Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Žan Ožbot

Reaktivni model razvoja aplikacij z uporabo ogrodja Vert.x

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Sebastijan Šprager

Ljubljana, 2017

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:
Reaktivni model razvoja aplikacij z uporabo ogrodja Vert.x

Tematika naloge:

Preučite koncept reaktivnega programiranja in ga primerjajte z objektnim pristopom. Raziščite in opišite ogrodja za razvoj reaktivnih aplikacij in jih primerjajte. Osredotočite se na ogrodje Vert.x in ga preučite. Izdelajte razširitev, ki bo omogočala uporabo ogrodja Vert.x pri razvoju mikrostoritev. Uporabo razvite rešitve prikažite na primeru.

Rad bi se zahvalil doc. dr. Sebastijanu Špragerju in prof. dr. Matjažu Branku Juriču za mentorstvo, pomoč in usmeritev pri izdelavi diplomskega dela. Zahvaljujem se tudi as. Janu Meznariču za pregled in nasvete v zvezi z izdelavo kodnega dela diplomske naloge.
Posebno bi se rad zahvalil družini in prijateljem, ki so me podpirali in vzpodbujali med pisanjem samega dela.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
AMQP	Advanced Message Queuing Protocol	napreden protokol za sporočilne vrste
API	Application Programming Interface	aplikacijski programski vmesnik
CSS	Cascading Style Sheets	predloge, ki določajo izgled spletnih strani
HTML	HyperText Markup Language	jezik za označevanje nadbesedila
HTTP	Hypertext Transfer Protocol	protokol za prenos hipertekst
JAR	Java Archive	arhiv java
JCA	Java EE Connector Architecture	arhitektura priključka v poslovni izdaji jave
JDBC	Java Database Connectivity	povezava z bazo podatkov v javi
JRE	Java Runtime Environment	izvajalno okolje java
JSON	JavaScript Object Notation	notacija objektov v jeziku JavaScript
JVM	Java Virtual Machine	javanski navidezni stroj
JWT	JSON Web Tokens	spletni žetoni JSON
Java EE	Java Enterprise Edition	poslovna izdaja jave

REST	Representational State Transfer	arhitekturna načela za izdelavo storitev
SQL	Structured Query Language	strukturiran povpraševalni jezik za delo s podatkovnimi bazami
STOMP	Streaming Text Oriented Message Protokol	protokol za pretok tekstovnih sporočil
TCP	Transmission Control Protocol	internetni protokol
XSS	Cross-site Scripting	večdomensko izvajanje kode

Povzetek

Naslov: Reaktivni model razvoja aplikacij z uporabo ogrodja Vert.x

Avtor: Žan Ožbot

Spletne in mobilne aplikacije postajajo čedalje bolj odzivne na dogodke v realnem času z namenom, da bi omogočile boljšo uporabnikovo izkušnjo. Za izdelavo takih aplikacij potrebujemo primerna orodja in reaktivno programiranje je ena izmed rešitev. Reaktivno programiranje postaja zaradi prednosti, ki jih ponuja, vedno večji razlog za opuščanje standardnega objektnega pristopa. V diplomskemu delu smo opisali koncepte reaktivnega programiranja in ga primerjali z objektnim pristopom. V nadaljevanju smo primerjali številna ogrodja za izdelavo reaktivnih aplikacij. Bolj podrobneje smo preučili ogrodje Vert.x, saj smo ga v sklopu diplomskega dela integrirali v odprtokodno ogrodje KumuluzEE, ki se uporablja za izdelavo mikrostoritev v Javi EE.

Ključne besede: Vert.x, reaktivno programiranje, mikrostoritve, KumuluzEE.

Abstract

Title: Reactive model for developing applications using Vert.x toolkit

Author: Žan Ožbot

Web and mobile applications consist of real-time events of different kinds in order to ensure the best possible user experience. To develop such applications, proper tools are needed and reactive programming is one of the possible solutions. Due to its many advantages, reactive programming is becoming an increasing reason to abandon standard object-oriented approach. Therefore, in this thesis we first describe the concepts of reactive programming and compare it to object-oriented programming. We continue by comparing numerous available frameworks for developing reactive applications. Framework Vert.x is described in more detail. Main contribution of this thesis is implementation of Vert.x in the open-source framework KumuluzEE which is used to develop microservices.

Keywords: Vert.x, reactive programming, microservices, KumuluzEE.

Poglavje 1

Uvod

1.1 Motivacija

Število spletnih aplikacij se iz dneva v dan povečuje. Vedno več uporabnikov zahteva skoraj ničelni odzivni čas ter stoodstotno dosegljivost kjer koli se nahajajo. Za uresničitev zgornjih ciljev je potrebno ustrezno zasnovati aplikacije in uporabiti ustrezna orodja. Eno izmed orodij, ki se je v zadnjih letih uveljavilo na področju izdelave odzivnih aplikacij, je reaktivno programiranje. Reaktivno programiranje je programiranje z asinhronimi podatkovnimi tokovi [25]. Omogoča izdelavo reaktivnih aplikacij, ki morajo po reaktivnem manifestu [15] vsebovati štiri lastnosti, ki so med sabo tesno povezane. Te so odzivnost, odpornost na odpovedi, elastičnost in sporočilna vodenost. Pri izdelavi reaktivnih aplikacij so nam v oporo raznovrstna ogrodja, ki implementirajo paradigmo reaktivnega programiranja.

1.2 Opis problema

Če želimo spletno aplikacijo ponuditi veliki množici končnih uporabnikov, moramo to tudi ustrezno zasnovati. Da bi omogočili lastnosti, ki jih opisuje reaktivni manifesto, moramo aplikacijo razdeliti na posamezne komponente, kjer vsaka opravlja svoje delo. Takšne komponente imenujemo mikrostori-

tve. Ogrodje, ki se uporablja za razvoj mikrostoritev in smo ga uporabili znotraj diplomske naloge, je ogrodje KumuluzEE[1]. Reaktivno programiranje nam znotraj mikrostoritev omogoči neblokirajoče izvajanje programske kode. Eno izmed ogrodij, ki omogoča uporabo funkcionalnosti reaktivnega programiranja, je ogrodje Vert.x[2]. Razvijalci velikokrat izbirajo ogrodja, ki jih bodo uporabili za izdelavo spletnih aplikacij glede na funkcionalnost in nabor komponent, ki jih te ponujajo. Z namenom, da bi dodatno prepričali razvijalce v uporabo ogrodja KumuluzEE, smo se odločili razširiti nabor komponent le-tega in razvili razširitev za enostavnejšo uporabo ogrodja Vert.x znotraj mikrostoritev.

1.3 Cilji in prispevki

V diplomskemu delu smo raziskali paradigmo reaktivnega programiranja in ga primerjali z objektnim pristopom. Na kratko smo opisali popularna ogrodja, ki jih uporabljamo za razvoj reaktivnih aplikacij. Podrobneje smo razčlenili ogrodje Vert.x, ki je dogodkovno vodeno in neblokirajoče.

Glavni prispevek, ki smo ga izdelali v sklopu diplomske naloge, je razširitev komponent ogrodja KumuluzEE z razvojem razširitve, ki omogoča enostavnejšo uporabo ogrodja Vert.x. Postopek integracije razširitve je podrobneje opisan v poglavju 4. Za prikaz delovanja smo izdelali spletno klepetalnico. Namen enostavne enostranske aplikacije (*angl. single page application*) je filtriranje uporabnikovih sporočil in s tem preprečitev zlonamernih napadov, ki ciljajo na končne uporabnike.

1.4 Struktura dela

Diplomsko delo smo razdelili na šest poglavij. Poglavje 2 začnemo z opisom reaktivnega programiranja in njegovih konceptov. V nadaljevanju reaktivno

[1] https://ee.kumuluz.com/
[2] http://vertx.io/

programiranje primerjamo s splošno bolj znanim objektnim pristopom. Poglavje zaključimo s kratkim opisom in primerjavo najbolj popularnih ogrodij, ki nam olajšajo delo pri izdelavi reaktivnih aplikacij. Ta ogrodja so Vert.x, ReactiveX[3], Bacon.js[4], Reactor[5] in Akka Streams[6]. V poglavju 3 podrobneje razčlenimo ogrodje Vert.x in predstavimo njegove komponente. Postopek integracije ogrodja Vert.x z ogrodjem KumuluzEE, ki se uporablja za izdelavo mikrostoritev v poslovni izdaji jave, je podrobno predstavljen v poglavju 4. Poglavje 5 je namenjeno predstavitvi delovanja izdelane razširitve ogrodja Vert.x. Diplomsko nalogo zaključimo s poglavjem 6, kjer na kratko povzamemo celotno delo in opišemo predloge, ki jih bomo uresničili v prihodnosti.

[3]http://reactivex.io/
[4]https://baconjs.github.io/
[5]https://projectreactor.io/
[6]http://akka.io/

Poglavje 2

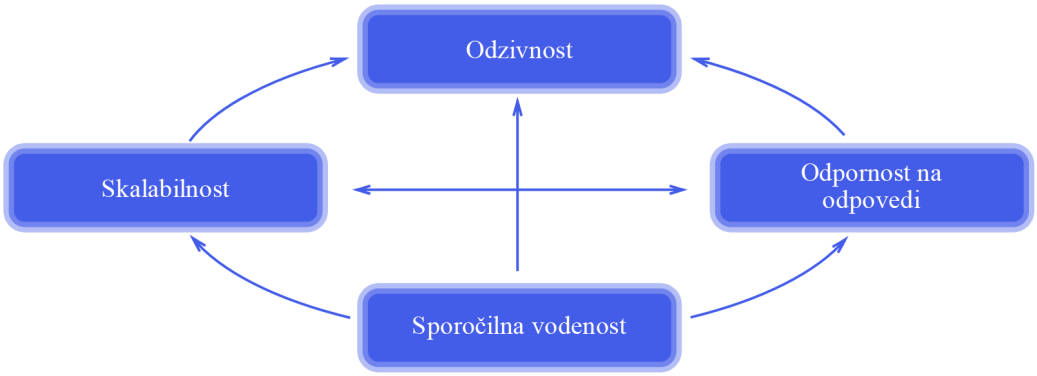
Reaktivno programiranje in razlike z objektnim pristopom

V tem poglavju predstavimo reaktivno programiranje, nato opišemo njegove koncepte ter s tem predstavimo razlike z objektnim programiranjem. Poglavje zaključimo s predstavitvijo glavnih ogrodij za reaktivno programiranje in izdelavo reaktivnih sistemov ter jih na kratko opišemo.

