Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Martin Čebular

Razvoj mikrostoritev v programskem jeziku Go

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: prof. dr. Matjaž Branko Jurič

Ljubljana, 2018

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Opišite ključne koncepte arhitekture mikrostoritev. Osredotočite se na odkrivanje storitev, konfiguracijo, upravljanje odvisnosti in pakiranje mikrostoritev v vsebnike. Opišite postopek razvoja mikrostoritev v programskem jeziku Go in identificirajte ter primerjajte ključna ogrodja. Izdelajte knjižnico za konfiguracijo in odkrivanje mikrostoritev. Opišite postopek razvoja, delovanje in evalvirajte uporabno vrednost.

Rad bi se zahvalil mentorju prof. dr. Matjažu Branku Juriču in as. Janu Meznariču za strokovno pomoč pri izdelavi diplomske naloge. Velika zahvala gre tudi moji družini in vsem prijateljem, ki so me spremljali v času študija in me pri tem neomajno podpirali.

Kazalo

Povzetek

Abstract

1 Uvod **1**

2 Relevantni pojmi in tehnologije za razvoj mikrostoritev **3**

2.1 Mikrostoritve 3

2.2 Odkrivanje storitev 5

2.3 Konfiguracija mikrostoritev 6

2.4 Programski jezik Go 7

2.4.1 Upravljanje odvisnosti 8

2.5 Virtualizacija Docker 10

3 Razvoj mikrostoritev v programskem jeziku Go **13**

3.1 Pregled obstoječih ogrodij za razvoj mikrostoritev 13

3.1.1 Kite 14

3.1.2 Go Micro 16

3.1.3 Go kit 18

3.1.4 Sklepne misli 21

3.2 Primer razvoja mikrostoritve 21

4 Konfiguracija in odkrivanje mikrostoritev Go **31**

4.1 Ogrodje KumuluzEE 31

4.2 Konfiguracija mikrostoritev Go 32

4.2.1 Implementacija strukture in metod *Util* 34
4.2.2 Implementacija načina *Bundle* 37
4.2.3 Primer implementacije konfiguracijskega vira 39
4.3 Registracija in odkrivanje mikrostoritev Go 42
4.3.1 Implementacija registracije mikrostoritve 44
4.3.2 Implementacija odkrivanja mikrostoritev 45
4.3.3 Primer implementacije vira 46
4.4 Razširitev primera mikrostoritve 50

5 Zaključek **57**

Literatura **60**

Seznam uporabljenih kratic

REST	Representational State Transfer
RPC	Remote Procedure Call
VCS	Version Control System
SQL	Structured Query Language
API	Application Programming Interface
HTTP	Hypertext Transfer Protocol
URL	Uniform Resource Locator
JSON	JavaScript Object Notation

Povzetek

Naslov: Razvoj mikrostoritev v programskem jeziku Go

Avtor: Martin Čebular

Arhitektura mikrostoritev je pristop k razvoju programske opreme, pri katerem celotno aplikacijo razdelimo na manjše dele, kjer vsak del opravlja podmnožico nalog aplikacije. Vsak ta del je nato razvit samostojno, na koncu pa te dele – mikrostoritve – povežemo skupaj v celoto. Posamezne mikrostoritve, ki sestavljajo aplikacijo, lahko razvijamo v različnih programskih jezikih in tehnologijah, med drugim tudi v programskem jeziku Go. V okviru diplomske naloge je bilo raziskano področje razvoja mikrostoritev v programskem jeziku Go, ter razvito ogrodje za konfiguracijo in odkrivanje mikrostoritev, namenjeno uporabi pri razvoju mikrostoritev v programskem jeziku Go.

Ključne besede: mikrostoritve, Go, konfiguracija, odkrivanje storitev.

Abstract

Title: Microservice development in Go programming language

Author: Martin Čebular

Microservice architecture is an approach for developing software as a service. The main point of this architecture is splitting an application into smaller parts, where each part performs a subset of application's tasks. Each of those parts is developed independently, and at the end, these parts – microservices – are connected together to form the complete application. Each microservice can be developed using different programming languages and technologies. Programming language Go is just one of the options. Goal of this thesis is to research the current state of microservices' development with Go, and developing a Go package for configuration and discovery of microservices, developed in the Go programming language.

Keywords: microservices, Go, configuration, service discovery.

Poglavje 1

Uvod

Razvoj programske opreme kot storitev je zelo zanimiva tematika. Obstajajo različni pristopi k razvoju storitev, poleg tega pa nam je danes na voljo kopica različnih programskih jezikov in drugih tehnologij, ki jih pri tem lahko uporabimo. Ob izbiri kateregakoli od uveljavljenih programskih jezikov so nam na voljo ogrodja, ki nam pripomorejo pri razvoju storitev. Ta ogrodja nam omogočajo osredotočanje na razvoj poslovne logike. Na primer, da želimo v okviru naše storitve pripraviti REST API po protokolu HTTP. Raba primernega ogrodja bi nam omogočala, da se ne ukvarjamo z implementacijo spletnega strežnika, ki bo odgovarjal na HTTP zahtevke, ampak samo definiramo končne točke REST, ki jih želimo, in poslovno logiko za temi točkami. Poleg tega pa nam je na voljo programska oprema, katere cilj je poenostaviti postopek namestitve storitev (ang. *deployment*), da lahko razvijalci teh storitev čim bolj enostavno in hitro ponudijo aplikacijo končnim uporabnikom. Tako lahko storitve razvijemo in pripeljemo do uporabe v zelo kratkem času. Pri tem pa seveda ni pomembno samo to, da je storitev na voljo, ampak da je zmožna vzdržati tudi večje pritiske, na primer ob velikem številu sočasnih zahtevkov. Storitev, ki ni na voljo, je namreč neuporabna storitev.

V diplomski nalogi se bomo posvetili razvoju programske opreme po principu razvoja mikrostoritev. Pri tem bomo uporabili programski jezik Go, pri razvoju pa se bomo poslužili nekaterih vzorcev, ki nam bodo kasneje ob

nameščanju mikrostoritev omogočali lažje upravljanje le-teh: lotili se bomo upravljanja konfiguracij mikrostoritev (ang. *configuration management*) in odkrivanja storitev (ang. *service discovery*). Pri namestitvi razvitih mikrostoritev si bomo pomagali z orodjem Docker. Naš cilj je izdelati mikrostoritev, ki jo bomo lahko konfigurirali s pomočjo konfiguracijskih datotek in drugih virov, mikrostoritvi pa bomo s pomočjo uporabe odkrivanja storitev dali možnost, da jo odkrijejo druge mikrostoritve.

Najprej bomo v poglavju 2 razjasnili pomembnejše pojme, ki se tičejo razvoja mikrostoritev. Nato si bomo v poglavju 3 pogledali tri različna ogrodja za razvoj mikrostoritev v programskem jeziku Go, v poglavju 3.2 pa bomo razvili enostavno mikrostoritev (v tem trenutku še brez konfiguracij in odkrivanja). Sledi kratka predstavitev javanskega ogrodja KumuluzEE (poglavje 4) ter predstavitev Gojevskih implementacij KumuluzEE konfiguracije in odkrivanja storitev (knjižnici `kumuluzee-go-config` in `kumuluzee-go-discovery` predstavimo v poglavjih 4.2 in 4.3). Kot zadnji del pa bomo našo mikrostoritev iz poglavja 3.2 razširili s predstavljenima knjižnicama (poglavje 4.4).

Poglavje 2

Relevantni pojmi in tehnologije za razvoj mikrostoritev

2.1 Mikrostoritve

Preden se lotimo česarkoli drugega, najprej malo bolje opredelimo osrednjo tematiko diplomske naloge. Arhitektura mikrostoritev je poseben pristop k razvoju programske opreme [14, 33]. Glavno načelo mikrostoritev je, da neko aplikacijo razvijemo kot skupino manjših storitev, pri čemer vsaka storitev opravlja nek del poslovne logike aplikacije in deluje kot samostojen proces [12, 13]. Pri tem je pomembno, da storitve niso (oziroma so čim manj) odvisne med seboj v smislu opravljanja svoje naloge – vsaka storitev lahko svojo nalogo opravlja samostojno, kar naredi storitve enostavne, celostne in neodvisne. Več razvitih mikrostoritev pa se povezuje v celoto, ki predstavlja končno, uporabniku vidno storitev oziroma aplikacijo.

Pred razvojem aplikacije kot množice mikrostoritev je potrebno celotno aplikacijo razdeliti na nekakšne logične dele, sklope funkcionalnosti. Vsak ta logični del naj bo nekakšna zaključena celota, saj želimo posamezne mikrostoritve razvijati neodvisno. Odločitev, kako velika je posamezna mikrostoritev v smislu funkcionalnosti, ki jih opravlja, je prepuščena razvijalcem aplikacije [25, 11]. Tu je potrebno najti pravo mero, da razvita mikrostoritev ne zaide

v eno od dveh skrajnosti:

- mikrostoritev postane monolitna aplikacija, ker opravlja več nalog, ki bi se lahko razdelile,
- mikrostoritve postanejo preveč "nadrobljene" v svojih funkcionalnostih [24]

Pri povezovanju mikrostoritev je pomembno, da je komunikacija med mikrostoritvami atomarna in brez stanja (ang. *stateless*) [32]. To pomeni, da se vsako prejeto sporočilo obravnava povsem ločeno od predhodnih ali prihodnjih sporočil. Sama mikrostoritev pa ob prejemu sporočila ne shrani v pomnilnik nobenih podatkov, ki bi na kakršen koli način spremenili obravnavo naslednjega sporočila (ne hrani stanja). Torej, če mikrostoritvi dvakrat pošljemo enako sporočilo (ali zaporedje sporočil), bo končni rezultat enak v obeh primerih.

Za komunikacijski protokol med mikrostoritvami je pomembno, da je čim bolj preprost [13]. Edina naloga komunikacijskega protokola je prenos sporočil od mikrostoritve do mikrostoritve. V protokolu si ne želimo, da se sporočilo na kakršen koli način obravnava ali spreminja, saj bi s tem implicitno implementirali del poslovne logike v samem protokolu, kar naenkrat pa bi komunikacija med mikrostoritvami postala odvisna od točno določenega načina oziroma poteka komunikacije. Preprost komunikacijski protokol omogoča, da je vsa poslovna logika implementirana v dejanskih mikrostoritvah in se nek del poslovne logike ne izvaja med prenosom sporočil.

Posamezne mikrostoritve se razvijajo samostojno in neodvisno od ostalih, ki skupaj sestavljajo enotno aplikacijo [12, 30, 32]. To pomeni, da ima vsaka mikrostoritev svoj razvojni cikel in svojo razvojno ekipo. Tehnologije, v katerih so razvite posamezne mikrostoritve, se lahko popolnoma razlikujejo. Eno od mikrostoritev lahko razvijemo v JavaScriptu in okolju Node, drugo pa na primer v Javi. Tu seveda nismo omejeni samo na prosto izbiro programskega jezika, ampak tudi vseh ostalih komponent posamezne mikrostoritve. Na primer, za shranjevanje podatkov lahko v enem primeru uporabimo MySQL,

v drugem pa eno od implementacij podatkovne baze NoSQL. Pomembno je le, da se pri razvoju mikrostoritev upoštevajo načela tega pristopa razvoja – vsaka mikrostoritev je zaključena celota, komunikacija med storitvami pa enostavna in standardizirana.

2.2 Odkrivanje storitev

Pri komunikaciji med mikrostoritvami imamo še eno težavo, ki jih pri monolitnih aplikacijah ni. Recimo, da imamo v sklopu aplikacije nameščenih več mikrostoritev, ki se nahajajo na različnih IP naslovih, ali pa na različnih vratih istega naslova. Ko ena mikrostoritev želi komunicirati z neko drugo mikrostoritvijo, mora vedeti njen naslov. Če sami poskrbimo za to, na katerih naslovih se nahaja katera mikrostoritev, seveda ni težko mikrostoritvam "povedati", kje se kaj nahaja. Kaj pa, če želimo postaviti več instanc iste mikrostoritve? Kaj pa, če ena od teh instanc postane nedosegljiva? Tudi ti dve težavi lahko rešimo – z implementacijo primerne logike v mikrostoritvi, ki bi omogočala dodajanje naslovov novih instanc, ter logike, ki bi omogočala izbiranje primerne instance, če je instanc več. To rešitev bi morali sedaj implementirati v vse mikrostoritve, pri tem pa sami vzdrževati seznam naslovov, kjer se mikrostoritve nahajajo. Pri tem je potrebno poudariti, da se ta seznam lahko pogosto spreminja, saj oblačna okolja dinamično dodajajo in ugašajo vsebnike ali jih prestavljajo med strežniki.

Seveda tu obstaja bolj preprosta rešitev: odkrivanje storitev [27]. Glavna ideja odkrivanja storitev je v tem, da imamo nek centralni register, ki hrani seznam aktivnih instanc vseh naših mikrostoritev. Vsaka instanca mikrostoritve se sama "prijavi"v ta register, s čemer mehanizem odkrivanja storitev vsem ostalim uporabnikom registra sporoči, na katerem naslovu se nahaja. Ko pa hoče mikrostoritev komunicirati s kako drugo mikrostoritvijo, pa register "povpraša", kje se določena mikrostoritev nahaja, in v odgovor dobi naslov ene od instanc iskane mikrostoritve.

