotovimo, da ena instanca storitve A ne zmore procesirati vseh klicev in zaženemo dve dodatni instanci. Čez nekaj časa opazimo, da je prva instanca še vedno preobremenjena, medtem ko sta preostali instanci večino časa nezasedeni. Ugotovimo, da četudi imamo v našem registru storitev shranjene vse tri instance mikrostoritve A, instance mikrostoritev B večinoma kličejo prvo instanco mikrostoritve A.

Za rešitev tega problema moramo v našem sistemu implementirati izenačevalca obremenitve [13], ki bo skrbel za enakomerno razporeditev zahtevkov, namenjenih določeni mikrostoritvi (v prejšnjem primeru mikrostoritev A), med instance te mikrostoritve. Če je ena instanca polno obremenjena, bo preprečil posredovanje zahtev tej instanci, dokler ne bo procesirala določenega odstotka svojih zahtev in postala manj ali enako obremenjena kot ostale instance.

Tega se lahko lotimo iz dveh zornih kotov - uporabimo centralni izenačevalec obremenitve, ki bo prejel vse zahtevke v sistemu in jih pravilno posredoval na neobremenjene instance storitev, ali implementiramo izenačevanje obremenitev že na strani klienta in se izognemo postavljanju nove kritične komponente v našem sistemu. Izbira je lahko pogojena tudi glede na arhitekturo celotnega sistema in na izvor zahtevkov. Če imamo del mikrostoritev, ki so izpostavljene navzven (v internet) in ob prejetem zahtevku sprožijo vrsto internih klicev na druge mikrostoritve znotraj sistema, lahko za izpostavljene mikrostoritve uporabimo centralni izenačevalec obremenitve, medtem ko lahko interne mikrostoritve integrirajo izenačevalca znotraj same mikrostoritve in se tako izognemo dodatnim skokom v omrežju in novi kritični komponenti.

Potrebi po izenačevalcu obremenitve se v celoti izognemo pri asinhroni komunikaciji preko sporočilnih vrst. Do neke mere lahko posplošimo in rečemo, da so sporočilne vrste v tem primeru izenačevalci obremenitve, saj prejemajo

zahtevke od mikrostoritev in jih hranijo, dokler druge mikrostoritve teh zahtevkov ne vzamejo iz vrste in jih sprocesirajo. Na ta način mikrostoritev, ki bere iz vrste, ne bo nikoli preobremenjena, saj bo prebrala naslednji zahtevek šele, ko obdela trenutnega. Na drugi strani se mikrostoritev, ki pošilja zahtevke v vrsto, ne zaveda kdo je na drugi strani vrste, pomembno ji je le to, da je bil prejem zahtevka v vrsto uspešen.

Poglejmo si še primer izenačevalca storitev na strežniku in na strani klienta.

4.3.1 Ribbon

Ribbon [7] je primer izenačevalca obremenitev na strani klienta, namenjen interni komunikaciji med storitvami znotraj sistema. Primarno je namenjen uporabi v oblaku, komunikacija poteka preko protokola REST. Je odprtokodni produkt Netflixa, ki uporablja Ribbon v kombinaciji z Eureko v AWS oblaku.

Na voljo imamo nekaj vgrajenih osnovnejših algoritmov za izenačevanje obremenitve [17]:

- **Simple Round Robin** - Ribbon kroži po seznamu instanc mikrostoritve in vsak zahtevek pošlje naslednji instanci v seznamu

- **Weighted Response Time** - glede na odziven čas instance mikrostoritve se instanci dodeli neko vrednost, ki predstavlja verjetnost, da bo zahtevek posredovan ravno tej instanci (večji kot je odzivni čas manjša je dodeljena vrednost)

- **Availability Filtering Rule** - storitve, ki so neodzivne ali imajo visoko število vzporednih povezav, bodo za določen čas izločene iz prejemanja zahtev

4.3.2 AWS Elastic Load Balancer

Amazon v svojem oblaku uporablja Elastic Load Balancer [11], centralni izenačevalec storitev, ki je postavljen med Amazonovimi EC2 instancami (virtualni strežniki, ki gostijo mikrostoritve) in končnim uporabnikom. Med drugim omogoča avtomatski zagon in ugašanje instanc storitev glede na obseg prometa. Komunikacija lahko poteka preko protokolov HTTP, HTTPS ali SSL; kot zanimivost povejmo, da lahko uporabljamo HTTPS za klic na ELB, ki potem prekine zaščito in v notranjost sistema posreduje klic preko protokola HTTP. S tem zmanjšamo zakasnitve znotraj sistema, ki jih prinaša dodatna zaščita v protokolu HTTPS. Podrobnejše se v ELB ne bomo spuščali, saj je razumevanje pogojeno z poznavanjem AWS oblaka.

4.4 Toleranca okvar (koncept odklopnika)

Do sedaj smo spoznali, kako poteka komunikacija v porazdeljenem svetu mikrostoritev. Omenili smo, da so lahko posamezne storitve preobremenjene in tako blokirajo vse nadaljne klice (v sinhronem sistemu) ali pa se zapolni sporočilna vrsta (asinhron sistem). Ko preobremenjena storitev v sinhronem sistemu ne zmore obdelati vseh klicev, blokira tudi vse storitve, ki so jo klicale in čakajo na njen odgovor. Posledično tudi te storitve ne morejo nadaljevati svojega dela in blokirajo ostale storitve, ki so jih klicale. Zaradi preobremenjenosti ene instance določene storitve je tako blokirana celotna veriga klicev, kar pri velikem prometu pomeni, da se bo sistem sesul v nekaj minutah.

Da preprečimo to situacijo, lahko uporabimo koncept odklopnika [30], ki je že dolgo znan v svetu elektrotehnike. Odklopnik zaznava, kdaj več (zaporednih) klicev na določeno instanco mikrostoritve ni bilo uspešnih in v takem primeru prepreči nadaljne klice na to instanco. Neuspeh klica je lahko posledica različnih dejavnikov - čas klica se je iztekel, mikrostoritev vrača vedno isto napako ali pa storitev sploh ni dosegljiva. Odklopniku lahko natančno definiramo pravila, kdaj naj posreduje klice in kdaj naj jih

blokira, s pomočjo različnih parametrov (npr. blokiraj storitev šele po desetih neuspešnih klicih, ki so vsi vrnili isto napako ali pa, ko uspešnost klicev pade pod prag 80%). Skoraj obvezno je tudi spremljanje odklopnika z orodji za nadzorovanje, saj je odklopnik prva linija obrambe in nam bo prvi povedal, da je v sistemu prišlo do napak.

Odklopnik ima 3 stanja: odprt, zaprt in polodprt [30]. Privzeto je odklopnik v zaprtem stanju, kar pomeni, da posreduje klice in vse deluje normalno. V primeru, da pride do napake ali serije napak, ki prekršijo nastavljena pravila, odklopnik preide v odprto stanje - vse nadaljne klice blokira. Nato počaka nekaj časa (nastavljivo) in preide v polodprto stanje, kjer ponovno posreduje del klicev naprej in preverja njihov uspeh. V primeru zadovoljivega uspeha, preide v zaprto stanje.

Ob odprtem stanju in blokiranju klicev moramo nekaj vrniti klicoči storitvi. Najbolj so v uporabi naslednji koncepti [53]:

- **Custom fallback** - uporabimo drugo metodo ali mikrostoritev (ki bo do neke mere popravila škodo, ki jo povzroča blokada nedelujoče mikrostoritve)

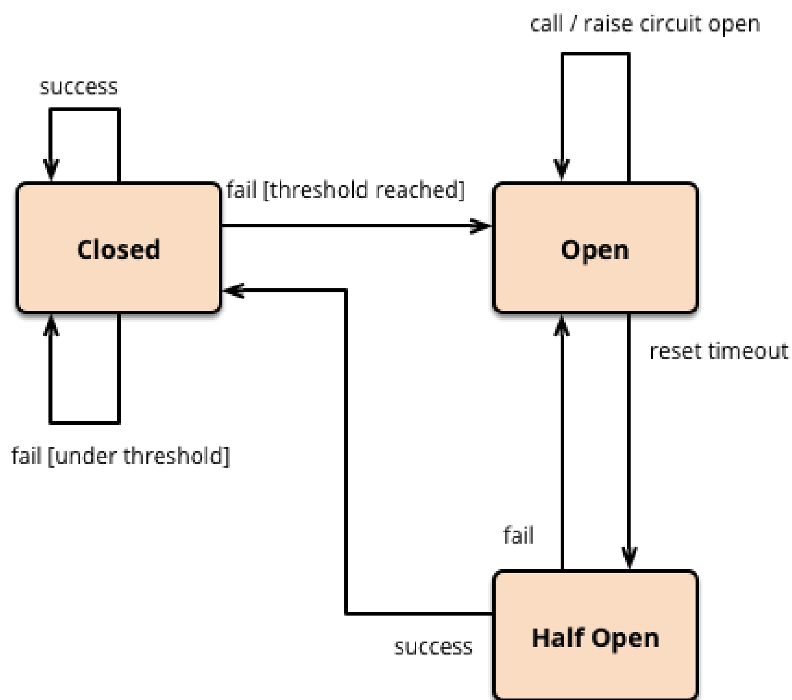
- **Fail silent** - vrnemo prazno vrednost

- **Fail fast** - nemudoma vrnemo zadnjo napako, ki jo je ta mikrostoritev vrnila (bolje da ima en uporabnik malce slabšo uporabniško izkušnjo, kot da se sesuje sistem)

Izmed naštetih je najbolj primerna uporaba "Custom fallback", ker v danem trenutku zagotavlja najboljšo rešitev, vendar je ob velikem številu mikrostoritev včasih težko zagotoviti popolno pokritost sistema na tak način.