2.1 Reaktivno programiranje

Reaktivno programiranje [8] je na abstraktnem nivoju asinhronska programska paradigma, ki se ukvarja s podatkovnimi tokovi in propagiranjem sprememb med vse opazovalce (*angl. observers*). Uporablja se za izdelavo sistemov, ki jim pravimo reaktivni šele takrat, ko vsebujejo štiri osnovne lastnosti. Te lastnosti so po reaktivnem manifestu [15] odzivnost, odpornost na odpovedi, elastičnost ter sporočilna vodenost. Na sliki 2.1 vidimo tesno povezanost zgornjih štirih lastnosti. Sistemi, zgrajeni na podlagi reaktivnega manifesta, so bolj fleksibilni in skalabilni, kar pomeni, da jih lažje razvijamo in posodabljamo. So bistveno bolj odporni na odpovedi ter zelo odzivni.

Sistem je odziven takrat, ko se pravočasno odziva na zahteve, ko je to le možno. Odzivnost pomeni, da se morebitni problemi odkrijejo hitro in se



Slika 2.1: Povezanost štirih glavnih lastnosti reaktivnega manifesta.

še hitreje razrešijo. Sistem se mora osredotočiti na zagotavljanje hitrih in doslednih odzivnih časov ter s tem omogočiti prijetno uporabniško izkušnjo.

Druga pomembna lastnost je odpornost na odpovedi, ki jo dosežemo z replikacijo, omejitvijo, izolacijo in delegacijo. Odpovedi so omejene znotraj posamezne komponente in s tem izolirane od okoliških komponent. Naloga obnovitve vsake padle komponente je prenesena na drugo komponento. Visoka razpoložljivost je zagotovljena z replikacijo, kjer je to možno.

Elastičnost pomaga, da sistem ostane odziven med različnimi delovnimi obremenitvami. Reaktivni sistemi se odzivajo na spremembe s povečevanjem ali zmanjševanjem svojih virov. Sistem se odzove na zunanje razmere s skaliranjem navzgor ali navzdol (*angl. scale up or down*). V tem primeru bo moral dodati ali odvzeti jedra enega računalnika. Lahko pa skaliranje opravi z dodajanjem ali odvzemanjem vozlišč v podatkovnih centrih. Slednji metodi pravimo skaliranje navzven ali navznoter (*angl. scale in or out*). Elastičnost poskrbi, da je sistem ločen na manjše komponente, ki jih lahko skaliramo neodvisno eno od druge.

Reaktivni sistemi se zanašajo na asinhrono posredovanje sporočil. Uporaba eksplicitnega posredovanja sporočil omogoča nadzor nad obremenitvijo, elastičnostjo ter kontrolo pretoka sistema. Neblokirajoča komunikacija omogoča prejemnikom, da uporabljajo sredstva, medtem ko so aktivna, kar vodi

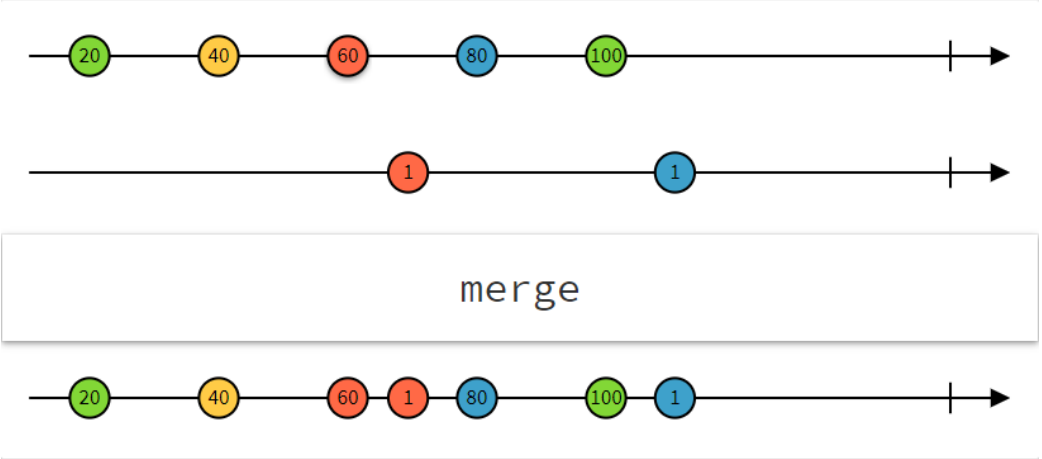
do manjše obremenitve sistema.

2.2 Koncepti reaktivnega programiranja

V reaktivnem programiranju se srečamo z različnimi koncepti [8], ki so na kratko opisani v spodnjih podpoglavjih.

2.2.1 Koncept opazovanega

Opazovani (*angl. observable*) omogoča obravnavo podatkovnih tokov asinhronih dogodkov z uporabo preprostih ali sestavljenih operatorjev, ki jih uporabljamo nad zbirkami elementov. Preprečujejo uporabo preglobokega gnezdenja povratnih klicev in so bolj odporni proti hroščem. Dogodke na podatkovnih tokovih prikažemo s pomočjo diagrama frnikol (*angl. marble diagram*). Diagram za operator *merge* je prikazan na sliki 2.2. Črta predstavlja čas, medtem ko frnikule predstavljajo dogodke, ki jih opazovani potisne na opazovalce (*angl. observers*) [23]. Vertikalna črta na časovni liniji pomeni, da se je podatkovni tok uspešno zaključil. V primeru napake mesto, kjer smo postavili znak za uspeh, označimo z znakom X.



Slika 2.2: Prikaz operatorja *merge* z diagramom frnikol.

2.2.2 Stopnja eksplicitnosti

Stopnja eksplicitnosti pravi, da se reaktivni programski jeziki gibljejo od zelo eksplicitnih, kjer so podatkovni tokovi nastavljeni z uporabo puščic, do implicitnih, kjer so podatkovni tokovi izpeljani iz jezikovnih konstruktov. Eksplicitno nastavljanje in kreiranje podatkovnih tokov zasledimo v programskem jeziku Haskell[1]. V Javi z ogrodjem Vert.x nastavljamo in kreiramo podatkovne tokove implicitno. Primer izdelave toka podatkov iz nabora nizov znakov je prikazan na delčku kode 2.1.

```
Observable<String> sentenceObservable = Observable.from(new
↪  String[] {"to", "je", "stavek"});
```

Delček kode 2.1: Izdelava podatkovnega toka v ogrodju Vert.x.

2.2.3 Statičnost ali dinamičnost

Reaktivno programiranje je lahko povsem statično, kjer so podatkovni tokovi nastavljeni tako, da so statični, ali dinamično, kjer se vsebnost podatkovnih tokov spreminja med izvajanjem programa. Ko želimo iz podatkovnega toka dobiti vedno enako vrednost, uporabimo statični podatkovni tok. Najpreprostejši statični podatkovni tok je prikazan na delčku kode 2.2. Dinamičnost podatkovnih tokov je povezana z reaktivnim programiranjem višjega reda, saj se izhodne vrednosti podatkovnega toka spreminjajo glede na vhodni podatkovni tok.

```
Observable<String> staticObservable = Observable.just("To je
↪  drugi stavek");
```

Delček kode 2.2: Izdelava statičnega podatkovnega toka v ogrodju Vert.x.

[1]https://www.haskell.org/

2.2.4 Reaktivno programiranje višjega reda

Za reaktivno programiranje lahko rečemo, da je višjega reda, če podpira idejo, da lahko podatkovni tok A uporabimo za izdelavo podatkovnega toka B s funkcijami kot so *map*, *filter*, *merge* ipd. To pomeni, da je vrednost podatkovnega toka B podatkovni tok A, nad katerim so bili izvedeni poljubni operatorji. Spremembe vhodnih vrednosti podatkovnega toka A odražajo spremembe izhodnih vrednosti podatkovnega toka B. Na delčku kode 2.3 je prikazano filtriranje podatkovnega toka A in s tem pridobitev novega podatkovnega toka B. Podatkovni tok B vsebuje vse vrednosti podatkovnega toka A, ki so večje od števila 7.

```
Observable<Integer> observableA = Observable.from(new Integer[]
↪  {6, 9, 10, 10, 9, 8, 7});
Observable<Integer> observableB = observableA.filter(x -> x >
↪  7);
```

Delček kode 2.3: Filtriranje vrednosti podatkovnega toka A v ogrodju Vert.x.

2.3 Primerjava z objektnim pristopom

Glavna korist reaktivnega programiranja [16] v primerjavi z objektnim pristopom je, da nudi večjo izkoriščenost računskih virov na večjedrnih centralno procesnih enotah in s tem poskrbi za učinkovito delovanja sistema. Reaktivno programiranje poskrbi za enostaven pristop pri pisanju asinhronih in neblokirajočih vhodno-izhodnih izračunov. Napisana koda je zaradi prej omenjene enostavnosti lepša in bolj berljiva.

Ker je izvajanje pri reaktivnem programiranju neblokirajoče, se nitim ni treba boriti za računske vire in s tem blokirati ostale, vendar lahko opravljajo druga koristna dela, medtem ko je vir zaseden. Ko se vir sprosti, nadaljujejo z izvajanjem operacij nad tem virom.

V položaju, ko podatkovni tok oddaja sporočila hitreje, kot jih opazovalec lahko sprejema, si pomagamo z metodo, imenovano protipritisk [15] (*angl. back-pressure*). Protipritisk je pomemben mehanizem za dajanje povratnih informacij, ki reaktivnemu sistemu omogoči, da se odzove na obremenitve in tako prepreči odpovedi.

Pri pisanju funkcij s povratnimi klici (*angl. callback*) se precej hitro zaplezamo in jih začnemo gnezditi. Ob zadostni globini gnezdenja nismo pokvarili le celotnega izgleda kode, temveč tudi ustvarili t. i. grozoto povratnih klicev [27] (*angl. callback hell*). Reaktivno programiranje se temu izogne, saj se osredotoči na medsebojno odvisne dogodke. Dogodki abstraktirajo potrebo po eksplicitnih povratnih klicih in gnezdenju, ki prihaja z njimi.