Omenjeno prijavo in povpraševanje lahko definiramo bolj strokovno: re-

gistracija storitve in odkrivanje storitev. Mikrostoritev mora vedeti le naslov registra, kamor naj se poveže za registracijo in odkrivanje ostalih mikrostoritev. Ko mikrostoritev namestimo, se ta poveže na register in se registrira. Register instanc mikrostoritev je potrebno vzdrževati, da v register niso vpisani naslovi, na katerih instance ni več, ali pa ima le-ta težave, zaradi katere ne more delovati pravilno. Obstaja več načinov za vzdrževanje registra [26], na primer:

- **TTL** (*time to live*): mikrostoritev je vpisana v register do preteka TTL. Mikrostoritev se registru javlja vsakih n sekund, če pa se po času TTL ne javi registru, se instanca odstrani iz registra.
- **Health check**: mikrostoritev izpostavi API, ki ob klicu sporoči svoje stanje. Če ima mikrostoritev REST API, se navadno izpostavi končna točka `GET /health`, ki vrne primeren status glede na stanje mikrostoritve (npr. status 200 za ok, status 500 za težave). Register nato vsakih n sekund pošlje zahtevek na to končno točko in si zapiše vrnjeno stanje. Če je stanje mikrostoritve m sekund kritično, se instanca odstrani iz registra.

Ko mikrostoritev želi odkriti kakšno drugo storitev, se ravno tako poveže na register, kjer poizve po določeni mikrostoritvi (tu navadno navedemo ime in verzijo mikrostoritve, ki jo želimo).

Kar smo omenili tu, je le del teorije, ki zajema odkrivanje storitev, a dovolj, da bomo v nadaljevanju razumeli pojem odkrivanja storitev.

2.3 Konfiguracija mikrostoritev

Pri konfiguraciji mikrostoritev je zaželeno, da je dejanska konfiguracija ločena od izvorne kode mikrostoritve. To pomeni, da konfiguracijski podatki niso vpisani neposredno v izvorno kodo (t.i. *hardcoded*), ampak jih ob primernem trenutku (na primer ob zagonu aplikacije, ob potrebi po podatku...) preberemo iz nekega konfiguracijskega vira. Mikrostoritev ima lahko dostop do več

različnih konfiguracijskih virov, na primer konfiguracijska datoteka, okoljske spremenljivke (ang. *environmental variables*), shramba tipa ključ-vrednost (ang. *key-value store*)... V primeru, da ima mikrostoritev na voljo več virov, mora obstajati nek red, da mikrostoritev ve, katero konfiguracijo upoštevati. To lahko rešimo z dodelitvijo prioritet posameznim konfiguracijskim virom, mikrostoritev pa bo upoštevala konfiguracijo v viru z najvišjo prioriteto, v katerem konfiguracija obstaja.

Primer: na voljo imamo dva vira, konfiguracijo iz datoteke in shramba tipa ključ-vrednost. Pri tem ima shramba tipa ključ-vrednost višjo prioriteto. Če nek konfiguracijski ključ obstaja samo v datoteki ali samo v shrambi tipa ključ-vrednost, se bo ta upošteval. Če pa obstaja nek ključ v obeh virih, pa se bo upoštevala vrednost v viru z višjo prioriteto, torej vrednost v shrambi tipa ključ-vrednost.

Tak način konfiguracije prinaša dve pomembni prednosti. Prvič, imamo hiter pregled nad trenutno konfiguracijo mikrostoritve – trenutna konfiguracija je razvidna iz vira, s čimer ni potrebe po brskanju po izvorni kodi. In drugič, upravljanje tekočih instanc mikrostoritev je preprostejše – konfiguracijo lahko spreminjamo brez postopka ponovnega prevajanja programa in namestitve mikrostoritve. Še eno pomembno prednost pa prinaša uporaba shrambe tipa ključ-vrednost. Uporaba tovrstnega vira konfiguracije nam daje možnost poenotene konfiguracije za množico mikrostoritev, ki se za konfiguracijo povezujejo na isti shrambo. Poleg tega pa nam je na voljo še nadzorovanje sprememb (ang. *watches*) v shrambi, kar omogoča spremembo konfiguracije mikrostoritve brez njenega ponovnega zagona.

2.4 Programski jezik Go

Programski jezik Go je prvo uradno izdajo doživel leta 2009, razvit je bil v podjetju Google. Gojev prevajalnik in ostala orodja za delo s programskim jezikom so brezplačna od odprtokodna [29].

Izvorna koda se s prevajalnikom prevaja v izvršljive datoteke (ang. *binary*

executeables), ki so lahko:

- statično povezane (ang. *statically linked*), kar pomeni, da izvršljiva datoteka sama vsebuje vse knjižnice, potrebne za izvajanje prevedenega programa.
- dinamično povezane (ang. *dynamically linked*), kar pomeni, da je izvršljiva datoteka lahko odvisna od zunanjih knjižnic, ki morajo za izvajanje biti na voljo v operacijskem sistemu, v katerem se program izvaja.

V diplomski nalogi bomo pri prevajanju izvorne kode Go predvideli prvi način prevajanja, s statičnim povezovanjem. Na ta način preveden program je lahko po velikosti nekoliko večji (odvisno od tega, koliko zunanjih knjižnic potrebuje), a je zaradi tega, ker sam program vsebuje vse potrebno za izvajanje, preprostejši za namestitev.

2.4.1 Upravljanje odvisnosti

Opazna lastnost jezika Go je tudi delo z odvisnostmi (ang. *dependencies*). Go nima nekega centralnega registra, kjer bi se vzdrževal seznam objavljenih knjižnic (kot npr. Javanski Maven ali Nodeov npm), temveč Gojevske knjižnice prenašamo neposredno z repozitorijev VCS (*version control system*). Na primer, če želimo uporabiti knjižnico, ki je objavljena na Githubu, v Go kodi zapišemo `import "github.com/user/repository"` ter nato poženemo ukaz `go get`, ki bo za nas prenesel kodo z Githuba, ki se nahaja v navedenem repozitoriju. Ker je seveda možno, da se koda nahaja še kje drugje kot na Githubu, lahko v `import` stavku navedemo tudi neposredno povezavo do repozitorija, ki se mora v takem primeru končati s končnico *.git* (ali za kakšen drug VCS, na primer za Subversion končnica *.svn*). Primer takega dodajanja knjižnice je `import "example.com/repos/user/repository.git"`. Prenesene knjižnice se ne prenesejo v mapo z našim programom, temveč se prenesejo v mapo, definirano s spremenljivko okolja `GOPATH`. Dobra lastnost

tega je, da knjižnice prenesemo na naš računalnik samo enkrat, ne glede na to, koliko različnih programov uporablja določeno knjižnico. Slaba lastnost pa je, da se nam lahko zalomi, če želimo v različnih programih uporabiti različne verzije knjižnic. V nadaljevanju si bomo pogledali rešitev tega problema.

Opazimo lahko, da orodje `go get` nima nobene možnosti po nadzoru nad verzijami knjižnic, temveč z repozitorija vedno prenese datoteke, ki se po znački (ang. *tag*) ujemajo z našo verzijo jezika Go (na primer, če uporabljamo jezik Go verzijo 1.9 in v repozitoriju obstaja značka "go9", se bodo prenesle datoteke ob tej znački). Če repozitorij take značke nima, se datoteke prenesejo s privzete veje (*default branch*, ki se v večini primerov imenuje master). To lahko predstavlja težavo, sploh če v repozitoriju knjižnice na *master* veji pride do sprememb, ki niso kompatibilne za nazaj. V tem primeru se naš program po morebitni posodobitvi prenesenih odvisnosti sploh ne more več prevesti.

Obstajajo različni načini za reševanje tega problema. Tu si bomo pogledali uporabo *vendoringa* [9] z orodjem `dep` [2], ki je namenjeno upravljanju verzij knjižnic Go. Ideja vendoringa je zelo preprosta: poleg prej omenjene centralne lokacije vseh prenesenih knjižnic (`GOPATH`), si v mapi z našo izvorno kodo ustvarimo mapo `vendor`, ki vsebuje datoteke knjižnic na točno določenih verzijah. Struktura vendor mape je identična strukturi mape, ki se nahaja na lokaciji `$GOPATH/src`. Pri prevajanju programa se najprej preveri, ali obstaja knjižnica v mapi vendor, in če obstaja, uporabi to. Sicer pa uporabi knjižnico, ki se nahaja na centralni lokaciji. Mapo vendor lahko s točno določenimi verzijami knjižnic napolnimo ročno, lahko pa si pomagamo s prej omenjenim orodjem `dep`.

Orodje `dep` nam omogoča upravljanje verzij v podobnem stilu, kot nam to omogoča na primer `npm`. V mapi z našo izvorno kodo Go poženemo ukaz `dep init`, ki med drugim ustvari datoteko `Gopkg.toml`. V tej datoteki vzdržujemo seznam knjižnic in njihovih verzij, pri čemer lahko kot verzijo navedemo vejo, značko, ali točen *commit*, ki ga želimo uporabiti. Podprto je tudi semantično verzioniranje. Knjižnice lahko v seznam dodamo ročno, ali

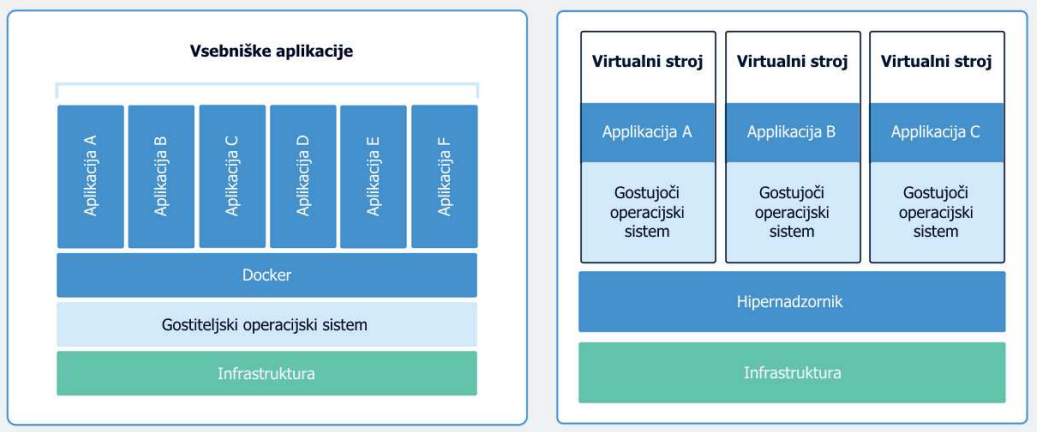
pa uporabimo ukaz `dep ensure -add path/to/repo`. Vsakič, ko poženemo ukaz `dep ensure`, se izbrane verzije knjižnic prenesejo v mapo vendor. Te pa so, ker se upošteva vendoring, na voljo Go prevajalniku.

Še opomba o terminologiji: omenili smo besedi odvisnost in knjižnica, s katerima označujemo neko zunanjo kodo Go, ki jo vključimo v naš program z uporabo stavka *import*. Po Gojevsko takim knjižnicam rečemo *package*, primeren slovenski prevod pa bi bil *paket*. V diplomski nalogi se bomo držali kar besede knjižnica, ker gre za bolj ustaljen izraz, ko se pogovarjamo o vključevanju zunanje kode v program.

2.5 Virtualizacija Docker

Docker je odprtokodna programska oprema, ki omogoča delo z virtualizacijo na nivoju operacijskega sistema (ang. *containerization*) [1, 3]. Tak način virtualizacije pomeni, da jedro operacijskega sistema omogoča obstoj več ločenih uporabniških prostorov (ang. *user-space*) [15]. Ti uporabniški prostori si lahko delijo vire strojne opeme (procesor, prostor na disku, druge povezane naprave...), a programi, ki tečejo v posameznem prostoru, imajo dostop le do vsebine prostora v katerem tečejo in virov, ki so dodeljeni prostoru. Za primerjavo, v dejanski virtualizaciji nam v gostiteljskem (*host*) operacijskem sistemu teče še gostujoči (*guest*) operacijski sistem, v katerem nato dela izbrani program. V zgoraj opisanem primeru pa preskočimo korak gostujočega operacijskega sistema in program deluje neposredno v gostiteljskem operacijskem sistemu, a so posamezni prostori, v katerih tečejo programi, strogo ločeni. Takemu uporabniškemu prostoru rečemo vsebnik (ang. *container*) [31]. Primerjava med virtualizacijo operacijskih sistemov in virtualizacijo na nivoju operacijskega sistema je prikazana na sliki 2.1.

