Reaktivno programiranje: utjecaj na dizajn strukture programskog proizvoda

Martinčević, Alen

Master's thesis / Diplomski rad

2020

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:136061

Rights / Prava: Attribution 3.0 Unported/Imenovanje 3.0

Download date / Datum preuzimanja: **2024-04-20**

Repository / Repozitorij:

Faculty of Organization and Informatics - Digital Repository

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Alen Martinčević

REAKTIVNO PROGRAMIRANJE: UTJECAJ NA DIZAJN STRUKTURE PROGRAMSKOG PROIZVODA

DIPLOMSKI RAD

Varaždin, 2020.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Alen Martinčević

Matični broj: 46426/17–R

Studij: Informacijsko i programsko inženjerstvo

REAKTIVNO PROGRAMIRANJE: UTJECAJ NA DIZAJN STRUKTURE PROGRAMRSKOG PROIZVODA

DIPLOMSKI RAD

Mentor:

Doc. dr. sc. Zlatko Stapić

Varaždin, rujan 2020.

Alen Martinčević

Izjava o izvornosti

Izjavljujem da je moj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Ovaj rad obrađuje reaktivno programiranje kao model (uzorak) razvoja modernih aplikacija temeljenih na asinkronim tokovima podataka. Naglasak je na promjeni načina dizajna i pisanja programskog kôda, rješavanju problema kompleksnosti, nesigurnosti, blokiranja i održivosti kôda te prednosti koje reaktivno programiranje pruža u odnosu na ostale paradigme programiranja. Praktični dio rada prikazuje konkretnu primjenu reaktivnog programiranja u kontekstu razvoja aplikacijskog softvera korištenjem razvojnog okvira Spring WebFlux za razvoj reaktivnih web aplikacija te ostalih modula Spring ekosustava. Cilj ovog rada je dati detaljni utjecaj primjene reaktivnog programiranja na razvoj aplikacije u odnosu na alternativne paradigme te prikazati njegovu primjenu u procesu izgradnje kompletnog programskog rješenja. Ovaj rad objedinjuje teorijski i praktični dio od razine koncepata pa sve do kompletnog programskog rješenja te iznosi zaključke u obliku prednosti i nedostataka reaktivnog programiranja od razine programskog kôd pa sve do razine cjelokupnog sustava uz poštivanje preporučenih smjernica i principa izrade web aplikacija.

Ključne riječi: asinkroni tokovi podataka; java; mikroservisi; project reactor; reaktivno programiranje; reaktivni sustavi; spring

Sadržaj

SADRŽAJ ..... **III**

1. UVOD **1**

2. METODE I TEHNIKE RADA **3**

3. REAKTIVNO PROGRAMIRANJE **5**

3.1. KONCEPTI REAKTIVNOG PROGRAMIRANJA 6
3.1.1. Uzorak dizajna - promatrač *6*
3.1.2. Gradivni blokovi *9*
3.1.2.1. Subjekt 9
3.1.2.2. Promatrač 10
3.1.2.3. Pretplata 11
3.1.2.4. Operatori 13
3.1.2.5. Raspoređivač 13
3.1.3. Asinkroni tokovi podataka *14*
3.1.4. Neblokirajuće izvršavanje *15*
3.2. PREDNOSTI 20
3.2.1. Jednostavnije asinkrono programiranje i čitljiviji kôd *21*
3.2.2. Bolje performanse *22*
3.2.3. Bolje rukovanje iznimkama i pogreškama *24*
3.2.4. Jednostavniji rad s povratnim tlakom *25*
3.3. NEDOSTACI 28
3.4. UTJECAJ NA DIZAJN ARHITEKTURE KLASA 29
3.5. KADA KORISTITI REAKTIVNO PROGRAMIRANJE 36

4. REAKTIVNI SUSTAVI **40**

4.1. REAKTIVNI MANIFEST 41
4.2. REAKTIVNI SUSTAVI NASPRAM REAKTIVNOG PROGRAMIRANJA 43
4.2.1. Rad s događajima naspram rada s porukama *45*
4.3. PREDNOSTI REAKTIVNIH SUSTAVA UZ PRIMJENU RP 45

5. RP U PROGRAMSKOM JEZIKU JAVA **47**

5.1. SPECIFIKACIJA REAKTIVNIH TOKOVA 47
5.2. REAKTIVNE EKSTENZIJE (REACTIVEX) 51
5.3. PROJEKT REACTOR 52
5.3.1. Flux i Mono *52*
5.3.2. Pretplata *55*
5.3.3. Procesori *56*
5.3.4. Operatori *58*
5.3.4.1. Stvaranje tokova 58
5.3.4.2. Programsko stvaranje toka 61
5.3.4.3. Transformacija toka 64
5.3.4.4. Spajanje tokova 69
5.3.4.5. Rukovanje greškama 75
5.3.4.6. Upravljanje dretvama 78

6. IZRADA REAKTIVNE WEB APLIKACIJE **81**

6.1. IDEJA APLIKACIJE 81
6.2. TEHNIČKI ASPEKTI APLIKACIJE 85
6.2.1. Arhitektura sustava *85*
6.2.2. Korišteni projekti, moduli, okviri i biblioteke *86*
6.2.2.1. Spring boot okvir i Spring Initializr platforma 88
6.2.2.2. Spring Reactive Web modul 91
6.2.2.3. Spring Security 94

6.2.2.4. Spring Data Reactive MongoDB 94
6.2.2.5. Spring Cloud i Cloud Stream 96
6.2.3. Komunikacija s udaljenim servisima 100
6.3. IMPLEMENTACIJA SPECIFIČNIH FUNKCIONALNOSTI 103
6.3.1. Reaktivna registracija 103
6.3.2. Ažuriranje prikaza na reaktivan način 107
6.3.3. Funkcionalnost oglašavanja i razgovora 111
6.3.4. Funkcionalnost recenziranja 115
7. ZAKLJUČAK 120
POPIS LITERATURE 122
POPIS SLIKA 127
POPIS TABLICA 129

1. Uvod

U današnje vrijeme u kojem se i one najjednostavnije radnje pokušavaju automatizirati, svaki poslovni proces informatizirati, vrijeme u kojem svaka obitelj posjeduje barem jedno računalo, u kojem svaka osoba svakodnevno koristi pametni mobilni uređaj za pristup raznim društvenim mrežama koje broje korisnike u milijunima i promete u stotinama terabajta podataka u minuti. Sve je to omogućeno brzim napretkom tehnologije, razvojem sve jačih procesora, povećanjem radne memorije računala, sve većim spremištima podataka, razvojem serverskih farmi samo kako bi se omogućilo posluživanje sve više rastućeg broja korisnika. Međutim, tehnologija može napredovati samo toliko brzo. Ponekad rješenja kao što su vertikalno i horizontalno skaliranje, i tu mislimo na povećavanje procesorske snage obrade, jednostavno nije zadovoljavajuće rješenje. Ova prepreka većinom je dosta izraženo kod start-upova te manjih i srednjih informatičkih firmi kod kojih većinom broj korisnika sustava može doživjeti nagli skok u vrlo kratkom vremenu. U takvim situacijama korištenjem rada u oblaku koje podržava vertikalno i horizontalno skaliranje može pomoći, međutim nekad takvo rješenje nema prevelikog ekonomskog smisla jer trošak zakupljivanja i korištenja usluge infrastrukture može biti daleko veći od dobiti koje programsko rješenje ostvaruje. U takvim slučajevima treba napraviti jedan korak unazad i napraviti detaljnu analizu utilizacije resursa te odgovoriti na pitanje što uzrokuje potrebu za velikom količinom raspoloživih resursa te kako možemo smanjiti njihovu količinu, a da u isto vrijeme njihovim smanjivanjem ne utječemo na korisničko iskustvo kod korištenja sustava ili aplikacije.

U većini slučajeva, neoptimiziranost sustava i loše korisničko iskustvo proizlazi iz lošeg arhitekturalnog rješenja i loše napisanog odnosno neoptimiziranog programskog kôda. U današnje vrijeme, gdje se zahtjevi korisnika mijenjaju velikom brzinom, gdje tržište svakog dana izbaci stotine novih rješenja za isti problem, programeri i arhitekti jednostavno nemaju vremena posvetiti se izradi kvalitetnog kôda koji će učiniti aplikaciju optimiziranom za korištenje od strane velikog broja korisnika. Tu u priču ulazi i tema ovog diplomskog rada, a to jedna drugačija paradigma programiranja zvana reaktivna paradigma ili reaktivno programiranje. U ovom radu vidjet ćemo kako nam reaktivno programiranje može pomoći u pisanju kvalitetnog i optimiziranog kôda. Na kojim principima se temelji te koje principe nadograđuje. Opisat ćemo glavnu ideju reaktivnog programiranja te koje su njezine prednosti ali i nedostaci kod izrade programskog rješenja te kakvu ono ima utjecaj na optimiziranost rada same aplikacije. Nakon detaljnog upoznavanja s reaktivnim programiranjem osvrnut ćemo se i na principe reaktivnih sustava, što su reaktivni sustavi, koja je njihova ideja te kako nam reaktivno programiranje može pomoći u njegovom ostvarenju odnosno izradi. U poglavlju nakon pogleda ćemo realnu primjenu reaktivnog programiranja i reaktivnih sustava u stvarnom

svijetu s naglaskom na probleme za koje nam ono može poslužiti kao jedno od mogućih rješenja. Praktični dio bit će realiziran korištenjem reaktivnog programiranja u programskom jeziku Java i to u sklopu određene reaktivne implementacije reaktivnih tokova zvane Reactor. Nakon što pokažemo osnovne koncepte reaktivnog programiranja korištenjem biblioteke Reactor prijeći ćemo na izradu reaktivne web aplikacije izgrađene korištenjem Spring okvira, projekata i modula za ostvarivanje mikroservisnog rješenja koje će u pozadini koristi reaktivnu paradigmu. U poglavlju koje će se baviti izradom mikroservisne reaktivne aplikacije, dati će se detaljno objašnjenje koji su svi okviri korišteni i zašto. Nakon što postavimo dobro temelje za razumijevanje rada sustava, prikazat ćemo nekolicinu funkcionalnosti koje posjeduju najviše prednosti od korištenja reaktivnog programiranja sa strane same implementacije ali isto tako i sa strane optimiziranosti rada sustava. Na kraju rada izreći će se kratki zaključak u kojem ćemo se osvrnuti na sve što je bilo obrađeno u radu.

2. Metode i tehnike rada

Metoda rada kojom je ovaj rad pisan jest metoda od gore prema dolje (eng. *Top-down*). Sama struktura rada podijeljena je na tri dijela: teorijski, semi-praktični i praktični dio. To znači da će se na početku obrađivati teorijski koncepti reaktivnog programiranja počevši od koncepata i gradivinih blokova, pa sve do zapažanja s aspekta prednosti i nedostataka primjene reaktivnog programiranja. Na kraju teorijskog djela vezanog uz reaktivno programiranje spomenuti ćemo i objasniti situacije u kojima reaktivno programiranje briljira te ćemo se kratko osvrnuti na utjecaje koje reaktivno programiranje ima na projekt u kontekstu strukture programskog kôda, povezanosti klasa i ovisnosti objekata. Nakon što su postavljeni dobri teorijski temelji za razumijevanje reaktivnog programiranja prijeći će se na idući teorijsku cjelinu koja predstavlja jednu apstrakcijsku razinu više i koja će nam objasniti što su to reaktivni sustavi, gdje i zašto se koriste i kako ih možemo postići. U tom poglavlju osim samog opisa reaktivnih sustava, osvrnut ćemo se i na različitosti reaktivnih sustava naspram reaktivnog programiranja. Nakon obrade i treće teorijske cjeline, prelazimo na semi-praktični dio u kojem ćemo dati uvod u tehnologije i potkrijepiti prethodno obrađene teorijske koncepte s programskim primjerima koristeći dostupne okvire i biblioteke za reaktivno programiranja. Kako reaktivni koncepti vrijede za sve implementacije reaktivnog programiranja te kako bi se izbjegla prezasićenost primjera i ponavljanja obradit će se samo jedan reaktivni okvir koristeći jedan programski jezik. Isto vrijedi i za sam kraj rada koji predstavlja čisti praktični dio u kojem će se primijeniti stečeno znanje iz prethodno obrađenih cjelina s ciljem razvoja mikro-servisne web aplikacije uz pomoć korištenja reaktivnog programiranja kako bismo zaokružili rad i prikazali na realnom primjeru koje sve mogućnosti i prednosti donosi korištenje reaktivnog programiranja ali isto tako kako bismo i prikazali utjecaje i promjene na arhitekturu sustava naspram standardnih web aplikacija.

Za izradu rada korišteno je sljedeće:

- IntelliJ IDEA Ultimate izdanje – IDE okolina (eng. *Integrated Development Environment*) koja će se koristiti za razvoj programskog rješenja.
- Java (verzija 1.8) – Programski jezik i verzija koji će se koristiti za prikazivanje primjera i izradu praktičnog dijela.
- MongoDB – NoSQL baza podataka. Razlog korištenja ove baze proizašao je iz toga što u vremenu pisanja diplomskog rada nije bilo produkcijsko spremnih reaktivnih drajvera za rad s relacijskim bazama podataka.
- MongoDB Compass – Sustav za upravljanje bazom podataka (eng. *DBMS – Database Management System*)

- Microsoft word – Alat korišten za pisanje rada
- Netty – NIO poslužiteljski okvir koji omogućuje brz i lak razvoj mrežnih aplikacija. Pojednostavljuje mrežno programiranje, a koristi se zbog svoje ne-blokirajuće karakteristike.
- RabbitMQ – Softver otvorenog kôda za razmjenu poruka koji implementira napredni protokol čekanja poruka (eng. *AMQP – Advance Message Queuing Protocol*).
- Spring Boot – Programski okvir za razvoj Spring web aplikacija baziranih na Java programskom jeziku.
- Spring Cloud Eureka – Poslužitelj koji će se koristiti za otkrivanje pojedinih servisa mikro-servisne arhitekture.
- Spring Cloud Stream – Okvir za izgradnju skalabilnih mikroservisa baziranih na radu s događajima povezanih sustavom razmjene poruka.
- Spring Initializr – Online alat za jednostavnije inicijaliziranje početnog Spring Boot projekta uz mogućnost jednostavnog definiranje potrebnih ovisnosti (eng. *Dependencies*).

3. Reaktivno programiranje

U ovom poglavlju detaljno će se obraditi reaktivna paradigma odnosno reaktivno programiranje s naglaskom na utjecaj razvoja programskog proizvoda, performanse i rad sustava, a u konačnici i na samu arhitekturu sustava. Za početak ćemo dati kratku definiciju i uvod u reaktivno programiranje, zatim ćemo detaljnije opisati same koncepte na kojima se ono temelji, spomenut ćemo specifičnosti paradigme, njene prednosti i nedostatke, u kojim situacijama nam primjena reaktivnog programiranja može pomoći te ćemo na kraju navesti koje to utjecaje reaktivno programiranje ima na implementaciju programskih rješenja i kako nam ono može pomoći u izgradnji reaktivnih sustava. Na samom kraju poglavlja dotaći ćemo se i pojma reaktivnih sustava čije će se detaljno objašnjenje prirodno nastaviti u idućem poglavlju.

Iako u današnje vrijeme možemo sve više čuti pojmove reaktivno i reaktivni sustavi u kontekstu programiranja i razvoju sustava, ovi pojmovi zapravo nisu ništa odveć novo. Prva spomen ovih dvaju pojmova datira iz čak 1969. godine u kojoj je Alan Kay objavio znanstveni rad pod nazivom „Reaktivni Stroj“ (eng. *The Reactive Engine*). Međutim, prvo modernije korištenje pojma reaktivnog programiranja možemo pronaći u znanstvenom radu objavljenom od strane Conal Elliota i Paul Hudaka 1997. godine pod nazivom „Funkcionalna Reaktivna Animacija“ (eng. *Functional Reactive Animation*) (Elliott & Hudak, bez dat.). Funkcionalna reaktivna animacija koja je poslije promijenila naziv u funkcionalno reaktivno programiranje ili FRP temelji se na ponašanjima i promjenama u skladu s događajima. Pokazalo se da uzorke koje FRP pruža za rukovanje transformacijama i za reagiranje na događaje (nuspojave koje FRP pokušava umanjiti) se mogu jednostavno prenijeti i iskoristiti u imperativnim jezicima. Isti uzorci pokazali su se korisnima za upravljanje i reagiranje na tok diskretnih događaja iako su ti tokovi pretežno bazirani na modelu guranja umjesto na modelu povlačenja kao što je slučaj kod funkcionalnih odnosno deklarativnih programskih jezika. U konačnici, to je rezultiralo nastankom nove paradigme za imperativne jezike naziva reaktivno programiranje, zamijetite nedostatak riječi funkcionalno u odnosu na FRP.