Pri objektnem pristopu [8] z izvajanjem ukaza `x = y + z` spremenljivka `x` dobi vrednost seštevka `y` in `z`. Kljub kasnejši spremembi `y` ali `z` ostane `x` nespremenjen. Njegova vrednost ostane enaka kot je bila pri prvi izvedbi ukaza. Pri reaktivnem programiranju vsaka sprememba spremenljivke `y` ali `z` avtomatično odraža spremembo spremenljivke `x` brez ponovnega zagona ukaza `x = y + z`. Reaktivno programiranje tako razbremeni razvijalce, da med izvajanjem programa ročno posododabljajo izhode, ko se vhodni podatki spremenijo [24].

2.3.1 Podobnost z vzorcem opazovalca

Vzorec opazovalca [7] (*angl. observer pattern*) je vzorec za načrtovanje programske opreme, pri katerem objekt vzdržuje seznam opazovalcev. Slednje avtomatično obvešča o vseh spremembah stanja običajno tako, da pokliče eno izmed njihovih metod. Asinhron podatkovni tok je v reaktivnem programiranju opazovani (*angl. observable*), na katerega se lahko opazovalec prijavi in s tem prične pridobivati ustrezne podatke. Potrebno je razlikovati med hladnimi [22] (*angl. cold observables*) ter vročimi podatkovnimi tokovi (*angl. hot observables*). Hladni podatkovni tok se začne izvajati šele, ko se opazovalec nanj prijavi. Vrednosti hladnega podatkovnega toka se ne delijo med druge opazovalce. Vroči podatkovni tok se od hladnega razlikuje po

tem, da prične z oddajanjem vrednosti še preden je kateri koli izmed opazovalcev aktiven. Po prijavi začne opazovalec s prejemanjem vrednosti, ki so trenutno v podatkovnem toku. Najbolj znan primer vročega podatkovnega toka je dogodek, ki se sproži s klikom na miško.

V objektno orientiranem programiranju predstavljeni vzorec deluje na celih objektih ter je namenjen poročanju stanja prijavljenim opazovalcem [18]. V reaktivnem programiranju se s spremembo vrednosti spremenljivke spremenijo tudi vse ostale vrednosti, ki so bile odvisne od nje. Stopnja granulacije doseže tudi primitivne tipe.

2.3.2 Primerjava po ključnih kriterijih

Kot smo prikazali v tabeli 2.1 stopnja granulacije reaktivnega programiranja doseže tudi primitivne tipe, medtem ko v objektnem programiranju upravljamo z objekti. Ob kakršni koli spremembi spremenljivke v reaktivnem programiranju se spremenijo vrednosti vseh spremenljivk, ki so bile odvisne od spremenjene spremenljivke. Reaktivno programiranje nam omogoči neblokirajoče izvajanje kode, kar poveča izkoriščenost računskih virov. Pri reaktivnem programiranju gnezdenje povratnih klicev nadomestimo s prijavo na dogodke in izboljšamo strukturo napisane kode. V objektnem programiranju z uporabo vzorca opazovalca objekt direktno posredujemo opazovalcu. Reaktivno programiranje nam nudi možnost transformacije posameznih dogodkov podatkovnega toka z uporabo raznolikih operatorjev.

kriterij	reaktivno programiranje	objektno programiranje
delovanje	nad primitivni tipi	nad objekti
spremembe	odvisne spremenljivke	brez sprememb
blokirajoče	kadar je nujno	da
povratni klici	uporaba dogodkov	gnezdenje
izkoriščenost	večja	-
transformacije	na posamezni dogodek	-
koda	bolj strukturirana	-

Tabela 2.1: Primerjava reaktivnega programiranja z objektnim po ključnih kriterijih.

2.4 Ogrodja za reaktivno programiranje

Obstaja veliko ogrodij, napisanih za različne jezike, ki nudijo podporo pri implementaciji reaktivnega programiranja. Trenutno so najbolj popularna ogrodja Vert.x, ReactiveX, Bacon.js, Reactor in Akka Streams. V tem poglavju našteta ogrodja na kratko opišemo in jih na koncu primerjamo po ključnih lastnostih.

2.4.1 Ogrodje Vert.x

Ogrodje Vert.x je odprtokodno, dogodkovno vodeno in neblokirajoče, kar pomeni, da omogoča izdelavo aplikacij z velikim številom vzporednosti pri uporabi majhnega števila procesnih jeder. Ogrodje je poliglot, saj ga lahko uporabimo v različnih jezikih. Jeziki, ki so podprti, so Java, JavaScript, Groovy itd. Ključna lastnost ogrodja Vert.x je možnost uporabe distribuiranega dogodkovnega vodila, ki lahko prodre celo v brskalnik. Na delčku kode 2.4 je prikazan reaktivni pristop pri postavitvi vertikla z ogrodjem Vert.x. Ogrodje Vert.x temelji na uporabi vzorca reaktorja in dogodkovne zanke, kar je podrobneje opisano v poglavju 3.

```
Observable<String> deploymentObservable =
↪  RxHelper.deployVerticle(vertx, new MojVertikel());
deploymentObservable.subscribe(id -> {
        // Vertikel uspesno postavljen
}, napaka -> {
        // Napaka pri postavitvi
});
```

Delček kode 2.4: Primer reaktivne postavitve vertikla v ogrodju Vert.x.

2.4.2 Ogrodje ReactiveX

Ogrodje ReactiveX [10] je odprtokodna knjižnica za izdelavo asinhronih programov ter programov, ki temeljijo na dogodkih. Podpira zaporedja podatkovnih tokov in dogodkov. Razširja vzorec opazovalca in ponuja operatorje, ki omogočajo opravljanje in sestavljanje novih zaporedij. Ogrodje ReactiveX lahko uporabimo v različnih jezikih, zato mu pravimo, da je poliglot. Ponuja uporabo reaktivnega programiranja v jezikih, kot so Java, JavaScript, .NET, Scala, Clojure, Swift in drugih. Glavna gradnika ogrodja ReactiveX sta opazovani (*angl. observable*) in naročnik (*angl. subscriber*). Opazovani predstavljajo vire podatkov, na katere se lahko prijavi poljubno mnogo naročnikov. Ko opazovani odda podatek, se pokliče metoda `onNext()` pri vsakemu izmed prijavljenih naročnikov. Na delčku kode 2.5 je razvidna kreacija opazovanega. V naslednji vrstici se na podatkovni tok prijavi naročnik in izpiše podatek ob uspešnem prejemu. Skupnost, ki stoji za ogrodjem, je poskrbela, da je začetek sila enostaven, saj je na voljo obilo dokumentacije.

```
Observable<String> observable = Observable.just("Pozdravljen
↪  svet!");
observable.subscribe(podatek -> {
        System.out.println(podatek);
});
```

Delček kode 2.5: Primer opazovanega in naročnika v ogrodju ReactiveX.

2.4.3 Ogrodje Bacon.js

Ogrodje Bacon.js [2] je odprtokodna knjižnica za funkcionalno reaktivno programiranje. Omogoča reaktivno programiranje samo v jeziku JavaScript. Bacon.js deluje tako, da se t. i. Property in t. i. EventStream povežeta s svojimi viri, če imata vsaj enega poslušalca. Prav tako samodejno prekineta povezavo z viri, če se odjavijo vsi poslušalci. Zato je s tega vidika naročanje na podatkovni tok zelo pomembno. Ogrodje ponuja uporabo dogodkovnega vodila, za katerega jamčijo, da dostavi vse dogodke brez napak (*angl. glitch-free*). Na delčku kode 2.6 je prikazan zajem dogodkov vnosnega polja. Naprej pretvorimo dogodek, ki se odda ob spustu tipke v podatkovni tok. Nato z operatorjem *map* izdelamo drug podatkovni tok, ki vrača vrednosti vnosnega polja. Na koncu pretvorimo podatkovni tok v lastnost (*angl. property*), ki ji nastavimo prazen niz kot začetno vrednost.

```
let username = $("#username
↪  input").asEventStream("keyup").map(function(event) { return
↪  $(event.target).val() }).toProperty("")
```

Delček kode 2.6: Primer zajema dogodkov vnosnega polja v ogrodju Bacon.js.

2.4.4 Ogrodje Reactor

Ogrodje Reactor [11] je knjižnica za izdelavo neblokirajočih aplikacij na javanskem navideznem stroju (JVM), ki temelji na specifikaciji reaktivnih tokov. Ogrodje je v neposrednem stiku z Java 8 funkcionalnim aplikacijskim programskim vmesnikom (API). Namen reaktivnih tokov [9] je zagotoviti standard za asinhrono obdelavo asinhronega podatkovnega toka z neblokirajočim protipritiskom. Ogrodje Reactor je sestavljeno iz različnih modulov. Nekateri izmed teh so Reactor Core, Reactor Test, Reactor Netty ipd. Ogrodje ponuja dva reaktivna sestavljiva aplikacijska programska vmesnika, to sta `Flux` in `Mono`, ki omogočata izdelavo in transformacijo podatkovnih tokov. `Flux` omogoča oddajo nič ali več elementov, medtem ko `Mono` ponuja oddajo največ enega elementa. Na delčku kode 2.7 izdelamo podatkovni tok, ki oddaja cela števila. Z operatorjem *map* vsako število povečamo. Operator *filter* omogoči fitriranje vhodnih vrednosti.

```
Flux.range(3,5).map(x -> i + 3).filter(x -> x > 7);
```

Delček kode 2.7: Primer uporabe operatorjev nad podatkovnim tokom v ogrodju Reactor.

2.4.5 Ogrodje Akka Streams

Ogrodje Akka Streams [28] je prav tako odprtokodno. Ponuja realizacijo reaktivnega programiranja v dveh jezikih, in sicer v Javi in v Scali. Ogrodje Akka Streams temelji na reaktivnih tokovih tako kot ogrodje Reactor. V ogrodju predstavijo obdelavo podatkov v obliki podatkovnega toka skozi poljuben kompleksen graf, kjer so vozlišča stopnje obdelave. Stopnje imajo nič ali več vhodov in nič ali več izhodov. Najpogostejši gradniki pri uporabi ogrodja Akka Streams so `Source`, `Sink` in `Flow`. `Source` ima natanko en izhodni podatkovni tok. `Sink` omogoča le en vhodni podatkovni tok. `Flow` je nekakšna kombinacija obeh in omogoča natanko en vhodni in izhodni podat-

kovni tok. Na delčku kode 2.8 je prikazana uporaba gradnika `Source`. Slednji sprejme dva parametra. Prvi nam pove tip elementa, ki ga bo `Source` oddajal. Drugi lahko signalizira, da zagon vira proizvede pomožne vrednosti. V primeru je nastavljen na `NotUsed`, saj podatkovni tok samo transformiramo.

```
Source<Integer, NotUsed> source = Source.range(1, 100).map(x ->
↪ {
        if((x % 2) == 0) return x;
        return -x;
});
```

Delček kode 2.8: Primer uporabe gradnika `Source` z ogrodjem Akka Streams.