Vsebino in delovanje vsebnika je seveda treba definirati. Osnova za to so slike (ang. *images*). Sliko si lahko predstavljamo kot paket, ki vsebuje vse potrebne podatke za njeno delovanje (program, potrebne knjižnice, ipd.). Ko sliko "poženemo", se iz slike ustvari nov vsebnik – nastane nova instanca



Slika 2.1: Razlika med virtualizacijo operacijskih sistemov in virtualizacijo na nivoju operacijskega sistema (ang. *containerization*). Vir: [31]

slike. Iz ene slike seveda lahko ustvarimo več vsebnikov – instanc, ki nato tečejo v ločenih uporabniških prostorih. Docker je orodje, ki nam omogoča upravljanje slik (prenos slik s spleta, kreiranje novih slik, itd.), ter upravljanje vsebnikov (instanciranje, poganjanje, ustavljanje, itd.). Če navežemo Docker na temo mikrostoritev lahko opazimo, da nam tehnologija, ki jo nudi Docker (slike in vsebniki), služi kot orodje za lažje nameščanje in upravljanje z mikrostoritvami, saj nam omogoča hitro in enostavno nameščanje in (vsaj začetno) konfiguracijo nove instance mikrostoritve.

Poglavje 3

Razvoj mikrostoritev v programskem jeziku Go

Ideja o mikrostoritvah v programskem jeziku Go seveda ni nova. Kot smo spoznali, Go lahko prevajamo v statično-povezane izvršljive datoteke, kar nam omogoča poganjanje programov Go praktično neposredno v operacijskem sistemu. Poleg tega pa prevedeni program Go ne potrebuje drugih odvisnosti, kot na primer Java potrebuje JVM ali pa Python potrebuje interpreter. Zaradi te lastnosti je Go primeren kandidat za razvoj majhnih storitev, ki opravljajo specifično nalogo – mikrostoritev.

3.1 Pregled obstoječih ogrodij za razvoj mikrostoritev

V tem poglavju si bomo pogledali obstoječa ogrodja za razvoj mikrostoritev – njihove glavne lastnosti, ter minimalni primer izdelane mikrostoritve, ki se registrira v register in nato odkrije iz neke druge mikrostoritve.

Po izvedenem brskanju lahko izpostavimo naslednja tri ogrodja kot najbolj znana: Kite, Go Micro in Go kit.

3.1.1 Kite

Kite (`koding/kite`) [7] bi lahko izmed izbranih treh ogrodij izpostavili za najpreprostejše. Kite omogoča komunikacijo med mikrostoritvami z uporabo oddaljenih klicev (RPC) ter odkrivanje storitev s pomočjo registra *Kontrol*, ki je tudi sam implementiran kot Kite mikrostoritev in temelji na uporabi shrambe tipa ključ-vrednost *etcd*. Komunikacija med mikrostoritvami poteka s pošiljanjem *dnode* sporočil [8] prek *WebSocket*a ali XHR. Poudarek knjižnice je na komunikaciji med mikrostoritvami in odkrivanjem mikrostoritev med seboj. Knjižnica ne vsebuje aktivne funkcionalnosti za konfiguracijo mikrostoritev (na primer konfiguracijo s pomočjo zunanjih datotek), omogoča pa omejeno konfiguracijo mikrostoritve s pomočjo spremenljivk okolja (branje ključev, ki so pomembni za registracijo storitve v Kontrol, kot so ime storitve in naslov registra Kontrol).

```
k := kite.New("example", "1.0.0")
k.Config = config.MustGet()

k.Config.Port = 6001
k.Config.IP = "127.0.0.1"
k.Config.Environment = "dev"

k.HandleFunc("square", func(r *kite.Request) (interface{},
 ↪  error) {
    a := r.Args.One().MustFloat64()
    return a * a, nil
})

_, err := k.Register(&url.URL{
    Scheme: "http",
    Host:   "127.0.0.1:6000",
})
if err != nil {
    panic(err)
}

k.Run()
```

Izsek 3.1: Primer mikrostoritve, razvite s pomočjo knjižnice Kite. (Razvito po: [16]).

V izseku 3.1 si poglejmo preprost primer mikrostoritve, razvite s pomočjo knjižnice.

Najprej naredimo novo instanco strukture *kite*, jo primerno konfiguriramo, pripravimo končno točko v obliki krmilne funkcije (*handler function*), jo registriramo v Kontrol register, ter na koncu zaženemo. V tem preprostem primeru smo pripravili končno točko *square*, ki sprejme število tipa `float64` in vrne kvadrat števila. Registrirano storitev v Kontrol lahko zdaj odkrijemo iz neke druge mikrostoritve. Primer kode, ki bi odkril našo mikrostoritev je v izseku 3.2.

```
k := kite.New("other", "1.0.0")

kites, err := k.GetKites(&protocol.KontrolQuery{
    Environment: "dev",
    Name:        "example",
})
if err != nil {
    panic(err)
}

client := kites[0]

err = client.Dial()
if err != nil {
    panic(err)
}

response, err := client.Tell("square", 4)
if err != nil {
    panic(err)
}

// response = 16
```

Izsek 3.2: Primer druge mikrostoritve, ki odkrije prvo mikrostoritev.

Tudi tu najprej naredimo instanco strukture *kite*, nato pa kličemo metodo `GetKites()`, s katero v Kontrol registru poizvemo po želeni mikrostoritvi. Metoda nam vrne seznam odkritih mikrostoritev. V zgornjem primeru izberemo kar prvo mikrostoritev v seznamu, ter najprej preverimo odzivnost mikrostoritve s klicem metode `Dial()`. Če je bil klic uspešen, lahko odkriti

mikrostoritvi pošljemo sporočilo s klicem metode `Tell()`. V zgornjem primeru kličemo funkcijo *square* (ki smo jo pripravili v prvem primeru) skupaj s parametrom 4 in v odgovor dobimo kvadrat števila 4, 16.

3.1.2 Go Micro

Go Micro (`micro/go-micro`) [6] je ogrodje za razvoj mikrostoritev, ki je del ekosistema Gojevskih knjižnic Micro [22]. Glavni značilnosti knjižnice sta implementacija odkrivanja storitev in komunikacija med mikrostoritvami z uporabo gRPC, pri čemer se za definicijo gRPC funkcij uporablja Googlov *protobuf* [23], ki za nas generira kodo Go iz definicije *proto*. Privzeta izbira implementacije odkrivanja storitev je Consul, na voljo pa so številne druge implementacije v obliki vtičnikov.

Ker Go Micro uporablja definicijo *proto*, najprej definiramo datoteko *square.proto* za našo mikrostoritev (glej izsek 3.3). V njej definiramo storitev (`service`) ter funkcijo, ki jo storitev izpostavlja. Poleg pa definiramo tudi tipa in vsebini zahteve (`request`) in odgovora (`response`).

```
1  syntax = "proto3";
2
3  service Example {
4      rpc Square(SquareRequest) returns (SquareResponse) {}
5  }
6
7  message SquareRequest {
8      float number = 1;
9  }
10
11 message SquareResponse {
12     float squared = 1;
13 }
```

Izsek 3.3: Vsebina datoteke *square.proto*.

```
1  // import stavki
2  import "context"
3  import "fmt"
4  import proto "example/proto"
```

```
5  import micro "github.com/micro/go-micro"
6  import "github.com/micro/go-micro/registry"
7  import "github.com/micro/go-micro/registry/consul"
8
9  type Example struct{}
10
11 func (g *Example) Square(ctx context.Context, req
    ↪  *proto.SquareRequest, rsp *proto.SquareResponse) error {
12     rsp.Squared = req.Number * req.Number
13     return nil
14 }
15
16 func main() {
17     service := micro.NewService(
18         micro.Name("example"),
19         micro.Registry( consul.NewRegistry(
            ↪  registry.Addrs("192.168.2.220") ) ),
20     )
21     service.Init()
22     proto.RegisterExampleHandler(service.Server(),
        ↪  new(Example))
23
24     if err := service.Run(); err != nil {
25         log.Fatal(err)
26     }
27 }
```

Izsek 3.4: Primer razvite mikrostoritve z uporabo generirane kode iz proto definicije.

S klicem ukaza (predvidevamo, da je v sistemu že nameščen `protoc`):

```
protoc --micro_out=. --go_out=. square.proto
```

generiramo datoteke Go, ki vsebujejo vse potrebno za uporabo definirane storitve in sporočil v naši izvorni kodi. V izseku 3.4 pripravimo mikrostoritev, ki bo implementirala storitev, definirano v datoteki *proto*. Tu je pomembno izpostaviti stavek *import* v vrstici 4, s katerim v naš program uvozimo generirano kodo (v tem primeru se generirane datoteke nahajajo v mapi `$GOPATH/src/example/proto`). Nato pripravimo tip `Example`, ki bo deloval kot implementacija naše storitve. Implementiramo mu metodo `Square()` tako, kot je bila definirana v proto datoteki. V funkciji `main()`

instanciramo novo storitev *Micro*, pri čemer poskrbimo tudi za registracijo mikrostoritve v register Consul, ki deluje na določenem naslovu. V storitev *Micro* registriramo še implementacijo proto storitve (`Example`), ter mikrostoritev poženemo.

V izseku 3.5 registrirano storitev odkrijemo, pri čemer uporabimo enako proto definicijo oz. generirane datoteke iz definicije. Tako kot prej, tudi tukaj pri instanciranju storitve *Micro* navedemo naslov registra Consul. Tu je ključen klic funkcije `proto.NewExampleService()`, ki v registru najde mikrostoritev s podanim imenom. Funckija nam neposredno vrne storitev, generirano po definiciji proto, zato lahko kar kličemo metodo `Square()` in prejmemo odgovor (ali napako) kot rezultat klica funkcije.

```
1  service := micro.NewService(
2      micro.Registry( consul.NewRegistry(
       ↪  registry.Addrs("192.168.2.220") ) ),
3  )
4  service.Init()
5
6  cl := proto.NewExampleService("example", service.Client())
7
8  rsp, err := cl.Square(context.Background(),
   ↪  &proto.SquareRequest{Number: 4})
9  if err != nil {
10     panic(err)
11 }
12
13 // rsp.Squared = 16
```

Izsek 3.5: Primer odkrivanja prej registrirane mikrostoritve.

3.1.3 Go kit

Go kit (`go-kit/kit`) [5] podobno kot Micro sestavlja več knjižnic. Go Kit za komunikacijo med mikrostoritvami uporablja protokol RPC, na voljo je uporaba gRPC, JSON-RPC ali `net/rpc`, ki je del standardne knjižnice Go. Kot bomo videli, sta programski kodi pri uporabi Go Micro in Go Kit do neke mere podobni. Izpostavljen koncept v Go Kitu je uporaba vmesnega

programja (ang. *middleware*) [10], s katerim se strogo ločuje posamezne aspekte mikrostoritve (t.i. *seperation of concerns*), kot so na primer definicije končnih točk, odkrivanje storitev, beleženje (ang. *logging*)...

V izseku 3.6 si poglejmo primer mikrostoritve, izdelane z Go Kit. Takoj lahko opazimo, da je tu kode precej več kot pri uporabi knjižnice Go Micro. To izhaja iz dejstva, da smo pri Go Micro uporabili definicijo proto, iz katere smo nato generirali določene dele kode (definicijo storitve, sporočil) in smo v našo mikrostoritev generirano kodo preprosto vključili z stavkom `import`. V tem primeru pa moramo vso šablonsko kodo (ang. *boilerplate code*), ki definira storitev in sporočila za protokol RPC, spisati sami.

```
1  package main
2
3  import (
4          "context"
5          "encoding/json"
6          "log"
7          "net/http"
8          "github.com/go-kit/kit/endpoint"
9          httptransport "github.com/go-kit/kit/transport/http"
10 )
11
12 // ExampleService definition
13 type ExampleService interface {
14         Square(context.Context, float64) float64
15 }
16
17 // service implementation
18 type exampleService struct{}
19
20 func (exampleService) Square(_ context.Context, number
    ↪  float64) float64 {
21         return number * number
22 }
23
24 // message definitions
25 type squareRequest struct {
26         Number float64 `json:"number"`
27 }
28
29 type squareResponse struct {
30         Squared float64 `json:"squared"`
31 }
```

```
32
33  // endpoint
34  func makeSquareEndpoint(svc ExampleService) endpoint.Endpoint
     ↪  {
35          return func(ctx context.Context, request interface{})
             ↪  (interface{}, error) {
36                  req := request.(squareRequest)
37                  sqrd := svc.Square(ctx, req.Number)
38                  return squareResponse{sqrd}, nil
39          }
40  }
41
42  func main() {
43          svc := exampleService{}
44
45          squareHandler := httptransport.NewServer(
46                  makeSquareEndpoint(svc),
47                  decodeSquareRequest,
48                  encodeResponse,
49          )
50
51          http.Handle("/square", squareHandler)
52          log.Println("Starting server on 8080...")
53          log.Fatal(http.ListenAndServe(":8080", nil))
54  }
55
56  func decodeSquareRequest(_ context.Context, r *http.Request)
     ↪  (interface{}, error) {
57          var request squareRequest
58          if err := json.NewDecoder(r.Body).Decode(&request);
             ↪  err != nil {
59                  return nil, err
60          }
61          return request, nil
62  }
63
64  func encodeResponse(_ context.Context, w http.ResponseWriter,
     ↪  response interface{}) error {
65          return json.NewEncoder(w).Encode(response)
66  }
```

Izsek 3.6: Primer razvite mikrostoritve z uporabo knjižnice Go Kit.