Reaktivno programiranje predstavlja paradigmu koja se temelji na događajima i njihovim emitiranjem, tokovima podataka i propagaciji tih događaja odnosno promjena. Podržava dekomponiranje problema na više diskretnih koraka od kojih se svaki može izvesti na asinkroni i ne-blokirajući način koji se potom mogu sastaviti kako bi stvorili tijek rada koji može biti neograničen u svojim ulazima ili izlazima (Klang, 2016). Možemo reći da ono predstavlja stil mikro-arhitekture koja uključuje inteligentno usmjeravanje i konzumiranje događaja (Syer, 2016). Primjenom reaktivnog programiranja, program se konstruira na takav način da se kod promjene stanja objekata generira događaj koji se potom prosljeđuje

podatkovnim tokom svim ostalim zainteresiranim objektima aplikacije. Ideja reaktivnog programiranja je da poveže dijelove aplikacije u smislu da kada se dogodi neka promjena u određenom objektu da drugi dio odnosno drugi ovisni objekti automatski reagiraju na određenu promjenu i da na temelju nje odrade neku drugu operaciju kao što je osvježavanje prikaza na korisničkom sučelju. Štoviše, dio koji reagira na promjenu može kao reakciju posjedovati događaj kojeg prate ostali dijelovi što u konačnici dovodi do povezanosti cijele aplikacije i čine ju reaktivnom (Tuđan, 2017).

Možda najvažnija razlika koja se može izdvojiti između funkcionalnog i imperativnog reaktivnog programiranja jest „vrijeme“. Vrijeme u FRP-u predstavlja vrijednost koja se kao parametar prosljeđuje svakoj operaciji te igra važnu ulogu u samom izvršavanju operacija. S druge strane kod imperativnog reaktivnog programiranja, redoslijed kojim se stječu relevantni mutexi[1] je važnije nego vrijeme. U računalnom programiranju mutexi predstavljaju međusobno isključive objekte koji omogućavaju većem broju dretvi korištenje istog ili istih resursa, ali ne u isto vrijeme. Jednostavnijim riječima, to znači da FRP funkcionira na principima simulacije gdje se može dogoditi istovremeno rukovanje dvama ili više događaja za razliku od reaktivnog programiranja koje se opisuje kao jednostavni niz blokova operacija poredanih po redoslijedu njihove obrade bez mogućnosti istovremenog rukovanja dvama ili više događaja bez uvođenja višedretvenosti (Gallagher, 2016).

Prva važnija odnosno popularnija implementacija reaktivnog programiranja u imperativnim jezicima bila je Reaktivna ekstenzija (Rx) za .NET programski okvir čija je prva verzije bila objavljenja 2009. godine. Ubrzo nakon toga uslijedile su i ostale reaktivne ekstenzija i za ostale jezike kao što su Java, JavaScript, Swift i ostali.

3.1. Koncepti reaktivnog programiranja

Reaktivno programiranje je kao što smo već rekli paradigma koja se bazira na više različitih ali podjednako važnih koncepata. Ti koncepti su sljedeći: asinkroni tokovi podataka, uzorak dizajna promatrač, konkurentnost i neblokirajuće izvršavanje. U idućim potpoglavljima detaljno ćemo opisati svaki pojedini koncept te ćemo navesti koji su to gradivni blokovi pojedinih koncepata kako bismo dobili detaljniju i širu sliku reaktivnog programiranja.

3.1.1. Uzorak dizajna - promatrač

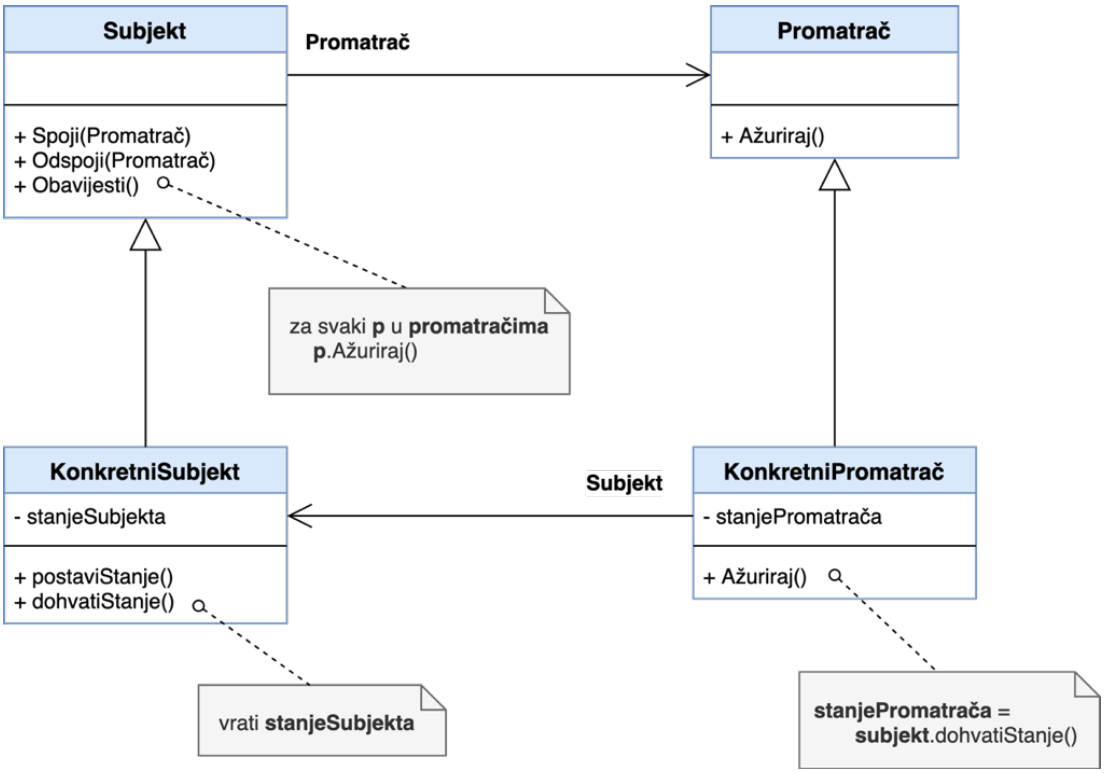
Uzorak dizajna promatrač (eng. *Observer Pattern*) jedan je od najčešće korištenih softverskih uzoraka dizajna u razvoju programskih rješenja te igra veoma važnu ulogu u

[1] Mutex – Objekt uzajamnog izuzimanja.

reaktivnom programiranju. Da bismo bolje razumjeli razlog njegove primjene u reaktivnom programiranju dati ćemo kratki opis, objasniti dijagram klasa te navesti svrhu primjene ovog uzorka.

Uzorak dizajna promatrač spada u uzorke ponašanja koji se bave algoritmima i dodjelom odgovornosti među objektima. Uzorci ponašanja ne opisuju samo predloške objekata ili klasa, nego i komunikaciju među njima te kako su ti objekti međusobno povezani. Ove uzorke karakterizira kompleksni kontrolni tok izvođenja kojeg je vrlo teško pratiti tijekom izvršavanja samog programa. Međutim fokus uzoraka ponašanja je na tome kako su objekti međusobno povezani, a ne na sam tijek izvršavanja. Uzorke ponašanja dijelimo na klasne uzorke i objektne uzorke. Promatrač pripada objektnim uzorcima ponašanja što znači da se za distribuiranje ponašanja među klasama koristi objektna kompozicija radije nego nasljeđivanje kao što je to slučaj kod klasnih uzoraka. Uzorak promatrača definira ovisnost jednog ili više objekata o promatranom objektu i koristi se za automatsko obavještavanje i ažuriranje zavisnih objekata (Gamma i ostali, 1994).

Uzorak je baziran na entitetima subjekta (eng. *Subject*) i promatrača (eng. *Observer*) kao što možemo vidjeti iz dijagrama klasa prikazanog na sljedećoj slici. Subjekt je jedinstveni objekt koji sadrži listu pretplaćenih objekata koje treba obavijestiti svaki put kada se promjeni njegovo stanje. Pretplaćeni objekti se nazivaju promatrači i oni sadrže javnu metodu za ažuriranje njihovog internog stanja koju subjekt poziva nakon što se dogodi promjena stanja. Metoda `obavijesti()` iterira listom pretplaćenih promatrača i izvršava njihovu javnu metodu `ažuriraj()` koja ažurira stanje promatrača sa subjektovim stanjem. Osim metode za obavještavanje, subjekt sadrži još minimalno dvije javne metode, a to su metoda `spoji(Promatrač)` i `odspoji(Promatrač)`. Obje ove metode kao argument primaju objekt promatrača. Iz njihovih naziva možemo vrlo jednostavno razumjeti da jedna metoda dodaje objekt novog promatrača dok druga uklanja objekt promatrača iz interne liste subjekta. Ostali sudionici ovog uzorka jesu `KonkretniSubjekt` (eng. *Concrete Subject*) i `KonkretniPromatrač` (eng. *Concrete Observer*). KonkretniSubjekt implementira sučelje subjekta te definira još dvije metode `postaviStanje()` i `dohvatiStanje()`. Ove dvije metode zadužene su za postavljanje i dohvaćanje internog stanja subjekta. `KonkretniPromatrač` implementira sučelje `Promatrač`, ne implementira dodatne metode ali zato sadrži vlastito interno stanje koje je zapravo jednako subjektovom stanju na kojeg je preplaćen nakon poziva metode `Ažuriraj()`.



Slika 1: Dijagram klasa uzorka ponašanja Promatrač (Prema: Gamma et al., 1994)

Ovaj uzorak se primjenjuje u sljedećim situacijama:

- U arhitekturi aplikacije postoje dva ili više entiteta koja ovise jedni o drugima te ih želimo zadržati odvojenima kako bi bili u mogućnosti ponovo ih iskoristiti.
- Drugi, češći razlog zbog kojeg se ovaj uzorak primjenjuje je kada objekt koji se mijenja mora obavijestiti o promjeni svog internog stanja nepoznati broj povezanih i ovisnih objekata.
- Subjekt treba obavijestiti ovisne objekte bez direktnog znanja koji su to točno objekti.

U konačnici, funkcioniranje uzorka promatrača možemo prikazati na jednostavan način putem sljedećeg dijagrama. Entiteti promatrača pretplaćuje se na određeni subjekt koji nakon što se dogodi događaj promjene njegovog internog stanja, okida događaj koji se propagira do svih zainteresiranih promatrača kako bi isti bili u mogućnosti reagirati na promjenu.

Slika 2: Uzorka dizajna - promatrač

Međutim, kakve veze ima uzorak dizajna promatrač s reaktivnim programiranjem. Za reaktivno programiranje često možemo čuti tvrdnju da ono zapravo predstavlja ispravnu implementaciju ovog uzorka. To znači da iste sudionike subjekta i promatrača možemo pronaći i u pozadini reaktivnog programiranja. Ovisno o reaktivnoj biblioteci ili okviru koji koristimo nazivi tih entiteta mogu biti drugačiji. U sljedećim potpoglavljima obradit ćemo detaljno značenje i korištenje subjekta i promatrača ali i ostalih značajnih gradivnih blokova reaktivnog programiranja.

3.1.2. Gradivni blokovi

Ovo potpoglavlje bavi se teorijskim opisom glavnih konstrukata koje je moguće pronaći u programskim okvirima i bibliotekama koje implementiraju koncepte reaktivnog programiranja. Detaljnija primjena i sami programski primjeri bit će detaljnije objašnjeni u poglavljima koja će obrađivati rad u jednom od reaktivnih programskih okvira te u samom praktičnom djelu ovog rada.

3.1.2.1. Subjekt

Subjekt je entitet koji predstavlja izvor podataka, odnosno događaja, koji se periodično emitiraju promatračima i time zapravo tvore tok podataka. Razlikujemo dvije vrste subjekta, **hladne subjekte** (eng. *Cold Observables*) i **tople subjekte** (eng. *Hot Observables*). Kod hladnih subjekta emitiranje podataka započinje onog trenutka kada se promatrač pretplati na subjekt, odnosno kažemo kada netko započne s konzumacijom tog subjekta. Mogli bismo još reći da su hladni subjekti zapravo lijeni (eng. *Lazy*) i asinkroni jer emitiraju podatke tek onda kada se od njih to eksplicitno zatraži. Važno je napomenuti da podaci emitirani od hladnih subjekata nisu djeljivi između više promatrača što znači da će svaki promatrač koji se pretplati na subjekt dobiti sve događaje predviđene za emitiranje neovisno o njihovoj količini, broju promatrača, vremenu i redoslijedu pretplate promatrača. Jednostavnije rečeno prvi i zadnji po redu pretplaćeni promatrač budu primili isti skup događaja kao i svi ostali pretplaćeni promatrači koji se nalaze između prvog i zadnjeg. S druge strane topli subjekti ne čekaju da se promatrač pretplati kako bi mogli emitirati događaje, već s emitiranjem počinju onog trenutka kada su stvoreni. Kažemo da su događaji neovisni o pojedinom promatraču, odnosno da nisu djeljivi. Ako za hladne subjekte kažemo da su lijeni, onda za tople subjekte možemo reći da su željni (eng. *Eager*) jer počinju s emitiranjem događaja neovisno da li netko na drugoj strani te događaje očekuje ili ne. To znači da ovisno o vremenu i redoslijedu pretplate promatrača na subjekt, dva različita promatrača mogu ali i ne moraju dobiti isti skup događaja od subjekta. Svaki promatrač koji se pretplati dobiti će samo one događaje koji su emitirani nakon trenutka njegove pretplate. Također, zbog željnog načina emitiranja, može se dogoditi da topli subjekt

završi s emitiranjem događaja, kažemo da se subjekt istrošio, prije nego li se i jedan promatrač pretplati i uspije dobiti emitirane događaje.

3.1.2.2. Promatrač

Promatrači s druge strane predstavljaju komponente koje konzumiraju emitirane događaje. Da bi primili događaj emitiran od subjekta, promatrači se moraju pretplatiti na određeni subjekt koristeći metodu za pretplatu. Kada god subjekt emitira pojedini događaj, svi registrirani promatrači će zaprimiti emitirani događaj. Iz ovog možemo vidjeti da su zapravo temeljni principi u reaktivnom programiranju preuzeti iz samog uzorka promatrača, međutim oni su podignuti na jednu višu razinu zahvaljujući asinkronoj implementaciji koja se nalazi u pozadini reaktivne paradigme. Nadalje, osim same asinkronosti, na uzorak su dodatne još dvije dodatne mogućnosti za koje se kaže da nedostaju kod primjene standardne OOP inačice uzorka promatrača. Te dvije nove mogućnosti odnose se na proširenu način komuniciranja između subjekta i promatrača koji se u reaktivnom programiranju odviju putem triju kanala umjesto samo jednog (Bagwala, 2019). Ti kanali su redom:

- Podatkovni kanal (eng. *Data Channel*),
- kompletni kanal (eng. *Complete Channel*),
- kanal pogrešaka (eng. *Error Channel*).

Na idućoj slici možemo vidjeti vizualni prikaz tih triju kanala i smjer njihove komunikacije od subjekta prema promatraču.

Slika 3: Vizualizacija triju kanala (Prema: Bagwala, 2019)

Podatkovni kanal kao što i samo ime govori predstavlja standardni kanal za prijenos podataka između subjekta i promatrača i njime se šalju emitirani događaji i podaci. Međutim ostala dva kanala predstavljaju novost uvedenu s reaktivnim programiranjem. S novim kanalima uvedena je mogućnost da subjekt signalizira svojim slušačima prestanak emitiranje novih događaja. Druga nova mogućnost jest sposobnost subjekta da signalizira promatračima da se dogodila pogreška na njegovoj strani te sposobnost propagiranja te iste pogreške nizvodnim komponentama odnosno slušačima koji potom mogu tu pogrešku dalje obraditi.

Podatkovni kao i ostala dva kanala stvaraju se u onom trenutku kada se promatrač pretplati na subjekt te ostaje otvoren sve dokle god se ne desi jedan od sljedećih scenarija:

- Subjekt je poslao signal kojim označava prestanak emitiranja novih podataka promatračima. Signal prekida emitiranja podataka se šalje putem signalnog kanala.
- Subjekt se iscrpio te nema više dostupnih podataka za emitirati ali subjekt nije imao pretplaćenih promatrača kojima bi poslao signal prekida (u slučaju da se radi o toplom subjektu).
- Dogodila se pogreška uzvodno na strani subjekta. Subjekt emitira ili prosljeđuje pogrešku nizvodno promatračima putem kanala pogreške. Razlog zašto je pogrešku moguće slati kanalom pogreške za kojeg smo rekli da zapravo predstavlja tok podataka leži u tome što se u reaktivnom programiranju na pogreške gleda kao i na svaki drugi podatak, stoga nije potreban dodatan mehanizam za njihovim rukovanjem.