2.4.6 Primerjava ogrodij

V tabeli 2.2 so prikazane podobnosti in razlike zgoraj opisanih ogrodij. Vsa opisana ogrodja so odprtokodna. Reaktivno programiranje v dveh ali več jezikih omogočajo ogrodja Vert.x, ReactiveX in Akka Streams. Ogrodje Vert.x implementira vzorec reaktorja. Ogrodji ReactiveX ter Bacon.js implementirata vzorec opazovalca. Ogrodji Reactor in Akka Streams implementirata specifikacijo reaktivnih tokov. Edino ogrodje, ki ne implementira protipritiska, je ogrodje Bacon.js. Dogodkovno vodilo lahko uporabimo v ogrodju Vert.x in ogrodju Bacon.js.

kriterij	**Vert.x**	**ReactiveX**	**Bacon.js**	**Reactor**	**Akka Streams**
poliglot	da	da	ne	ne	da
dogodkovno vodilo	da	ne	da	ne	ne
vzorec	reaktor	opazovalec	opazovalec	reaktivni tok	reaktivni tok
odprtokodno	da	da	da	da	da
protipritisk	da	da	ne	da	da

Tabela 2.2: Primerjava ogrodij Vert.x, ReactiveX, Bacon.js, Reactor in Akka Streams.

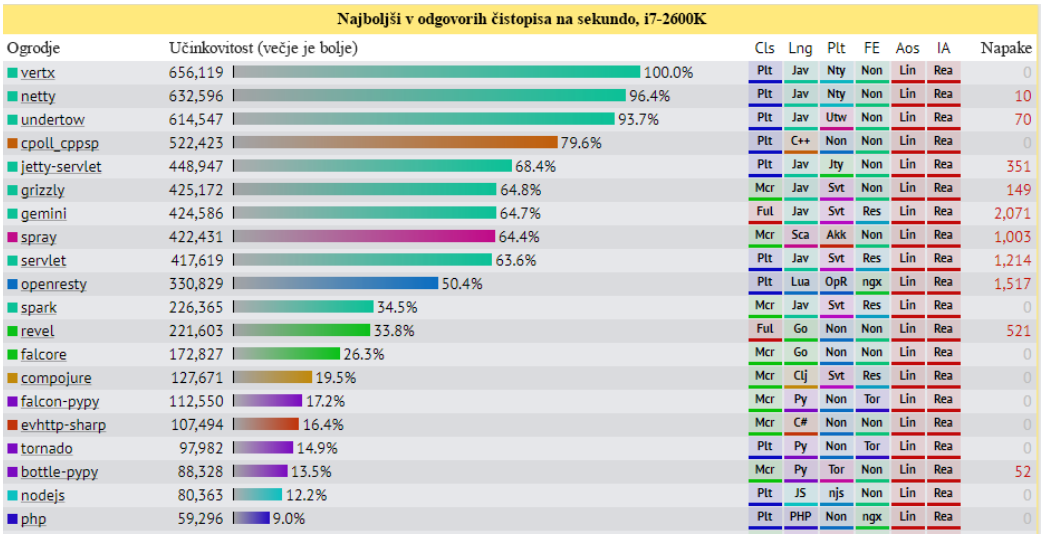
Poglavje 3

Ogrodje Vert.x

Vert.x [12] je ogrodje za izdelavo reaktivnih aplikacij na javanskem virtualnem stroju (JVM). Ogrodje je dogodkovno vodeno in neblokirajoče, kar pomeni, da omogoča aplikaciji obvladovanje velikega števila vzporednosti z uporabo majhnega števila procesnih jeder. Vert.x omogoča, da je aplikacija skalabilna z uporabo minimalne strojne opreme. Ogrodje je poliglot. Vključuje podporo za jezike Java, JavaScript, Groovy, Ruby, Ceylon, Scala in Kotlin. Ponuja možnost uporabe distribuiranega dogodkovnega vodila, ki se razteza ne samo na stran strežnika, temveč tudi na stran odjemalca. Dogodkovno vodilo lahko predre celo v brskalnik in tako omogoči izdelavo t. i. aplikacij s takojšnjim odzivom (*angl. real-time applications*). Ogrodje Vert.x je izjemno hitro. Na sliki 3.1 je razvidno, da se je uvrstilo na prvo mesto. Ogrodje je moralo na zahtevke odgovarjati z enostavnim sporočilom „Hello, World“, generiranim v čistopisu (*angl. plain text*) [14].

3.1 Zgodovina

Ogrodje Vert.x [13] je Tim Fox, takratni uslužbenec pri VMware, začel razvijati leta 2011. Najprej je svoj projekt poimenoval Node.x, pri čemer je `x` pomenil, da bo projekt v prihodnosti poliglot. Kasneje je bil preimenovan v Vert.x, da bi se izognil morebitnim pravnim težavam. Sredi leta 2012 je



Najboljši v odgovorih čistopisa na sekundo, i7-2600K

Ogrodje	Učinkovitost (večje je bolje)	Cls	Lng	Plt	FE	Aos	IA	Napake
vertx	656,119 100.0%	Plt	Jav	Nty	Non	Lin	Rea	0
netty	632,596 96.4%	Plt	Jav	Nty	Non	Lin	Rea	10
undertow	614,547 93.7%	Plt	Jav	Utw	Non	Lin	Rea	70
cpoll_cppsp	522,423 79.6%	Plt	C++	Non	Non	Lin	Rea	0
jetty-servlet	448,947 68.4%	Plt	Jav	Jty	Non	Lin	Rea	351
grizzly	425,172 64.8%	Mcr	Jav	Svt	Non	Lin	Rea	149
gemini	424,586 64.7%	Ful	Jav	Svt	Res	Lin	Rea	2,071
spray	422,431 64.4%	Mcr	Sca	Akk	Non	Lin	Rea	1,003
servlet	417,619 63.6%	Plt	Jav	Svt	Res	Lin	Rea	1,214
openresty	330,829 50.4%	Plt	Lua	OpR	ngx	Lin	Rea	1,517
spark	226,365 34.5%	Mcr	Jav	Svt	Res	Lin	Rea	0
revel	221,603 33.8%	Ful	Go	Non	Non	Lin	Rea	521
falcore	172,827 26.3%	Mcr	Go	Non	Non	Lin	Rea	0
compojure	127,671 19.5%	Mcr	Clj	Svt	Res	Lin	Rea	0
falcon-pypy	112,550 17.2%	Mcr	Py	Non	Tor	Lin	Rea	0
evhttp-sharp	107,494 16.4%	Mcr	C#	Non	Non	Lin	Rea	0
tornado	97,982 14.9%	Plt	Py	Non	Tor	Lin	Rea	0
bottle-pypy	88,328 13.5%	Mcr	Py	Tor	Non	Lin	Rea	52
nodejs	80,363 12.2%	Plt	JS	njs	Non	Lin	Rea	0
php	59,296 9.0%	Plt	PHP	Non	ngx	Lin	Rea	0

Slika 3.1: Hitrosti pri pošiljanju odgovorov, sestavljenih iz čistopisa.

bila izdana prva verzija ogrodja Vert.x. V letu 2013 je prišel Vert.x pod okrilje korporacije Eclipse Foundation [1], katere projekti so osredotočeni na izgradnjo in razvoj odprtokodnih rešitev. Eclipse Foundation je neprofitna organizacija, ki pomaga pri obvladovanju odprtokodnih rešitev skupnosti. V letu 2014 je ogrodje Vert.x prejelo nagrado za najbolj inovativno tehnologijo, napisano v Javi (*angl. Most Innovative Java Technology*) [17].

3.2 Delovanje

Vert.x [12] deluje tako, da prenese dogodke na ustrezne obdelovalce (*angl. handlers*), ko so ti na voljo. Vert.x najpogosteje pokliče obdelovalce z uporabo niti, ki se imenuje dogodkovna zanka (*angl. event loop*). Ker je Vert.x neblokirajoč, lahko dogodkovna zanka v kratkem času dostavi ogromne količine dogodkov. Temu pravimo vzorec reaktorja. V standardni implementaciji vzorca obstaja samo ena dogodkovna zanka, ki skrbi za dostavo dogodkov na ustrezne obdelovalce. Težava take implementacije je, da lahko teče hkrati samo na enem izmed računalniških jeder. Implementacija ogrodja Vert.x se

razlikuje v tem, da vsaka instanca Vert.x ohranja več dogodkovnih zank. To število je nastavljivo, vendar ima privzeto vrednost enako dvakratnemu številu jeder, ki so na voljo. Takšni implementaciji pravimo multi-reaktorski vzorec (*angl. multi-reactor pattern*).

3.3 Vertikli

Vertikli (*angl. Verticles*) so delčki kode, ki jih Vert.x postavi (*angl. deploy*) in zažene. Vertikel je lahko napisan v katerem koli jeziku, ki ga podpira ogrodje Vert.x. Aplikacija lahko vsebuje tudi več vertiklov, ki so napisani v različnih jezikih. Vertikel si lahko predstavljamo kot igralca (*angl. actor*) v modelu igralca (*angl. actor model*). V modelu igralca [21] se igralec obravnava kot primitiv sočasnih izračunov. Vsak igralec ohranja stanje in lahko komunicira z drugimi s pošiljanjem sporočil. Sporočila se shranjujejo v poštni nabiralnik ter so ena za drugo obdelana s strani igralca. Po končani obdelavi sporočila lahko igralec spremeni svoje stanje, opravi nadaljnje izračune, pošlje sporočilo drugemu igralcu ali celo kreira novega igralca. Vertikli se med sabo sporazumevajo s pošiljanjem sporočil preko dogodkovnega vodila. Vert.x razdeli vertikle na tri tipe.

Standardni vertikel (*angl. standard verticle*) je najbolj pogost in uporaben tip. Vsakemu vertiklu se ob nastanku dodeli dogodkovna zanka, ki pokliče metodo `start` tega vertikla. Pri izvajanju drugih metod, ki sprejmejo obdelovalca, je zagotovljeno, da se obdelovalci izvedejo na isti dogodkovni zanki. Vert.x s tem poskrbi, da razvijalci lahko pišemo enonitno kodo (*angl. single threaded*) ter ogrodju Vert.x prepustimo skaliranje in nitnost (*angl. threading*).