Spoznajmo še realni primer odklopnika - Hystrix [8]. Je stabilen odprtokodni projekt pod okriljem Netflixa in trenutno eden izmed najbolj razširjenih odklopnikov za java mikrostoritve. Poleg odklopnika Hystrix ponuja tudi druge možnosti za toleranco okvar v sistemu, kot so semaforji, iztek časa in ponoven poskus (klica) in posebne niti za klice, prav tako pa omogoča

spremljanje aktivnosti in statusa sistema. Implementacija večinoma poteka s pomočjo anotacij, posebej so se z integracijo Hystrixa potrudili pri Springu.



Slika 4.4: Skica delovanja odklopnika. Levo zgoraj imamo zaprto stanje, katero bo obstalo dokler morebitne napake ne presežejo meje, ki smo jo nastavili. Nato bo odklopnik prešel v odprto stanje (desno zgoraj), kjer bo ostal nekaj časa, nakar bo prešel v pol odprto stanje (desno spodaj). V primeru, da se napake ne ponavljajo, bo prešel v zaprto stanje, drugače se vrne nazaj v odprto stanje.

Poglavje 5

Implementacija

Do sedaj smo se spoznali s teoretičnimi osnovami mikrostoritev, predvsem arhitekturnimi koncepti, in si pogledali nekaj orodij, ki so v takšni in drugačni obliki skoraj obvezna za uspešen razvoj in uporabo mikrostoritev v produkciji. Spoznali smo razliko med monolitnim in storitvenim pristopom h gradnji aplikacij. Sedaj pa je čas da pridobljeno znanje uporabimo v praksi. Preoblikovali bomo del obstoječe monolitne java EE aplikacije

Še prej pa si poglejmo, kdaj je primeren čas za prehod na mikrostoritve in komu je to že uspelo. Da pa ne zanemarimo monolitske strukture, ki je še vedno najbolj razširjena in uporabljena v java EE svetu, si bomo pogledali še primer monolita, ki ustreza kriterijem za preoblikovanje v mikrostoritveni sistem, a vendar uspešno in brez težav teče v trenutni obliki.

5.1 Kdaj je pravi čas za prehod?

"*Microservices are not a free lunch!*" (Wootton, 2014) [56].

G. Wotton je s to mislijo želel povedati, da mikrostoritve niso univerzalna rešitev za naše težave. Če nam prehod na mikrostoritve na eni strani odpravi en problem, nam bo na drugi strani povzročil tri nove. Sedaj se moramo samo vprašati, ali se nam bolj splača trpeti prvi problem ali pa zagristi in rešiti tri

nove [54].

Razvoj nove aplikacije z mikrostoritvenim pristopom, od začetka in brez predhodnih izkušenj z porazdeljenimi sistemi, je močno odsvetovan. Kot prvo se soočimo s problemom, kako zahtevano aplikacijo pravilno razdeliti v več problemskih domen in posledično mikrostoritev. Imamo zahteve celotne aplikacije v specifikaciji, do določene mere lahko zarišemo zasnovo in meje mikrostoritev, vendar obstaja velika možnost, da bomo tekom razvoja ugotovili napako in bo potrebno spreminjati meje nekaterih mikrostoritev, morda kreirati novo mikrostoritev in posledično prilagoditi obstoječe. Naslednji problem je izkušenost ekipe z razvojem in vzdrževanjem porazdeljenega sistema [21]. Spremeni se stil programiranja (tolerantni moramo biti do okvar, podvajati kodo, pravilno definirati vmesnike ...), poveča se potreba po sistemskih administratorjih (načrtovanje, nadzorovanje in vzdrževanje množice mikrostoritev ter registrov storitev, izenačevalcev obremenitve in tako naprej je zahtevno delo), testiranje postane zelo oteženo in še bi lahko naštevali. V kolikor nismo na vse to zelo dobro pripravljeni, smo obsojeni na neuspeh.

Za začetek vzemimo za primer zagonsko podjetje, ki razvija svoj produkt, ki na začetku sicer ne bo imel veliko uporabnikov, vendar obstaja možnost, da bo nekoč postal naslednji Netflix, Amazon ali Google. V pričakovanju take rasti podjetja bi bilo skoraj logično, da že od začetka razvijajo porazdeljen sistem s stevilnimi mikrostoritvami. Vendar bi ravno to znala biti njihova ključna napaka, zaradi katere bodo propadli. Predpostavimo lahko, da tako podjetje nima visoko usposobljenih razvijalcev, sistemskih administratorjev oziroma vsaj razvijalcev, ki so že kdaj razvijali porazdeljen sistem. Čeprav se kasneje izkaže za izjemno učinkovitega, tak sistem na začetku vzame veliko časa za postavitev in uskladitev, medtem ko je cilj vsakega zagonskega podjetja čimprejšnji vstop na trg in testiranje produkta. Če se odločijo za sistem mikrostoritev bodo dodatno zamaknili vstop na trg in jih lahko v tem času že prehiti konkurenca. V danem primeru je veliko bolj primerno začeti z monolitom in ga v primeru uspeha produkta kasneje preoblikovati v mikrostoritve.

Vzemimo še (hipotetično) spletno aplikacijo, ki je bila na začetku razvita kot monolit, vendar je zaradi potreb in porasta uporabnikov čez čas zrasla tako v smislu nabora funkcionalnosti kot v prometu. Razvija jo izkušena ekipa, ki dobro pozna njeno drobovje in zna začrtati jasne meje znotraj aplikacije (razdelitev na module). Zaradi nenehnega porasta prometa se spopadajo z velikimi stroški pri skaliranju aplikacije (horizontalno skaliranje z dodajanjem novih strežniških instanc). V danem primeru je prehod na mikrostoritve primeren, vendar se ga je treba lotiti pazljivo. Namesto popolnega preoblikovanja aplikacije bi bil primeren postopen prehod [23]. Najprej bi se sprejela odločitev, da vse dodane funkcionalnosti od tega trenutka dalje postanejo mikrostoritve (v kolikor, seveda, gre za "večjo" funkcionalnost, in ne npr. dodajanje novega gumba na spletno stran). To je tudi najlažje, saj ne potreba posega v obstoječo aplikacijo, prav tako bi ekipa na manjšem primeru spoznala potrebe in pasti mikrostoritev, katera tehnologija jim odgovarja in kako so se (razvijalci) odzvali na nov pristop. Nadaljevali bi lahko s postopnim razgrajevanjem obstoječega monolita. Najprej bi ga razdelili na dva dela in nato postopoma drobili vsak del posebej, dokler je še smiselno seveda. Podoben prehod sta izvedli podjetji Netflix in Soundcloud [23].

5.2 Uspešni primeri po svetu

5.2.1 Primer mikrostoritev: Netflix

Netflix je svetovno znani ponudnik videa na zahtevo. Dandanes vse njegove storitve tečejo na Amazon AWS oblaku. Celoten sistem sestoji iz več kot 500 mikrostoritev, ki procesirajo več kot 2 milijardi zahtevkov dnevno [20].

Netflix je prehod iz monolita, ki ga je hkrati razvijalo okoli 100 razvijalcev, na mikrostoritve v oblaku začel leta 2009 in ga končal leta 2011, ko pojem "mikrostoritve" še ni bil poznan. Tekom prehoda je podjetje razvilo več orodij za pomoč pri delu in razvoju mikrostoritev in jih kasneje tudi dalo na razpolago ostalim, v obliki odprte kode [20]. Nekaj od teh orodij smo že omenili (Eureka, Ribbon, Hystrix).

Glavni razlogi za prehod so bili dostopnost, skalabilnost in hitrost razvoja [20].

Pred prehodom je lahko celotna Netflix aplikacija postala nedosegljiva za več ur ob najmanjši napaki v programu [45], ki je zaustavila strežnik (kot npr. NullPointerException). Razvijalci so morali pregledati vsak svoj del kode in najti napako, kar je zahtevalo pozornost vseh razvijalcev. Dandanes, ko se zgodi napaka, je ta omejena le na določeno mikrostoritev, zaradi česar ne pride do izpada celotne Netflix aplikacije, prav tako je omejen nabor razvijalcev, ki morajo najti napako.