Onog trenutka kada promatrač zaprimi završni signal, ili zaprimi pogrešku kroz dedicirani kanal, promatrač zatvara podatkovni kanal ali ujedno i prekida sve otvorene pretplate promatrača. Razlog tomu je u tom što zatvoreni podatkovni kanal označava da subjekt više neće emitirati nove podatke stoga nema ni smisla čuvati veze između subjekta i promatrača u obliku pretplata.

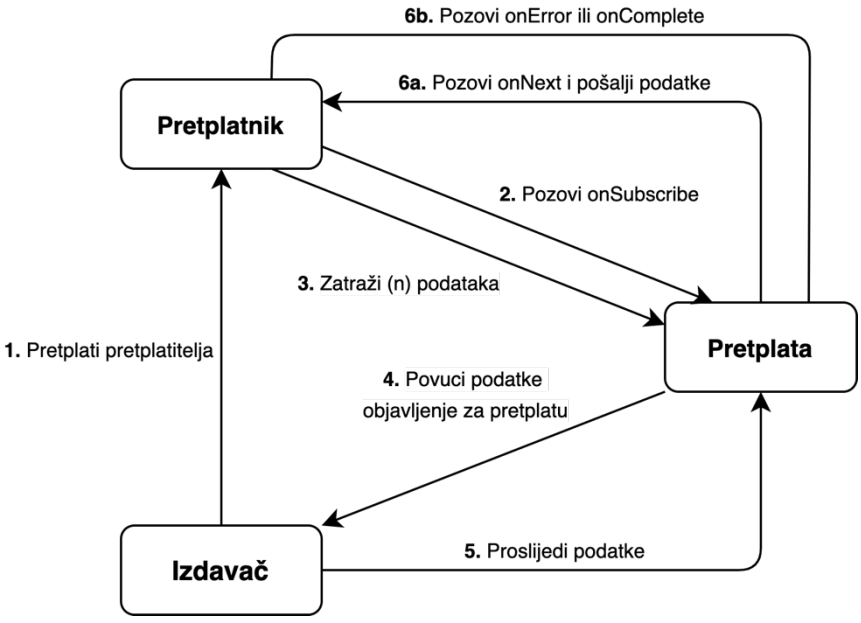
U mnogim ostalim programskim paradigmama, više ili manje se očekuje od instrukcija da se izvršavaju inkrementalno odnosno slijedno, jedna nakon druge, onim redoslijedom kojim su i napisane. Međutim u reaktivnom programiranju, tijekom izvršavanja, instrukcije se ne moraju nužno izvršavati redoslijedom kojim su napisane. Razlog tomu leži u asinkronom izvršavanju koje rezultira kasnijim izvršavanjem pojedinih instrukcija smještenih prije u strukturi programskog kôda od trenutačno izvršavane instrukcije. Rezultati tih instrukcija su poslije uhvaćeni proizvoljnim redoslijedom od strane promatrača. Umjesto da se koriste uzvratni pozivi (eng. *Fallback methods*), obećanja (eng. *Promises*) ili neki drugi asinkroni mehanizmi, reaktivno programiranje implementira poseban mehanizam za dohvaćanje i pretvaranje podataka u obliku spomenutog koncepta subjekta i promatrača. U trenutku pretplate promatrača na subjekt, prethodno definirani mehanizam se aktivira te promatrač koji stražari je spreman uhvatiti i reagirati na podatke emitirane od strane subjekta.

3.1.2.3. Pretplata

Pretplata predstavlja konekciju između izdavača i pretplatnika ili mogli bismo reći da predstavlja kontroler poruka. Izdavač stvara pretplatu za svakog pretplatnika koji se zatim pokušava na nju pretplatiti. Pretplata rukuje zahtjevima od strane pretplatnika. Drugim riječima,

izdavač služi kao spremište podataka iz kojeg će pretplata dobivati podatke. Na sljedećoj slici možemo vidjeti rad mehanizma reaktivnog programiranja s aspekta pretplatnika i izdavača uz dodatak nove apstrakcije pretplate. Ova slika nam može olakšati razumijevanje toka podatka od izdavača do pretplatnika od trenutka pretplate. U idućim natuknicama možemo vidjeti detaljnije objašnjenje dijagrama prikazanog na slici:

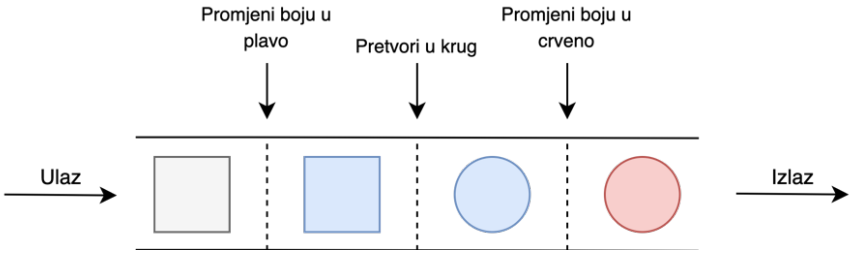
- Izdavač koristi metodu `subscribe()` koja označava da se pretplatnik može pretplatiti na izdavača (Korak 1).
- Pretplatnik se pretplaćuje na izdavača pozivanjem metode `onSuscribe()`. Pozivom ove metode pretplatniku se dodjeljuje pretplata. Svaki izdavač definira vlastitu pretplatu te posjeduje logiku za emitiranje podataka dotičnoj pretplati (Korak 2).
- Pretplata predstavlja vezu između izdavača i pretplatnika koja se koristi za potraživanje podataka od strane pretplatnika pozivanjem metode `request(n)` koja za parametar prima broj podataka koje pretplatnik želi dobiti od izdavača (Korak 3).
- Izdavač svojim tempom emitira podatke. Svaki put kada se emitira pojedini podatak, pretplata poziva `onNext()` metodu kojom se prosljeđuje emitirani podatak (Korak 4, Korak 5, Korak 6a)
- U slučaju pojave greške ili istrošenosti izdavača (označavaju prestanak emitiranja podataka) pretplata može pozvati `onError()` ili `onComplete()` metodu pretplatnika (Korak 6b).



Slika 4: Prikaz rada sustava Izdavača i Pretplatnika u reaktivnom programiranju (Prema: Mandar jog, 2017)

3.1.2.4. Operatori

Operatori predstavljaju jednostavne funkcije koje izvršavaju određene akcije nad vrijednostima te vraćaju modificiranu vrijednost. Većina operatora rade nad subjektima te kao povratnu vrijednost vraćaju modificirani subjekt što omogućuje lančanu primjenu operatora. Svaki sljedeći ulančati operator modificira objekt subjekta koji je proizašao kao rezultat prethodno izvršenog operatora.



Slika 5: Vizualizacija rada operatora (Prema: Ari, 2019)

Na slici iznad možemo vidjeti vizualizaciju načina rada operatora i njihove primjene. Ulančani operatori zajedno s tokom podataka tvore cjevovod kod kojeg se ulazne vrijednosti transformiraju što dalje teku cjevovodom. Slika iznad prikazuje jednostavne postepene promjene ulaznih vrijednosti od sivog pravokutnika pa sve do crvenog kruga. Kažemo da u cjevovodu postoje takozvane kontrolne točke odnosno operatori koji izvršavaju određene transformacije podataka te iste prosljeđuju sljedećim operatorima koji se nadovezuju i koji ponavljaju postupak sve dok se ne dođe do zadnjeg operatora koji oblikuje krajnji izlaz (Ari, 2019).

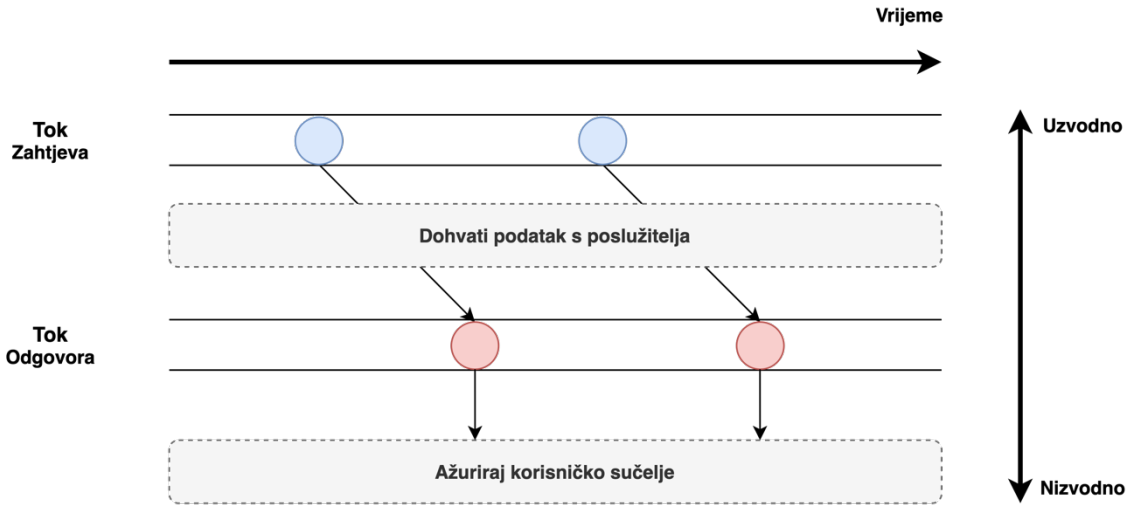
3.1.2.5. Raspoređivač

Većina reaktivnih okvira, iako se temelje na asinkronom načinu rada i dalje po standardu koriste samo jednu dretvu za rad. To znači da će promatrači biti obaviješteni na istoj dretvi na kojoj su se pretplatili na subjekt. Ako želimo iskoristiti puni potencijal asinkronosti u reaktivnom programiranju putem korištenje višedretvenosti za postizanje paralelnog izvršavanja, onda moramo koristiti dodatnu apstrakciju zvanu raspoređivači (eng. *Schedulers*). Raspoređivači predstavljaju mehanizam višedretvenosti koji omogućuje dodavanje i jednostavno korištenje dretvi u radu aplikacije. Na njih možemo gledati kao na bazene dretvi iz kojeg se uzima dretva na kojoj će se izvršavati zadaci. Detaljno objašnjenje raspoređivača s prikazom njihove primjene na razini programskog kôda moguće je pronaći u poglavlju ***5.3.4.6. Upravljanje dretvama***.

3.1.3. Asinkroni tokovi podataka

Kada koristimo reaktivno programiranje za razvoj programskog rješenja, tokovi podataka (eng. *Data streams*) predstavljaju kostur naše aplikacije. Događaji, poruke, pozivi, pa čak i pogreške se prenose putem tokova podataka. U reaktivnom programiranju, sve je tok podataka i sve kreira tokove podataka. Kako je sve tok i kako sve kreira novi tok te kako su tokovi međusobno neovisni, događa se jedna zanimljiva nuspojava, a to je da sama aplikacija postaje asinkrona (Escoffier, 2017). Asinkronost u računalnom programiranju definiramo kao pojavu događaja neovisnih o glavnom toku programa u ovom slučaju neovisnih o glavnoj niti programa. To u ovom kontekstu znači da se obrada događaja može dogoditi u proizvoljno vrijeme pa čak i u budućnosti što posljedično omogućuje neblokirajući način izvršavanja (Klang, 2016).

Kada pričamo o tokovima podatka oni nisu ništa drugo nego niz događaja poredanih vremenski po nastajanju. Na idućoj slici možemo vidjeti primjer realnih tokova neke web aplikacije. Prvi tok predstavlja tok zahtjeva i on sadrži podatke odnosno događaje nastale od strane korisnika, bilo da su ti događaji nastali putem klika mišem, pritiskom tipke na tipkovnici ili nekim trećim načinom koji generira određenu vrstu događaja. Nakon što je događaj kreiran, isti se emitira. Pretpostavimo da imamo neku komponentu u ulozi promatrača koja promatra tok zahtjeva. U web aplikacijama takva komponenta je kontroler. Kontroler zaprima emitirani događaj putem korisničkog toka te na njega reagira. Kako će kontroler reagirati na određeni tip događaja, u ovom slučaju HTTP zahtjeva, ovisi o poslovnoj logici, međutim pretpostavimo da se radi o jednostavnoj logici kod koje se poziva jednostavna komponenta servisa koja zatim poziva komponentu koja je zadužena za komuniciranje s bazom i dohvaćanja rezultata na temelju upita. U pravilu, svaka komponenta generira novi tok podataka te svaka komponenta nizvodno osluškuje komponentu uzvodno te reagira na njezine emitirane događaje i emitira svoje nove događaje u sklopu novog podatkovnog toka. Na slici je prikazan pojednostavljeni primjer takvog načina rada u kojem bismo proces „Dohvati podatke s poslužitelja“ mogli razbiti na veći broj manjih procesa kao što su slanje upita bazi podataka i emitiranje rezultata i slično. Međutim iz slike je važno vidjeti da tok zahtjeva i tok odgovora nisu ovisni jedan o drugome, što znači da se izvršavaju asinkrono. Dok se obrađuje jedan pristigli zahtjev, korisnička nit, odnosno glavna nit programa nije blokirana te aplikacija može zaprimiti idući događaj zahtjeva i krenuti ga izvršavati. Isto vrijedi i za sve ostale novo pristigle događaje.



Slika 6: Prikaz tokova i reagiranja na emitirane događaje (Prema: Escoffier, 2017)

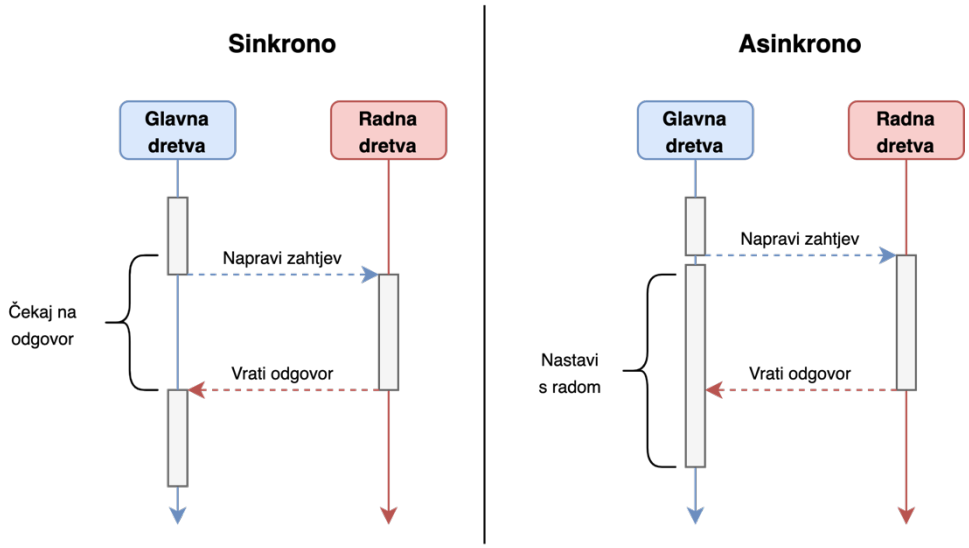
Laički rečeno u reaktivnom programiranju tok podataka možemo poistovjetiti s rijekom koju se može promatrati, filtrirati, manipulirati, spajati s drugim rijekama ali isto tako i kreirati sasvim novu rijeku iz jedne ili kombinacijom više postojećih. Stručnijim jezikom, to znači da tok podataka s emitiranim podacima može predstavljati ulazne vrijednosti za neki drugi tok, također je moguće filtrirati tok i raditi manipulacije nad njime. Više o mogućnostima upravljanja i manipuliranja tokovima bit će objašnjeno dalje u radu.

3.1.4. Neblokirajuće izvršavanje

Da bismo razumjeli kakvo je to neblokirajući izvršavanje, prvo moramo razumjeti kakvo je to blokirajuće izvršavanje i koji su sve načini za izbjegavanje njegove pojave.