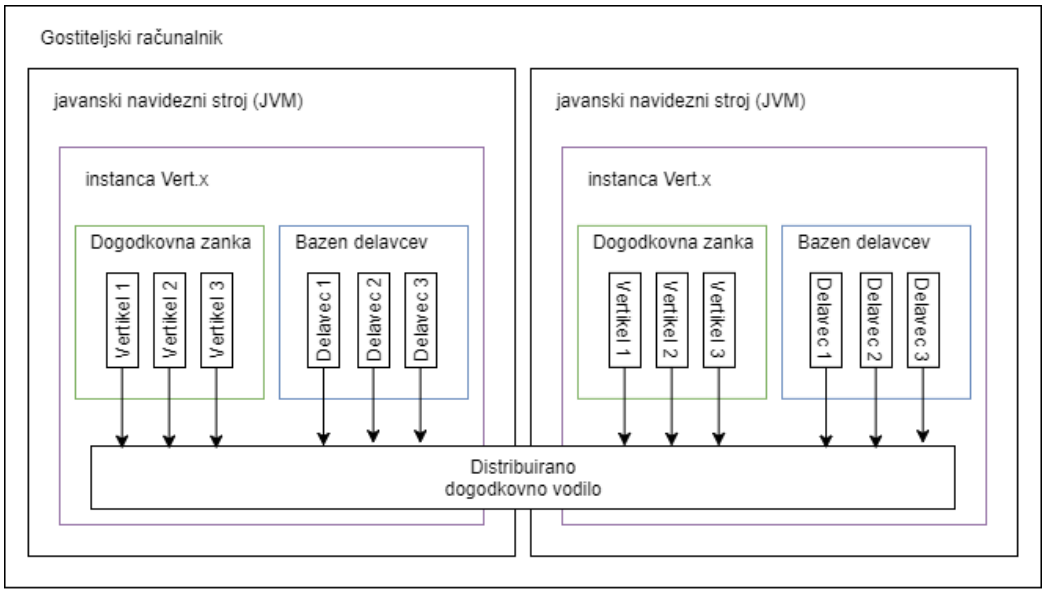
Vertikel delavec (*angl. worker verticle*) je enak kot standardni vertikel s to razliko, da se ne izvede s pomočjo dogodkovne zanke, ampak z uporabo niti iz bazena niti (*angl. worker thread pool*). Delavci so namenjeni izvajanju blokirajoče kode.

Večnitni vertikel delavec (*angl. multi-threaded worker verticle*) se razli-

kuje od navadnega delavca v tem, da se ga lahko izvede hkrati z različnimi nitmi.

3.4 Arhitektura

Tipična arhitektura aplikacije, zgrajene z ogrodjem Vert.x, je prikazana na sliki 3.2. Enaka arhitektura velja tudi za našo implementacijo, katere delovanje bomo prikazali v poglavju 5. Iz slike razločimo, da smo pognali dve instanci Vert.x, ki tečeta vsaka v svojem javanskem navideznem stroju na gostiteljevem računalniku. Znotraj vsake instance Vert.x se nahaja dogodkovna zanka, ki skrbi za prenašanje dogodkov na ustrezne obdelovalce. Izvajanje blokirajoče kode nam omogoči vertikel delavec, ki ga pridobimo iz bazena delavcev. Levo instanco Vert.x smo z desno povezali z uporabo gručenja. Gručenje omogoči, da se instanci povežeta in tvorita distribuirano dogodkovno vodilo, preko katerega poteka komunikacija med instancama.



Slika 3.2: Prikaz arhitekture aplikacije, izdelane z ogrodjem Vert.x.

3.5 Komponente

Ogrodje Vert.x ponuja ogromno zbirko komponent, ki razvijalcem olajšajo pisanje poljubnih aplikacij. Ogrodje je zelo modularno, kar pomeni, da nam omogoči uporabo samo tistih komponent, ki jih potrebujemo glede na naravo projekta. Komponente so razdeljene na Core, Web, Web Client, sklop za dostop do podatkov, sklop za integracijo, sklop za mostove dogodkovnega vodila, sklop za avtentikacijo in avtorizacijo, sklop za reaktivnost, sklop za mikrostoritve, sklop za razvijalce, testiranje, sklop za gručenje, sklop za storitve, sklop za oblak in sklop za interno uporabo. V nadaljevanju na kratko opišemo glavne komponente oziroma sklope komponent.

V središču ogrodja Vert.x je niz aplikacijskih programskih vmesnikov, ki jih imenujemo Vert.x Core. Funkcionalnosti komponente Core so precej nizko nivojske. Omogočajo funkcionalnosti, kot so pisanje odjemalcev in strežnikov TCP, pisanje odjemalcev in strežnikov HTTP, vključno s podporo za tehnologijo WebSocket, dogodkovno vodilo idr.

Komponenta Web ponuja niz gradnikov za gradnjo modernih in skalabilnih spletnih aplikacij z uporabo ogrodja Vert.x. Komponento uporabimo za izdelavo klasičnih zalednih aplikacij, aplikacij REST, spletnih aplikacij, ki se izvajajo v realnem času, idr. Ponuja funkcionalnosti, kot so usmerjanje (*angl. routing*), ekstrakcija parametrov poti, limitiranje velikosti telesa, manipuliranje s piškotki idr. V primeru, ko želimo uporabljati naprednejše funkcionalnosti odjemalca HTTP, uporabimo komponento Web Client, ki nam omogoča kodiranje in dekodiranje telesa JSON, oddajo obrazcev, enotno obvladovanje napak, uporabo HTTP/2 idr.

Sklop komponent za dostop do podatkov zagotavlja asinhrone odjemalce za dostopanje do raznolikih podatkovnih baz. Vert.x odjemalci nam pomagajo pri dostopu do MongoDB, JDBC, SQL, Redis in MySQL.

Sklop komponent za integracijo zagotavlja pošiljanje e-pošte, implementacijo STOMP protokola, adapter JCA, odjemalca za sporočilno vrsto RabbitMQ, odjemalca za Apache Kafka, odjemalca za Consul ter most za interakcijo z AMQP.

Za varnost je z ogrodjem Vert.x poskrbljeno, saj je sklop komponent za avtentikacijo in avtorizacijo eden največjih. Avtentikacija preveri identiteto uporabnika, medtem ko avtorizacija poskrbi za preverbo uporabnikovih pooblastil. Sklop ponuja aplikacijske programske vmesnike za JDBC, JWT, Shiro, MongoDB, OAuth 2 ter .htdigest.

Vert.x ponuja različne mostove za razširitev tako dogodkovnega vodila kot tudi komponente za odkrivanje storitev (*angl. service discovery*), ki igra pomembno vlogo pri izdelavi mikrostoritev.

V primeru želje po izdelavi še bolj reaktivnih aplikacij nam Vert.x omogoči sklop reaktivnih komponent. Ponuja podporo za RxJava, reaktivne tokove in uporabo Vert.x Sync, ki omogoča zagon vertiklov s pomočjo vlaken (*angl. fibers*). Vlakna so lahke niti, ki jih je mogoče blokirati, ne da bi blokirali nit jedra.

Poglavje 4

Integracija ogrodja Vert.x z Javo in mikrostoritvami

V tem poglavju predstavimo glavni prispevek diplomske naloge. Podrobno opišemo postopek izdelave razširitve ogrodja Vert.x in njegove integracije v ogrodje KumuluzEE, ki se uporablja za izdelavo mikrostoritev. Za boljše razumevanje namena izdelave razširitve na kratko opišemo arhitekturni slog mikrostoritev in ogrodje KumuluzEE.

4.1 Mikrostoritve

Arhitekturni slog mikrostoritev je pristop, v katerem razvijemo aplikacijo kot zbirko manjših storitev, imenovanih mikrostoritve [20]. Vsaka mikrostoritev se izvaja ločeno, v svojem procesu, in je zadolžena za opravljanje točno določenega dela, ki ga opravlja po najboljših močeh. Najpogosteje je naloga mikrostoritev komuniciranje z aplikacijskimi programskimi vmesniki (API) za pridobivanje virov HTTP. Vsaka mikrostoritev je neodvisna celota, ki je ponavadi zagnana s pomočjo avtomatiziranih zagonskih mehanizmov. Upravljanje mikrostoritev je skoraj popolnoma decentralizirano. Pišemo jih lahko v različnih jezikih in uporabimo specifične prednosti izbranega jezika.

Premik k izdelavi mikrostoritev najlažje razumemo, če ga primerjamo z

arhitekturo monolitne aplikacije. Monolitna aplikacija najpogosteje vsebuje poslovno logiko celotne aplikacije, povezavo do podatkovne baze ter uporabniški vmesnik, ki ga aplikacija prikaže končnemu uporabniku. Celoten monolit se zapakira v eno izvršljivo datoteko, ki je pripravljena za postavitev (*angl. deployment*). Ena sama sprememba uporabniškega vmesnika ali najmanjši dodatek kode nas prisili, da celotno aplikacijo ponovno zapakiramo in postavimo. Problem kodiranja monolitnih aplikacij, ki se pojavi na dolgi rok, je nezmožnost učinkovite skalabilnosti. V primeru, ko je določen modul aplikacije preobremenjen, moramo postaviti še eno instanco celotne aplikacije, da preprečimo kakršne koli odpovedi sistema. S takim ravnanjem porabljamo veliko več virov kot bi jih, če bi postavili samo novo instanco preobremenjenega modula. To sta samo dve glavni težavi, ki sta pripeljali do uporabe arhitekturnega sloga mikrostoritev.

Mikrostoritev je neodvisna celota, ki se jo lahko postavi, skalira in testira neodvisno [26]. Osredotočena je na izvajanje določene naloge, ki jo opravlja po najboljših močeh.

4.2 Ogrodje KumuluzEE

Ogrodje KumuluzEE je lahko (*angl. lightweight*) odprtokodno ogrodje za razvijanje mikrostoritev z uporabo standardnih Java EE tehnologij. Poleg razvijanja omogoča tudi preureditev obstoječih Java EE aplikacij v mikrostoritve [4]. KumuluzEE zapakira mikrostoritve kot samostojne datoteke JAR, ki se jih lahko zažene kjer koli. Ogrodje za zagon mikrostoritev potrebuje le namestitev programske opreme JRE. Mikrostoritve, zapakirane z ogrodjem KumuluzEE, so lahke in optimizirane glede na velikost in začetni zagonski čas. Ogrodje je sestavljeno iz več različnih komponent, ki jih lahko selektivno uporabimo v mikrostoritvah [19]. To nam omogoči, da uporabimo samo tiste odvisnosti, ki jih resnično potrebujemo. S tem zmanjšamo velikost končne datoteke JAR in onemogočimo neuporabljenim odvisnostim zasedanje pomnilniškega prostora. KumuluzEE omogoča enostavno konfigu-

racijo mikrostoritev, saj je vsa konfiguracija na voljo znotraj konfiguracijske datoteke `pom.xml`.