Pri skalabilnosti so se soočali z razkorakom med potrebami po skaliranju med posameznimi deli monolita in imeli težave pri gradnji dovolj hitrih podatkovnih centrov. Dodajanje novih instanc strežnikov je lahko trajalo po več ur. Na oblaku pa se nove instance sedaj zaženejo v parih minutah.

Porazdeljen sistem je Netflixu omogočil večjo modularnost znotraj podjetja, saj so ustvarili več kot 30 samostojnih razvijalskih ekip, vsaka je zadolžena za določene mikrostoritve. Na ta način se je pohitril čas razvoja in olajšalo konstantno nameščanje nove kode na produkcijsko okolje.

Netflix se je izkazal kot izvrsten primer prehoda na mikrostoritve in daje zgled ostalim podjetjem, kako izvesti prehod in upravljati kompleksen porazdeljen sistem [20]. Z množico svojih odprtokodnih orodij so približali mikrostoritve ostalim ter jim omogočili lažji in hitrejši prehod.

5.2.2 Primer monolitne aplikacije: ETSY

Etsy je spletna trgovina, ki jo mesečno obišče 60 milijonov obiskovalcev, ki opravijo več kot 1.5 milijarde ogledov strani. Celotna aplikacija je razvita kot velik modularen monolit (uporabljajo tehnologije PHP za API in MySQL za shranjevanje podatkov). Aplikacijo razvija več kot 150 razvijalcev, ki opravijo 50 in več novih namestitev na aplikacijski strežnik dnevno [55].

Njihov razvoj monolitne aplikacije bazira na več, pretežno organizacijskih, razlogih. Preferirajo preverjena orodja (PHP, MySQL ...), v nasprotju z novimi in vznemirljivimi tehnologijami ter in se s tem izognejo problemu, da

vsak razvijalec razvija v svojem najljubšem orodju oziroma tehnologiji, ko pa mora nekdo drug razumeti to kodo, se stvari upočasnijo, če ne celo ustavijo [39]. Nova orodja vpeljejo le, ko je skrajno potrebno, ko podrobno definirajo prednosti tega orodja in ko imajo dovolj razvijalcev z znanjem uporabe tega orodja. Nekaj orodij so morali celo razviti znotraj podjetja, ker na trgu ni bilo zadovoljive rešitve.

Uporabljajo lastne strežnike (edina izjema je gostovanje slik v Amazon S3 oblaku), ki jih s pomočjo izkušene ekipe in globokim poznavanjem tehnologije lahko izredno učinkovito optimizirajo in upravljajo.

To so glavni razlogi, ki jih v Etsyu dojemajo kot razlog za uspeh njihovega monolitnega pristopa. Na koncu dodajo, da redno preverjajo njihov nabor tehnologij in zasnovo sistema in primerjajo realne stroške in potencialni prehranek pri uporabi drugih arhitekturnih rešitev (kot so mikrostoritve). Zaenkrat še niso prišli do točke preloma.

Vidimo lahko, da je tudi monolit mogoče uspešno razvijati v njegovih kasnejših fazah (ko bi, brez odstranjevanja tehničnega dolga, že ratal preveč kompleksen) in ga skalirati. To lahko jemljemo kot dokaz, da katero koli pot izberemo (mikrostoritve ali monolit), je v prvi vrsti uspeh odvisen od znanja in (razvijalske) kulture v podjetju.

5.3 Primer preobrazbe dela obstoječega monolita v sistem mikrostoritev

Opremljeni z znanjem o mikrostoritvah bomo sedaj to znanje uporabili na praktičnem primeru. Da ne ustvarjamo teoretične monolitne aplikacije samo za potrebe te diplomske naloge, si izberimo primer iz realnega sveta. Tako lahko iz prve roke vidimo, kako poteka preoblikovanje monolita v mikrostoritve in spoznamo, da priporočila in pravila kako graditi mikrostoritve, v realnem svetu ne moremo vedno upoštevati. Osredotočili se bomo na arhitekturno zasnovo in izbiro primernih tehnologij. Podrobnosti tehnične implementacije (programska koda) ne bomo obravnavali.

Izbrana monolitna aplikacija je zelo obsežna, zato se bomo osredotočili le na del aplikacije in ga preoblikovali v sistem mikrostoritev. Najprej bomo opisali obstoječo funkcionalnost, nato na njeni podlagi zarisali sistem mikrostoritev in na koncu še izbrali tehnologije za implementacijo. Naš cilj je zasnovati sistem mikrostoritev, ki bodo omogočale večjo izolacijo napak in hitrejše avtomatsko nameščanje na aplikacijski strežnik, bodo organizirane okoli posameznih poslovnih domen in bo upravljanje z njimi posledično lažje. Prav tako želimo decentralizirati shranjevanje podatkov tega dela aplikacije in omejiti dostop do zunanjih sistemov.

5.3.1 Obstoječa monolitna aplikacija

Za primer bomo uporabili spletni portal Enotna Kontaktna Točka (v nadaljevanju EKT), ki jo je razvilo podjetje Medius, v katerem sem trenutno zaposlen. EKT je namenjen elektronskemu urejanju zadev v zvezi s podjetji (odpiranje, zapiranje, spremembe ...) in ga uporabljajo podjetniki (v nadaljevanju vlagatelji) in uradniki na ministrstvih Republike Slovenije. Uporablja tehnologijo java EE in teče na aplikacijskem strežniku Wildfly. Mi se bomo osredotočili na proces ustvarjanja, izpolnjevanja in oddaje obrazcev v odločanje uradnikom (v nadaljevanju: Proces). Proces smo izbrali zaradi različnega povpraševanja uporabnikov po posameznih korakih znotraj Procesa in povezovanja z zunanjimi sistemi, ki se tičejo le Procesa in ne preostale EKT aplikacije.

Na sliki 5.1 lahko vidimo arhitekturo obstoječega monolita, ki je lepo zasnovan in razdeljen v več modulov. Sestavljen je iz jedra projekta (znotraj območja EKT2 Server), baz podatkov (znotraj območja Oracle DB) in zunanjih sistemov (izven omenjenih območij). Nas zanima predvsem modul Urejevalnik obrazcev, ki vsebuje logiko za ustvarjanje novih vrst obrazcev in izpolnjevanje ter pregledovanje (izpolnjenih) obrazcev. Proces oddaje obrazca se zaključi z elektronskim podpisom in elektronskim vročanjem obrazca uradnikom v odločanje. Ti dve funkcionalnosti sta vsebovani v modulu Front.

Urejevalnik obrazcev je povezan z moduloma Administracija in Front.

Preko modula Administracija uradniki ustvarjajo nove obrazce, katerih XML struktura se shrani v bazo EKT2-Jedro kot CLOB. Preko modula Front vlagatelji izpolnjujejo, pregledujejo, elektronsko podpisujejo in vročajo obrazce. Podatki iz teh obrazcev se nato preko modula EDV (Enotni Dokumentni Vmesnik) shranijo v bazo podatkov Jedro-EDV (shranijo se samo podatki obrazca, XML struktura obrazca je še vedno shranjena v EKT2-Jedro bazi podatkov). Avtentikacija uradnikov poteka v okviru modula Administracija preko zunanjega sistema Varnostna shema, avtentikacija vlagateljev pa v okviru modula Front preko zunanjega sistema SI-CAS ali podpisne komponente na vlagateljevem računalniku (ni na sliki 5.1).