Za blokirajuće izvršavanje kažemo da nastaje izvršavanjem programskog kôda koji uzrokuje blokadu izvršavanja ostatka dijela aplikacije zbog čekanja na izvršavanje trenutnog odsječka radi obavljanja neke intenzivne operacije. Za takav program kažemo da se izvršava na sinkroni/slijedni način. Primjer takvog programskog kôda može biti nešto jednostavno kao što su petlje koje su resursno intenzivne, a ne koriste konstrukte za izbjegavanje slijednog izvršavanja. Drugi primjer može biti slučaj pozivanje nekih vanjskih servisa, izvršavanje upita nad bazom podataka, čitanje i pisanje u datoteku, otvaranje mrežne konekcije, čitanje s utičnice (eng. *Socket*) i slično. Svi ovi primjeru uzrokuju čekanje dretve dok se trenutno aktivna operacija ne izvrši. Takav način rada rezultira aplikacijom slabih performansi, sporim izvršavanjem, a u konačnici može zamrznuti aplikaciju i učiniti ju ne odzivnom na duži vremenski period. Na kraju, blokirajuće metode predstavljaju značajnu prijetnju skalabilnosti sustava (Paul, 2017).

Prema Webber i Harrington (Webber & Harrington, 2019) postoje nekoliko načina koji mogu ublažiti pojavu programskog kôda koji blokira. Jedan od tih načina je primjenom konstrukata kao što su korištenje povratnih poziva ili obećanja koji omogućuju asinkrono izvršavanje programa. Korištenjem takvih konstrukata postiže oslobođenje glavne dretve tako da se izvršavanje operacije prepusti radnoj dretvi tako da glavna dretva može nastaviti neometano sa svojim izvođenje. Nakon što je operacija završena, radna dretva vratit će rezultat glavnoj dretvi. Međutim, korištenje navedenih konstrukata dodatno narušava strukturu programskog kôda i čini ga teško razumljivim. Na idućoj slici možemo vidjeti vizualni prikaz razlike između sinkronog i asinkronog izvršavanja opisanog u tekstu iznad.



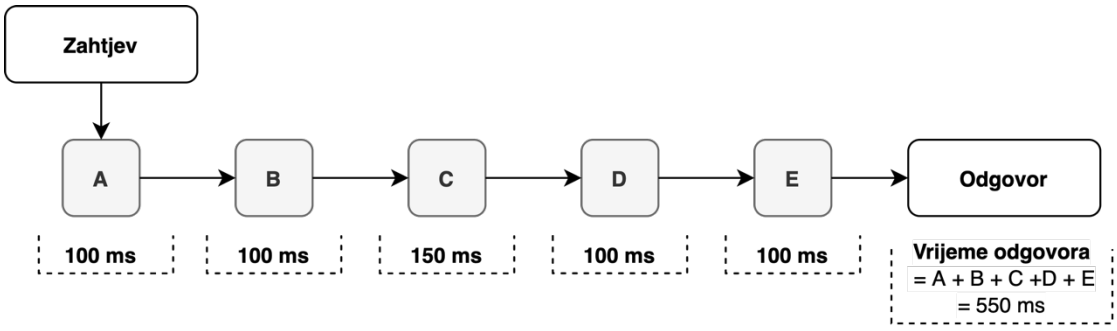
Slika 7: Prikaz razlike sinkronog i asinkronog izvršavanja s pogleda glavne i radne dretve

Drugi način kako možemo izbjeći pojavu blokiranja je korištenjem većeg broja dretvi za posluživanje većeg broja klijenata. Međutim ovaj način ima nekoliko nedostataka i zamki na koje moramo paziti. Prva zamka, definitivno predstavlja ograničenost resursa, ovisno o jačini hardvera, radne memorije i broja jezgri procesora, moguće je stvoriti samo određeni broj dretvi prije nego li se potroše svi računalni resursi. Druga zamka leži u samoj implementaciji višedretvenosti kod koje ako nismo pažljivi može rezultirati pojavom problema kao što su mrtvo-zaključavanje (eng. *Dead Lock*), utrku uvjeta (eng. *Race Conditions*), desinkronizaciju i ostalih pojava koje mogu narušiti konzistentnost aplikacije i u najgorem slučaju dovesti do zamrzavanja, a potom i do prestanka rada aplikacije.

Iako asinkroni kôd posjeduje svoje specifične probleme i nedostatke, također sa sobom donosi i neke prednosti. Te prednosti ogledaju se u većem broju aspekata. Jedan od tih aspekata je zasigurno vrijeme potrebno za izvršavanje određenog broja zadataka ili skupa zadataka. Vremenska razlika potrebna za obavljanje skupa zadataka najbolje se vidi kada se skup sastoji od velikog broja pojedinačnih zadataka koji su međusobno neovisni jedan o

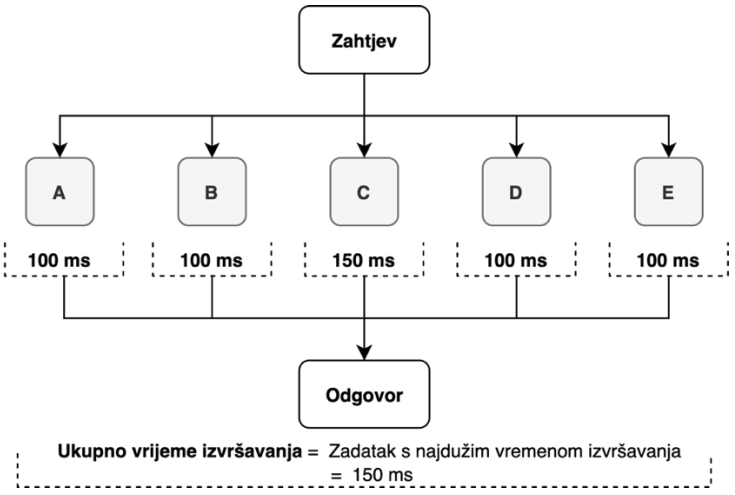
drugome. Pošto su zadaci neovisni, odnosno ne utječu jedan na drugog, logičkim razmišljanjem, dolazimo do zaključka da bi se ti zadaci trebali moći istovremeno izvršavati radije nego da se čeka završetak svakog pojedinog prije nego li se započne s izvršavanjem sljedećeg. Na taj način, izvršavanje cijelog skupa zadataka traje onoliko dugo koliko je potrebno da se izvrši vremenski najzahtjevniji zadatak, umjesto da je vrijeme izvršavanja jednako sumi vremena izvršavanja svih zadataka.

Na sljedećim dvjema slikama možemo vidjeti razliku u trajanju izvršavanja skupa zadataka na sekvencijalni način koji predstavlja blokirajući način te na asinkroni koji predstavlja neblokirajući način. Kod sekvencijalnog izvršavanja, možemo vidjeti da je ukupno vrijeme izvršavanja svih zadataka A, B, C, D i E jednako sumi vremena izvršavanja pojedinih zadataka što u ovom slučaju iznosi 550 milisekundi ili nešto malo više od pola sekunde. Ovo je primjer izvršavanja koje je kao što smo već prije rekli blokirajuće. Kada kažemo blokirajući način izvršavanja, to znači da su naši zadaci iako neovisni jedna o drugom u ovom slučaju postali ovisni jer svaki zadatak osim prvog je primoran čekati na završetak izvršavanja prethodnog. U slučaju da u slijedu zadataka imamo jedan koji predstavlja usko grlo (eng. *Bottleneck*) u smislu da njegovo izvršavanja traje puno duže nego ostalih, ti nedostaci sekvencijalnog izvršavanja postaju još izražajniji.



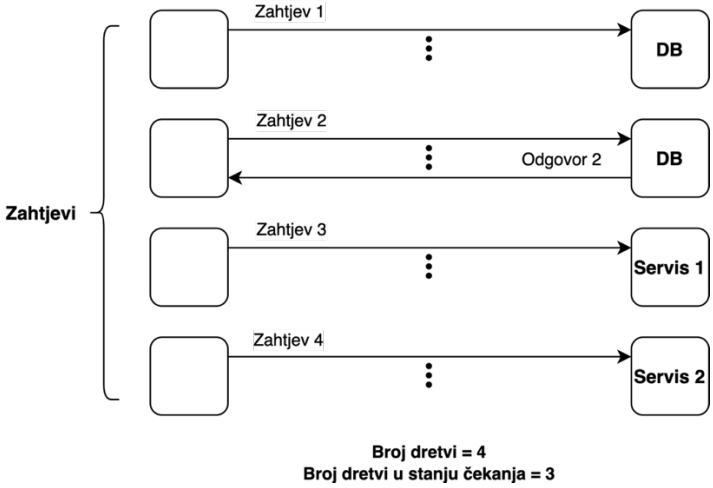
Slika 8: Prikaz vremena trajanja sekvencijalnog izvršavanja zadataka (Prema: Webber & Harrington, 2019)

Druga slika prikazuje primjer asinkronog načina izvršavanja koji je po prirodi neblokirajući. Isti zadaci s istim vremenom izvršavanja u ovom slučaju bili su gotovi za vrijeme koje je jednako vremenu izvršavanja vremenski najdužeg zadatka. U ovom primjeru to je zadatak C za koji je potrebno 150 milisekundi da se izvrši. Stoga ukupno vrijeme potrebno do dobivanja odgovora iznosi također 150 milisekundi, što je 3.6 puta brže vrijeme odgovora nego što je to bilo u prethodnom primjeru. Zaključujemo, ako je programski kôd napisan na asinkroni način, on je ujedno i neblokirajući.



Slika 9: Prikaz vremena konkurentnog izvršavanja zadataka (Prema: Webber & Harrington, 2019)

Idući aspekt koji možemo promatrati je aspekt iskoristivosti računalnih resursa potrebnih za rad same aplikacije. Kod tradicionalnog blokirajućeg kôda, velika je šansa da će se desiti blokiranje dretve uzrokovane čekanjem izračuna/dohvaćanja traženog rezultata. Međutim, kako je dretva stvorena, iako se trenutno ne koristi, ona će i dalje trošiti procesorske cikluse budeći se i provjeravajući da li je izvršavanje završeno te se vraćajući u stanje čekanja sve dokle god rezultat ne pristigne. Na idućoj slici vidimo kako to u stvarnosti može izgledati.



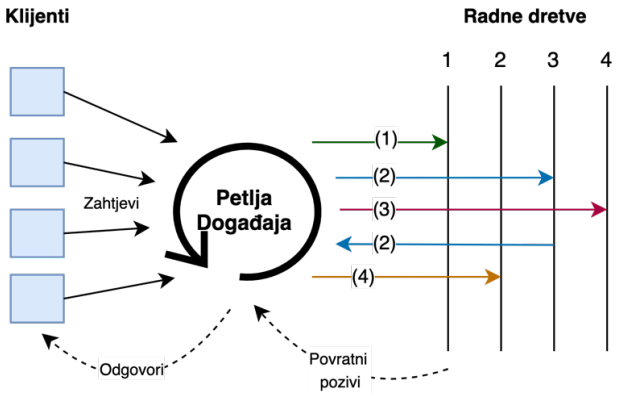
Slika 10: Prikaz blokirajućeg izvršavanja s gledišta dretvi (Prema: Webber & Harrington, 2019)

Iz slike vidimo da je aplikacija dobila četiri zahtjeva u istom vremenskom periodu, prva dva zahtjeva vezana su za dohvaćanje traženog podatka iz baze podataka, dok druga dva

zahtjeva komuniciraju s nekim servisom za obradu podataka. Kako su sva četiri zahtjeva stigla u vrlo kratkom vremenskom razmaku, server je morao iskoristiti četiri različite dretve kako bi bio u mogućnosti poslužiti sva četiri zahtjeva u tom trenutku. U ovom slučaju dogodilo se da je samo drugi zahtjev po redu uspio završiti te se s njegovim završetkom dodijeljena dretva oslobodila i vratila u bazen slobodnih dretvi. Ostale tri dretve za preostale zahtjeve završile su u stanju čekanja i u tom stanju će ostati sve dokle god ne dobiju odgovor. To je s gledišta iskoristivosti resursa veoma loše jer se preostale tri dretve koje su u stanju čekanja ne mogu ponovo iskoristiti za obradu drugih pristiglih zahtjeva. To pak znači da se naš bazen dretvi efektivno smanjio. U slučaju da je bazen u potpunosti iskorišten, u smislu da su sve raspoložive dretve blokirane, postoje vrlo velike šanse da idući zahtjev koji pristigne neće moći biti obrađen što će rezultirati ili njegovim ispuštanjem ili dugim vremenom čekanja na oslobođenje dretve.

Većina programskih okvira i rješenja danas za zadovoljavanje kriterija neblokirajućeg izvršavanja implementiraju jedno od sljedećih dva rješenja („Event Driven and Reactive Architecture“, 2015). Oba načina rješavaju problem obrade pristiglih zahtjeva na svoj način:

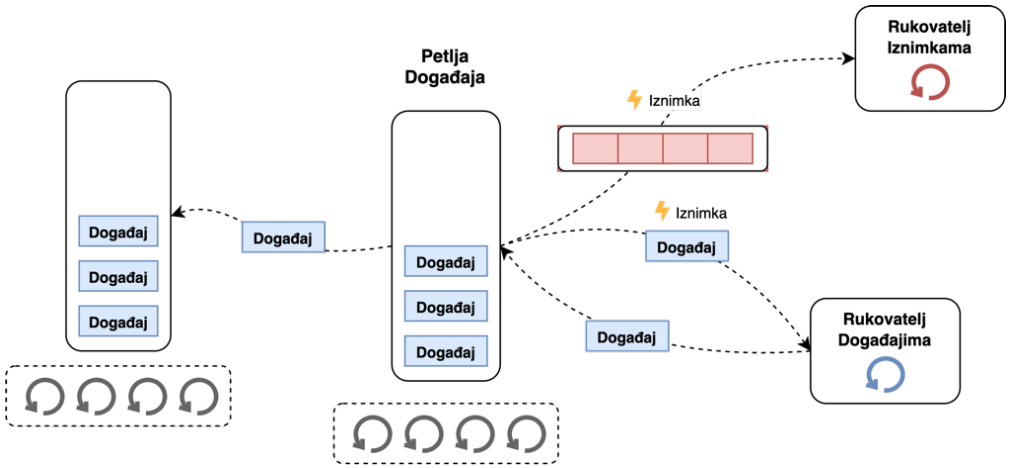
- **Petlja događaja** (eng. *Event Loop*) – Petlja događaja u suštini predstavlja jednu dretvu koja izvršava i obavještava proces koji se sljedeći treba izvršiti na način da ne blokira ostale. Stoga umjesto da se koristi veći broj dretvi koji rukuju ulaznim/izlaznim operacijama te koje troše resurse dok se nalaze u stanje čekanja, koristi se samo jedna poslužiteljska dretva koja se naziva petlja događaja. Primjer ovakve implementacije možemo pronaći u okvirima kao što su Verte.x i node.js.



Slika 11: Prikaz koncepta petlje (Prema: outsource, 2018)

- **Bazen dretvi temeljen na događajima** – Mehanizam temeljen na događajima u suštini sadrži i koristi mali broj dretvi, osim dretvi obično koristi i dodatne virtualne konstrukte kao što su redovi ili akteri (eng. *Actors*) u sklopu nadogradnje na okvir zaduženog za upravljanje dretvama kako bi se postiglo što efikasnije rukovanje s dretvama ali i s događajima. Kao i u prethodnom primjeru, možemo primijetiti implementaciju petlje događaja koja prima, održava

i upravlja s događajima namijenjenih za određene rukovatelje koji obrađuju te događaje. Rukovatelj može reagirati na događaj na dva načina, uspješno ili s greškom koja će uzrokovati generiranja iznimke. Iznimka se također prosljeđuje petlji događaja kao i svaki drugi događaj. Primjeri implementacije ovakvog mehanizma mogu se pronaći u okviru Akka te reaktivnoj biblioteci napravljene od strane Pivotala za korištenje u njihovom okvira za izradu web aplikacija baziranih na Java programskom jeziku. Okvir se naziva Spring WebFlux, a reaktivna biblioteka nosi naziv Reactor.



Slika 12: Prikaz bazena dretvi temeljen na događajima (Prema: „Event Driven and Reactive Architecture“, 2015)

3.2. Prednosti

Prema Latcu (Latcu, 2017) prednosti reaktivnog programiranja prvenstveno se očituju u tome što je reaktivno programiranje deklarativna programska paradigma koja se temelji na radu s tokovima podataka. Pošto spada u deklarativnu paradigmu, to znači da je programski kôd napisan na reaktivan način jednostavniji i čitljivi ljudima jer govori priču što želiš učiniti, a ne kako želimo nešto učiniti, kao što je to slučaj kod imperativnih paradigmi. Zbog toga što se reaktivno programiranje temelji na asinkronom načinu te zbog toga što tokovi podataka predstavljaju okosnicu same paradigme, svi reaktivni okviri i biblioteke nude pozamašan skup apstrakcija u obliku operatora koji olakšavaju programerima rad i programiranje. Također, zbog asinkronog načina, reaktivno programiranje donosi prednosti i u području performansi u situacijama kada se aplikacija nalazi pod konkurentnim opterećenjem. Ostale prednosti koje se mogu istaknuti jesu: olakšani višedretveni rad, jednostavnost kod implementacije koncepta povratnog tlaka, izbjegavanje pojave *pakla povratnih poziva* i lakše rukovanje greškama. U idućem potpoglavlju prikazat ćemo razliku između sinkronog, asinkronog i programskog kôda napisanog primjenom reaktivnog programiranja uz korištenje tokova podataka (eng. *Streams*).