4.3 Integracija in delovanje

Eden izmed ciljev diplomske naloge je omogočiti razvijalcem čim lažji začetek pri implementaciji in uporabi ogrodja Vert.x znotraj mikrostoritev. Ker ogrodje KumuluzEE omogoča enostavno izgradnjo poljubnih razširitev, smo se odločili povečati nabor razširitev ogrodja KumuluzEE in razviti svojo, ki omogoča zagon in enostavno uporabo določenih značilnosti ogrodja Vert.x.

Integracija vsebuje dve anotaciji, ki poskrbita za enostavno izbiro funkcij ogrodja Vert.x in predvsem polepšata končno kodo. Anotacije so v Javi oblika metapodatkov, ki zagotavljajo podatke o programu in niso del samega programa [6].

4.3.1 Izdelava anotacije `@VertxEventPublisher`

Anotacija `@VertxEventPublisher` sprejme kot vhodni parameter naslov in zagotovi, da se bo preko tovarniških razredov (*angl. factory classes*) ustvaril nov oziroma vrnil že obstoječ ustvarjalec sporočil (*angl. message producer*). Izdelava novega ustvarjalca sporočil je prikazana na delčku kode 4.1. Najprej s pomočjo razreda *VertxConfigurationUtils* pridobimo referenco na dogodkovno vodilo in se povežemo na podatkovni tok z naslovom, ki smo ga prejeli preko zgornje anotacije. Ob uspešni povezavi na podatkovni tok lahko nemudoma pričnemo z oddajanjem sporočil. Iz delčka kode 4.2 je razvidno, da ustavimo novega ustvarjalca sporočil le v primeru, ko na naslov ni prijavljen še nobeden ustvarjalec. V primeru že prijavljenega ustvarjalca sporočil na dani naslov, ga preprosto vrnemo.

```
@Override
public MessageProducer<Object> createPublisher(String address)
{
        MessageProducer<Object> messageProducer = null;
        EventBus eventBus =
        ↪  VertxConfigurationUtils.getInstance()
        ↪  .getVertx().eventBus();
        messageProducer = eventBus.publisher(address);
        return messageProducer;
}
```

Delček kode 4.1: Metoda tovarniškega razreda, ki vrne novega ustvarjalca sporočil.

```
@Produces
@VertxEventPublisher
public MessageProducer<Object> getPublisher(InjectionPoint
↪  injectionPoint)
{
        String address = injectionPoint.getAnnotated()
        ↪  .getAnnotation(VertxEventPublisher.class)
        ↪  .address();
        if(producers.containsKey(address)) {
                return producers.get(address);
        } else {
                MessageProducer<Object> messageProducer =
                ↪  vertxPublisherFactory
                ↪  .createPublisher(address);
                producers.put(address, messageProducer);
                return messageProducer;
        }
}
```

Delček kode 4.2: Izdelovalec ustvarjalcev sporočil.

Anotacijo za pridobitev ustvarjalca sporočil lahko uporabimo samo na polju (*angl. field*). Na delčku kode 4.3 je razvidno, da želimo pridobiti oziroma referencirati ustvarjalca sporočil, ki bo sporočila pošiljal na naslov „tacos“.

```
@Inject
@VertxEventPublisher(address = "tacos")
MessageProducer<Object> tacos;
```

Delček kode 4.3: Prikaz uporabe anotacije `@VertxEventPublisher`.

4.3.2 Izdelava anotacije `@VertxEventListener`

Anotacija `@VertxEventListener` ima prav tako en vhodni parameter. Za pravilno delovanje anotacije smo morali izdelati nov razred, ki razširja `javax.enterprise.inject.spi.Extension`. Naloga razreda je, da ob zagonu ogrodja KumuluzEE pregleda vsa zrna in shrani tista, ki imajo metode označene z zgornjo anotacijo. Po postavitvi ogrodja KumuluzEE iteriramo skozi vse reference na shranjena zrna in za vsakega ustvarimo novega izvršitelja (*angl. runnable*). Naloga izvršitelja je prikazana na delčku kode 4.4. Najprej pridobimo referenco na dogodkovno vodilo in se prijavimo na podatkovni tok z danim naslovom. Po prijavi pričnemo s poslušanjem in ob prejemu novega sporočila pokličemo metodo, nad katero smo uporabili anotacijo.

```
messageConsumer =
↪  vertxUtils.getVertx().eventBus().consumer(address);
messageConsumer.handler(message -> {
        if (message.body() != null) {
                try {
                        method.invoke(instance, message);
                } catch (...) {
                        // napaka
                }
        }
});
```

Delček kode 4.4: Prikaz izdelavave potrošnika sporočil preko izvršitelja.

Anotacijo lahko postavimo le na metode, ki imajo en vhodni parameter tipa `Message<Object>`. Na delčku kode 4.5 prijavimo metodo `onMessage()` na podatkovni tok z naslovom „tacos“. Vsako sporočilo ob prejemu zabeležimo z objektom za beleženje sporočil (*angl. logger*). V naslednji vrstici

pošljemo pošiljatelju nazaj prejeto sporočilo.

```
@VertxEventListener(address = "tacos")
public void onMessage(Message<Object> event) {
        log.info("Prejeto sporocilo: " + event.body());
        event.reply(event.body());
}
```

Delček kode 4.5: Prikaz uporabe anotacije `@VertxEventListener`.

4.3.3 Napredne možnosti uporabe

V primeru, da nam anotacije niso dovolj, je na voljo razred `VertxConfigurationUtils.java`. Ta je edinec (*angl. singleton*), ki ponuja dodatne možnosti za upravljanje z ogrodjem Vert.x. Iz poglavja 3 vemo, da ogrodje Vert.x ponuja dve možnosti zagona. Izbiramo lahko med zagonom v normalnem ali v gručastem načinu. Izbiranje načina in spreminjanje ostalih lastnosti ogrodja Vert.x smo razvijalcem zapakirali v konfiguracijsko datoteko *config.yml*. Vsebnost datoteke vidimo na delčku kode 4.6. Naštete lastnosti so poljubno nastavljive, ki se med zagonom razširitve preberejo s pomočjo pomožnih konfiguracijskih razredov ogrodja KumuluzEE. Na delčku kode 4.7 je razvidna koda, potrebna za pridobitev lastnosti „clustered“.

4.3.4 Končna struktura projekta

Končna struktura projekta vsebuje dva modula. V modulu `common` se nahajajo anotacije in ostali vmesniki (*angl. interface*). Odvisnosti, uporabljene v modulu `common`, so prikazane na delčku kode 4.8. Modul `vertx` vsebuje implementacijo razširitve ogrodja Vert.x. Na delčku kode 4.9 so prikazane uporabljene odvisnosti znotraj modula `vertx`. Končna struktura, kot je prikazana v orodju Eclipse, je razvidna na sliki 4.1.

```
kumuluzee:
  reactive:
    vertx:
      blocked-thread-check-interval: 1000
      event-loop-pool-size: 16
      file-caching-enabled: true
      ha-enabled: false
      ha-group: __DEFAULT__
      internal-blocking-pool-size: 20
      max-event-loop-execute-time: 2000000000
      max-worker-execute-time: 60000000000
      quorum-size: 1
      worker-pool-size: 20
      clustered: true
      cluster-host: localhost
      cluster-port: 0
      cluster-ping-interval: 20000
      cluster-ping-reply-interval: 20000
      cluster-public-port: null
      cluster-public-port: -1
```

Delček kode 4.6: Konfiguracijska datoteka *config.yml* z vsemi nastavljivimi lastnostmi.

```
ConfigurationUtil configurationUtil =
↪  ConfigurationUtil.getInstance();
boolean clustered = configurationUtil
↪  .getBoolean("kumuluzee.reactive.vertx.clustered")
↪  .orElse(VertxOptions.DEFAULT_CLUSTERED);
```

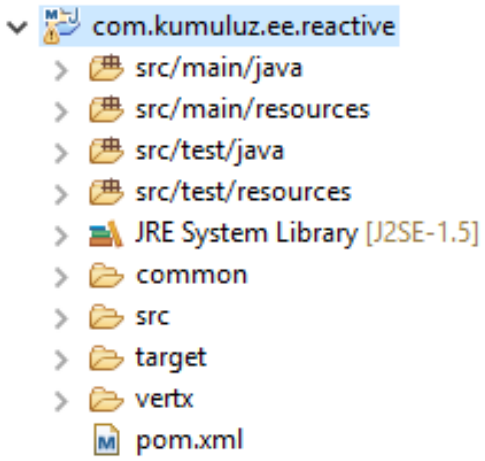
Delček kode 4.7: Prikaz kode za pridobitev lastnosti „clustered“.

```
<dependency>
        <groupId>com.kumuluz.ee</groupId>
        <artifactId>kumuluzee-cdi-weld</artifactId>
        <scope>provided</scope>
</dependency>
```

Delček kode 4.8: Odvisnosti znotraj konfiguracijske datoteke *pom.xml* modula `common`.

```
<dependency>
        <groupId>io.vertx</groupId>
        <artifactId>vertx-web</artifactId>
        <version>${vertx.version}</version>
</dependency>
<dependency>
        <groupId>io.vertx</groupId>
        <artifactId>vertx-hazelcast</artifactId>
        <version>${vertx.version}</version>
</dependency>
<dependency>
        <groupId>com.kumuluz.ee.reactive</groupId>
        <artifactId>kumuluzee-reactive-common</artifactId>
</dependency>
<dependency>
        <groupId>com.kumuluz.ee</groupId>
    <artifactId>kumuluzee-cdi-weld</artifactId>
         <scope>provided</scope>
</dependency>
```

Delček kode 4.9: Odvisnosti znotraj konfiguracijske datoteke *pom.xml* modula `vertx`.



Slika 4.1: Prikaz končne strukture razširitve Vert.x.