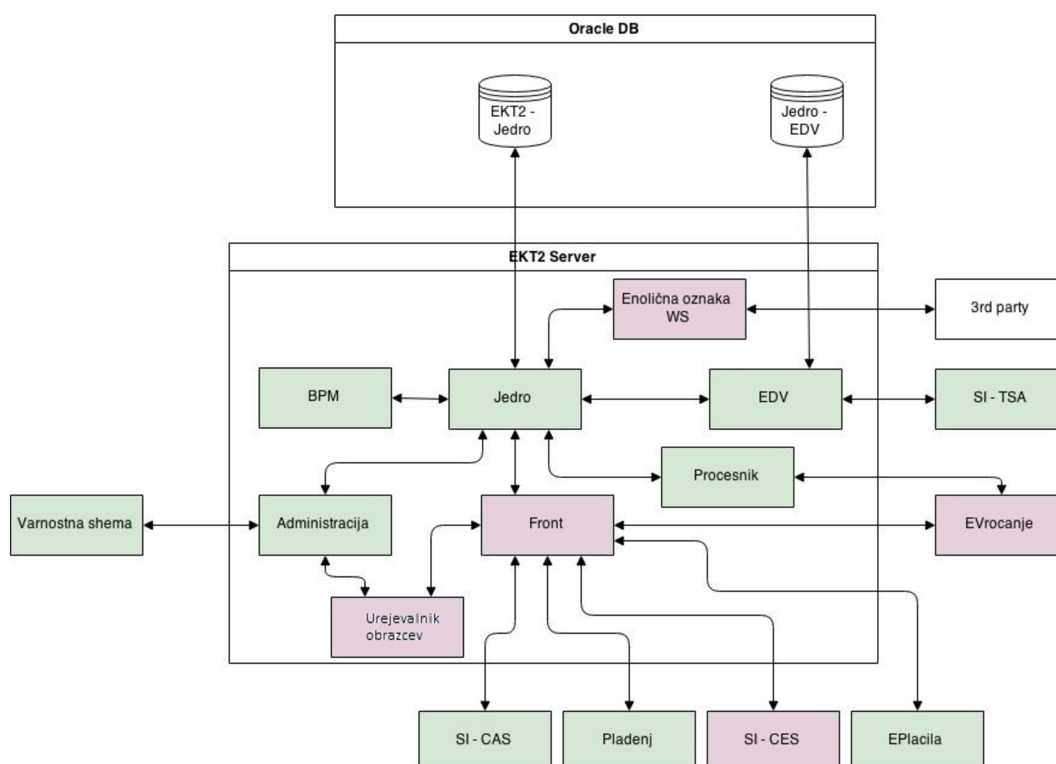
Naj najprej razložimo še razliko med pojmoma obrazec in vloga. Obrazec je del vloge. Vloga predstavlja celoten proces urejanja neke poslovne zadeve s strani vlagatelja. Vsebuje pridobivanje informacij (izpolnjevanje obrazca), podpisovanje, odločanje o vlogi, vročanje odločitve, plačilo stroškov s strani vlagatelja in tako dalje.

Strnimo celoten postopek ustvarjanja in izpolnjevanja obrazca iz stališča uporabnika (uradnik ali vlagatelj):

1. Ustvarjanje obrazca preko modula Administracija (uradnik):
   - (a) avtentikacija (preko zunanjega sistema Varnostna shema)
   - (b) ustvarjanje obrazca (preko uporabniškega vmesnika ali neposredno v XML datoteki)
2. Izpolnjevanje obrazca preko modula Front (vlagatelj):
   - (a) preusmeritev iz drugih spletnih portalov na EKT
   - (b) avtentikacija (preko zunanjega sistema SI-CAS)
   - (c) izpolnjevanje obrazca (določeni podatki so lahko predizpolnjeni, pridobijo se iz zunanjega sistema Pladenj)
   - (d) pregled izpolnjenega obrazca

(e) podpis obrazca (elektronsko preko zunanjega sistema SI-CES, preko podpisne komponente proXSign ali fizični podpis, skeniranje in nalaganje na EKT)

(f) vročanje uradnikom (preko zunanjega sistema eVročanje ali preko modula Procesnik)



Slika 5.1: Arhitekturna zasnova obstoječe monolitne aplikacije EKT. Deli se na jedro proijekta (znotraj območja EKT2 Server), baze podatkov (znotraj območja Oracle DB) in zunanjih sistemov (zunaj omenjenih področij)

Vlagatelj lahko vstopi v Proces v katerikoli točki od 2b do 2e. Točka vstopa je odvisna od tega, kateri je naslednji korak v izpolnjevanju vloge. Za primer vzemimo vlagatelja, ki je v ponedeljek izpolnil obrazec, nato pa čakal do srede, da ga dokončno odda. V sredo bi ga sistem ob prijavi usmeril v podpisovanje obrazca in ne v izpolnjevanje.

Poglejmo še, kaj se zgodi s podatki pri tem Procesu. Pri koraku 1b se XML struktura ustvarjenega obrazca zapiše v bazo podatkov EKT2-Jedro. Kasneje pri koraku 2c se XML struktura obrazca prebere iz baze EKT2-Jedro in združi s programsko kodo ter se prikaže uporabniku. Pri istem koraku se nato izpolnjeni podatki (in morebitne dodane priponke) preko modula EDV zapišejo v bazo Jedro-EDV. Pred tem se priponke še pregledajo z antivirusnim programom. Pri koraku 2d poteka branje XML strukture iz baze EKT2-Jedro in podatkov (ter morebitnih priponk) iz baze Jedro-EDV. Pri koraku 2e se iz baze Jedro-EDV prenese PDF obrazec in podpiše, ki se nato shrani v isto bazo. Pri vročanju se podpisane podatke skupaj z morebitnimi priponkami posreduje v zunanji sistem eVročanje ali v modul Procesnik in se jih odstrani iz baze Jedro-EDV.

Vidimo lahko, da je Proces poslovno neodvisen od ostale aplikacije. Deli si skupno bazo z ostalo aplikacijo. Do njega se dostopa preko dveh uporabniških vmesnikov (Administrator in Front), ki hkrati predstavljata dostop tudi do drugih delov aplikacije. Proces je edini del EKT, ki potrebuje antivirusni program in se povezuje z zunanjim sistemom Pladenj Znotraj Procesa obstajajo različne potrebe po povezovanju z zunanjimi sistemi iz česar sledi, da imajo dostop do vseh zunanjih sistemov tudi tisti deli Procesa, ki določenega zunanjega sistema ne potrebujejo (obstaja možnost hroščev ali zlorab).

5.3.2 Funkcionalna razgradnja dela monolitne aplikacije v mikrostoritve

Sedaj, ko smo se podrobneje spoznali z monolitno aplikacijo in poslovno logiko Procesa, ga lahko z upoštevanjem domensko usmerjenega načrtovanja in principa enojne odgovornosti razbijemo na več mikrostoritev. Če pogledamo seznam korakov v prejšnjem razdelku, lahko izločimo naslednje problemske domene:

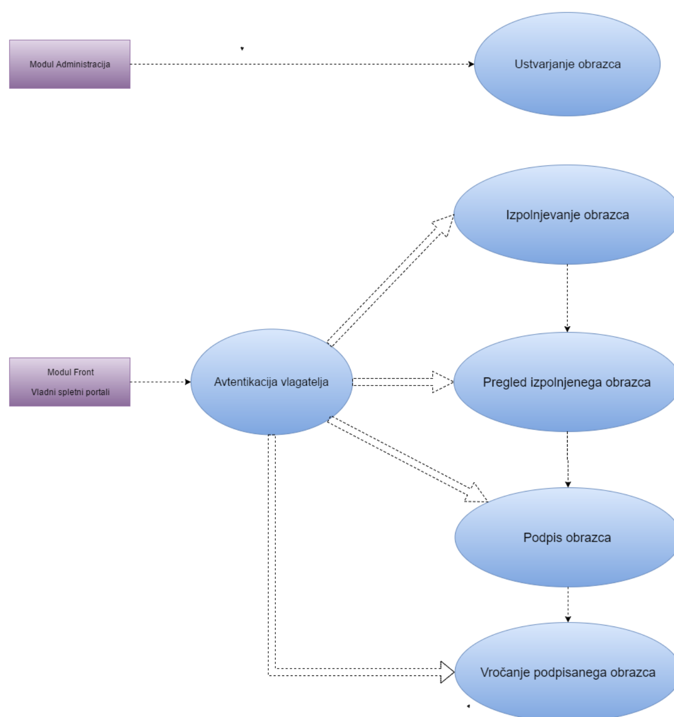
- Avtentikacija vlagatelja
- Ustvarjanje obrazca

- Izpolnjevanje obrazca
- Pregled izpolnjenega obrazca
- Podpis obrazca
- Vročanje podpisanega obrazca

Vsaka od teh domen je odgovorna za eno funkcionalnost. Tako smo ugodili obema principoma omenjenima v uvodnem odstavku. Avtentikacijo uradnikov nismo uvrstili med problemske domene, saj se uradniki pred ustvarjanjem novega obrazca zagotovo prijavijo v sistem, drugače nimajo dostopa do te funkcionalnosti. Vlagatelji pa lahko v EKT pridejo preko spletne povezave do vloge iz drugih spletnih portalov, torej predhodno niso prijavljeni v sistem. Na podlagi problemskih domen lahko sedaj narišemo diagram na sliki 5.2.