3.2.1. Jednostavnije asinkrono programiranje i čitljiviji kôd

Asinkrono programiranje može se pokazati kao iznimno težak zadatak i za iskusnije programere. Ovisno o složenosti poslovne logike i iskustvu programera, vrlo često se kod rada s asinkronim programskim kôdom dešava pojava zlouporabe, odnosno pretjeranog korištenja povratnih poziva (eng. *Callback*). Razlog tomu je što većina programera pokušava konstruirati asinkroni kôd na takav način da se dobije vizualna predodžba da se kôd izvršava redoslijedom od gore prema dolje, što kod asinkronog programiranja nije uvijek primjenjivo. Pretjerano korištenje povratnih poziva u ove svrhe dovodi do višestrukog ugnježđivanja povratnih poziva koji kreiraju kompleksne strukture koje su vrlo teško čitljive, a samim time čine i tijek izvršavanja programa nerazumljivim. Takva pojava u industriji razvoja softvera naziva se *pakao povratnih poziva* (eng. *Callback Hell*) (Kambona i ostali, 2013).

Većina programskih jezika, biblioteka i okvira nudi svoje apstrakcije koje olakšavaju rad s asinkronim programskim kôdom. U Javi neki od tih konstrukata su klase Future i Promise koje olakšavaju programerima rad na način da pružaju dodatnu razinu apstrakcije. Međutim, u nastavku se nećemo time baviti, već ćemo prikazati tri programska isječka koja će nam pokazati razlike u strukturi i čitljivosti kôda napisanog na sinkroni način, asinkroni način koristeći povratne pozive i asinkroni način koristeći reaktivno programiranje.

```
Product product = getProductFromDB(id);
product = convertProduct(product);
product = processResult(product);
showResult(product);
```

Kôd 1: Primjer sinkronog kôda za dohvaćanje i obradu rezultata

```
getProductFromDB(id, product ->{
    convertProduct(product, convertedProduct -> {
        processResult(convertedProduct, processedProduct -> {
            showResult(processedProduct);
        });
    });
});
```

Kôd 2: Primjer asinkronog kôda za dohvaćanje i obradu rezultata

```
getProductFromDB(id)
        .map(product -> convertProduct(product))
        .map(convertedProduct -> processResult(convertedProduct))
        .subscribe(processedProduct -> showResult(processedProduct));
```

Kôd 3: Primjer reaktivnog kôda za dohvaćanje i obradu rezultata

Kao što možemo vidjeti iz programskih isječaka. Sinkroni programski kôd je vrlo lagano za pratiti i razumjeti jer se izvršava slijedno od najgornje instrukcije prema najdonjoj. U drugom programskom isječku, sinkroni programski kôd napisali smo na asinkroni način koristeći povratne pozive. Već na prvi pogled možemo primijetiti da nam za razumijevanje svrhe i samog tijeka izvršavanja treba puno više vremena nego što je to bilo u prvom primjeru. Ovaj primjer predstavlja jednostavnu poslovnu logiku, međutim ona može biti daleko kompleksnija. Zadnji primjer nam prikazuje isti asinkroni kôd napisan uz pomoć korištenja tokova podataka (eng. *Streams*) i reaktivnog programiranja. Ovaj programski kôd je puno čitljiviji i jednostavniji za razumjeti jer je fino strukturiran i prati vizualizaciju izvršavanja od gore prema dolje.

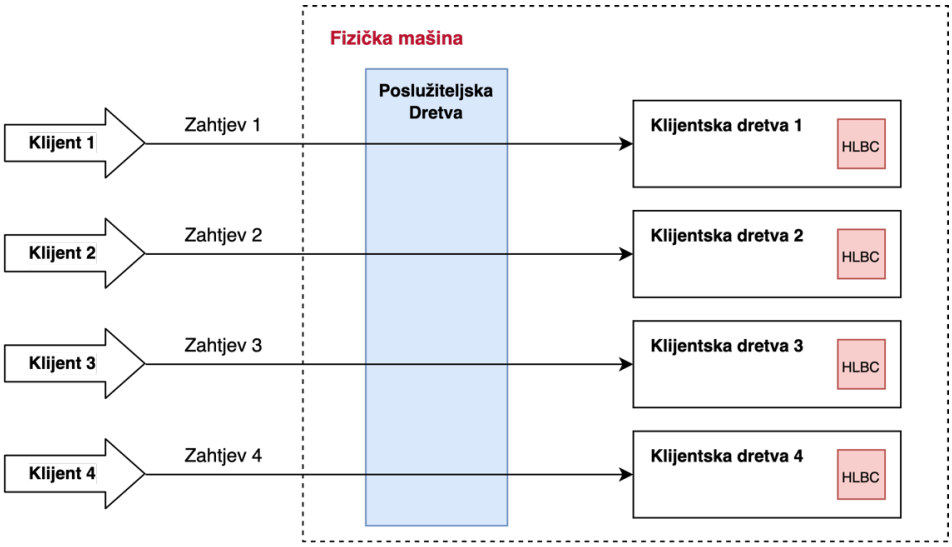
Razlog zašto je zadnji primjer strukturiraniji od ostalih leži u tome što se reaktivno programiranje temelji na radu s tokovima podataka te ujedno podržava korištenje operatora kao što je operator `map` (operatori su detaljnije obrađeni u poglavlju *5.3.4 Operatori*) koji predstavlja koncept funkcionalnog programiranja, a koji se koristi za transformaciju objekta. Funkcija `getProductFromDB` vraća reaktivni tip podataka koji predstavlja niz vrijednosti (detaljnije o reaktivnim tipovima objašnjeno je u poglavlju *5.3.1 Flux i Mono*) koji će se u ovom slučaju sastojati od samo jedne vrijednosti u obliku objekta proizvoda. Nad tim objektom vrši se prva transformacija uz pomoć `map` operatora gdje dohvaćeni model proizvoda pretvaramo u nama neki koristan oblik podatka. Nakon prve transformacije, odmah nadovezujemo i drugu transformaciju u kojoj možemo pretvoreni podataka provući kroz određenu logike provjere ili slično i na temelju toga ponovo transformirati ili ažurirati dohvaćeni proizvod. Na kraju koristimo operator `subscribe` specifičan za reaktivno programiranje koji povezuje objekt izdavača i pretplatnika koji reagira na emitirane vrijednosti.

3.2.2. Bolje performanse

U ovom poglavlju objasnit će se zašto su performanse programskih okvira koji koriste reaktivno programiranje bolje od performansa okvira koji koriste sinkroni model izvršavanja u slučajevima visoke konkurentnosti.

Naime, većina današnjih poslužitelja kao što je Tomcat i dalje koristi pristup „Dretva po konekciji“ (eng. *Thread-per-connection*) odnosno arhitekturu temeljenu na dretvama. To znači da će svaki zahtjev koji pristigne na server biti dodijeljen posebnoj dretvi koja će biti zadužena za izvršavanje pojedinačnog zahtjeva. Takav model rada aplikacije možemo vidjeti na sljedećoj slici 13 koja prikazuje model rada u kojem pojedinačni zahtjev klijenta biva dodijeljen zasebnoj radnoj dretvi iz bazena dostupnih dretvi. Ovakav način rada je iznimno resursno intenzivan u slučajevima kada pristiže veliki broj zahtjeva od korisnika jer se za obradu svakog novog pristiglog zahtjeva koristi zasebna dretva iz bazena dostupnih dretvi. Skraćenica HLBC unutar klijentske dretve predstavlja poslovni slučaj visoke razine (eng. *HLBC - High Level*

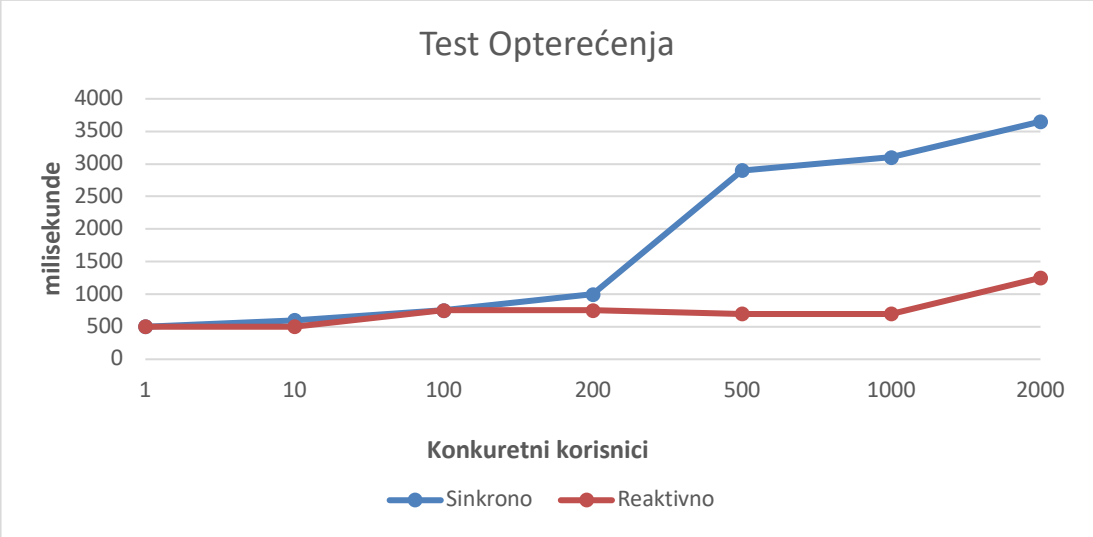
Business Case) kojeg dretva izvršava. U slučajevima kada se sve dretve iscrpe iz bazena dostupnih dretvi pojavljuje se čekanje što uzrokuje duže vrijeme potrebno za odgovor klijentu. Ovaj problem moguće je riješiti na više načina, ili skaliranjem vertikalno ili horizontalno aplikacije, ili implementacijom neblokirajućeg načina izvršavanja primjenom modela petlje događaja ili modelom bazena dretvi temeljenim na događajima. Ovi primjeri objašnjeni su u poglavlju 3.1.4 Neblokirajući izršavanje te se isti neće dodatno objašnjavati.



Slika 13: Model dretva po zahtjevu (Prema: Dmytro Melnychuk, 2019)

Kako bi se lakše vidio nedostatak primjene modela dretve po zahtjevu u primjeni sinkronog izvršavanja naspram neblokirajućeg asinkronog izvršavanja u situacijama visoke konkurentnosti, dati će se realni primjer na java Spring web aplikaciji napisane koristeći Spring Boot okvir za razvoj. Prvi slučaj predstavljat će jednostavnu web aplikaciju napisanu na temelju Spring MVC okvira koristeći ugrađeni Tomcat poslužitelj s maksimalnim brojem dretvi od 10.000 koji dolazi sklopu okvira. Spring MVC okvir temelji se na sinkronom izvršavanju, a sam Tomcat server radi na principu dretva po zahtjevu. U kontrastu, drugi slučaj će također predstavljati java Spring web aplikaciju, međutim, ova aplikacija napisana je koristeći Spring WebFlux okvir za reaktivno programiranje. Reaktivna aplikacija koristi za rad Netty poslužitelj koji radi na neblokirajućem principu, odnosno arhitekturi baziranoj na događajima. Obje aplikacije simulirat će poziv vanjskog servisa s definiranim kašnjenjem od 500 milisekundi. Obje aplikacije promatraju se s gledišta koliko korisničkih zahtjeva mogu obraditi istovremeno prije nego li se vrijeme odziva počne pogoršavati, odnosno povećavati. Na sljedećim grafikonima možemo vidjeti rezultate u slučaju kada je vrijeme odziva za poziv vanjskog servisa pola sekunde i drugog slučaja gdje je vrijeme odziva vanjskog servisa 2 sekunde. U oba grafikona, x-os predstavlja broj konkurentnih korisnika počevši s minimalnim brojem od jednog korisnika, pa sve do 2000 istovremenih korisnika s pretpostavkom da svaki korisnika

generira točno jedan zahtjev. Y-os prikazuje vrijeme odaziva servisa, odnosno koliko je dugo trebalo da korisnik dobije odgovor od poslužitelja nakon što je zaprimljen zahtjev. Vrijednosti su prikazane u milisekundama gdje 1000 milisekundi iznosi 1 sekunda (Trincao Cunha, 2018).



Grafikon 1: Prikaz opterećenja sinkrone i reaktivne aplikacije s vremenom odaziva od 500 milisekundi (Prema: Trincao Cunha, 2018)

Iz grafikona možemo vidjeti da obje aplikacije, sinkrona i reaktivna imaju relativno jednako vrijeme odaziva za posluživanje do 100 konkurentnih korisnika. Razliku primjećujemo nakon što broj konkurentnih korisnika prijeđe brojku od 100. Što je brojka konkurentnih korisnika veća to se sve više primjećuje razina degradacije sinkrone aplikacije naspram reaktivne s gledišta vremena potrebnog za odziv. Kod brojke od 500 konkurentnih korisnika vrijeme odaziva sinkrone aplikacije skače na pune 3 sekunde, dok za isti broj korisnika vrijeme odziva za reaktivnu aplikaciju ostaje stabilno pri 750 milisekundi. Pri najvećem broju konkurentnih korisnika od 2000, vrijeme odziva za sinkronu aplikaciju probilo je vrijeme od 3.5 sekundi, dok reaktivna aplikacija, iako isto pokazuje skok na skoro pa sekundu i pol, i dalje daleko bolje podnosi opterećenje.

3.2.3. Bolje rukovanje iznimkama i pogreškama

Rukovanje s greškama u reaktivnom programiranju temelji se na tri konstrukta. Prva dva koncepta većini programera su poznati jer su prisutni u većini programskih jezika. To su iznimke i pogreške. Važno je spomenuti da iznimke i pogreške ne predstavljaju isto.

Iznimka: Označava uvjete koje bi odgovorna aplikacija htjela uloviti i time spriječiti rušenje aplikacije tako da se iznimka programski obradi. Ovisno o programskom jeziku, mogu se definirati dvije vrste iznimka: provjerene i neprovjerene iznimke međutim njihova razlika nije od veće važnosti za ovaj rad.

Pogreška: Predstavlja ozbiljan problem za koji aplikacija nema programskih mogućnosti za oporavak, kao što je to slučaj kod iznimaka te takve pogreške trebaju biti u stanju terminirati program. Primjeri takvih pogrešaka su greške nedostatka memorije (eng. *OutOfMemoryError*) i prelijevanje stoga (eng. *StackOverflowError*).

Ista pravila vezana uz iznimke i pogreške vrijede i u svijet reaktivnog programiranja. Međutim postoji jedna specifična razlika u odnosu na ostale paradigme, a to je da su pogreške i iznimke enkapsulirane u dodatnu razinu apstrakcije zvanu signal pogreške i to predstavlja naš treći konstrukt. Signal pogreške nije isto što i obična pogreška i ne bi se trebao poistovjećivati. U reaktivnom svijetu, signali pogreške spadaju u koncepte prve klase (eng. *First-Class citizens*) što znači da podržavaju sve operacije kao i ostali entiteti te grupacije nekog programskog jezika. Važna karakteristika signala pogreške je da označava terminalni događaj, odnosno događaj koji označava prestanak emitiranja podataka nakon što je emitiran, neovisno o tome da li se koristi operator za rukovanjem pogreškom. Ovisno o korištenju određenog okvira i biblioteke, postoji više načina na koji se može rukovati pogreškama uz korištenje određenih operatora (Senanayake, 2019).

3.2.4. Jednostavniji rad s povratnim tlakom

Povratni tlak (eng. *Backpressure*) predstavlja analogiju koja je preuzeta iz stručnog polja dinamike fluida. U dinamici fluida, definicija povratnog tlaka glasi na sljedeći način:

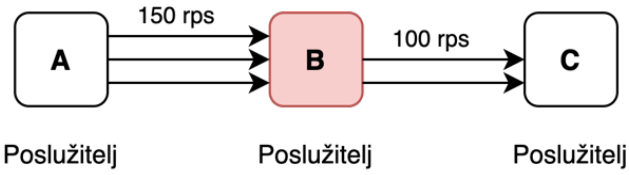
„Povratni tlak se definira kao otpor ili sila koja se suprotstavlja željenom protoku tekućine kroz cijev.“ (Phelps, 2020).