3.1.4 Sklepne misli

Pogledali smo si tri različna ogrodja za izdelavo mikrostoritev v jeziku Go in z vsakim izdelali majhno mikrostoritev. To še zdaleč ne pomeni, da so to edine tri opcije, ko potrebujemo ogrodje za razvoj mikrostoritev, so pa tri bolj znane, ki bi jih našli po kratkem brskanju na temo mikrostoritev Go. Če postavimo vsa tri pregledana ogrodja druga ob drugo, lahko opazimo, da:

- je *Kite* osredotočen na odkrivanje in komunikacijo med storitvami, omejen pa je na uporabo etcd (seveda nam odprtokodnost projekta omogoča, da bi pripravili implementacijo katerega drugega registra za odkrivanje storitev),
- *Go Micro* zahteva uporabo `protoc` (kar seveda ni narobe ali slabo, je pa dodatna zahtevana odvisnost, ki je ostali dve ogrodji nimata),
- *Go Kit* v začetku potrebuje nekoliko več šablonske kode (definicije storitev, sporočil), njegova glavna lastnost pa je uporaba vmesnega programja.

Skratka, vsako od treh preizkušenih ogrodij ima svoj bolj ali manj specifičen namen in cilj (ki se generalizira na razvoj mikrostoritev). Temu primerne pa so tudi pozitivne in (morebitne) negativne lastnosti, kar se tiče načina razvoja mikrostoritev Go z določenim ogrodjem.

3.2 Primer razvoja mikrostoritve

V tem poglavju prej pregledanih knjižnic ne bomo uporabili, ker bomo pripravili minimalen primer izdelane mikrostoritve – omejili se bomo le na uporabo standardnih knjižnic Go, ter knjižnice za lažjo inicializacijo REST APIja `gorilla/mux`, katera sama uporablja le standardne knjižnice.

Pred samim začetkom razvoja je seveda pomembno, da mikrostoritev načrtujemo. V ta namen definirajmo, kaj želimo, da naša razvita mikrostoritev obsega:

- izpostavljen API za komunikacijo v obliki REST prek protokola HTTP,
- dostop do vira podatkov, ki bo za naš primer kar tabela v spominu programa,
- poslovna logika, ki se kliče prek APIja, omogoča delo s podatki v viru: poizvedovanje, dodajanje, popravljanje in brisanje podatkov.

Izmislimo si še nek primer uporabe naše mikrostoritve. Recimo, da želimo v sklopu razvoja spletne trgovine izdelati mikrostoritev, ki bo upravljala s podatki o uporabnikih spletne trgovine – kupcih. V ta namen bomo v izbranem viru podatkov hranili naslednje podatke: ID kupca, ime, priimek, naslov in poštno številko.

Metoda	**Pot**	**Akcija**
GET	/kupci	vrne seznam vseh kupcev
GET	/kupci/{id}	vrne podatke po IDju izbranega kupca
POST	/kupci	zapiše novega kupca v vir podatkov
PUT	/kupci/{id}	posodobi podatke z IDjem izbranega kupca
DELETE	/kupci/{id}	izbriši podatke z IDjem izbranega kupca

Tabela 3.1: Končne točke REST APIja za primer mikrostoritve

V tabeli 3.1 zapišemo končne točke (ang. *endpoints*) naše mikrostoritve. Končne točke smo načrtovali po principu *CRUD: create, read, update, delete*. To pomeni, da vsaka končna točka izraža eno izmed štirih navedenih akcij, končne točke pa so namenjene upravljanju s točno določeno entiteto [21]. Pot do končne točke izraža entiteto, s katero upravljamo (t.j. kupci), metodo HTTP pa izraža akcijo, ki naj se izvede ob klicu. Na primer, s klicem `GET /kupci/6` poizvemo po podatkih o kupcu z IDjem 6, s klicem `PUT /kupci/6` pa podatke o kupcu z IDjem 6 posodobimo. Metodi `POST` in `PUT`, ki dodajata in popravljata podatke, pa pri klicu pričakujeta, da telo klica (ang. *request body*) vsebuje potrebne podatke za vnos, ki so navedeni v neki določeni obliki (na primer JSON ali XML).

Načrtovan API je zelo enostaven, zato lahko opazimo nekaj pomanjkljivosti, ki jih v nekem produkcijskem scenariju API ne bi imel:

- dobra praksa je, da se API primerno verzionira [21]. Ponavadi to storimo s predpono na poti do končnih točk, na primer `/v1/kupci`.
- `GET /kupci` metoda vedno vrne seznam vseh kupcev. V primeru, da bi imeli podatkov veliko, bi bila to slaba ideja – problem bi lahko rešili z implementacijo omejitev in/ali ostranjevanja (ang. *pagination*). Druga rešitev bi bila tudi omogočiti poizvedovanje po podatkih, opisano v naslednji točki.
- `GET /kupci/{id}` metoda omogoča le poizvedbo po ID kupca. Primerno bi bilo, da omogočimo poizvedovanje tudi po drugih podatkih. To bi lahko naredili na primer z uporabo GET parameterov: `/kupci?priimek={q}`, kjer je `{q}` iskan priimek.

Po načrtovanju REST APIja se lahko lotimo dela. Ustvarimo datoteko `service.go`, ki bo vsebovala celotno logiko naše mikrostoritve.

```
1  package main
2
3  import (
4      // importi iz standardne knjižnice
5      // ...
6
7      "github.com/gorilla/mux"
8  )
9
10 type Kupec struct {
11     ID      int
12     Ime     string
13     Priimek string
14     Naslov  string
15     Posta   int `json:"postna-stevilka"`
16 }
17
18 var dataSource []Kupec
19
20 func main() {
21     initDataSource()
```

```
22      r := initHTTPRouter()
23      log.Fatal(http.ListenAndServe(":9000", r))
24  }
```

Izsek 3.7: Osnovna struktura mikrostoritve v datoteki *service.go*.

Osnovna struktura programa je v izseku 3.7. V vrsticah 10-16 definiramo strukturo (`struct`), ki jo bomo uporabili za delo s podatki o kupcih. Ta struktura definira polja, ki jih vsebuje entiteta kupec, uporablja pa se tudi pri generiranju izhodnega JSONa in razčlenjevanju (ang. *parse*) vhodnega JSONa.

V vrstici 18 definiramo naš vir podatkov – kot smo omenili prej, bomo za potrebe primera uporabili kar polje s podatkovnim tipom naše strukture. V glavni funkciji (`main()`) nato kličemo funkcijo, ki inicializira naš vir podatkov (glej izsek 3.8).

```
1   func initDataSource() {
2       dataSource = make([]Kupec, 2)
3       dataSource[0] = Kupec{
4           ID:      1,
5           Ime:     "Janez",
6           Priimek: "Novak",
7           Naslov:  "Primerna cesta 6",
8           Posta:   "1000",
9       }
10      // ...
11  }
```

Izsek 3.8: Funkcija, ki inicializira naš vir podatkov.

S pomočjo omenjene knjižnice `gorilla/mux` pripravimo HTTP usmerjevalnik (ang. *router*). V izseku 3.9 definiramo poti, ki smo jih zastavili pri načrtovanju APIja v tabeli 3.1. Usmerjevalniku definiramo poti, za vsako pot pa HTTP metodo in njeno krmilno funkcijo (ang. *handler function*).

```
1   func initHTTPRouter() *mux.Router {
2       router := mux.NewRouter()
3       router.StrictSlash(true).Host(":9000")
4       router.HandleFunc("/kupci/", kupciGet).Methods("GET")
5       router.HandleFunc("/kupci/{id}", kupciGet).Methods("GET")
```

```
    router.HandleFunc("/kupci/", kupciPost).Methods("POST")
    router.HandleFunc("/kupci/{id}", kupciPut).Methods("PUT")
    http.Handle("/", router)
    return router
}
```

Izsek 3.9: Inicializacija končnih točk REST APIja.

Poglejmo si še primer ene izmed krmilnih funkcij. V izseku 3.10 implementiramo krmilno funkcijo `kupciGet()`.

```
func kupciGet(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    id, prs := vars["id"]

    if prs {
        log.Printf("KupecID: %s", id)
        kupecID, err := strconv.Atoi(id)
        if err != nil {
            w.WriteHeader(400)
            fmt.Fprint(w, err.Error())
            return
        }

        for _, k := range dataSource {
            if k.ID == kupecID {
                data, err := json.Marshal(k)
                if err != nil {
                    w.WriteHeader(500)
                    fmt.Fprint(w, err.Error())
                    return
                }
                fmt.Fprint(w, string(data))
                return
            }
            w.WriteHeader(404)
            return
        }
    } else {
        data, err := json.Marshal(dataSource)
        if err != nil {
            w.WriteHeader(500)
            fmt.Fprint(w, err.Error())
            return
        }
        fmt.Fprint(w, string(data))
```

```
36          return
37      }
38  }
```

Izsek 3.10: Primer implementacije krmilne funkcije `kupciGet()`.

V tretji vrstici se pozanimamo po podanem IDju v URL parameteru. Spremenljivka `prs` hrani binarno vrednost, ki nam pove, ali je bil ID sploh definiran, v spremenljivko `id` pa se shrani vrednost parametra, če obstaja. Torej, če je bil ID definiran, ga v vrstici 7 pretvorimo iz tipa `string` v tip `int`. Če pretvorba ni uspela, vrnemo odgovor s statusom `400`, v telo odgovora pa zapišemo opis napake. Če je pretvorba uspela, iteriramo skozi naš vir podatkov (vrstice 14-27) in primerjamo IDje. Ko najdemo kupca s podanim ID, strukturo entitete pretvorimo v JSON (vrstica 16). V primeru, da pretvorba v JSON ne uspe, vrnemo odgovor s statusom `500` (in v telo odgovora zapišemo opis napake), sicer pa lahko v telo odgovora zapišemo generirani JSON (vrstica 22). Če kupca s podanim IDjem ni v našem podatkovnem viru, vrnemo odgovor s statusom `404`.

Druga možnost je, da ID v URLju sploh ni bil podan. V tem primeru v JSON pretvorimo kar celoten podatkovni vir in ga zapišemo v telo odgovora (vrstice 29-36). To seveda ni najboljša ideja, ker je potencialno podatkov lahko veliko, a je za naš primer v redu. Pri nekem produkcijskem razvoju mikrostortive bi ta problem rešili na primer z implementacijo ostranjevanja (ang. *pagination*) in/ali kakih drugih parametrov poizvedbe (ang. *query parameters*).

Go program, oz. našo mikrostoritev lahko neposredno zaženemo z ukazom `go run service.go`, pri čemer se z ukazno vrstico nahajamo v mapi z datoteko `service.go`. Ko mikrostoritev teče, lahko obiščemo naslov `localhost:9000/kupci/`, ki nam vrne seznam vseh kupcev v podatkovnem viru, ali pa zahtevamo podatke specifičnega kupca z navedbo IDja, npr. `localhost:9000/kupci/1`.

Izvršljivo datoteko lahko ustvarimo z ukazom

```
go build service.go
```

V isti mapi se ustvari izvršljiva datoteka `service`, ki jo lahko zaženemo iz ukazne vrstice. Ta datoteka je privzeto dinamično povezana (razliko med statično in dinamično povezanimi izvršljivimi datotekami smo omenili v poglavju 2.4). Če želimo ustvariti statično povezano izvršljivo datoteko, ukazu pripnemo še posebno spremenljivko okolja, takole:

```
CGO_ENABLED=0 go build service.go
```

Zadnji korak, ki ga želimo opraviti, je zapakiranje naše mikrostoritve v sliko Docker. To lahko storimo na več različnih načinov:

- izvorno kodo prenesemo v sliko, ob instanciranju slike se izvorna koda prevede in prevedeni program zažene,
- dinamično povezano izvršljivo datoteko prenesemo v sliko, pri čemer za bazo naše slike uporabimo obstoječo sliko, npr. sliko Debian Linux, ki vsebuje potrebne deljene knjižnice,
- statično povezano izvršljivo datoteko prenesemo v sliko, pri čemer za bazo naše slike uporabimo prazno sliko `scratch`.

Sliko pripravimo s pomočjo datoteke *Dockerfile*, ki vsebuje posebne ukaze, ki se izvedejo ob ustvaritvi nove slike. V nadaljevanju si bomo pogledali datoteke Dockerfile in pripadajoče ukaze za pakiranje mikrostoritve v sliko Docker, za vse tri prej omenjene različne načine.

```
1 FROM golang
2 ADD . /go/src/service
3
4 WORKDIR /go
5 RUN go get
6 RUN go install service
7
8 ENTRYPOINT /go/bin/service
9 EXPOSE 9000
```

Izsek 3.11: Primer datoteke Dockerfile, pri kateri izvorno kodo prenesemo v sliko.

Primer prvega načina pakiranja, kjer izvorno kodo prenesemo v sliko, je opisan tudi na spletni strani The Go Blog [28]. Na spletni strani je podan tudi primer datoteke Dockerfile, ki smo ga v izseku 3.11 priredili, da se ujema z našim primerom.