Sedaj se vprašajmo, kateri podatki krožijo po sistemu in na podlagi tega določimo baze podatkov. Omenili smo, da se tekom Procesa podatki zapisujejo v dve ločeni bazi, EKT2-Jedro in Jedro-EDV (obe sta relacijski). V prvo se zapisuje XML struktura obrazca, medtem ko druga vsebuje izpolnjene podatke in morebitne priponke. Poleg XML strukture obrazca se v bazo EKT2-Jedro zapiše še unikatni identifikator vloge, kateri pripada obrazec. Namesto da to zapisujemo v relacijsko bazo, raje uporabimo ključ/vrednost bazo podatkov, ki je za ta način shranjevanja bolj primerna. Poimenujmo jo EKT2-XML. Glede shranjevanja v bazo Jedro-EDV ne moremo veliko narediti, saj zapisovanje poteka preko modula EDV, ki ga potrebujejo tudi drugi deli aplikacije. Torej pustimo to bazo tako kot je, saj bomo zapisovali in brali iz nje preko modula EDV. Domene, ki bodo komunicirale z modulom EDV so Izpolnjevanje obrazca, Pregled izpolnjenega obrazca, Podpis obrazca in Vročanje podpisanega dokumenta.

Do sedaj še nismo omenili, vendar je v sistemu potrebno vzdrževati konsistentnost podatkov, kar pomeni, da moramo pri shranjevanju podatkov in komunikaciji izbirati tehnologije, ki to omogočajo (imeti moramo CP sistem,



Slika 5.2: Diagram problemskih domen in povezav med njimi.

če se sklicujemo na CAP teorem). To je pomembno zato, ker XML strukturo obrazca potrebuje več problemskih domen - ustvarjanje, izpolnjevanje in pregled obrazca. Če bi imeli AP sistem, bi lahko vsaka domena imela svojo bazo, ki bi jo sproti osveževala, ko bi bil ustvarjen nov obrazec. Vendar si pri izpolnjevanju obrazca ne moremo privoščiti, da vlagatelj izpolne obrazec, ki ga naslednji korak, pregled obrazca, morebiti še nima shranjenega v svoji bazi (pride do napake in uporabnik bi moral počakati na osvežitev baze podatkov domene Pregled obrazca). Torej morajo vse tri problemske domene uporabljati bazo EKT2-XML. Naleteli smo na novo problemsko domeno:

- branje in zapisovanje v bazo EKT2-XML

Elegantna rešitev je dodatna mikrostoritev, ki bo izpostavila API, preko katerega bo možno branje in zapisovanje v bazo EKT2-XML. S tem se izgo-

nemo direktnemu dostopu do baze s strani vseh treh problemskih domen in morebitnim hroščem, ki jih prinaša neomejen dostop do baze. Lahko bi uporabili tudi alternativen pristop - domena Ustvarjanje obrazca bi imela svojo bazo in bi izpostavila API za dostop do podatkov. Slabost tega pristopa je skaliranje, saj se ustvari veliko manj obrazcev kolikor se jih izpolni. Iz tega sledi, da bi bilo potrebno ob skaliranju izpolnjevanja ali pregledovanja obrazcev skalirati tudi ustvarjanje obrazcev, samo zato, da lahko obdelamo vse zahtevke za branje.

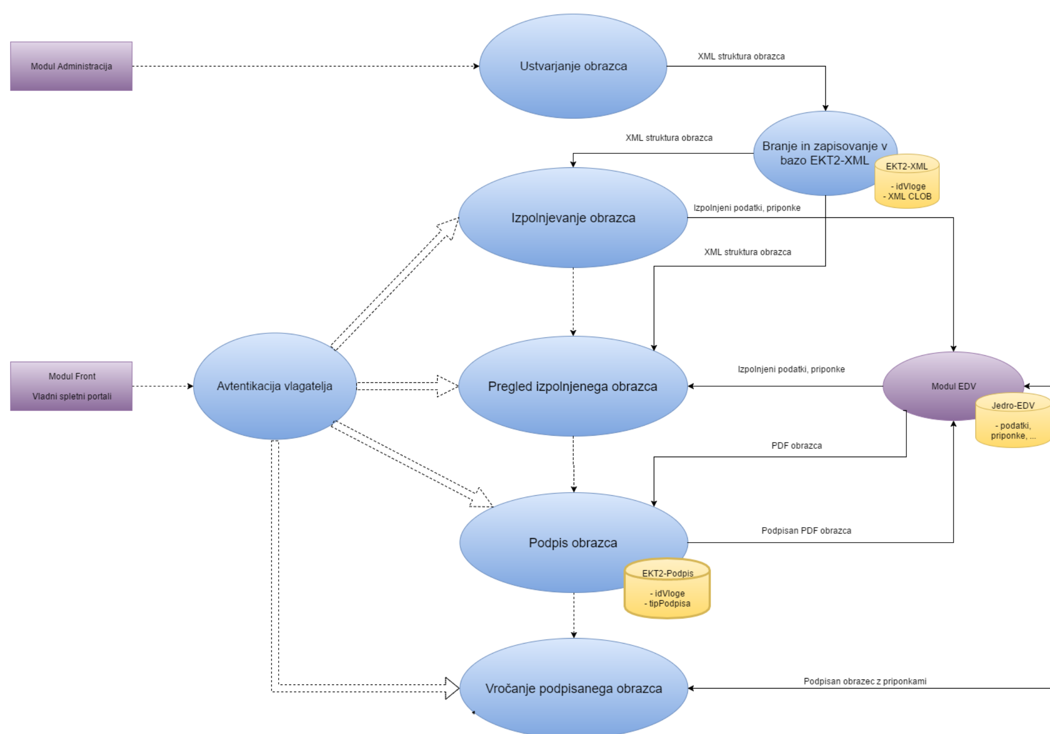
Podpisovanje obrazca lahko poteka na več načinov - elektronsko preko zunanjega sistema SI-CES ali preko podpisne komponente, nameščene na računalniku vlagatelja ali ročno (tisk obrazca, podpis, skeniranje, nalaganje na EKT2). Nekatere vloge omogočajo vse vrste podpisovanja obrazcev, nekatere pa le določene. Podatki o tem so shranjeni v bazi EKT2-Jedro, vendar ker jih uporablja le domena podpisovanja obrazcev, je bolj primerno, da jih premaknemo v ločeno bazo. Zapis je sestavljen iz unikatnega identifikatorja vloge in tipa podpisovanja, torej je baza podatkov ključ/vrednost povsem primerna. Poimenujmo jo EKT2-Podpis.

Dopolnimo prejšnji diagram z novo problemsko domeno, bazami podatkov in podatki, ki se pošiljajo po povezavah (slika 5.3).

Sedaj je čas, da problemske domene zamenjamo z mikrostoritvami in jih primerno poimenujemo:

- Avtentikacija vlagatelja → Avtentikacija
- Ustvarjanje obrazca → Ustvarjanje
- Branje in zapisovanje v bazo EKT2-XML → EJB (ker predstavlja zaledni, uporabniku neviden sistem)
- Izpolnjevanje obrazca → Izpolnjevanje
- Pregled izpolnjenega obrazca → Pregledovanje
- Podpis obrazca → Podpisovanje

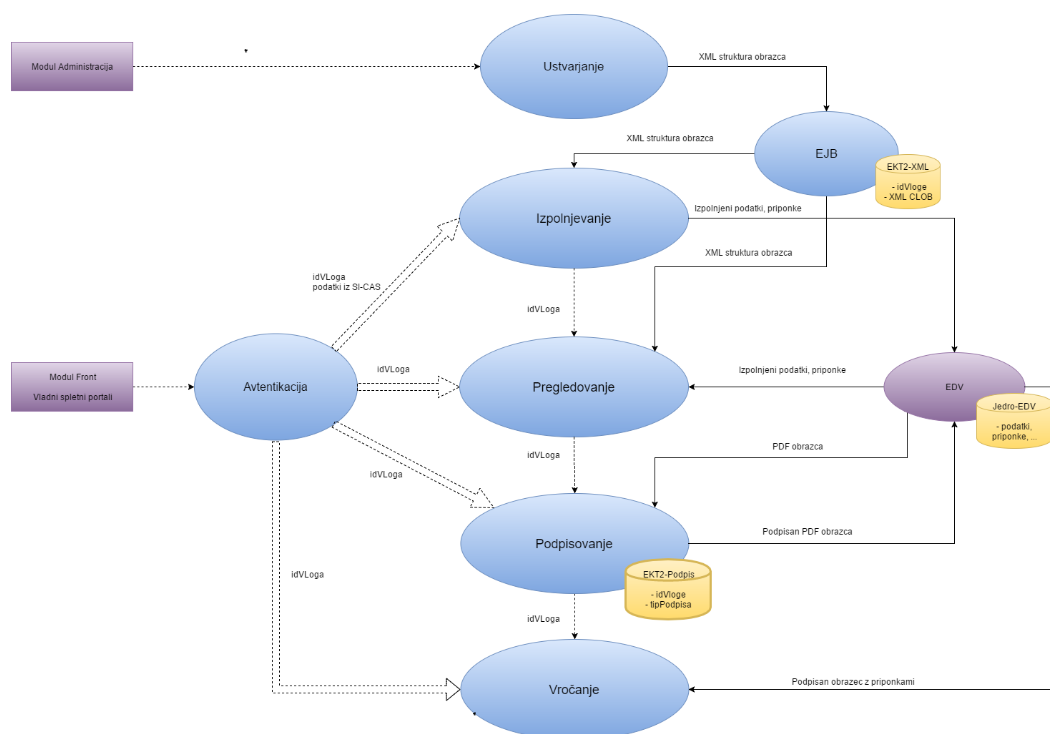
- Vročanje podpisanega obrazca → Vročanje



Slika 5.3: Diagramu s slike 5.2 smo dodali problemsko domeno Branje in zapisovanje v bazo EKT2-XML, baze podatkov, zunanji sistem EDV in informacije, kateri podatki se pretakajo po povezavah.