Poglavje 5

Prikaz delovanja

Za prikaz delovanja smo implementirali razvito razširitev ogrodja Vert.x in izdelali enostavno klepetalnico, kjer si uporabniki izmenjujejo sporočila. Spletna klepetalnica je bila napisana s pomočjo spletnih tehnologij, kot so HTML, CSS ter JavaScript in ogrodja Vert.x, ki deluje kot strežnik. Funkcija implementacije je filtriranje zlonamernih sporočil, ki bi lahko napadalcu omogočila izvedbo napada XSS.

Napad XSS oziroma večdomensko izvajanje kode (*angl. cross-site scripting*) je eden izmed najpogostejših napadov na spletnem aplikacijskem sloju [3]. Napadalci izkoriščajo ranljivosti vdelanih (*angl. embedded*) skript na spletnih straneh, ki se izvedejo na strani odjemalca, natančneje v spletnem brskalniku odjemalca.

Rešitev smo prikazali postopoma. Najprej podrobneje razčlenimo vertikel, ki skrbi za povezavo med spletno klepetalnico in ogrodjem Vert.x. Nato opišemo delovanje spletne klepetalnice. Na koncu rešitev združimo z mikrostoritvijo, ki vsebuje implementacijo razširitve ogrodja Vert.x. Slednja opravlja delo filtriranja uporabniških sporočil.

Na delčku kode 5.1 odpremo dogodkovno vodilo navzven in omogočimo komunikacijo odjemalec-strežnik. Nastavimo vsa potrebna dovoljenja za vhodne in izhodne naslove podatkovnih tokov. V primeru, da dovoljenj ne bi nastavili, komunikacija med odjemalcem in strežnikom ne bi bila možna. V

delčku kode 5.2 poslušamo podatkovni tok z naslovom „chat.to.server“, skozi katerega se pretakajo sporočila uporabnikov.

```
BridgeOptions bridgeOptions = new BridgeOptions()
        .addInboundPermitted(new
        ↪  PermittedOptions().setAddress("chat.to.server"))
        .addOutboundPermitted(new
        ↪  PermittedOptions().setAddressRegex("chatroom\\..+"))
        .addOutboundPermitted(new
        ↪  PermittedOptions().setAddress("tacos"));

// Dogodkovno vodilo povezemo z usmerjevalnikom
SockJSHandler sockJSHandler =
↪  SockJSHandler.create(vertx).bridge(bridgeOptions);
router.route("/eventbus/*").handler(sockJSHandler);
```

Delček kode 5.1: Odjemalcu omogočimo dostop do dogodkovnega vodila.

```
EventBus eventBus = vertx.eventBus();
eventBus.consumer("chat.to.server").handler(message -> {
        // Pridobitev sporocila
        eventBus.send("tacos", message.body(), ar -> {
                if (ar.succeeded()) {
                        // Pridobitev filtriranega sporocila
                        // ...
                } else {
                        System.out.println("Filtering failed.
                        ↪  Fallback...");
                        // ...
                }
        });
});
```

Delček kode 5.2: Prikaz sprejema in filtriranja sporočil.

Na strani odjemalca se z delčkom kode 5.3 najprej prijavimo na navzven odprto dogodkovno vodilo in ob postavitvi pričnemo poslušati na naslovu, ki je sestavljen iz predpone „chatroom.“ in poljubnega zaporedja znakov. Sporočilo ob prejemu razčlenimo v format JSON in ga prikažemo vsem naročnikom. Odjemalec poslano sporočilo dostavi strežniku z delčkom kode 5.4.

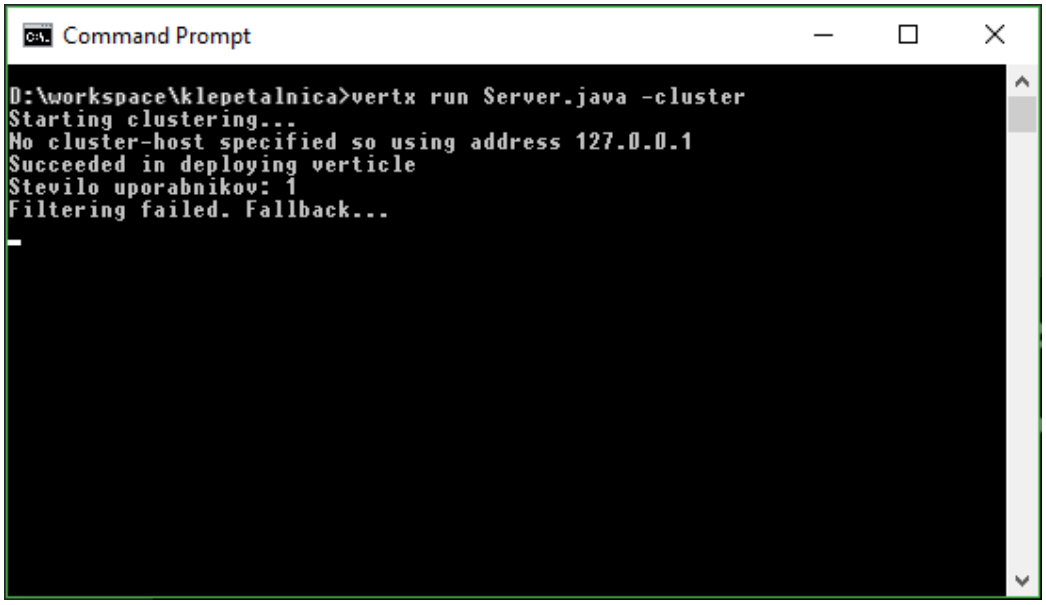
```
eventbus = new EventBus("/eventbus/");
eventbus.onopen = function() {
        eventbus.registerHandler("chatroom." + chatroom,
        ↪  function(error, message) {
                // Prejeto sporocilo
                message = JSON.parse(message.body);
                $('#messages').append(...);

        });
};
```

Delček kode 5.3: Prijava na dogodkovno vodilo na strani odjemalca.

```
eventbus.publish("chat.to.server", /* Sporocilo */);
```

Delček kode 5.4: Prikaz pošiljanja sporočil na strani odjemalca.

Vertikel, ki vsebuje implementacijo spletne klepetalnice, zaženemo z ukazom, prikazanim na sliki 5.1. Iz slike je razvidno, da trenutno spletno klepetalnico uporablja en uporabnik. Informacija „Filtering failed. Fallback...“ nam pove, da je uporabnik poslal sporočilo, ki ga ni bilo moč filtrirati. Razlog za neuspeh je še ne izvajajoča instanca mikrostoritve, ki vsebuje implementacijo razširitve ogrodja Vert.x. Slednja je prikazana z delčkom kode 5.5. Prejeto sporočilo najprej spremenimo v niz znakov in preverimo, če ni prazno. V primeru, da sporočilo ni prazno, sporočilo zabeležimo, filtriramo in pošljemo pošiljatelju nazaj. S tem onemogočimo prikaz informacije „Filtering failed. Fallback...“, ki smo je predhodno dobili pri pošiljanju sporočil, saj se je filtriranje uspešno izvedlo.



Slika 5.1: Prikaz zagona spletne klepetalnice.

```
@VertxEventListener(address = "tacos")
public void onMessage(Message<Object> event) {
        String message = (String) event.body();
        if(message != null) {
                log.info("Received message " + message);
                /* Filtriranje */
                event.reply(/* Filtrirano sporocilo */);
        }
}
```

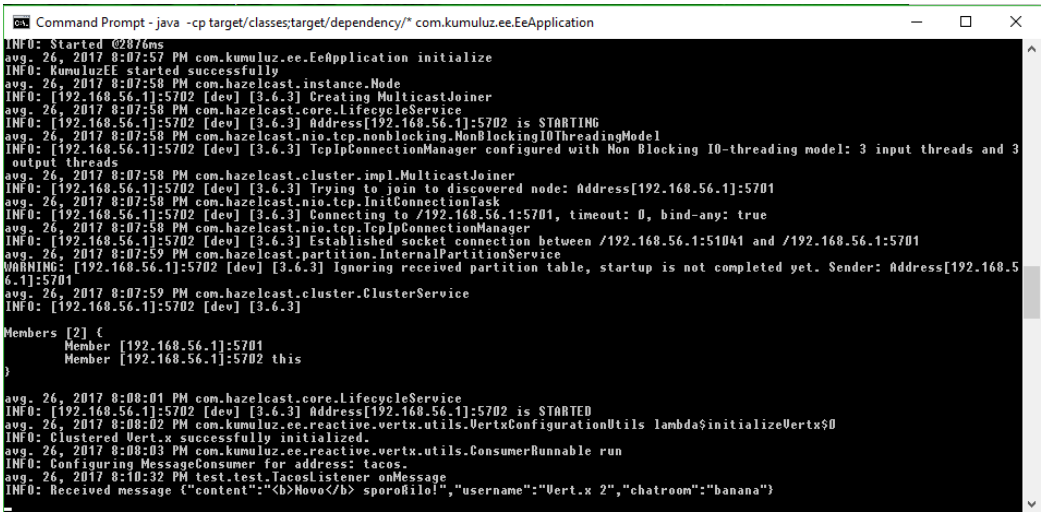
Delček kode 5.5: Prikaz prejema in filtriranja sporočil v mikrostoritvi z uporabo anotacije.

Spletno klepetalnico pustimo teči in poskusimo poslati sporočilo še enkrat, vendar prej zaženemo mikrostoritev. Zagon opravimo z ukazom, prikazanim na delčku kode 5.6. Iz razdelka `Members` na sliki 5.2 je razvidno, da sta se

instanca Vert.x, zagnana prek mikrostoritve, ter zunanja instanca Vert.x, ki skrbi za prikaz spletne klepetalnice, uspešno združili v gručo. Na sliki 5.2 sta razvidni tudi vsebnost in struktura prejetega sporočila. Sporočilo ima strukturo JSON in vsebuje polja `content`, `username` in `chatroom`. Na delčku kode 5.7 so prikazane odvisnosti, ki se nahajajo v konfiguracijski datoteki *pom.xml*.

```
java -cp target/classes;target/dependency/*
↪  com.kumuluz.ee.EeApplication
```

Delček kode 5.6: Ukaz za zagon mikrostoritve.