Sprehodimo se po ukazih: ukaz `FROM` definira sliko, ki jo bomo uporabili kot bazo za našo novo sliko. V tem primeru je to slika `golang`, ki jo lahko najdemo na spletnem mestu Docker Hub. Ko bomo kasneje pognali ukaz za ustvaritev slike, se bo slika `golang` prenesla s spletnega mesta Docker Hub in bo za nadaljnjo delo na voljo lokalno. Ukaz `ADD` datoteke iz izbrane mape (v tem primeru `.` – trenutna mapa) prekopira v sliko v izbrano mapo. Z ukazom `WORKDIR` se premaknemo v izbrano mapo (v tem primeru `/go`) – vsi nadaljnji ukazi se bodo izvedli v tej mapi. Nato z ukazom `RUN` specificiramo ukaz, ki se zažene v ukazni vrstici ob instanciranju novega vsebnika Docker iz te slike. V tem primeru je to najprej `go get`, ki prenese vse odvisnosti, ki jih zahteva naš service, nato pa še `go install`, ki prevede kodo in "namesti" izvršljivo datoteko v mapo `/go/bin`. Za tem sledi ukaz `ENTRYPOINT`, ki specificira pot do programa, ki naj se zažene ob zagonu vsebnika Docker. To je v tem primeru prej nameščena izvršljiva datoteka `/go/bin/service`. Zadnji ukaz, `EXPOSE`, pa ne igra aktivne vloge, ampak z njim samo dokumentiramo vrata, ki jih uporablja slika, v našem primeru mikrostoritev uporablja vrata 9000.

Ko je datoteka Dockerfile pripravljena, novo sliko ustvarimo z ukazom

```
docker build -t my-service .
```

pri čemer je `my-service` ime oz. *tag* slike. Iz te slike pa lahko instanciramo nov vsebnik z ukazom

```
docker run -i -p 9000:9000 my-service
```

(seznam opcij ukaza in njihov pomen si lahko pogledamo z ukazom `docker help run`).

Druga dva načina, pri katerih v sliko prenesemo že prevedeni program, imata datoteki Dockerfile preprostejši in skorajda enaki. V prvem primeru,

kjer ustvarimo dinamično povezano izvršljivo datoteko, za bazo slike uporabimo sliko `debian` (izsek 3.12).

V drugem primeru je Dockerfile datoteka praktično enaka, razlikuje se le v prvi vrstici, kjer določimo, na kateri sliki bazira naša slika. V tem primeru izvršljiva datoteka ne potrebuje nobenih zunanjih knjižnic, zato lahko kot bazno sliko uporabimo prazno sliko, ki se v Dockerju imenuje `scratch`. Primer datoteke Dockerfile je v izseku 3.13.

```
1 FROM debian
2 ADD . /
3 ENTRYPOINT ["./service"]
4 EXPOSE 9000
```

Izsek 3.12: Primer datoteke Dockerfile, pri kateri v sliko prenesemo dinamično povezano izvršljivo datoteko.

```
1 FROM scratch
2 ADD . /
3 ENTRYPOINT ["./service"]
4 EXPOSE 9000
```

Izsek 3.13: Primer datoteke Dockerfile, pri kateri v sliko prenesemo statično povezano izvršljivo datoteko.

Pri teh dveh načinih moramo biti pozorni, da program prevedemo, preden ustvarimo sliko. V prvem primeru smo v sliko prenesli kar izvorno kodo, s čimer nam prevajanje programa ni bilo potrebno. V drugih dveh primerih pa moramo program najprej prevesti (z ukazom `go build`), ter šele na to ustvariti sliko.

Ukaza za ustvarjanje nove slike in instanciranje vsebnika iz te slike pa sta enaka, kot smo jih videli malo prej, pri prvem načinu pakiranje mikrostoritve v sliko.

Poglavje 4

Konfiguracija in odkrivanje mikrostoritev Go

V predhodnih poglavjih smo spoznali nekaj obstoječih ogrodij za razvoj mikrostoritev v programskem jeziku Go, izdelali pa smo tudi primer enostavne mikrostoritve. Zdaj pa se bomo posvetili dvema pomembnima tematikama razvoja mikrostoritev: konfiguraciji mikrostoritev in odkrivanju (mikro)storitev.

4.1 Ogrodje KumuluzEE

KumuluzEE, ogrodje za razvoj mikrostoritev v Javi, se je začelo razvijati v sklopu diplomske naloge leta 2015 [4]. KumuluzEE temelji na uporabi tehnologij Java EE, pomembna lastnost ogrodja pa je modularnost – v mikrostoritev, izdelano s KumuluzEE, lahko vključimo samo tiste komponente Java EE, ki jih za dano mikrostoritev potrebujemo. Mikrostoritev z izbranimi komponentami se nato zapakira v javanski paket JAR, ki ga lahko zaženemo samostojno, brez aplikacijskega strežnika.

Ker je poudarek diplomske naloge na konfiguraciji mikrostoritev in odkrivanju storitev, nas zanimata naslednja dva modula:

- KumuluzEE Config [17]: modul, ki omogoča konfiguracijo mikrosto-

ritev iz različnih konfiguracijskih virov (po prioriteti, najprej najpomembnejši): sistemske nastavitve (ang. *system properties*), spremenljivke okolja (ang. *environmental variables*), shrambe tipa ključ-vrednost in konfiguracija iz datoteke.

- KumuluzEE Discovery [18]: modul, ki omogoča uporabo implementacij odkrivanja storitev (registracija in odkrivanje storitev), trenutno sta podprti implementaciji etcd in Consul.

Vsebina teh dveh modulov je bila implementirana tudi za NodeJS [19, 20]. To nam omogoča souporabo konfiguracij in medsebojno odkrivanje javanskih KumuluzEE mikrostoritev ter NodeJS mikrostoritev – kar je korak bližje k dejstvu, da se posamezne mikrostoritve iste aplikacije lahko razvijajo z različnimi tehnologijami. V naslednjih dveh poglavjih bomo implementirali modula še za programski jezik Go in s tem ogrodju KumuluzEE poleg podpore za Javo in JavaScript dodali še podporo za razvoj KumuluzEE mikrostoritev v programskem jeziku Go. Za primera uporabe obeh modulov lahko preskočimo na poglavje 4.4.

4.2 Konfiguracija mikrostoritev Go

Celotna implementacija modula za konfiguracijo je odprtokodna in je na voljo na GitHubu:

`https://github.com/mc0239/kumuluzee-go-config`

V tem poglavju si bomo pogledali le ključne točke implementacije modula.

Kot omenjeno, nam modul za konfiguracijo omogoča uporabo različnih konfiguracijskih virov. Zato je pri implementaciji tega modula pomembno, da lahko tudi kasneje v razvoju relativno preprosto dodamo nov konfiguracijski vir k obstoječim virom, če je to potrebno.

Branje konfiguracije je možno na dva načina:

- Z uporabo strukture *Util* in njenih metod. *Util* inicializiramo s klicem funkcije `config.NewUtil()`, strukturo *Util* lahko nato uporabimo za poizvedovanje po konfiguracijskih vrednostih s klici metode `util.Get()` (ter priročnimi metodami, ki vračajo bolj specifičen tip, na primer `util.GetString()` in `util.GetInt()`).

- Z definicijo lastne strukture, katere oblika zrcali obliko konfiguracije (izsek 4.1). Najprej definiramo in instanciramo našo strukturo, nato pa jo s klicem funkcije `config.NewBundle()` napolnimo z vrednostmi iz konfiguracije. Uporaba načina *Bundle* pride prav, kadar vnaprej točno poznamo obliko konfiguracije in želimo naložiti večje število konfiguracijskih vrednosti – s tem se izognemo pogostim klicem `util.Get()`, ki bi jih izvajali pri uporabi metod strukture *Util*.

```yaml
some-config:
  protocol: "tcp"
  address:
    ip: "127.0.0.2"
    port: 3000
version: "1.0.0"
some-boolean: true
```

```go
type someConfig struct {
    Protocol string
    Address  struct {
        IP   string `config:"ip"`
        Port int
    }
    Version  string
    SomeBool bool `config:"some-boolean"`
}
```

Izsek 4.1: Primer konfiguracijske strukture. Zgoraj definirana struktura v *yaml* datoteki, spodaj struktura v *go* datoteki.

Strukturno bomo modul razdelili na:

- *config.go*, ki je glavna datoteka in vsebuje tipe in funkcije, ki so na voljo zunaj knjižnice, namenjene uporabi.

- **_config.go*, kjer zvezdica označuje kratko ime za konfiguracijski vir, primeri: *file_config.go*, *consul_config.go*

Taka razdelitev nam omogoča logično dodajanje novih virov tako, da vsak vir definiramo v svoji datoteki.

Poglejmo si najprej datoteko *config.go*.

Najprej definirajmo `interface`, ki ga morajo implementirati vsi konfiguracijski viri (izsek 4.2).

```
type configSource interface {
    Name() string
    ordinal() int
    Get(key string) interface{}
    Subscribe(key string, callback func(key string, value
     ↪  string))
}
```

Izsek 4.2: `interface` za konfiguracijske vire.

Če gremo po vrsti: vsak konfiguracijski vir mora implementirati funkciji `Name()` in `ordinal()`, ki vrneta ime vira in ordinalno število vira. Slednje označuje pomembnost oziroma prioriteto vira (večje število označuje bolj pomemben vir). Funkcija `Get()` je tista funkcija, ki nam vrne vrednost ključa, po katerem poizvedujemo. Funkcija `Subscribe()` pa je funkcija, ki nad podani ključ zažene nadzorovanje sprememb (*watch*). Vsakič, ko pride do spremembe vrednosti podanega ključa, se sproži funkcija `callback`, katere vsebino določimo sami. Kasneje bomo videli, da je funkcija `Subscribe()` relevantna le za shrambe tipa ključ-vrednost.

4.2.1 Implementacija strukture in metod *Util*

V datoteki *config.go* definirajmo tip, strukturo `Util`, ki bo hranila seznam konfiguracijskih virov (izsek 4.3).

Nato definirajmo še funkcijo `NewUtil()`, ki nam vrne instanco ravnokar definirane strukture (izsek 4.4). Funkcija poskrbi, da se inicializirajo konfiguracijski viri glede na opcije, ki jih podamo. Privzeto se inicializirata

konfiguracijska vira *environment* (spremenljivke okolja) in *file* (konfiguracija iz datoteke).

```
type Util struct {
    configSources []configSource
    Logger        logm.Logm
}
```

Izsek 4.3: Definicija strukture *Util*.

```
func NewUtil(options Options) Util {
    configs := make([]configSource, 0)

    envConfigSource := initEnvConfigSource(&lgr)
    if envConfigSource != nil {
        configs = append(configs, envConfigSource)
    }

    fileConfigSource :=
    ↪  initFileConfigSource(options.ConfigPath, &lgr)
    if fileConfigSource != nil {
        configs = append(configs, fileConfigSource)
    }

    switch options.Extension {
    case "consul":
        extConfigSource :=
        ↪  initConsulConfigSource(fileConfigSource, &lgr)
        if extConfigSource != nil {
            configs = append(configs, extConfigSource)
        }
        break
    default:
        break
    }

    // insertion sort
    for i := 1; i < len(configs); i++ {
        for k := i; k > 0 && configs[k].ordinal() >
        ↪  configs[k-1].ordinal(); k-- {
            temp := configs[k]
            configs[k] = configs[k-1]
            configs[k-1] = temp
        }
    }

```

```
34      k := Util{
35          configs,
36      }
37
38      return k
39  }
```

Izsek 4.4: Funkcija `NewUtil()`, ki inicializira konfiguracijske vire.

Pripravimo še strukturo `config.Options` (izsek 4.5), ki jo uporabimo ob klicu funkcije `NewUtil()`. Omogoča nam navedbo dodatnega konfiguracijskega vira (`Extension`) in navedbo poti do konfiguracijske datoteke (`ConfigPath`).

```
1  type Options struct {
2      Extension string
3      ConfigPath string
4      LogLevel int
5  }
```

Izsek 4.5: Definicija strukture `Options`.

Po klicu funkcije `NewUtil()` nam je na voljo instanca strukture *Util*. Opazimo, da konfiguracijski viri v strukturi *Util* niso izpostavljeni izven knjižnice (t.j. *exported*), kar pomeni, da jih uporabniki knjižnice ne morejo neposredno nasloviti. To je tudi naš cilj, saj želimo, da na primer z uporabo metode `util.Get()` naslovimo vse konfiguracijske vire hkrati in pridobimo konfiguracijo iz vira z najvišjo prioriteto.

V ta namen implementiramo metodo `util.Get()` (izsek 4.6), ki jo kličemo nad inicializirano strukturo *Util*. Metoda iterira skozi seznam konfiguracijskih virov in vrne prvo vrednost, ki jo najde – pri tem so konfiguracijski viri razporejeni od najbolj do najmanj pomembnega, saj smo jih v funkciji `NewUtil()` razporedili z *insertion sort*om).

```
1  func (c Util) Get(key string) interface{} {
2      var val interface{}
3
4      for _, cs := range c.configSources {
5          val = cs.Get(key)
```

```
6           if val != nil {
7               break
8           }
9       }
10      return val
11  }
```

Izsek 4.6: Implementacija metode `util.Get()`.