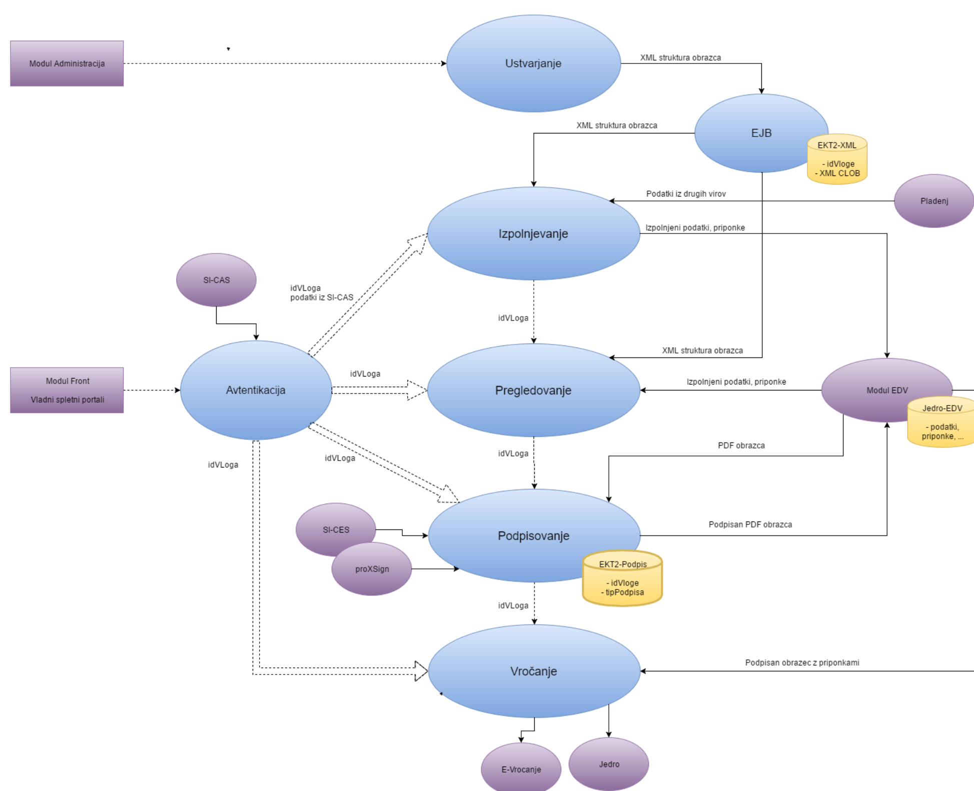
V naslednjem koraku definirajmo načine komunikacije med mikrostoritvami. Najprej poudarimo, da so vse mikrostoritve, razen EJB, namenjene interakciji z uporabnikom. Torej se zahteva po posamezni mikrostoritvi pojavi le, če imamo uporabnika s to zahtevo. V vmesnem času (ko ni nobenega uporabnika) mikrostoritve mirujejo. Potreba po posamezni mikrostoritvi je torej sinhrona. Tudi komunikacija z zunanjimi sistemi (trenutno imamo sicer samo EDV) je večinsko sinhrona, tudi v primeru, ko le pošiljamo podatke. Bolje je, da uporabnika ne spustimo naprej, dokler se ne prepričamo, da so

bili poslani podatki uspešno prejeti. V nasprotnem primeru bi v naslednjem koraku Procesa vlagatelj čakal, da se asinhrono poslani podatki obdelajo. Za sinhrono komunikacijo izberimo protokol REST, ki bo uporabljen na vseh povezavah na sliki []. Mikrostoritve Pregledovanje, Podpisovanje in Vročanje potrebujejo za svoje delovanje le unikatni identifikator vloge (v nadaljevanju idVloga), ki jim ga posredujemo ob klicu. Izpolnjevanje pa poleg idVloga potrebuje še podatke iz sistema SI-CAS (kasneje ga bomo povezali z Avtentikacijo), ki se bodo vnesli v obrazec in s katerimi bo lahko pridobil dodatne podatke o vlagatelju iz zunanjega sistema Pladenj. Na sliki 5.4) vidimo diagram s slike 5.3) dopolnjen s posredovanimi podatki med mikrostoritvami in uporabo novega poimenovanje mikrostoritev.



Slika 5.4: Spremenjeno poimenovanje mikrostoritev in dodan parameter idVloga, ki se posreduje med mikrostoritvami.

Našemu sistemu mikrostoritev manjka še povezovanje z zunanjimi sistemi iz slike 5.1). Pri komunikaciji smo omenili zunanja sistema SI-CAS in Pladenj. Prvega uporablja Avtentikacija, drugega pa Izpolnjevanje. Pri podpisovanju dokumentov sodeluje zunanji sistem SI-CES, če ta ni dostopen se kot rezerva uporabi podpisna komponenta na vlagateljevem računalniku, s katero Podpisovanje komunicira preko javascript knjižnice v brskalniku. Vročanje pošilja vloge v zunanji sistem eVročanje ali pa jih posreduje nazaj v EKT2, v modul Jedro (ki jih posreduje modulu Procesnik), kjer sledi nadaljna obravnava. Končen diagram lahko vidimo na sliki 5.5).



Slika 5.5: Končen diagram skupaj z zunanjimi sistemi, na katere se povezujejo naše mikrostoritve.

Iz monolitne aplikacije EKT nam je uspelo izločiti Proces in ga preobraziti v sistem mikrostoritev, ki komunicira z EKT in ostalimi zunanjimi sistemi,

ki jih potrebuje. Sledili smo načelom domensko usmerjenega načrtovanja in principu enojne odgovornosti in tako dobili mikrostoritve, ki predstavljajo zaključene funkcionalnosti. Med njimi smo uvedli preprosto sinhrono REST komunikacijo. Do neke mere smo decentralizirali upravljanje s podatki. Zaradi zahtev po konsistentnosti podatkov imamo samo eno bazo za shranjevanje XML strukture obrazcev. Antivirusni program lahko omejimo na mikrostoritvi Izpolnjevanje in Podpisovanje ter se s tem izognemo obremenitvi EKTja in ostalih mikrostoritev.

Zavedati se moramo, da je dan sistem skalabilen samo do določene mere. Vse povezave z EKT monolitom in zunanjimi viri so potencialna ozka grla, saj potekajo sinhrono. Če pride do povečanega prometa in povečamo število instanc mikrostoritev, je potrebno horizontalno skalirati tudi EKT. V tem primeru bi bilo vseeno, če Proces ostane znotraj EKT. Vendar naš primarni cilj ni bila večja skalabilnost sistema, temveč predvsem lažje razumevanje poslovne logike, hitrejše nameščanje na aplikacijski strežnik, omejitev dostopa do zunanjih sistemov in izolacija napak.

5.3.3 Tehnična implementacija

Obstoječi monolit EKT uporablja tehnologijo java EE in teče na aplikacijskem strežniku Wildfly. Uporablja sinhrono komunikacijo in relacijske baze podatkov. V naši arhitekturni zasnovi smo že predvideli sinhrono komunikacijo preko protokola REST, medtem ko smo za hranjenje nekaterih podatkov vpeljali ključ/vrednost bazo podatkov. Da bo izločitev mikrostoritev iz monolita preprosta, za mikrostoritveno šasijo uporabimo Wildfly Swarm. Za odkrivanje storitev se bomo poslužili orodja Consul, ki bo hkrati skrbel tudi za preprosto izenačevanje obremenitev. Morebitne okvare v sistemu in izolacijo napak bo zagotavljal Hystrix. V nadaljevanju še pojasnimo, zakaj so bile izbrane omenjene tehnologije.

- **Hranjenje podatkov - ključ/vrednost**

  Za bazi EKT2-XML in EKT2-Podpis, ki sta tipa ključ/vrednost, kot

implementacijo izberimo MongoDB [9]. MongoDB je popularna NoSQL baza, ki omogoča shranjevanje podatkov brez vnaprej predvidenih shem, je skalabilna in primerna za večje zapise.

Zaradi narave aplikacije nismo izkoristili možnosti poliglotnega shranjevanja podatkov, ker ni potrebe za to iz poslovnega vidika, poleg tega pa to poenostavi sam razvoj in vzdrževanje, ker je uporabljena ista tehnologija v vseh bazah podatkov.