Istu definiciju uz sitne preinake Phelps koristi i za opis povratnog tlaka u kontekstu razvoja softvera:

„Povratni tlak se definira kao sila koja se suprotstavlja željenom protoku podataka kroz aplikaciju, odnosno sustav.“ (Phelps, 2020).

S praktične strane gledišta, povratni tlak predstavlja situacije kod kojih se za vrijeme izvršavanja aplikacije događa pojava sprječavanja ili opiranja pretvorbe ulaznih podataka u izlazne rezultate. U većini aplikacija takva vrsta otpora se pojavljuje kao nenamjerna nuspojava nemogućnosti aplikacije da na dovoljno brzi način obradi i pruži rezultate za pristigle ulazne podatke. Ovo nije jedina situacija koja može uzrokovati pojavu povratnog tlaka u radu aplikacija. Ostali vrlo česti primjeri koji također uzrokuju nenamjernu pojavu povratnog tlaka jesu operacije čitanja i pisanja u datoteke, komunikacija između većeg broja poslužitelja te kada tehnologija za ažuriranje korisničkog sučelja ne može istom brzinom ažurirati prikaz kao i brzinom kojom dobiva nove podatke. Na sljedećoj slici možemo vidjeti situaciju kada imamo slijednu komunikaciju između tri poslužitelja A, B i C gdje poslužitelj A šalje zahtjeve

poslužitelju B koji ih potom obrađuje i šalje izlazne rezultate kao ulazne zahtjeve poslužitelju C. Iz slike vidimo da poslužitelj A radi 150 rsp[2]-a prema poslužitelju B. Međutim, poslužitelj B, zbog potrebe za obradom ulaznih zahtjeva ne može na dovoljno brzi način poslati rezultate na poslužitelj C što uzrokuje njegovu mogućnost slanja od samo 100 rps-a, što je 50 rps-a manje nego na ulazu. Kažemo da komunikacija između poslužitelja doživljava posljedice povratnog tlaka u poslužitelju B koji uzrokuje otpor u toku komunikacije.



Slika 14: Pojava sile otpora u komunikaciji između više poslužitelja

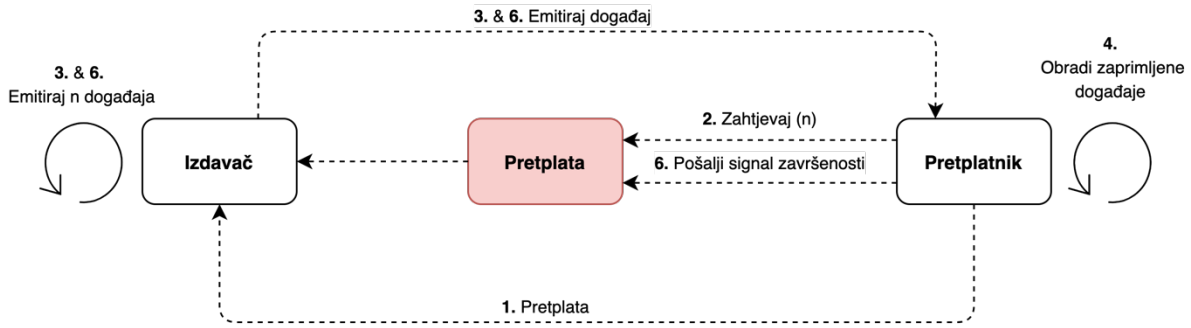
Problemi povratnog tlaka prema Phelps (Phelps, 2020) mogu se riješiti korištenjem jedne od četiri mogućih strategija:

- **Vertikalno ili horizontalno skaliranje** (eng. *Vertical/Horizontal scaling*): Skaliranje aplikacije vertikalno rezultira povećanjem raspoloživih resursa u smislu jačeg hardvera kojeg će aplikacija moći dodatno iskoristiti za obradu povećanog broja dolaznih zahtjeva. Skaliranje aplikacije horizontalno predstavlja rješenje u kojem će se stvoriti nove instance aplikacije koje će služiti za preraspodijele dolaznog prometa s ciljem smanjenja zasićenosti pojedine komponente. Ovaj način u većini slučajeva iziskuje dodatne troškove vezane uz nabavu dodatne hardverske opreme ili zakupljivanje dodatnih resursa na poslužitelju.
- **Akumuliranje putem spremnika** (eng. *Buffering*): Akumuliranje podataka je rješenje na razini programa u kojem se dolazni ili izlazni podaci spremaju u određenu vrstu spremnika (lista, polje i slično) na strani poslužitelja ili klijenta kako bi se spriječio gubitak podataka koji se može desiti zbog nemogućnosti jedne strane da se prilagodi drugoj s brzinom obrade podataka. Ovaj način ne dolazi kao ugrađena značajka i u većini okvira ju je potrebno samostalno implementirati. Ova strategija je također iznimno opasna ako se za spremanje podataka koriste memorijski i/ili vremenski neograničeni spremnici. Neispravno korištenje neograničenih spremnika su česti uzroci pada poslužitelja zbog iscrpljivanja sve raspoložive radne memorije fizičke mašine.

[2] rsp (eng. *Request per seconds*) - broj napravljenih zahtjeva u sekundi

- **Ispuštanje** (eng. *Dropping*): Ova strategija predstavlja ispuštanje dolaznih podataka koji nakon ispuštanja postaju bespovratno izgubljeni. Ova strategija koristi se u kombinaciji s akumuliranjem putem ograničenih spremnika nakon što se isti napune.
- **Upravljanje proizvođačem** (eng. *Producer control*): Zadnja strategija predstavlja ujedno i najbolju opciju od ponuđenih u slučajevima kada postoji mogućnost za njenom implementacijom. Ideja iza upravljanja proizvođačem je ta da komponenta potrošača, odnosno nizvodna komponenta koja prima podatke, na temelju svoje mogućnosti za obradom podataka, obavještava uzvodnu komponentu proizvođača kako bi ista usporila slanje podataka. Korištenjem ove strategije, nije potrebno koristiti nikakve dodatne memorijske spremnike niti iziskuje ispuštanje podataka.

Prethodno navedeni primjeri predstavljaju situacije u kojima se događa nenamjerna pojava povratnog tlaka. Međutim, kao razvojnim inženjerima kojima je u cilju izgradnja robusnog i odzivnog sustava takve situacije se ne bi smjele događati. Štoviše, na mehanizam povratnog tlaka trebali bi gledati s aspekta njegove primjene s ciljem sprečavanja pojave zasićenosti komponenti potrošača sustava na prvom mjestu. U tom slučaju, spomenuti mehanizam predstavlja rješenje koje nam omogućuje da krajnje komponente potrošača ali i posredničkih operatora da zahtijevaju određeni broj događaja od strane proizvođača. U tom slučaju kontrola toka podataka se djelomično prebacuje na nizvodne komponente te se princip rada prebacuje s čistog modela guranja gdje primjerice hladni proizvođač emitira događaje što brže može na kombinaciju modela guranja i povlačenja gdje nizvodne komponente zahtijevaju od uzvodnog proizvođača određeni broj događaja. Koraci ovog procesa mogu se prikazati na sljedeći način.



Slika 15: Prikaz rada mehanizma povratnog tlaka u reaktivnom programiranju (Prema: Mandar jog, 2017)

Objašnjenja koraka na dijagramu su sljedeći:

1. **Pretplatnik**: Pretplaćuje se na izdavača.
2. **Pretplatnik**: Definira broj događaja koji želi zaprimiti.
3. **Izdavač**: Nakon pretplate pretplatnika, izdavač počinje emitirati događaje. Emitira događaje vlastitom brzinom, jedan po jedan, dok ne emitira onoliki broj događaja koje je pretplatnik definirao, nakon toga staje s emitiranjem.
4. **Pretplatnik**: Zaprima emitirane događaje i obrađuje ih.
5. **Pretplatnik**: Nakon što je obradio sve pristigle događaje, šalje signal završenosti izdavaču.
6. **Izdavač**: Nakon zaprimanja signala završenosti od pretplatnika počinje ponovo emitirati događaje.
7. Proces se ponavlja dok se proizvođač ne iscrpi ili potrošač ne odjavi.

Implementacijom ovog načina, sprječava se mogućnost dešavanja zasićenosti komponente pretplatnika jer sama komponenta određuje koliko događaja može obraditi te na temelju toga određuje i broj događaja koje želi zaprimiti. Komponenta izdavača, iako ima spremne događaje za emitiranje, neće emitirati više događaja nego što je zahtijevano sve dok ne primi signal od nizvodne komponente da je gotova s obradom prethodnih. Razlog lagane implementacije mehanizma pomoću reaktivnog programiranja proizlazi iz toga što se mehanizam temelji na istim konceptima kao i reaktivno programiranje. Štoviše, reaktivni okviri posjeduju već ugrađene apstrakcije povratnog tlaka u sklopu operatora i sučelja što znači da se programeru utoliko olakšava posao što se ne moraju baviti tehničkom implementacijom jezgre mehanizma (Christensen & Nurkiewicz, 2016).

3.3. Nedostaci

Kao i sve, tako i primjena reaktivnog programiranja posjeduje svoje prednosti ali i nedostatke. Ovisno o scenariju i problemu za čije se rješenje reaktivno programiranje pokušava primijeniti tih nedostataka može biti manje ali i više. Kako ne bismo išli previše u širinu i detalje, pretpostavit ćemo da je sustav zamišljen na takav način da njegova implementacija korištenjem reaktivnog programiranja odgovara svim postavljenim zahtjevima. Zbog toga ćemo navesti samo zapažene nedostatke s aspekta same primjene reaktivnog programiranja, a ne i nedostatke koje donosi primjena reaktivnog programiranja u korištenju nad krivim domenama problema. Međutim, spomenut ćemo samo da primjena reaktivnog programiranje u domene koje ne iskorištavaju u potpunosti prednosti koje ono donosi, utječe na povećanja kompleksnosti sustava i uvođenje dodatnog napora. Uzevši to u obzir,

nedostataka reaktivnog programiranja nema puno, jer ipak ono predstavlja rješenje osmišljeno za rješavanje specifičnih problema.

Veliki nedostatak s kojim se većina programera susreće kod korištenja reaktivnog programiranja je krivulja učenja koja može biti veoma strma. Strma krivulja učenja postoji iz dva razloga. Prvi razlog leži u tome što se reaktivna paradigma temelji na asinkronim tokovima podataka, koji se podosta razlikuju od programiranja temeljenim na sinkronim instrukcijama. Većini programera teško je pojmiti da u reaktivnom programiranju sve predstavlja tok podataka i taj proces prebacivanja na takav način razmišljanja i pisanja programskog kôda može potrajati.

Drugi razlog je taj što za sada još uvijek nedostaje dobre i kvalitetne literature koja bi dala potpuniju i detaljniju sliku reaktivnog programiranja. Zbog toga, većina programera završi s polovičnim razumijevanjem koncepata što posljedično uzrokuje lošu implementaciju sustava. To zatim povlači dodatni nedostatak, koji se ogleda u teškoći pronalaženja grešaka. Pošto je sve tok podataka, i k tome asinkrono, vrlo je teško pa čak i nemoguće pratiti tok izvođenja programa što rezultira vrlo teškim pronalaženjem logičkih grešaka.

Idući nedostatak, djelomično se nadovezuje na nedostatak literature, a to je da se reaktivnog programiranje u raznim literaturama često krivo poistovjećuje s funkcionalno reaktivnim programiranje, koje predstavlja paradigmu samu za sebe. To dovodi do frustracije i poteškoće u razumijevanju koncepata jer se čitače navodi na krive zaključke. Idući nedostatak odnosi se na korištenje računalnih resursa. Kako je u reaktivnom programiranju sve tok podataka, to znači da ovisno o situaciji, rad aplikacije ili sustava može postati memorijski intenzivan zbog potrebe za njihovom pohranom.

3.4. Utjecaj na dizajn arhitekture klasa

U prethodnim poglavljima prvenstveno smo pričali o reaktivnom programiranju s aspekta konstrukata koje čine reaktivno programiranje, aspekta teorije i principa koji se nalaze u pozadini reaktivne paradigme te o utjecaju reaktivnog programiranja na performanse i rad same aplikacije. Međutim nismo se dotakli previše promjena koje ono donosi u strukturu naše aplikacije s točke gledišta programskog kôda, povezanosti klasa i promjena ovisnosti objekata aplikacije. U ovom poglavlju pozabavit ćemo se načinom na koje reaktivno programiranje utječe i mijenja način implementacije poslovne logike sustava.

Vrlo jednostavan primjer može nam pomoći u dočaravanju promjena koje reaktivno programiranje donosi u kontekstu ovisnosti objekata ali i povezanosti klasa. Uzmimo za primjer implementaciju logike prekidača i žarulje. Ovaj primjer prikazuje nam međusobnu povezanost

dvaju entiteta čija se povezanost manifestira na način da promjenom stanja jednog utječemo na promjenu stanja drugog entiteta. U ovom kontekstu to znači da akcija prebacivanja prekidača u različite položaje rezultira promjenom rada same žarulje gdje kada se prekidač nalazi u jednoj poziciji žarulju radi ili daje svjetlost dok u drugoj poziciji je žarulja ugašena i ne daje svjetlost. Ovo je tipični školski primjer koji se koristi za opisivanje različitih GOF (eng. *Gang of Four*) uzoraka dizajna kao što su uzorak stanja (eng. State Pattern) i uzorak mosta (eng. Bridge Pattern). Implementacijom ovog primjera korištenjem bilo kojeg uzorka rezultirat će klasama koje će biti do određene razine međusobno povezane. Povezanost klasa možemo klasificirati u dvije krajnosti: čvrsto povezane i slabo povezane. **Čvrsta povezanost** klasa znači da su klase veoma ovisne jedna o drugoj što kao posljedicu ima sljedeće nuspojave: Promjena jedne klase uobičajeno uzrokuje prisilnu promjenu ostalih povezanih klasa te utječe na težu ponovnu iskoristivost pojedine klase. Drugim riječima, ako klasa A zna više o implementacije klase B nego što bi trebala, promjenom klase B rezultirat će posljedičnom promjenom i klase A. Pravilo je da se ako je ikako moguće izbjegne pojava čvrste povezanosti radi prethodno navedenih nuspojava. Umjesto čvrste povezanosti, preporučeno je da se kod pisanja programskog rješenja teži ostvarivanju slabe povezanosti. **Slaba povezanost** s druge strane sugerira da su klase većinom neovisne jedna o drugoj. Ako klasa A zna samo koje metode klasa B pruža kroz svoje sučelje, onda kažemo da su klase A i B slabo povezane. Promjene unutarnje implementacije postojećih javnih metoda klase B neće rezultirati posljedičnom promjenom klase A.