Spremenljivka, ki jo dobimo pri klicu metode `util.Get()`, je tipa `interface{}`. To je sicer pravilno, ker ne vemo, kakšnega tipa bo vrednost v shranjenem ključu, ki ga bomo naložili. Je pa uporabniku knjižnice nadležno, ker je nad spremenljivko potrebno izvesti primeren *type assertion* pred nadaljnjo uporabo. Za rešitev te težave implementiramo nekaj priročnih metod, ki namesto tipa `interface{}` vrnejo spremenljivko bolj specifičnega tipa. Vse te priročne metode so implementirane na zelo podoben način: najprej pridobimo vrednost s pomočjo prej omenjene metode `util.Get()`, nato pa poskušamo nad spremenljivko izvesti *type assertion* in/ali *type cast* v ciljni tip. Primer implementacije ene izmed metod je v izseku 4.7.

```
1   func (c Util) GetString(key string) (value string, ok bool) {
2       // try to type assert as string
3       svalue, ok := c.Get(key).(string)
4       if ok {
5           return svalue, true
6       }
7
8       // can't assert to string, return nothing
9       return "", false
10  }
```

Izsek 4.7: Implementacije priročne metode `util.GetString()`.

4.2.2 Implementacija načina *Bundle*

Implementacija načina Bundle deluje po naslednjih korakih:

1. Uporabnik definira strukturo, v katero se bo naložila konfiguracija. Ta

struktura zrcali obliko konfiguracije (glej izsek 4.1). Pomembna podrobnost je, da struktura lahko vsebuje več gnezdenih struktur.

2. S pomočjo knjižnice *mitchellh/mapstructure* podano strukturo pretvorimo v asociativno polje (*map*), pri tem lahko to polje vsebuje gnezdena asociativna polja (tako kot so gnezdene strukture).

3. Po novo nastalem asociativnem polju se rekurzivno sprehodimo in ključem s pomočjo prej opisane uporabe metod strukture *Util* določimo vrednosti.

4. S pomočjo knjižnice *mitchellh/mapstructure* z vrednostmi napolnjeno asociativno polje uporabimo, da zapolnimo vrednosti na začetku podani strukturi.

Kot omenjeno, uporaba načina *Bundle* pri nalaganju vrednosti ključev uporablja instanco strukture *Util*, kot smo to videli v poglavju 4.2.1. To pomeni, da funkcija `NewBundle()` (v izseku 4.8) ravno tako uporablja strukturo `config.Options` in omogoča navedbo enakih opcij, kot smo jih opisali pri funkciji `NewUtil()`.

```
1  func NewBundle(prefixKey string, fields interface{}, options
   ↪  Options) Bundle {
2      util := NewUtil(options)
3
4      bun := Bundle{
5          prefixKey: prefixKey,
6          fields:    &fields,
7          conf:      util,
8      }
9
10     // convert fields struct to map
11     var fieldsMap map[string]interface{}
12
13     decoder, err :=
       ↪  mapstructure.NewDecoder(&mapstructure.DecoderConfig{
14         Result:  &fieldsMap,
15         TagName: "config",
16     })
17     if err != nil {
18         return bun
```

```
19      }
20
21      err = decoder.Decode(fields)
22      if err != nil {
23          return bun
24      }
25
26      // recursively traverse all fields and set their values
        ↪  using Util.Get()
27      if prefixKey != "" {
28          prefixKey += "."
29      }
30      traverseFillBundle(fieldsMap, prefixKey, util)
31
32      // convert map back to struct
33      recoder, err :=
        ↪  mapstructure.NewDecoder(&mapstructure.DecoderConfig{
34          Result:  &fields,
35          TagName: "config",
36      })
37      if err != nil {
38          return bun
39      }
40
41      err = recoder.Decode(fieldsMap)
42      if err != nil {
43          return bun
44      }
45
46      return bun
47  }
```

Izsek 4.8: Implementacija funkcije `NewBundle()`.

4.2.3 Primer implementacije konfiguracijskega vira

Poglejmo si primer implementiranega vira konfiguracije, konfiguracije iz datoteke, v datoteki *file_config.go*.

Najprej definiramo nov tip, strukturo `fileConfigSource`, ki bo implementirala prej omenjene metode konfiguracijskega vira (izsek 4.9). Poleg tega bo struktura vsebovala še spremenljivko `config map[string]interface{}`, ki vsebuje iz datoteke naloženo konfiguracijo.

Opazimo, da sta funkciji `Name()` in `ordinal()` v izseku 4.10 zelo preprosti, saj le vračata primerno vrednost. Ker funkcijo `Subscribe()` dejansko implementiramo le pri shrambah tipa ključ-vrednost, tu funkcija ne stori ničesar.

```
type fileConfigSource struct {
    config map[string]interface{}
    logger *logm.Logm
}
```

Izsek 4.9: Struktura `fileConfigSource`.

```
func (c fileConfigSource) ordinal() int {
    return 100
}

func (c fileConfigSource) Subscribe(key string, callback
 ↪  func(key string, value string)) {
    return
}

func (c fileConfigSource) Name() string {
    return "File"
}
```

Izsek 4.10: Implementacije metod `ordinal()`, `Subscribe()` in `Name()` za vir konfiguracije iz datoteke.

Bolj zanimivi sta funkciji `initFileConfigSource()` (izsek 4.11) in `Get()` (izsek 4.12). Za prvo lahko opazimo, da funkcija ni metoda konfiguracijskega vira in jo kličemo neposredno, ob klicu pa nam vrne inicializiran konfiguracijski vir. V tej funkciji se napolni prej omenjen `config map`, seveda če konfiguracijsko datoteko uspešno preberemo. Druga funkcija, `Get()`, pa v napolnjenem asociativnem polju (*map*) poišče in vrne podani ključ, če ta obstaja.

```
func initFileConfigSource(configPath string, lgr *logm.Logm)
 ↪  configSource {
    var c fileConfigSource
```

```
    c.logger = lgr

    var joinedPath string
    if configPath == "" {
        // set default
        joinedPath = "config/config.yaml"
    } else {
        joinedPath = configPath
    }

    bytes, err := ioutil.ReadFile(joinedPath)
    if err != nil {
        return nil
    }

    err = yaml.Unmarshal(bytes, &c.config)
    if err != nil {
        return nil
    }

    return c
}
```

Izsek 4.11: Implementacija funkcije `initFileConfigSource()`.

```
func (c fileConfigSource) Get(key string) interface{} {
    tree := strings.Split(key, ".")

    // move deeper into maps for every dot delimiter
    val := c.config
    var assertOk bool
    for i := 0; i < len(tree)-1; i++ {
        if val == nil {
            return nil
        }
        val, assertOk = val[tree[i]].(map[string]interface{})

        if !assertOk {
            return nil
        }
    }

    return val[tree[len(tree)-1]]
}
```

Izsek 4.12: Implementacija metode `Get()`.

4.3 Registracija in odkrivanje mikrostoritev Go

Celotna implementacija modula za odkrivanje mikrostoritev je odprtokodna in je na voljo na GitHubu:

`https://github.com/mc0239/kumuluzee-go-discovery`

Modul omogoča interoperabilnost z javanskimi in NodeJS mikrostoritvami, ki uporabljajo modul za odkrivanje mikrostoritev ogrodja KumuluzEE.

V tem poglavju si bomo pogledali le ključne točke implementacije modula. Tako kot smo to videli pri implementaciji konfiguracijskega modula, tudi tukaj knjižnico strukturno razdelimo na glavno datoteko *discovery.go*, in posamezne datoteke za različne implementacije registrarjev za odkrivanje storitev (trenutno lahko v repozitoriju najdemo implementacijo za Consul, *consul_discovery.go*).

Osnoven postopek uporabe knjižnice je razmeroma enostaven: s klicem funkcije `discovery.New()` pridobimo instanco strukture *Util*. Ta struktura ima na voljo metodi `RegisterService()` in `DiscoverService()`, ki sprejmeta opcijske parametre in/ali konfiguracijo s podajanjem strukture `RegisterOptions` in `DiscoverOptions`. Torej, če želimo storitev registrirati, kličemo prvo metodo, če pa želimo neko storitev odkriti, kličemo drugo metodo.

Konfiguracija mikrostoritve pred registracijo je tu ključnega pomena. Mikrostoritve, ki jih želimo registrirati skladno z ogrodjem KumuluzEE, zahtevajo nekaj specifičnih definiranih vrednosti v konfiguraciji. Primer konfiguracije mikrostoritve, ki jo registriramo, je v izseku 4.13.

Tri vrednosti, ki identificirajo našo mikrostoritev, so *kumuluzee.name* (ime storitve), *kumuluzee.version* (verzija storitve) in *kumuluzee.env.name* (ime okolja storitve). Verzija in ime okolja sicer imata predvidene privzete vrednosti, v primeru, da ju ne navedemo, obvezno pa moramo navesti ime

storitve. Pri tem omenimo še, da teh treh ključev ni nujno potrebno nevesti v konfiguraciji, ker, kot bomo videli kasneje, jih lahko navedemo tudi v parametrih pri klicu `util.RegisterService()`.

```
kumuluzee:
  name: test-service
  version: 1.0.0
  server:
    base-url: http://localhost:9000
    http:
      port: 9000
  env:
    name: dev
  discovery:
    consul:
      hosts: http://localhost:8500
```

Izsek 4.13: Primer konfiguracije mikrostoritve za registracijo.

Ostali so ključi, ki jih ob registraciji ne moremo navesti, ampak morajo biti navedeni v konfiguraciji. Ključa *kumuluzee.server.base-url* in *kumuluzee.server.http.port* navajata, na katerem naslovu (in vratih) se naša mikrostoritev nahaja. Ključ *kumuluzee.discovery.consul.hosts* pa navaja naslov, kjer se nahaja orodje za odkrivanja storitev, kamor pošljemo zahtevo po registraciji naše mikrostoritve.

```
type Util struct {
    discoverySource discoverySource
    Logger          logm.Logm
}

type discoverySource interface {
    RegisterService(options RegisterOptions) (serviceID
    ↪  string, err error)
    DiscoverService(options DiscoverOptions) (Service, error)
}
```

Izsek 4.14: Definicija strukture *Util* in tipa `discoverySource`.

V datoteki *discovery.go* definiramo strukturo *Util*, ki vsebuje instanco "vira odkrivanja storitev"(simetrično konfiguraciji, kjer smo imeli "vir konfi-

guracije", poglavje 4.2). Vsak vir odkrivanja storitev pa mora implementirati metodi za registracijo in odkrivanje storitev (izsek 4.14).

Nato v izseku 4.15 definiramo že omenjeno funkcijo `New()`, ki nam ustvari pripravljeno instanco strukture *Util*.

```
1  func New(options Options) Util {
2      conf = config.NewUtil(config.Options{
3          ConfigPath: options.ConfigPath,
4          LogLevel:   logm.LvlWarning,
5      })
6
7      var src discoverySource
8      if options.Extension == "consul" {
9          src = initConsulDiscoverySource(conf, &lgr)
10     }
11
12     k := Util{
13         src,
14     }
15
16     return k
17 }
```

Izsek 4.15: Implementacija funkcije `New()`, ki vrne instanco strukture *Util*.

4.3.1 Implementacija registracije mikrostoritve

Mikrostoritev registriramo s klicem metode `util.RegisterService()` (izsek 4.16), ki kot parameter sprejme strukturo `discovery.RegisterOptions`.

Opazimo, da je metoda enostavna, saj zahtevo po registraciji le delegira dejanskemu viru odkrivanja storitev. V strukturi pa imamo na voljo kar nekaj parametrov. Omenili smo, da lahko ime, verzijo in ime okolja mikrostoritve navedemo tu, namesto v konfiguraciji – to so polja `Value`, `Environment` in `Version`. Na voljo sta še polji `PingInterval` in `TTL`, ki definirata, na koliko časa se mikrostoritev javlja registru in koliko časa mikrostoritev ostane v registru, če se ne javlja več (*TTL* smo omenili v poglavju 2.2).

```
1  type RegisterOptions struct {
2      Value string
```

```
    TTL int64
    PingInterval int64
    Environment string
    Version string
        Singleton bool
}

func (d Util) RegisterService(options RegisterOptions)
 ↪  (string, error) {
    return d.discoverySource.RegisterService(options)
}
```

Izsek 4.16: Struktura `RegisterOptions` in implementacija metode `RegisterService()`.

4.3.2 Implementacija odkrivanja mikrostoritev

Mikrostoritve, ki so registrirane v registru, lahko odkrijemo s klicem metode `util.DiscoverService()`, ki kot parameter sprejme strukturo `discovery.DiscoverOptions` (izsek 4.17).

```
type DiscoverOptions struct {
    Value string
    Environment string
    Version string
    AccessType string
}

func (d Util) DiscoverService(options DiscoverOptions)
 ↪  (Service, error) {
    return d.discoverySource.DiscoverService(options)
}
```

Izsek 4.17: Struktura `DiscoverOptions` in implementacija metode `DiscoverService()`.