- **Mikrostoritvena šasija - Wildfly Swarm**
  Ker je EKT monolit razvit v tehnologiji java EE, je bila podpora java EE najbolj pomemben faktor pri izbiri šasije. S tem nemudoma izločimo Spring Boot in Dropwizard. Spring Boot bi zahteval, da temeljito preobrazimo celotno programsko kodo z uporabo Spring programskih knjižnic, Dropwizard pa bi zahteval podobno preobrazbo kot Spring, saj prav tako razvijalca sili k uporabi določenih knjižnic. Ostaneta nam le še Wildfly Swarm in KumuluzEE, ki oba temeljita na tehnologiji java EE. Zaradi svoje stopnje razvitosti in podpore različnim java EE tehnologijam je najbolj primeren Wildfly Swarm. Nezanemarljivo je tudi dejstvo, da je Wildfly Swarm razgrajen aplikacijski strežnik Wildfly, ki je bil uporabljen že za prvotno aplikacijo. S tem se prehod še dodatno pohitri.

- **Odkrivanje storitev in preprosto izenačevanje obremenitve - Consul**
  Pri izbiri orodij za odkrivanje storitev vzemimo za glavni pogoj čim lažjo implementacijo z Wildfly Swarm šasijo. Na žalost izmed omenjenih orodij edino Consul ponuja integracijo z Wildfly Swarm. Pri Eureki in ZooKeeperju bi bilo iskanje in registracijo potrebno ročno implementirati v programski kodi, poleg tega pa Consul ponuja tudi boljše preverjanje zdravja mikrostoritev. Dodaten minus Eureke je tudi prilagojenost za Amazon AWS oblak, ki ga v našem primeru ne uporabljamo.

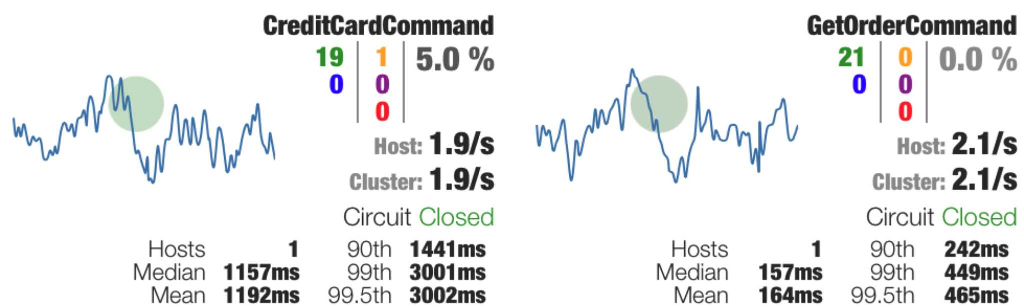
Consul je tako najprimernejše orodje za odkrivanje storitev v našem primeru, istočasno pa lahko izkoristilmo tudi preprosto izenačevanje obremenitve, ki jo Consul ponuja ob DNS poizvedbah. Integracija z Wildfly Swarm šasijo omogoča sila preprosto registracijo storitev. V konfiguracijski datoteki Swarma le definiramo IP naslov in port, kjer se nahaja Consul, za samo registracijo storitve bo poskrbel Swarm v ozadju [10].

Ob DNS poizvedbah nam Consul vrne le eno instanco posamezne mikrostoritve, ki jo naključno izbere izmed obstoječih instanc. Glede na predvidene potrebe našega sistema je tako preprosto izenačevanje obremenitve zadostno.

- **Toleranca okvar - Hystrix**

  Kot odklopnik je za naše potrebe najbolj primeren Hystrix, zaradi razširjenosti, stabilne verzije, preproste integracije in spremljanja zdravja mikrostoritev (v sklopu te diplomske naloge tega nismo podrobneje obravnavali). Zahteva preproste posege v programsko kodo - v kolikor želimo iz določenega java razreda klicati neko mikrostoritev, moramo implementirati razred HystrixCommand, ki potem v ozadju poskrbi za toleranco na morebitne okvare in za posredovanje informacij za lažje spremljanje sistema [51].

  Hystrix nato iz vsake instance mikrostoritev agregira podatke in nam omogoča vizualni pregled uspešnosti klicev na mikrostoritve. Za agregiranje podatkov skrbi Turbine, vizualni pregled pa omogoča Hystrix Dashboard, primer vidimo na sliki 5.6.



Slika 5.6: Primer, kako zgleda Hystrix Dashboard. Vidimo lahko dve metodi, za kateri Hystrix spremlja zdravje.

Poglavje 6

Sklepne ugotovitve

Uporaba mikrostoritev v svetu vztrajno raste. Največje korake na tem področju delajo veliki igralci (Netflix, Amazon ...), ki s tem spodbujajo tudi preostala podjetja, da jim sledijo. Veliko k temu pripomorejo tudi številna orodja in knjižnice za pomoč pri razvoju in upravljanju mikrostoritev, ki so že razvita do te mere, da se lahko varno uporabljajo v produkciji. Vendar je odstotek mikrostoritvenih aplikacij kljub vsemu relativno nizek. Monolitne aplikacije ne bodo izginile čez noč in tako je tudi prav, saj mikrostoritve (oziroma porazdeljeni sistemi, če posplošimo) niso primerni za vse vrste aplikacij.

V diplomski nalogi smo predstavili različne načine komunikacije med mikrostoritvami, kako hranijo podatke, načine združevanja v večje sisteme in različna orodja, ogrodja in knjižnice za lažji in hitrejši razvoj mikrostoritev. Če se ozremo nazaj, lahko vidimo, da ni enotnega konsenza, kako razvijati mikrostoritve. Veliko je odvisno tudi od poslovnih potreb, ki jih hočemo uresničiti z mikrostoritvenim sistemom. Zaradi želje po šibki sklopljenosti je priporočena uporaba asinhrone komunikacije, vendar smo kasneje spoznali, da asinhrona komunikacija ni primerna za vsak poslovni problem. Kljub temu zopet omenimo zanimivi alternativi sinhroni komunikaciji - dogodkovno vodeno programiranje in tudi koncept ločene poizvedbe in zapisa podatkov.

Poleg vsega, kar smo zajeli v diplomski nalogi, obstaja še nekaj drugih

področij, ki smo jih le bežno omenili. Govorimo o nadzorovalskih in upravljalskih orodjih, brez katerih bi bilo nadorovanje delovanja mikrostoritev prava nočna mora. (Integracijsko) testiranje mikrostoritev je tudi svojevrsten izziv v katerega se nismo poglobili. Obe področji sta zelo pomembni za vzdrževanje sistema mikrostoritev, medtem ko smo ju lahko pri samem razvoju malce zapostavili.

Na koncu še opozorimo, da ta diplomska naloga ne sme biti edini vir in izhodišče za razvoj sistema mikrostoritev. Služi naj bolj za uvod v svet mikrostoritev (z manjšim poudarkom na java programskem jeziku), na podlagi katerega bo bralec sposoben kritične raziskave področja in izbire najprimernejših konceptov in tehnologij za postavitev lastnega sistema mikrostoritev. Pri zbiranju informacij smo se oprli na spletne vire (strokovnih knjig o razvoju mikrostoritev ni veliko) k branju katerih vabimo tudi bralca. S pomočjo diplomske naloge, virov in brskanja po internetu si drznemo trditi, da bo bralec pridobil dovolj znanja za postavitev sistema mikrostoritev.

Literatura

[1] Dosegljivo: http://www.dropwizard.io/1.0.0/docs/. [Dostopano 25. 6. 2016].

[2] Dosegljivo: http://wildfly-swarm.io/. [Dostopano 25. 6. 2016].

[3] Dosegljivo: https://ee.kumuluz.com/. [Dostopano 25. 6. 2016].

[4] Dosegljivo: https://github.com/Netflix/eureka. [Dostopano 25. 7. 2016].

[5] Dosegljivo: https://zookeeper.apache.org/. [Dostopano 25. 7. 2016].

[6] Dosegljivo: https://www.consul.io/. [Dostopano 25. 7. 2016].

[7] Dosegljivo: https://github.com/netflix/ribbon. [Dostopano 2. 8. 2016].

[8] Dosegljivo: https://github.com/Netflix/Hystrix. [Dostopano 2. 8. 2016].

[9] Dosegljivo: https://www.mongodb.com/. [Dostopano 29. 8. 2016].

[10] Dosegljivo: http://wildfly-swarm.io/tutorial/step-4/. [Dostopano 16. 8. 2016].

[11] Elastic load balancing. Dosegljivo: https://aws.amazon.com/elasticloadbalancing/. [Dostopano 2. 8. 2016].

[12] Gossip protocol. Dosegljivo: https://www.consul.io/docs/internals/gossip.html. [Dostopano 25. 7. 2016].