Međutim, da se vratimo na naš primjer prekidača i žarulje. Jedan način na koji se ovaj primjer može implementirati je korištenjem OOP paradigme i uzorka stanja u kojem objekt mijenja svoje ponašanje na temelju akcije prijelaza stanja/naredbe koje prima od drugog objekta. U ovom primjeru to znači da prekidač modificira stanje žarulje na proaktivan način, odnosno da gura novo stanje prema žarulji koja je za razliku od prekidača, pasivna komponenta. Za entitet žarulje kažemo da je pasivan zbog toga što na promjenu njezina stanja utječe vanjski entitet. Drugim riječima, stanje žarulje ostat će nepromijenjeno sve dok neki vanjski utjecaj ne bude utjecao na njenu promjenu. Iz sljedećeg isječka možemo vidjeti jedan dio proaktivnog rješenja. Zamijetimo u isječku da instanca prekidača unutar sebe sadrži instancu žarulje čije stanje instanca prekidača modificira svaki put kada se njegovo stanje promjeni, odnosno kada se promijeni položaj prekidača.

```
public class Switch
{
    LightBulb lightBulb;
    ...
    public void onFlip(boolean enabled){
        lightBulb.onFlip(enabled);
    }
    ...
}
```

Kôd 4: Proaktivni način promjene stanja žarulje

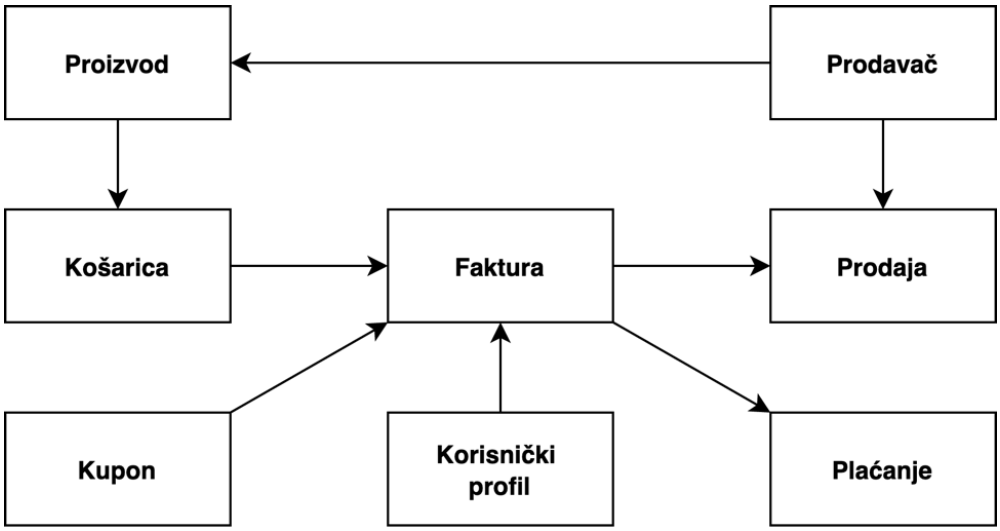
Drugi način na koji možemo povezati klase, a time stvoriti i ovisnost između objekata je na način da žarulja sluša na promjene stanja prekidača i sukladno tomu ažurira svoje stanje. U ovom modelu žarulja više ne predstavlja pasivnu komponentu već kažemo da je ona postala reaktivna zbog toga što promatra komponentu promatrača i reagira sukladno na njegove promjene. Idući isječak kôda prikazuje reaktivno rješenje gdje objekt žarulje primat objekt prekidača kao slušača događaja emitiranog od strane prekidača.

```
public class LightBulb
{
    ...
    public LightBulb create(Switch switchy){
        LightBulb lightBulb = new LightBulb();
        switchy.addOnFlipListener(enabled -> lightBulb.power(enabled));
        return  lightBulb;
    }
    ...
}
```

Kôd 5: Reaktivni način promjene stanja

Primjenom oba načina proaktivnog ili reaktivnog dovodi do istog krajnjeg rezultata paljenja ili gašenja žarulje. Međutim postoje suptilne razlike kod primjene oba rješenja. Prva razlika leži u tome tko kontrolira objekt žarulje. Kod proaktivnog modela kontrolu nad žaruljom preuzima neka vanjska komponenta koja poziva metodu za promjenu stanja rada u ovom slučaju metoda `power()`. S druge strane, kod primjene reaktivnog modela, objekt žarulje upravlja sam svojim stanjem. Druga razlika odnosi se na to koja komponenta određuje što objekt prekidača kontrolira. U proaktivnom modelu, objekt prekidača određuje koje objekte žarulje kontrolira. Dok kod reaktivnog modela, objekt prekidača nema eksplicitnog znanja o objektima koje kontrolira jer se objekti žarulja priključuju preko slušatelja. Drugim riječima, u proaktivnom modelu, entiteti kontroliraju jedni druge dok u reaktivnom modelu, entiteti kontroliraju sami sebe i spajaju se jedni na druge indirektno (Bagwala, 2019).

Kako ne bismo ostali samo na ovakvom jednostavnom, školskom primjeru, u nastavku ćemo prikazati malo kompleksniju strukturu modula vezanu uz poslovnu logiku e-trgovine koja se primjenjuje u realnim poslovnim sustavima. Na sljedećem dijagramu možemo vidjeti strukturu modula s prikazom njihovog međusobnog utjecaja jedne na druge.



Slika 16: Prikaz ovisnost i međusobnog utjecaja modula poslovne logike e-trgovine (Prema: Staltz, 2017)

Na dijagramu možemo vidjeti 8 zasebnih modula: Proizvod, Košarica, Kupon, Faktura, Korisnički profil, Prodavač, Prodaja i Plaćanje. Ovo su neki od modula koje aplikacije u domeni e-trgovine većinom koriste u svom radu. Osim modula na dijagramu vidimo i strijelice koje izlaze iz jednog modula i ulaze u drugi. Strijelica predstavlja ovisnosti modula u smislu da pokazuje koji modul će utjecati na neku promjenu drugog modula. Uzmimo za primjer modul košarice i fakture. Vidimo da modul košarice ima izlaznu strelicu koja ulazi u modul fakture. To znači da košarica na neki način mijenja fakturu. Jedan primjer takve promjene može biti dodavanje proizvoda u košaricu. Normalno je za očekivati kada dodamo novi proizvod u košaricu da će faktura biti ažurirana sukladno s ispravnom količinom dodanih proizvoda i izračunatom ispravnom krajnjom cijenom. Pošto je arhitektura aplikacije koncipirana na rad s modulima, to znači da sve što je isprogramirano živi unutar modula odnosno njegovih klasa, isto vrijedi i za samu strelicu. Međutim, gdje točno živi strelica? Strelica može živjeti ili u repu strelice ili u glavi strelice. Ako živi u repu to znači da ćemo u modulu košarice imati sljedeći programski kôd u kojem možemo vidjeti da je modul košarice zadužen za ažuriranje fakture.

```
public class Cart
{
    Invoice invoice; //Injected
    ...
    public void addProduct(Product product){
        invoice.updateInvoice(product);
    }
    ...
}
```

Kôd 6: Modul košarice (pasivno rješenje)

Kao i u prethodnom primjeru, kažemo da je modul košarice proaktivan jer je on odgovoran za uzrokovanje promjene, dok je s druge strane modul fakture pasivan iz razloga što nije svjestan postojanja ovisnosti između sebe i modula košarice. Da bi promjena stanja bila moguća, modul fakture mora imati javnu metodu za svoje ažuriranje. Takav način programiranja možemo nazvati pasivnim programiranjem kojeg karakteriziraju ovakve udaljene imperativne promjene, odnosno delegacija odgovornosti. Drugačiji pristup jest da naša strjelica živi u glavi strjelice. To znači da će modul fakture imati ovisnost prema modulu košarice. Ovim pristupom modul košarice preuzima ulogu izdavača događaja koje ostali moduli mogu osluškivati i reagirati na njih, dok s druge strane modul fakture preuzima reaktivnu ulogu na način da reagira na emitirane događaje od strane košarice čime postaje odgovoran sam za promjenu svog stanja. Važno je naglasiti, da reaktivnim načinom postižemo zaokruživanje poslovne logike modula u smislu da se sva poslovna logika nalazu unutar jednog modula, a ne raštrkana kroz više njih. Jednostavnijim riječima, svaki put kada se modul mijenja on je sam odgovoran za definiranje te promjene. U idućem programskom isječku možemo vidjeti reaktivno rješenje u kojem modul fakture sam ažurira svoje stanje krajnje cijene nakon što se dogodi događaj dodavanja novog proizvoda u košaricu. To je postignuto pretplatom fakture na košaricu koja nakon napravljene pretplate osluškuje emitirane događaje od strane košarice i nakon svakog emitiranja promjene ažurira svoje stanje.

```
public class Invoice
{
    Float total = 0f;
    ...
    public Observable getInvoice(){
        return Cart
                .getProductAddedObservable()
                .subscribe(product -> {
                    total += product.getPrice();
                });
    }
    ...
}
```

Kôd 7: Modul fakture (reaktivno rješenje)

Usporedimo sada primjenu pasivnog programiranja naspram reaktivnog s aspekta našeg primjera e-trgovine s pretpostavkom da se naš sustav sastoji od velikog broja klasa i modula. Prema Staltz (Staltz, 2017) ako želimo razumjeti rad takvog sustava te razumjeti pojedine module, potrebno je postaviti si dva temeljna pitanja. Prvo pitanje je na što sve pojedini modul utječe i drugo pitanje kako pojedini modul radi. Pretpostavimo da je sustav koncipiran pasivno. To znači da će svaka klasa ažurirati drugu klasu što rezultira pojavom neodgovornih modula. Ako pogledamo prethodnu sliku 16, te ako se fokusiramo na modul fakture vidimo da se faktura ažurira od strane minimalno tri modula (Košarica, Kupon, Korisnički profil), štoviše faktura dalje ažurira minimalno dva ostala modula (Prodaja, Plaćanje). Ako želimo otkriti koje module ili klase modul fakture mijenja, potrebno je ući unutar modula i manualno pogledati. Ovaj primjer je relativno plitak što se tiče ovisnosti, međutim u realnim sustavima on može biti daleko dublji i širi. Što se ovisnosti modula protežu dublje to je sam sustav i logiku rada modula teže za razumjeti prvenstveno iz razloga što se postaje vrlo lako za izgubiti u svim tim ovisnostima. Međutim, ovo daje odgovor samo na prvo pitanje. Ako želimo razumjeti kako modul fakture radi, potrebno je pronaći sva korištenja njegovih javnih metoda koje se koriste u ostalim modulima. Što je broj klasa i modula sustava veći te što su ovisnosti kompliciranije i šire, dobiti odgovor na pitanje kako nešto radi postaje proporcionalno teže. Pretpostavimo sada da smo umjesto pasivnog koncepta odlučili primijeniti reaktivni koncept na sve module i klase. Ako sada postavimo isto pitanje tko na što utječe odgovor i dalje ostaje isti, ništa se nije promijenilo. U našem primjeru modul fakture i dalje utječe na modul prodaje i plaćanja. Međutim razlika je u tome kako dolazimo do te informacije. Sada više ne moramo ulaziti u modul i provjeravati koje ovisnosti su definirane u njemu, već je

potrebno pronaći korištenja odnosno reagiranje na emitirane događaja u drugim modulima. Zapravo se postupak pronalaženja odgovora obrnuo u odnosu na postupak u pasivnom primjeru. Ovisno o preferencijama ovo može biti nešto što će nam više odgovarati ili ne. Međutim, ovo izvrtanje rezultira puno lakšim odgovorom na drugo pitanje kako nešto funkcionira. U reaktivnom primjeru, ako želimo saznati kako neki modul funkcionira, više nije potrebno pretraživati cijelo projektno rješenje već je dovoljno da pogledamo u sami modul koji nas zanima. Razlog tomu je što je svaki modul odgovoran sam za sebe i za svoje ažuriranje koje se izvršava na temelju emitiranih događaja ostalih modula. Da bi pojednostavili objašnjenje, u sljedećoj tablici možemo vidjeti u skraćenoj verziji odgovore na pitanja za pasivno i reaktivno rješenje ali isto tako nedostatke i prednosti pojedinog rješenja.

	Pasivno	***Reaktivno***
Na što utječe?	▪ *Pogledaj unutra*	▪ *Pronađi konzumiranje događaja*
Kako nešto radi?	▪ *Pronađi korištenja javnih metoda*	▪ *Pogledaj unutra*
Nedostatci	▪ *Pojava neodgovornih moduli* ▪ *Teže za razumjeti rad modula*	▪ *Korištenje singletona*
Prednosti	▪ *Podatkovne strukture* ▪ *Ubrizgavanje ovisnosti*	▪ *Modul je odgovoran za promjenu samog sebe* ▪ *Jednostavnije za shvatiti funkcionalnost modula*

Tablica 1: Pasivno rješenje naspram reaktivnog (Prema: Staltz, 2017)

Iz perspektive razvojnih inženjera većinom nam je prvo važnije shvatiti kako nešto radi, a tek poslije se pitamo na što sve ostalo utječe. S tom pretpostavkom, reaktivno programiranje ima veću vrijednost jer je daleko lakše vidjeti kako neki modul funkcionira zato što se sve nalazi unutar njega. Međutim pasivno rješenje također ima svoje određene prednosti koje se ogledaju u mogućnosti lakšeg definiranja i korištenja podatkovnih struktura i svima nama dobro poznatog ubrizgavanja ovisnosti koje je iznimno lako za koristiti u okvirima koji ga podržavaju.

3.5. Kada koristiti reaktivno programiranje

Da bismo razumjeli kada i zašto koristiti reaktivno programiranje, prvo se moramo osvrnuti na današnje zahtjeve sustava koje mi kao programeri razvijamo te koji su to mogući problemi na koje moramo pripaziti i uzeti ih u obzir kod početnog planiranja, analize i konstruiranja arhitekture budućeg sustava. Ovisno za koju domenu razvijamo rješenje, ti problemi odnosno zahtjevi mogu biti brojni i po prirodi veoma različiti. Međutim, kako ne bismo otišli previše u dubinu, fokusirat ćemo se samo na podskup problema koji su za današnje vrijeme interesantni i sveprisutni. To su problemi rada s velikim količinama podataka (eng. *Big data*) te problem stabilnosti i odzivnosti sustava s velikim populacijama korisnika. Ako bi se vratili oko 10 godina u prošlost, brzina i odzivnost sustava, iako doduše i tada važan kriterij, ne bi bilo nešto o čemu bi ovisila uspješnost naše aplikacije na tržištu. U prošlosti, od aplikacije se očekivalo da radi svoj posao i da ga radi ispravno, brzina i odzivnost same aplikacije ako pričamo o vremenski ne kritičnim aplikacijama, su bili kriteriji koji su bili manje važni korisniku. Naravno, slabije performanse aplikacije u to vrijeme su bili izravno povezani s puno manjim te slabijim računalnim resursima koje je sama aplikacija imala na raspolaganju za korištenje. Međutim, s globalnim proširenjem interneta, a pogotovo s pojavom pametnih telefona, brzina i odzivnost su postali jedni od važnijih kriterija ako ne i najvažniji za prihvaćanje aplikacije od strane korisnika (Mandar jog, 2017). Korisnici u današnje vrijeme imaju puno manju toleranciju za rad same aplikacije od koje više ne očekuju da samo ispravno radi već ujedno žele i najbolje korisničko iskustvo koje je moguće dobiti. Dobro korisničko iskustvo se može ogledati u više kriterija kao što su: jednostavnost aplikacije za korištenje, vizualni izgleda, korištenje opće prihvaćenih dizajna ali isto tako i brzina, stabilnost, dostupnost te fluidnost korištenja aplikacije. Drugim riječima, korisnici očekuju optimiziranu aplikaciju s brzinom izvršavanja u milisekundama, bez previše zastajkivanja (eng. *Lag*) te aplikaciju koja će biti stalno dostupna za korištenje. U slučaju da aplikacija ne zadovoljava navedene kriterije, što zbog loše arhitekture, neoptimiziranog kôda, ili nemogućnosti usluživanja velikog broja istovremeno aktivnih korisnika, šanse da će korisnici prestati koristiti aplikaciju te prijeći na korištenje alternativnog rješenja su veoma velike. Stoga si malo tko može dopustiti išta manje od savršenstva.

Međutim reaktivno programiranje kao i sve ostale programske paradigme nije uvijek odgovarajuće i primjenjivo rješenje za sve probleme. Neke od situacija u kojima nam primjena reaktivnog programiranja može uvelike pomoći te ujedno utjecati na bolje performanse našeg rješenja su sljedeće. Situacija koja je već prije spomenuta, a vezana je uz procesuiranje vanjskih događaja. To mogu biti različiti korisnički generirani događaji od pokreta i klika mišem, pritisak tipke na tipkovnici, pritisak zaslona osjetljivog na dodir, potom događaji generirani od

neke vanjske komponente ili senzora kao što je GPS signal za praćenje kretanja korisnika, podaci generirani različitim senzorima i slično (Latcu, 2017). Sve su ovo događaji bazirani na modelu guranja, gdje proizvođač šalje podatke potrošaču i to u obliku događaja koji su po svojoj prirodi asinkroni.

Druga situacija je reagiranje i procesuiranje događaja s latencijom, događaja kod kojeg postoji interval vremena između podražaja i reagiranja. Takvi događaji su većinom vezani uz pisanje i čitanje podataka s memorijskog diska, poziva udaljenih servisa, komunikacije preko internet mreže i slično. Svojstvo događaja upita i odgovora je to da su po svojoj prirodi asinkroni što znači da će nakon kreiranja zahtjeva ili upita, proći neko određeno vrijeme prije nego li će odgovor biti zaprimljen, pod pretpostavkom da uopće bude zaprimljen. Zbog mogućnosti da odgovor na upit nikad ne stigne do aplikacije, što je realan slučaj kod upita koji se šalju nekom vanjskom servisu, takve operacije se nikada ne bi smjele implementirani na sinkroni način. Razlog tomu je što se vrlo lako može dogoditi da aplikacija potroši sve raspoložive resurse čekajući na odgovor kojeg nikad neće dobiti. Kako bi se to izbjeglo, primjenjuje se asinkroni način komunikacije, a kako smo već spomenuli da je reaktivno programiranje bazirano na asinkronosti ono nam se pruža kao jedno moguće rješenje (Mandar jog, 2017).