Opazimo, da tudi tukaj metoda zahtevo po odkrivanju le delegira dejanskemu viru odkrivanja storitev. Struktura `DiscoverOptions` pa vsebuje polja, ki definirajo parametre poizvedbe po storitvah. Pri registraciji smo

definirali parametre ime, verzijo in ime okolja, pri odkrivanju pa iščemo mikrostoritve po točno teh treh parametrih. V odgovor kot rezultat funkcije `util.DiscoverService()` dobimo mikrostoritev, ki se je ujemala z definiranimi parametri ali pa dobimo napako (na primer, da se nobena storitev ni ujemala s poizvedbo). V primeru, da je v registru na voljo več instanc mikrostoritve, ki se ujema s podanimi parametri, bo izmed ujemajočih instanc izbrana in vrnjena naključna instanca mikrostoritve. V kodi vidimo, da je vrnjena spremenljivka, ki hrani instanco mikrostoritve, tipa `Service`. To je preprosta struktura, ki vsebuje naslov in vrata mikrostoritve, ki smo jo želeli odkriti (izsek 4.18).

```
type Service struct {
    Address string
    Port    int
}
```

Izsek 4.18: Definicija strukture `Service`.

4.3.3 Primer implementacije vira

Poglejmo si primer implementacije vira odkrivanja storitev, v datoteki *consul_discovery.go*. Najprej v inicializacijski funkciji naložimo konfiguracijo iz datoteke (ali okoljskih spremenljivk) in pripravimo novega odjemalca Consul z uporabo knjižnice `consul/api` (izsek 4.19).

```
func initConsulDiscoverySource(options config.Options)
 ↪  discoverySource {
        var d consulDiscoverySource
        d.configOptions = options

        consulClientConfig := api.DefaultConfig()
        conf := config.NewUtil(options)

        if addr, ok :=
         ↪  conf.GetString("kumuluzee.config.consul.hosts");
         ↪  ok {
            consulClientConfig.Address = addr
        }
```

```
11          startRetryDelay, ok :=
             ↪  conf.Get("kumuluzee.config.start-retry-delay-ms")
             ↪  .(float64)
12          if !ok {
13                  startRetryDelay = 500
14          }
15          d.startRetryDelay = int64(startRetryDelay)
16
17          maxRetryDelay, ok :=
             ↪  conf.Get("kumuluzee.config.max-retry-delay-ms")
             ↪  .(float64)
18          if !ok {
19                  maxRetryDelay = 900000
20          }
21          d.maxRetryDelay = int64(maxRetryDelay)
22
23          client, err := api.NewClient(consulClientConfig)
24          d.client = client
25          return d
26  }
```

Izsek 4.19: Inicializacija vira odkrivanja storitev Consul.

Poglejmo si metodo `RegisterService()` v izseku 4.20. Tu pripravimo vse potrebne parametre za registracijo mikrostoritve v Consul – ime, verzija, okolje mikrostoritve, ter interval osveževanja vrednosti TTL v registru. Pripravimo spremenljivko `regconf`, v kateri bomo te parametre hranili. V vrsticah 4-9 naložimo privzete vrednosti za parametre, ki jih shranimo jih v `regconf`. V vrstici 12 preberemo konfiguracijo iz datoteke, t.j. naložimo parametre, ki so v datoteki definirani. Shranimo jih v `regconf`, pri čemer vrednosti parametrov iz datoteke nadomestijo vrednosti, trenutno shranjene v `regconf`. V vrsticah 15-29 preverimo, ali je bila podana kakšna vrednost kot parameter funkcije `RegisterService()` (v strukturi `RegisterOptions`), in morebitne podane vrednosti naložimo, s čemer spet nadomestimo vrednosti, ki so trenutno shranjene v `regconf`.

Nato kličemo metodo `register()`, ki s pomočjo knjižnice `consul/api` registrira mikrostoritev v register, za vzdrževanje vnosa v register pa se uporablja način TTL (omenili smo ga v poglavju 2.2). Ko je mikrostoritev registrirana, se v novi niti (ang. *thread*) kliče metoda `checkIn()`, ki se v rednih

intervalih (v tem primeru na 20 sekund) javlja registru, da mikrostoritev deluje. Metoda `checkIn()` je implementirana rekurzivno (repna rekurzija) in v izbranem intervalu kliče sama sebe.

```
1  func (d consulDiscoverySource) RegisterService(options
   ↪  RegisterOptions) (serviceID string, err error) {
2
3          // Load default values
4          regconf := registerConfiguration{}
5          regconf.Server.HTTP.Port = 80
6          regconf.Env.Name = "dev"
7          regconf.Version = "1.0.0"
8          regconf.Discovery.TTL = 30
9          regconf.Discovery.PingInterval = 20
10
11         // Load from configuration file, overriding defaults
12         config.NewBundle("kumuluzee", &regconf,
           ↪  d.configOptions)
13
14         // Load from RegisterOptions, override file
           ↪  configuration
15         if options.Value != "" {
16                 regconf.Name = options.Value
17         }
18         if options.Environment != "" {
19                 regconf.Env.Name = options.Environment
20         }
21         if options.Version != "" {
22                 regconf.Version = options.Version
23         }
24         if options.TTL != 0 {
25                 regconf.Discovery.TTL = options.TTL
26         }
27         if options.PingInterval != 0 {
28                 regconf.Discovery.PingInterval =
                   ↪  options.PingInterval
29         }
30
31         regService := registerableService{
32                 options:   &regconf,
33                 singleton: options.Singleton,
34         }
35         d.registerableServices =
           ↪  append(d.registerableServices, regService)
36
37         uuid4, err := uuid.NewV4()
38         if err != nil {
```

```
39                  d.logger.Error(fmt.Sprintf(err.Error()))
40          }
41
42          regService.id = regService.options.Name + "-" +
             ↪  uuid4.String()
43          regService.name = regService.options.Env.Name + "-" +
             ↪  regService.options.Name
44          regService.versionTag = "version=" +
             ↪  regService.options.Version
45
46          d.register(&regService, d.startRetryDelay)
47          go d.checkIn(&regService, d.startRetryDelay)
48
49          return regService.id, nil
50  }
```

Izsek 4.20: Implementacija metode `RegisterService()` v datoteki *consul_discovery.go*.

Poglejmo si še metodo `DiscoverService()` v izseku 4.21. Tu podane parametre v obliki strukture `DiscoverOptions` uporabimo za poizvedbo po mikrostoritvah v registru. V ta namen tudi uporabimo `consul/api`, klic funckije v vrstici 3 nam vrne seznam vseh zdravih mikrostoritev v registru, ki se ujemajo z našo poizvedbo po imenu mikrostoritve in okolju mikrostoritve – ta trenutek so v seznamu mikrostoritve vseh morebitnih verzij. V vrstici 11 kličemo metodo `extractService()`, ki kot argumenta sprejme omenjen seznam ujemajočih mikrostoritev, ter verzijo oz. območje verzij, vrne pa nam mikrostoritev, ki se verzijsko nahaja v podanem območju (oz. se verzija mikrostoritve ujema s podano verzijo).

```
1   func (d consulDiscoverySource) DiscoverService(options
     ↪  DiscoverOptions) (Service, error) {
2           queryServiceName := options.Environment + "-" +
             ↪  options.Value
3           serviceEntries, _, err :=
             ↪  d.client.Health().Service(queryServiceName, "",
             ↪  true, nil)
4           if err != nil {
5                   return Service{}, fmt.Errorf("Service
                     ↪  discovery failed: %s", err.Error())
6           }
7           versionRange, err := d.parseVersion(options.Version)
```

```
8           if err != nil {
9                   return Service{}, fmt.Errorf("wantVersion
                     ↪  parse error: %s", err.Error())
10          }
11          return d.extractService(serviceEntries, versionRange)
12  }
```

Izsek 4.21: Implementacija metode `DiscoverService()` v datoteki *consul_discovery.go*.

4.4 Razširitev primera mikrostoritve

Primer mikrostoritve, ki smo ga izdelali v poglavju 3.2, bomo sedaj razširili z opisanima knjižnicama. Najprej bomo s pomočjo modula za konfiguracijo našo mikrostoritev konfigurirali iz datoteke, nato pa bomo s pomočjo modula za odkrivanje storitev našo mikrostoritev registrirali v Consul.

Najprej si zamislimo primer, kaj glede naše mikrostoritve si sploh želimo konfigurirati. Recimo, da želimo imeti na voljo konfiguracijski ključ, ki določi, ali pri vračanju podatkov kupcev vrnemo naslov ali ne. Pripravimo datoteko *config.yaml*, ki se nahaja v isti mapi kot datoteka *service.go* (izsek 4.22).

```
1  rest-config:
2    return-address: false
```

Izsek 4.22: Vsebina datoteke *config.yaml*.

V datoteki *service.go* najprej dodamo stavek *import*, s katerim uvozimo knjižnico:

```
import "github.com/mc0239/kumuluzee-go-config/config"
```

Poleg že obstoječega `dataSource` dodamo `config.Util`, s katerim bomo dostopali do konfiguracije:

```
var conf config.Util
```

Nato definiramo funkcijo, v kateri inicializiramo dostop do konfiguracije (izsek 4.23).

```
func initConfig() {
    conf = config.NewUtil(config.Options{
        //Extension:  "consul",
        ConfigPath: "config.yaml",
    })
}
```

Izsek 4.23: Inicializacija modula za konfiguracijo.

Funkcijo `initConfig()` pa nato kličemo v funkciji `main()`, poleg obstoječih inicializacijskih funkcij (izsek 4.24).

```
func main() {
    initDataSource()
    initConfig()
    r := initHTTPRouter()
    log.Fatal(http.ListenAndServe(":9000", r))
}
```

Izsek 4.24: Funkcija `main()` z dodanim klicem `initConfig()`.

Ko je konfiguracija uspešno inicializirana, lahko do naše konfigracijske vrednosti dostopamo s klicem:

```
val, ok := conf.GetBool("rest-config.return-address")
```

Spremenljivka `ok` nam pove, ali je vrnjena konfiguracijska vrednost v spremenljivki `val` veljavna ali ne (vrednost je lahko neveljavna na primer, če navedemo ključ, za katerega vrednost ne obstaja).

Konfiguracijo sedaj lahko uporabimo v funkciji `kupciGet()`, kjer vračamo podatke o kupcu (izsek 4.25). Če je vrednost konfiguracije za ključ `rest-config.return-address` enaka `false`, naslova ne vrnemo, zato ga v vrstici 5 nastavimo na nič.

```
for _, k := range dataSource {
    if k.ID == kupecID {
        val, ok := conf.GetBool("rest-config.return-address")
        if ok && !val {
            k.Naslov = ""
        }
        data, err := json.Marshal(k)
```

```
8           if err != nil {
9               w.WriteHeader(500)
10              fmt.Fprint(w, err.Error())
11              return
12          }
13          fmt.Fprint(w, string(data))
14          return
15      }
16      w.WriteHeader(404)
17      return
18  }
```

Izsek 4.25: Del kode v krmilni funkciji `kupciGet()`, kjer uporabimo konfiguracijo.

Primer odgovora JSON, ki ga dobimo, je v izseku 4.26.

```
1  {
2      "id": 1,
3      "ime": "Janez",
4      "priimek": "Novak",
5      "naslov": "",
6      "postna-stevilka": 1000
7  }
```

Izsek 4.26: Primer odgovora JSON funkcije `KupciGet()`.

Opazimo, da v generiranem JSONu še vedno dobimo polje *naslov*, čeprav je njegova vrednost prazna. Če nas to moti, lahko to popravimo tako, da znački pri polju Naslov v definiciji strukture `Kupec` dodamo besedo `omitempty` (izsek 4.27).

```
1  type Kupec struct {
2      ...
3      Naslov  string `json:"naslov,omitempty"`
4  }
```

Izsek 4.27: Popravek v strukturi *Kupec*, kjer znački pri polju Naslov dodamo besedo `omitempty`.

Tako. Našo mikrostoritev smo sedaj uspešno konfigurirali iz datoteke, zdaj pa bomo z uporabo modula za odkrivanje storitev našo mikrostoritev

registrirali – tako bo naša mikrostoritev na voljo za odkrivanje drugim mikrostoritvam.

Prvi pogoj za uporabo odkrivanja storitev je seveda to, da imamo na voljo nek register. V nadaljevanju se bomo poslužili uporabe registra Consul, lokalno razvojno (*development*) verzijo pa lahko poženemo tudi z Dockerjem:

```
docker run -d --name=dev-consul --net=host consul
```

Kot smo omenili v poglavju 4.3, registracija vključuje tudi določeno konfiguracijo. Našo obstoječo *config.yaml* datoteko dopolnimo s potrebnimi vrednostmi (izsek 4.28).

```
1  kumuluzee:
2    name: kupci-service
3    server:
4      http:
5        port: 9000
6    discovery:
7      consul:
8        hosts: http://localhost:8500
9  rest-config:
10   return-address: false
```

Izsek 4.28: Dopolnjena datoteka *config.yaml* s parametri za registracijo mikrostoritve.