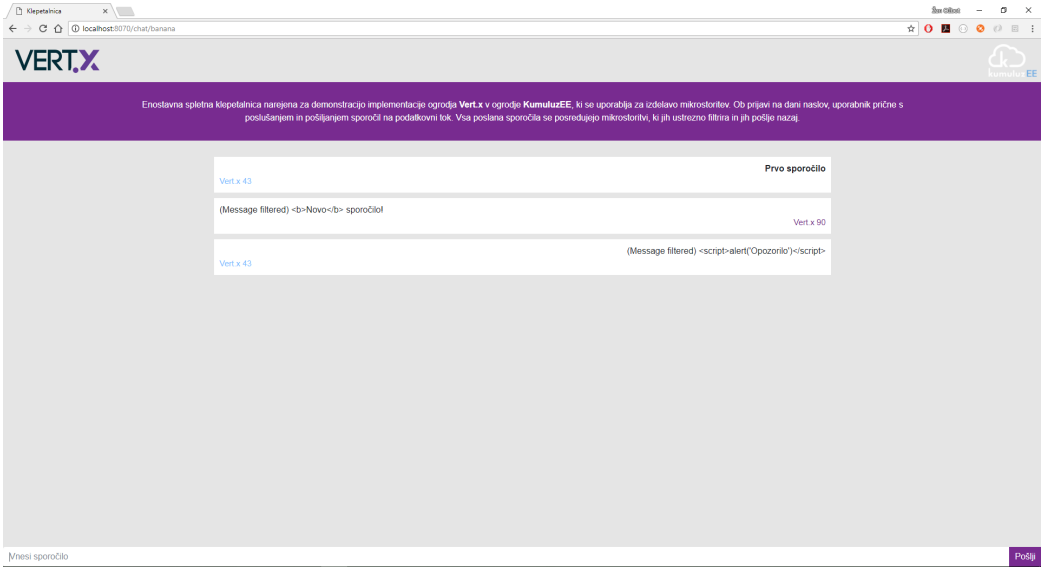
Slika 5.2: Prikaz uspešnega gručenja mikrostoritve z zunanjo instanco Vert.x.

```
<dependency>
    <groupId>com.kumuluz.ee</groupId>
    <artifactId>kumuluzee-core</artifactId>
</dependency>
<dependency>
    <groupId>com.kumuluz.ee</groupId>
    <artifactId>kumuluzee-servlet-jetty</artifactId>
</dependency>
<dependency>
    <groupId>com.kumuluz.ee</groupId>
    <artifactId>kumuluzee-jax-rs-jersey</artifactId>
</dependency>
<dependency>
    <groupId>com.kumuluz.ee</groupId>
    <artifactId>kumuluzee-cdi-weld</artifactId>
</dependency>
<dependency>
        <groupId>com.kumuluz.ee.reactive</groupId>
        <artifactId>kumuluzee-reactive-vertx</artifactId>
        <version>${kumuluzee-reactive-vertx.version}</version>
</dependency>
```

Delček kode 5.7: Odvisnosti znotraj konfiguracijske datoteke *pom.xml* mikrostoritve.

Uporabniški vmesnik je prikazan na sliki 5.3. Deležni smo toka sporočil dveh uporabnikov, ki želita testirati delovanje filtriranja. Iz slike je razvidno, da bi bil prvi poskus napada uspešen, saj je sporočilo odebeljeno. Opazimo, da se drugo in tretje sporočilo od prvega razlikujeta po predponi `(Message filtered)`. Predpona nam pove, da se je zagnana mikrostoritev uspešno postavila in povezala v gručo z zunanjo instanco Vert.x ter pričela opravljati

nalogo filtriranja.



Slika 5.3: Prikaz maske uporabniškega vmesnika.

Poglavje 6

Zaključek

V diplomskem delu smo pregledali področje reaktivnega programiranja v Javi, analizirali ogrodje Vert.x in razvili integracijo med ogrodjem Vert.x in ogrodjem za mikrostoritve KumuluzEE. Razširitev je bila izdela z namenom, da bi razvijalcem omogočili enostavnejši začetek uporabe ogrodja Vert.x znotraj mikrostoritev. Integracijo razširitve smo prikazali na enostavnem primeru spletne klepetalnice, katere namen je filtriranje uporabniških sporočil.

Med izdelavo diplomskega dela smo pridobili nova znanja s področja reaktivnega programiranja. Pregledali smo koncepte takšnega programiranja in ga primerjali z objektnim pristopom. Na kratko smo opisali vodilna ogrodja, ki nam omogočajo uporabo reaktivnega programiranja v raznolikih programskih jezikih.

Ogrodje Vert.x smo podrobneje razčlenili in ugotovili, da je dogodkovno vodeno, neblokirajoče in omogoča poleg izdelave reaktivnih aplikacij tudi obvladovanje velikega števila vzporednosti. Ponuja velik nabor komponent in omogoča uporabo zgolj tistih, ki jih resnično potrebujemo znotraj našega projekta.

Prikazali smo potek izdelave in uporabo dveh anotacij `@VertxEventPublisher` in `@VertxEventListener`, ki jih lahko uporabimo znotraj arhitekturnega sloga mikrostoritev. Prva anotacija omogoči izdajo sporočil na

podatkovni tok. Namen druge anotacije je prijava na podatkovni tok in morebiten odziv na prejeto sporočilo. Zagon, delovanje in izgled končne rešitve smo prikazali v poglavju 5.

6.1 Nadaljnje delo

Ogrodje KumuluzEE vsebuje razširitev, imenovano KumuluzEE Discovery, ki zagotavlja podporo za registracijo in odkrivanje storitev ter omogoča izenačevanje obremenitve (*angl. load balancing*) na odjemalčevi strani [5]. Ogrodje Vert.x vsebuje podobno komponento, ki se imenuje Vert.x Service Discovery. Slednja prav tako zagotavlja infrastrukturo za prijavo in odkrivanje storitev. Naša naloga za nadaljnje delo je razširiti izdelano rešitev s poenotenjem komponent KumuluzEE Discovery in Vert.x Service Discovery.

Literatura

[1] About the Eclipse Foundation. Dosegljivo: https://eclipse.org/org/. [Dostopano: 24. 08. 2017].

[2] Bacon.js. Dosegljivo: https://github.com/baconjs/bacon.js/blob/master/README.md. [Dostopano: 23. 08. 2017].

[3] Cross-site scripting (XSS) tutorial: Learn about XSS vulnerabilities, injections and how to prevent attacks. Dosegljivo: https://www.veracode.com/security/xss. [Dostopano: 26. 08. 2017].

[4] kumuluzEE. Dosegljivo: https://ee.kumuluz.com/. [Dostopano: 25. 08. 2017].

[5] KumuluzEE Discovery. Dosegljivo: https://github.com/kumuluz/kumuluzee-discovery. [Dostopano: 27. 08. 2017].

[6] Lesson: Annotations. Dosegljivo: https://docs.oracle.com/javase/tutorial/java/annotations/. [Dostopano: 25. 08. 2017].

[7] Observer pattern. Dosegljivo: https://en.wikipedia.org/wiki/Observer_pattern. [Dostopano: 21. 08. 2017].

[8] Reactive programming. Dosegljivo: https://en.wikipedia.org/wiki/Reactive_programming. [Dostopano: 21. 08. 2017].

[9] Reactive streams. Dosegljivo: http://www.reactive-streams.org/. [Dostopano: 23. 08. 2017].

[10] ReactiveX. Dosegljivo: http://reactivex.io/intro.html. [Dostopano: 23. 08. 2017].

[11] Reactor. Dosegljivo: https://projectreactor.io/. [Dostopano: 23. 08. 2017].

[12] Vert.x. Dosegljivo: http://vertx.io/. [Dostopano: 24. 08. 2017].

[13] Vert.x history. Dosegljivo: https://github.com/vert-x3/wiki/wiki/Vert.x-History. [Dostopano: 24. 08. 2017].

[14] Web framework benchmarks. Dosegljivo: https://www.techempower.com/benchmarks/#section=data-r8&hw=i7&test=plaintext. [Dostopano: 24. 08. 2017].

[15] Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson. The Reactive Manifesto. Dosegljivo: http://www.reactivemanifesto.org/. [Dostopano: 21. 08. 2017].

[16] Jonas Bonér and Viktor Klang. Reactive programming versus reactive systems. Dosegljivo: https://www.lightbend.com/reactive-programming-versus-reactive-systems. [Dostopano: 23. 08. 2017].

[17] Lucy Carey. Jax Innovation Awards 2014 champions declared. Dosegljivo: https://jaxenter.com/jax-innovation-awards-2014-champions-declared-107796.html. [Dostopano: 31. 08. 2017].

[18] Fernando Doglio. *Reactive Programming with Node.js*. Springer, 2016.

[19] Tilen Faganel and Matjaz B. Jurič. Microservices with Java EE and KumuluzEE. Dosegljivo: https://blog.kumuluz.com/architecture/2015/06/04/microservices-with-java-ee-and-kumuluzee.html. [Dostopano: 25. 08. 2017].

[20] Martin Fowler. Microservices. Dosegljivo: https://martinfowler.com/articles/microservices.html. [Dostopano: 25. 08. 2017].

[21] Fasih Khatib. Hello, actors. Dosegljivo: https://medium.com/akka-for-newbies/hello-akka-f0a908d4a859. [Dostopano: 24. 08. 2017].

[22] Ben Lesh. Creating and subscribing to simple observable sequences. Dosegljivo: https://github.com/Reactive-Extensions/RxJS/blob/master/doc/gettingstarted/creating.md#cold-vs-hot-observables. [Dostopano: 22. 08. 2017].

[23] Dan Lew. An introduction to functional reactive programming. Dosegljivo: http://blog.danlew.net/2017/07/27/an-introduction-to-functional-reactive-programming/. [Dostopano: 30. 08. 2017].

[24] Guido Salvaneschi and Mira Mezini. Debugging for reactive programming. In *Proceedings of the 38th International Conference on Software Engineering*, pages 796–807. ACM, 2016.

[25] André Staltz. The introduction to reactive programming you've been missing. Dosegljivo: https://gist.github.com/staltz/868e7e9bc2a7b8c1f754. [Dostopano: 21. 08. 2017].

[26] Johannes Thönes. Microservices. *IEEE Software*, 32(1):116–116, 2015.

[27] Harold Treen. What is reactive programming? Why should we use it and where? Dosegljivo: https://www.quora.com/What-is-reactive-programming-Why-should-we-use-it-and-where. [Dostopano: 23. 08. 2017].

[28] Ivan Yurchenko. About Akka Streams. Dosegljivo: https://jobs.zalando.com/tech/blog/about-akka-streams/?gh_src=4n3gxh1. [Dostopano: 23. 08. 2017].