[13] Load balancing (computing). Dosegljivo: https://en.wikipedia.org/wiki/Load_balancing_(computing). [Dostopano 11. 8. 2016].

[14] Monolithic application. Dosegljivo: https://en.wikipedia.org/wiki/Monolithic_application. [Dostopano 22. 7. 2016].

[15] Service registration and discovery. Dosegljivo: https://spring.io/guides/gs/service-registration-and-discovery/. [Dostopano 13. 8. 2016].

[16] Spring boot. Dosegljivo: http://projects.spring.io/spring-boot/. [Dostopano 15. 7. 2016].

[17] Working with load balancers. Dosegljivo: https://github.com/Netflix/ribbon/wiki/Working-with-load-balancers, 2014. [Dostopano 2. 8. 2016].

[18] Oracle announces winners of the 2015 duke's choice award. Dosegljivo: https://www.oracle.com/corporate/pressrelease/dukes-award-102815.html, 2015. [Dostopano 25. 6. 2016].

[19] Service discovery overview. Dosegljivo: http://www.simplicityitself.io/getting/started/with/microservices/2015/06/10/service-discovery-overview.html, 2015. [Dostopano 25. 7. 2016].

[20] Why you can't talk about microservices without mentioning netflix. Dosegljivo: http://blog.smartbear.com/microservices/why-you-cant-talk-about-microservices-without-mentioning-netflix/, 2015. [Dostopano 22. 7. 2016].

[21] Daniel Bryant. Is it time for your 'microservices checkup'? Dosegljivo: https://opencredo.com/is-it-time-for-a-microservices-checkup/, 2016. [Dostopano 22. 7. 2016].

[22] Radu Butnaru. Fault-tolerant microservices with netflix hystrix. Dosegljivo: http://www.todaysoftmag.com/article/1531/fault-tolerant-microservices-with-netflix-hystrix. [Dostopano 2. 8. 2016].

[23] Phil Calçado. Building products at soundcloud —part i: Dealing with the monolith. Dosegljivo: https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith, 2014. [Dostopano 15. 8. 2016].

[24] Richard Clayton. Failing at microservices. Dosegljivo: https://rclayton.silvrback.com/failing-at-microservices, 2014. [Dostopano 15. 7. 2016].

[25] Melvin E. Conway. How do committees invent? Dosegljivo: http://www.melconway.com/Home/Committees_Paper.html, 1968. [Dostopano 20. 6. 2016].

[26] Peter Deutsch. The eight fallacies of distributed computing. Dosegljivo: https://blogs.oracle.com/jag/resource/Fallacies.html. [Dostopano 15. 7. 2016].

[27] Tilen Faganel. Ogrodje za razvoj mikrostoritev v javi in njihovo skaliranje v oblaku. Diplomska naloga, Fakulteta za računalništvo in informatiko, Univerza v Ljubljani, 2015.

[28] Martin Fowler. Event sourcing. Dosegljivo: http://martinfowler.com/eaaDev/EventSourcing.html, 2005. [Dostopano 15. 7. 2016].

[29] Martin Fowler. Polyglotpersistence. Dosegljivo: http://martinfowler.com/bliki/PolyglotPersistence.html, 2011. [Dostopano 15. 7. 2016].

[30] Martin Fowler. Circuitbreaker. Dosegljivo: http://martinfowler.com/bliki/CircuitBreaker.html, 2014. [Dostopano 5. 8. 2016].

[31] Martin Fowler. Microservices. Dosegljivo: http://martinfowler.com/articles/microservices.html, 2014. [Dostopano 5. 6. 2016].

[32] Martin Fowler. Microservice trade-offs. Dosegljivo: http://martinfowler.com/articles/microservice-trade-offs.html, 2015. [Dostopano 25. 6. 2016].

[33] Martin Fowler. Monolithfirst. Dosegljivo: http://martinfowler.com/bliki/MonolithFirst.html, 2015. [Dostopano 25. 6. 2016].

[34] M. Gradišnik and Č. Majer. Mikrostoritve in zabojniki docker. *TS 2016 - Sodobne tehnologije in storitve*, 1(2):10–20, 2016.

[35] Jim Gray. A conversation with werner vogels. Dosegljivo: https://queue.acm.org/detail.cfm?id=1142065, 2006. [Dostopano 11. 7. 2016].

[36] Robert Greiner. Cap theorem: Revisited. Dosegljivo: http://robertgreiner.com/2014/08/cap-theorem-revisited/, 2014. [Dostopano 25. 7. 2016].

[37] Arun Gupta. Microservice design patterns. Dosegljivo: http://blog.arungupta.me/microservice-design-patterns/, 2015. [Dostopano 11. 6. 2016].

[38] Arun Gupta. Microservices, monoliths, and noops. Dosegljivo: http://blog.arungupta.me/microservices-monoliths-noops/, 2015. [Dostopano 11. 6. 2016].

[39] Derrick Harris. Microservices, monoliths and laser nail guns: Etsy tech boss on finding the right focus. Dosegljivo: https://medium.com/s-c-a-l-e/microservices-monoliths-and-

laser-nail-guns-how-etsy-finds-the-right-focus-in-a-sea-of-cf718a92dc90#.xrokomge6, 2015. [Dostopano 17. 8. 2016].

[40] Joab Jackson. How synchronous rest turns microservices back into monoliths. Dosegljivo: http://thenewstack.io/synchronous-rest-turns-microservices-back-monoliths/, 2016. [Dostopano 2. 8. 2016].

[41] Daniel Jagielski. Spring boot and dropwizard in microservices development. Dosegljivo: http://www.schibsted.pl/blog/spring-boot-and-dropwizard-in-microservices-development/, 2015. [Dostopano 25. 6. 2016].

[42] Peter Kelley. Eureka! why you shouldn't use zookeeper for service discovery. Dosegljivo: https://tech.knewton.com/blog/2014/12/eureka-shouldnt-use-zookeeper-service-discovery/, 2014. [Dostopano 25. 7. 2016].

[43] Florian Motlik. Monolithic core versus full microservice architecture. Dosegljivo: https://blog.codeship.com/monolithic-core-vs-fully-microservice-architecture/, 2015. [Dostopano 15. 8. 2016].

[44] Ketan Parmar. Monolithic vs microservice architecture. Dosegljivo: https://www.linkedin.com/pulse/20141128054428-13516803-monolithic-vs-microservice-architecture, 2014. [Dostopano 11. 6. 2016].

[45] John Piela. Why netflix moved to a microservices architecture. Dosegljivo: http://www.programmableweb.com/news/why-netflix-moved-to-microservices-architecture/elsewhere-web/2016/04/02, 2016. [Dostopano 22. 7. 2016].

[46] Andrey Redko. Packing your java application as one (or fat) jar. Dosegljivo: https://www.javacodegeeks.com/2012/11/packing-your-

java-application-as-one-or-fat-jar.html, 2012. [Dostopano 7. 6. 2016].

[47] C. Richardson and F. Smith. *Microservices: From Design to Deployment*. NGINX, Inc., 2016.

[48] Chris Richardson. Pattern: Microservice chassis. Dosegljivo: http://microservices.io/patterns/microservice-chassis.html. [Dostopano 11. 7. 2016].

[49] Chris Richardson. Pattern: Service registry. Dosegljivo: http://microservices.io/patterns/service-registry.html. [Dostopano 2016].

[50] Chris Richardson. Whyeventsourcing. Dosegljivo: https://github.com/cer/event-sourcing-examples/wiki/WhyEventSourcing, 2015. [Dostopano 15. 8. 2016].

[51] Siamak Sadeghianfar. Building microservices with wildfly swarm and netflix oss on openshift. Dosegljivo: https://blog.openshift.com/building-microservices-wildfly-swarm-netflix-oss-openshift/, 2016. [Dostopano 16. 8. 2016].

[52] Bryce Merkl Sasaki. Graph databases for beginners: Acid vs. base explained. Dosegljivo: https://neo4j.com/blog/acid-vs-base-consistency-models-explained/, 2015. [Dostopano 12. 8. 2016].

[53] Ben Schmaus. Making the netflix api more resilient. Dosegljivo: http://techblog.netflix.com/2011/12/making-netflix-api-more-resilient.html, 2011. [Dostopano 29. 7. 2016].

[54] Sinclair Schuller. Why monolithic apps are often better than microservices. Dosegljivo: https://gigaom.com/2015/11/06/why-monolithic-apps-are-often-better-than-microservices/, 2015. [Dostopano 22. 7. 2016].

[55] Jan Stenberg. Microservices vs monolithic applications. Dosegljivo: https://www.infoq.com/news/2014/08/microservices-monoliths, 2014. [Dostopano 11. 6. 2016].

[56] Benjamin Wootton. Microservices - not a free lunch! Dosegljivo: http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html, 2014. [Dostopano 3. 8. 2016].