V datoteki *service.go* dodamo nov stavek *import*, s katerim uvozimo knjižnico:

```
import "github.com/mc0239/kumuluzee-go-discovery/discovery"
```

Spremenljivkama `dataSource` in `conf` dodamo še spremenljivko `disc` tipa `discovery.Util`, s katerim bomo našo storitev kasneje registrirali:

```
var disc discovery.Util
```

Nato definiramo funkcijo, s katero inicializiramo orodje za delo z odkrivanjem storitev, pri čemer tako kot prej pri konfiguraciji tu navedemo pot do konfiguracijske datoteke (izsek 4.29).

```
func initDiscovery() {
    disc = discovery.New(discovery.Options{
        Extension:  "consul",
        ConfigPath: "config.yaml",
    })
}
```

Izsek 4.29: Inicializacija modula za odkrivanje.

Funckijo `initDiscovery()` pa nato kličemo v funkciji `main()` (izsek 4.30). Po klicu funkcije pa lahko uporabimo `disc` za registracijo naše mikrostoritve tako, da kličemo metodo `disc.RegisterService()`.

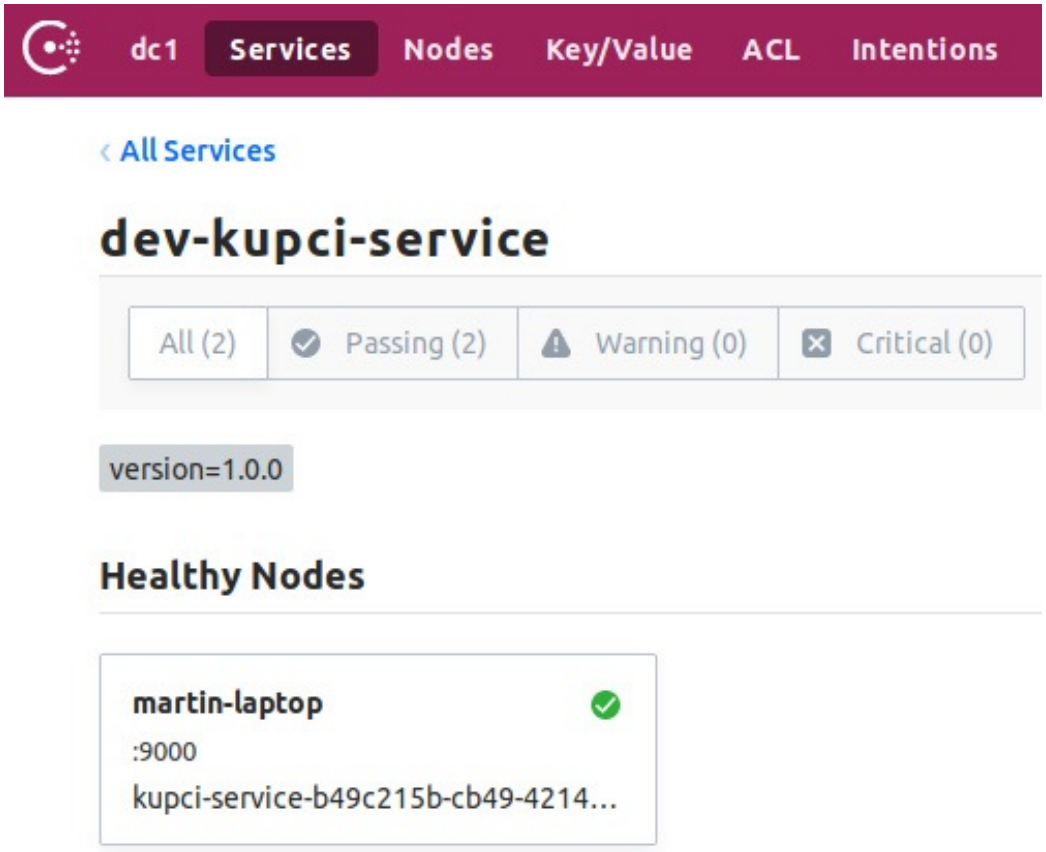
```
func main() {
        initDataSource()
        initConfig()
        initDiscovery()
        _, err :=
         ↪  disc.RegisterService(discovery.RegisterOptions{})
        if err != nil {
                panic(err)
        }

        r := initHTTPRouter()
        log.Fatal(http.ListenAndServe(":9000", r))
}
```

Izsek 4.30: Dopolnjena funkcija `main()` s klicem funkcije `initDiscovery()`.

Če je naša mikrostoritev uspešno registrirana, lahko preverimo v Consulu (slika 4.1).

Tako smo primer mikrostoritve, ki smo jo razvili v poglavju 3.2, razširili z moduloma za konfiguracijo in odkrivanje storitev. S pomočjo modula za konfiguracijo smo prebrali konfiguracijo iz datoteke in nato to konfiguracijo uporabili pri logiki v končni točki REST. Z modulom za odkrivanje storitev pa smo našo mikrostoritev registrirali v register Consul, s čimer je mikrostoritev na voljo za odkrivanje drugim mikrostoritvam.

Ta dva modula sta odprtokodna in na voljo uporabi pri razvoju mikrostoritev v programskem jeziku Go. Medtem ko sta bila modula spisana z



Slika 4.1: Registrirana mikrostoritev v Consulu

interoperabilnostjo z javanskim ogrodjem KumuluzEE v mislih, nič ne preprečuje splošne uporabe teh dveh modulov pri razvoju mikrostoritev Go.

Poglavje 5

Zaključek

V uvodnih poglavjih naloge smo se spoznali z nekaterimi ključnimi pojmi, ki jih srečamo pri razvoju mikrostoritev. Nato smo razvili tri enostavne mikrostoritve z uporabo treh različnih ogrodij za razvoj mikrostoritev in s tem predstavili nekaj funkcionalnosti, ki jih ta ogrodja ponujajo. Zatem smo se lotili izdelave dveh modulov – Gojevskih knjižnic, enega za konfiguracijo mikrostoritev in drugega za odkrivanje storitev, na koncu pa smo prikazali tudi primer uporabe teh dveh knjižnic.

Razvoj programske opreme je proces, ki se neprestano izvaja – programsko opremo, ki jo razvijamo in uporabljamo, nenehno nadgrajujemo s popravki in novimi funkcionalnostmi. Razlogov za to nadgrajevanje je veliko, na primer spremenjene želje uporabnikov (skozi čas), ali pa poenostavitev uporabe ali vključevanje novih in izključevanje starih funkcionalnosti. Možno je, da bo z leti vsebina te diplomske naloge postala nerelevantna – preprosto zato, ker se primeri obstoječih ogrodij mikrostoritev, ki smo jih pogledali, lahko spremenijo. Na trgu se lahko pojavijo neka nova ogrodja, ki bodo nadomestila ta, ki smo jih omenili tu. Mogoče se bo uveljavil kakšen nov programski jezik, ki bo preferenčno uporabljen pred programskim jezikom Go.

V vsakem primeru pa programska oprema, ki je spisana danes, se čez noč ne migrira na neko novo tehnologijo ali programski jezik. Ampak se, če se ne

nadgrajuje več, zagotovo vzdržuje, dokler bo programska oprema v (aktivni) rabi.

Implementaciji modulov za konfiguracijo in odkrivanje storitev v programskem jeziku Go, predstavljeni v diplomski nalogi, sta le osnovi za nadaljnji razvoj: z uporabo teh modulov se bo zagotovo pojavila želja po novi funkcionalnosti v samih modulih, ali pa bo treba odpraviti kakšno napako v kodi, ki v času pisanja diplomske naloge ni bila najdena. Zagotovo bo v knjižnicah skozi njun razvoj prihajalo do sprememb (mogoče tudi bolj fundamentalnih), zato, da bosta knjižnici bolje služila svojemu namenu.

Literatura

[1] Charles Anderson. "Docker [software engineering]". V: *IEEE Software* 32.3 (2015), str. 102–c3.

[2] *dep, Dependency management for Go*. Dosegljivo: `https://golang.github.io/dep/`. [Dostopano: 9. 8. 2018]. 2018.

[3] *Docker*. Dosegljivo: `https://www.docker.com/`. [Dostopano: 4. 9. 2018]. 2018.

[4] Tilen Faganel. "Ogrodje za razvoj mikrostoritev v Javi in njihovo skaliranje v oblaku". Diplomska naloga. Univerza v Ljubljani, 2015.

[5] *Github: Go Kit*. Dosegljivo: `https://github.com/go-kit/kit`. [Dostopano: 4. 9. 2018]. 2018.

[6] *Github: Go Micro*. Dosegljivo: `https://github.com/micro/go-micro`. [Dostopano: 4. 9. 2018]. 2018.

[7] *Github: Kite*. Dosegljivo: `https://github.com/koding/kite`. [Dostopano: 4. 9. 2018]. 2018.

[8] *Github: Kite - dnode protocol*. Dosegljivo: `https://github.com/substack/dnode-protocol/blob/master/doc/protocol.markdown`. [Dostopano: 4. 9. 2018]. 2018.

[9] *Go 1.5 Vendor Experiment*. Dosegljivo: `golang.org/s/go15vendor`. [Dostopano: 9. 8. 2018]. 2018.

[10] *Go Kit: FAQ*. Dosegljivo: `https://gokit.io/faq/`. [Dostopano: 4. 9. 2018]. 2018.

[11] Jean-Philippe Gouigoux in Dalila Tamzalit. "From monolith to Microservices: Lessons learned on an industrial migration to a web oriented architecture". V: *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*. IEEE. 2017, str. 62–65.

[12] Thomas Hunter II. *Advanced Microservices: A Hands-on Approach to Microservice Infrastructure and Tooling*. 1. izd. Apress, 2017.

[13] Martin Fowler in James Lewis. *Microservices*. 2014. URL: `https://www.martinfowler.com/articles/microservices.html`. (dostopano: 20. 8. 2018).

[14] Pooyan Jamshidi in sod. "Microservices: The Journey So Far and Challenges Ahead". V: *IEEE Software* 35.3 (2018), str. 24–35.

[15] Ahmet Alp Balkan Jeff Nickoloff. *Docker in Action*. Manning, 2016.

[16] *Kite: Library for writing distributed microservices*. Dosegljivo: `https://blog.gopheracademy.com/birthday-bash-2014/kite-microservice-library/`. [Dostopano: 4. 9. 2018]. 2018.

[17] *kumuluz/kumuluzee-config*. Dosegljivo: `https://github.com/kumuluz/kumuluzee-config`. [Dostopano: 15. 6. 2018]. 2018.

[18] *kumuluz/kumuluzee-discovery*. Dosegljivo: `https://github.com/kumuluz/kumuluzee-discovery`. [Dostopano: 15. 6. 2018]. 2018.

[19] *kumuluz/kumuluzee-nodejs-config*. Dosegljivo: `https://github.com/kumuluz/kumuluzee-nodejs-config`. [Dostopano: 15. 6. 2018]. 2018.

[20] *kumuluz/kumuluzee-nodejs-discovery*. Dosegljivo: `https://github.com/kumuluz/kumuluzee-nodejs-discovery`. [Dostopano: 15. 6. 2018]. 2018.

[21] David Heinemeier Hansson Leonard Richardson Sam Ruby. *RESTful Web Services*. O'Reilly, 2007.

[22] *Micro - Simplifying Cloud Native Development*. Dosegljivo: `https://micro.mu/explore/`. [Dostopano: 4. 9. 2018]. 2018.

[23] *Protocol Buffers, Google*. Dosegljivo: `https://developers.google.com/protocol-buffers/`. [Dostopano: 4. 9. 2018]. 2018.

[24] Mark Richards. *Microservices AntiPatterns and Pitfalls*. O'Reilly, 2016.

[25] Mark Richards. *Microservices vs. Service-Oriented Architecture*. O'Reilly, 2016.

[26] Chris Richardson. *Pattern: Service registry*. 2017. URL: `https://microservices.io/patterns/service-registry.html`. (dostopano: 22. 8. 2018).

[27] Chris Richardson. *Service Discovery in a Microservices Architecture*. 2015. URL: `https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/`. (dostopano: 20. 8. 2018).

[28] *The Go Blog: Deploying Go servers with Docker*. Dosegljivo: `https://blog.golang.org/docker`. [Dostopano: 14. 8. 2018]. 2018.

[29] *The Go Project*. Dosegljivo: `https://golang.org/project/`. [Dostopano: 15. 6. 2018]. 2018.

[30] Johannes Thönes. "Microservices". V: *IEEE Software* 32.1 (2015), str. 116–116.

[31] *What is a Container, Docker*. Dosegljivo: `https://www.docker.com/resources/what-container`. [Dostopano: 4. 9. 2018]. 2018.

[32] Eberhard Wolff. *Microservices Flexible Software Architecture*. Addison-Wesley Professional, 2016.

[33] Mazin Yousif. "Microservices". V: *IEEE Cloud Computing* 3.5 (2016), str. 4–5.