Sljedeći situacija je slična prethodno navedenima i odnosi se ponovo na guranja podataka aplikaciji od strane proizvođača, međutim ovdje je naglasak na tome da proizvođač emitira veću količinu informacija nego što ih potrošač može obraditi u danom trenutku. Takvi događaji su učestali kod zaprimanja raznih signala s hardvera uzrokovanim senzorima ili slično. Primjer realne situacije koji može rezultirati ovakvim slučajem je sustav koji zaprima podatke od jednog ili većeg broja uređaja koji te podatke generiraju te ih šalju aplikaciji putem interneta. Takve sustave nazivamo Internet stvarima (eng. *Internet of things*). Ovisno o tome koliki broj uređaja sustav koristi te koliko i kojom brzinom isti generiraju i šalju podatke aplikaciji može se vrlo lako prouzrokovati zagušenje sustava. Tu u spas dolazi reaktivno programiranje koje može pomoći kod brzine obrade veće količine podataka u kratkom vremenu zahvaljujući konceptima asinkronosti, konkurentnosti i paralelnosti na kojima je reaktivno programiranje građeno te koje je zbog toga vrlo lako za implementirati. Osim same primjene reaktivnog programiranja, u ovakvim situacijama veoma je korisno iskoristiti i dodatni koncept koji se s velikom jednostavnošću može koristiti u kombinaciji s reaktivnim programiranjem, a to je koncept **povratnog tlaka** (*eng. Backpressure*). Koncept povratnog tlaka se koristi kako bi se komponentama potrošača omogućilo rukovanje s ulaznim podacima koje potrošač nije u mogućnosti dovoljno brzo procesuirati te ih pretvoriti u izlazni podatak i to na jedan od nekoliko mogućih načina sve s ciljem sprečavanja preopterećenja komponente potrošača. (Christensen & Nurkiewicz, 2016)

Idući scenariji govori više o izradi arhitekture sustava i kako olakšati programerima izmjenu arhitekture ili načina rada aplikacije nego o rješavanju nekog specifičnog problema sustava. U samom početku izrade aplikacije, vrlo je teško pa čak i nemoguće predvidjeti kolika će opterećenost sustava biti jednom kada isti ode u produkciju te kakve će performanse aplikacija imati u produkciji s gledišta brzine izvršavanja u odnosu na broj aktivnih korisnika. Ovisno o znanju i stručnosti arhitekta i razvojnih inženjera koji razvijaju aplikaciju, performanse mogu biti izvrsne, zadovoljavajuće ali i veoma loše. Štoviše, tijekom dužeg perioda razvoja i višestrukih nadogradnja aplikacije, većina aplikacija izgubi na performansama što izravno utječe na korisničko iskustvo. U tom trenutku nastaje problem i javlja se potreba za optimizacijom programskog kôda kako bi se brzina izvršavanja dovela ponovo na zadovoljavajuću razinu. Prema Subramaniam (Subramaniam, 2017) za većinu aplikacija koje su pisane na temeljima sekvencijalnog izvršavanja u većini slučajeva znači da je sada potrebno uvesti višedretvenost i preoblikovati sekvencijalni kôd u konkurenti. Zamjena sekvencijalnog izvršavanja konkurentnim nije lak posao, i u većini slučajeva predstavlja teško vrijeme za projekt zbog toga što nestručni programeri u radu s dretvama, procesima i ostalim konstruktima, uzrokuju više štete nego koristi. Način na koji se moglo izbjeći mijenjanje sekvencijalnog s konkurentnim načinom rada kasnije u projektu je taj da se krenulo s istim od samog početka izrade rješenja. Međutim ako bi se odlučili ići tim putem, to znači da bi nam vrlo vjerojatno za razvoj samog rješenje trebalo puno više vremena i uloženog truda jer koliko god je moguće dobiti na performansama korištenjem višedretvenosti za obavljanje poslovne logike isto toliko možemo i narušiti ispravnost i stanje samog programa. Štoviše, dodatno uloženo vrijeme i trud za razvoj takvog rješenja u konačnici može biti uzalud jer nismo uspjeli predvidjeti ispravnu opterećenost sustava što u konačnici znači da sustav nikad neće biti opterećen u tolikoj mjeri da performanse sekvencijalnog načina izvršavanja ne bi bile zadovoljavajuće na prvom mjestu. Treći način kako se ovo pitanje može razriješiti, je upotrebom reaktivnog programiranja. Naime primjenom reaktivnog programiranja strukturalne razlike između sekvencijalnog i konkurentno kôda su minimalne i zanemarive i to sve zahvaljujući korištenjem tokova podataka čijom primjenom zapravo postižemo tu minimalnu sintaktičku i strukturalnu razliku (Subramaniam, 2017). S druge strane, ako za primjer uzmemo objektno orijentiranu paradigmu, struktura kôda između jednog i drugog se uvelike razlikuje. Stoga primjenom reaktivnog programiranja, veoma lako možemo sekvencijalno izvršavanje prebaciti u paralelno i obratno na početku ali isto tako i kasnije u razvoju bez potrebe za ulaganjem velikog truda, puno vremena i velikih promjena.

Sljedeći, ujedno i zadnji scenariji koji ćemo navesti kod kojeg nam reaktivno programiranje može koristiti je scenarij izrade reaktivnih sustava. Veliki broj ljudi kada čuje pojmove reaktivni sustav i reaktivno programiranje zaključi da se radi o istoj stvari, odnosno

da oni predstavljaju sinonime. Međutim, to nije točno. Reaktivni sustav označava jednu višu razinu, ono označava arhitekturalni stil izgradnje odzivnih i distribuiranih sustava. Zabuna proizlazi iz toga što se reaktivno programiranje nameće kao jedno od mogućih rješenja za implementaciju reaktivnih sustava. Međutim, samom primjenom reaktivnog programiranja ne dobivamo reaktivni sustav kao što ni zadovoljenje principa reaktivnog sustava ne znači da se u pozadini primjenjuje reaktivno programiranje (V. Klang & Bonér, 2016). Više o reaktivnim sustavima bit će objašnjeno u idućem poglavlju.

4. Reaktivni sustavi

Reaktivni sustavi predstavljaju skup načela arhitekturalnog dizajna za izgradnju modernih poslovnih sustava koji su u mogućnosti podnijeti veliki broj zahtjeva s kojima se aplikacije susreću u današnje vrijeme. Principi reaktivnih sustava koji su navedeni i detaljno objašnjeni u reaktivnom manifestu nisu nešto što već nije od prije poznato. Prvo spominjanje principa reaktivnih sustava možemo pronaći u seminarskom radu napisanog od Jim Graya i Pat Hellanda iz 1986. godine na temu tolerancije kvarova u Tandem računalnom sustavu. Osim Jima i Pata, temom reaktivnih sustava bavio se i Joe Armstrong koji je napisao doktorsku tezu o „Postizanje pouzdanih distribuiranih sustava u prisutnosti programskih grešaka“ s ciljem pronalaženja boljih načina programiranja telekomunikacijskih aplikacija ali i za postizanje robusnijih i otpornijih aplikacija. Na temelju njegove teze nastao je i programski jezik Erlang koji se koristi za izgradnju masivnih skalabilnih sustava visoke dostupnosti.

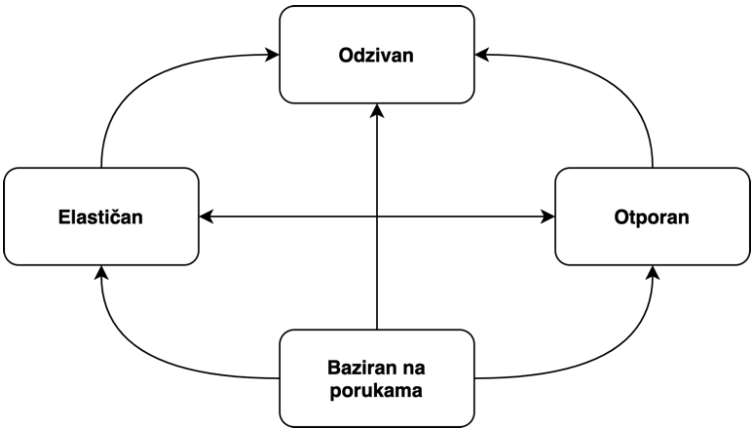
Kao što možemo vidjeti već davnih 1980-ih godina su se ljudi bavili tematikom robusnih sustava, međutim svi ovi ljudi bili su daleko ispred svoga vremena. Tek u zadnjih 5-10 godina se desio značajniji zaokret s pojavom računalstva u oblaku i internet stvari (eng. *Internet of Things*). Tehnološka industrija bila je primorana promijeniti svoje stajalište i prakse u razvoju poslovnih aplikacija koje su postale daleko više od samo običnih programa, postali su poslovni sustavi. Za razliku od običnih programa, sustavi predstavljaju daleko složeniji konstrukt koji se mogu sastojati i temeljiti na radu većeg broja komponenti koje same za sebe mogu predstavljati samostalni sustav. Pojavom takvih složenijih konstrukata dolazimo do zavisnosti rada jednog softvera o radu drugog softvera. Realni primjer takve ovisnosti je Microsoft Azure[3] usluga računalstva u oblaku i aplikacija koje se vrte na istom. Štoviše, sustavi koji se razvijaju danas, mogu se vrtjeti na različitim računalima od slabih pa do superračunala, a sustavi moraju biti dostupni na korištenje korisnicima koji se mogu nalaziti locirani u različitim dijelovima svijeta. Da bismo bili u stanju isporučiti sustav na kojeg se korisnici ali i tvrtke mogu osloniti, sustavi moraju biti odzivni. Prošla su vremena kada se od sustava očekivalo samo da vrati ispravan rezultat bez očekivanja da se isti rezultat dobije što je vremenski prije moguće. Da bismo postigli ispravnost ali i brzu odzivnost sustava, kao arhitekti sustava moramo se pobrinuti da odzivnost sustava ostane brza i konstantna pa čak i u situacijama pojave grešaka ili vremenima kada sustav doživljava opterećenja. Da bi to bilo moguće, osmišljena je arhitektura sustava baziranim na porukama koju nazivamo reaktivnim sustavom. Više o ostalim karakteristikama takvog sustava i načina njihova postizanja bit će detaljnije objašnjeni u idućem poglavlju koji

[3] Usluga računalstva u oblaku stvorena od strane Microsofta za izgradnju, raspoređivanje testiranje i upravljanje aplikacijama i servisima putem podatkovnih centara.

se bavi arhitekturom reaktivnih sustava objašnjenih u dokumentu pod nazivom reaktivni manifest (V. Klang & Bonér, 2016).

4.1. Reaktivni Manifest

Reaktivni Manifest predstavlja dokument čija je druga verzija objavljena 2014. godine. Ovaj dokument nastao je kao zajednički napor sa svrhom da se predstavi drugačija arhitektura sustava koja bi zadovoljila sve potrebne aspekte današnjice za njihov razvoj i rad. Autori ovog manifesta tvrde da su očekivanja od sustava postala znatno veća za razliku od prije nekoliko godina gdje su se dugotrajna vremena odziva, sporost aplikacije, česti ispadi i održavanja s poslužiteljima izvan mreže po nekoliko sati smatrali prihvatljivim. Kako bi se zadovoljila današnja očekivanja korisnika, autori su sa svojim manifestom predstavili novu vrstu sustava, zvanu reaktivni sustav. Reaktivni sustavi građeni su na temelju četiriju karakteristika: odzivnost, elastičnost, otpornost i baziranost na porukama (*The Reactive Manifesto*, 2014).



Slika 17: Četiri karakteristike reaktivnog sustava i njihov međuodnos (Prema: The Reactive Manifesto, 2014)

- **Odzivnost:** Odzivni sustav je sustav koji reagira pravodobno, koji je fokusiran na pružanje brzog i konzistentnog vremena odgovora čime uspostavlja pouzdanu gornju granicu te time isporučuje konzistentnu kvalitetu usluge koja zauzvrat rezultira boljim korisničkim iskustvom. Predstavlja temelj upotrebljivosti i korisnosti.
- **Otpornost:** Da bi sustav bio neprestano dostupan, veoma je važno da događaj kvara ne utječe na njegovu dostupnost. Sustav bi trebao ostati odzivan i u slučaju kvara. Otpornost nije dovoljno zadovoljiti samo za kritične dijelove sustava ili dijelove od kojih se očekuje da imaju visoku dostupnost. Bilo koji

sustav koji nije otporan na greške, prestat će reagirati nakon što se dogodi greška. Otpornost je moguće postići *replikacijom*, *zatvorenošću*, *izolacijom* i *delegacijom*.

- **Replikacija:** Predstavlja istovremeno izvršavanje komponente na različitim mjestima. To znači izvršavanje komponente na različitim dretvama ili bazenima dretvi, izvršavanje u sklopu različitih procesa, mrežnih čvorova ili računalnim centrima (izvršavanje u privatnom ili javnom oblaku). Korištenjem replikacije, sustav dobiva na skalabilnosti, odnosno otpornosti zahvaljujući raspoređivanju dolaznog radnog opterećenja na više instanci komponente.
- **Izolacija i zatvorenost:** Predstavlja vremensko i prostorno razdvajanje komponenti pošiljatelja i primatelja. Vremenska razdvojenost je zadovoljena kada komponente pošiljatelja i primatelja ne dijele isti životni ciklus, njihovi ciklusi su neovisni jedna o drugome, međutim njihova međusobna komunikacija je i dalje moguća. Postiže se uvođenjem asinkronih granica te uvođenja komuniciranja putem poruka između komponenti. S druge strane, prostornu razdvojenost je moguće postići izvođenjem komponenti u sklopu različitih procesa. Ispravna implementirana izolacija rezultira kompartmentalizacijom i zatvorenošću stanja, ponašanja i pogrešaka komponenti sustava.
- **Delegacija:** Označava asinkrono prepuštanje izvršavanja nekog zadataka drugim komponentama. Zadatak se izvršava unutar konteksta komponente te u skladu s njezinim životnim ciklusom, međutim nije nužno da se delegirana komponenta izvodi na istoj dretvi, istom procesu ili čak istom mrežnom čvoru kao i komponenta koja je prepustila zadatak. Svrha delegacije je prepuštanje odgovornost za obradu zadatka drugoj komponenti kako bi se prvotna komponenta oslobodila za obavljanje drugog posla.

- **Baziranost na porukama:** Asinkrona komunikacija porukama između komponenti uspostavlja granicu koja osigurava slabu povezanost, izolaciju i lokacijsku transparentnost sustava. Značenje lokacijske transparentnosti najbolje je promatrati u kontekstu skaliranja sustava. Lokacijski transparentni sustav moguće je izvršavati na više čvorova na posve identični način kao i na jednom. To znači da ne postoji razlika između skaliranja sustava vertikalno ili horizontalno. Nadalje, asinkrono slanja poruka između komponenti pomaže u delegaciji pogrešaka te pruža mogućnost zadovoljavanja elastičnosti i odzivnosti uz implementaciju dodatnog konstrukta kao što su redovi poruke i

koncepta povratnog tlaka. O konceptu povratnog tlaka bit će više rečeno u idućem potpoglavlju.

- **Elastičnost:** Definiramo kao mogućnost sustava da ostane odzivan i u slučaju promjenjivog radnog opterećenja. Za reaktivne sustave je karakteristično da imaju sposobnost automatskog povećanja ili smanjenja alociranih resursa ovisno o opterećenja samog sustava. Takvi sustavi zahtijevaju dizajn koji u sebi ne sadrži uska grla koja bi mogla uzrokovati nemogućnost repliciranja komponenti te distribuiranje ulaznog prometa na njih. Iako većina ljudi pojam elastičnost i skalabilnosti koristi kao sinonime, između njih ipak postoje suptilne razlike. Elastičnost je sposobnost automatskog povećanja ili smanjenja resursa infrastrukture i propusnosti sustava s ciljem prilagodbe radnom opterećenju sustava tako da se maksimizira uporaba dostupnih resursa koji će odgovarati što približnije trenutačnim potrebama. Da bi sustav mogao biti elastičan, isti mora prvo biti skalabilan, međutim obrnuto ne vrijedi. Stoga možemo reći da se elastičnost temelji na skalabilnosti te ju proširuje dodavanjem automatskog upravljanja resursima.

Na temelju opisanih koncepata reaktivnih sustava, možemo izreći sljedeći zaključak. Osnovu reaktivnih sustava predstavlja prenošenje poruka čija primjena stvara vremensku ali i prostornu razdvojenost između komponentu sustava. Postizanje vremenske i prostorne razdvojenosti čine sustav otpornijim i elastičnijim. Sustav koji je skalabilan, elastičan, otporan i razdvojen u neovisne komponente predstavljaj reaktivan sustav.

4.2. Reaktivni sustavi naspram reaktivnog programiranja

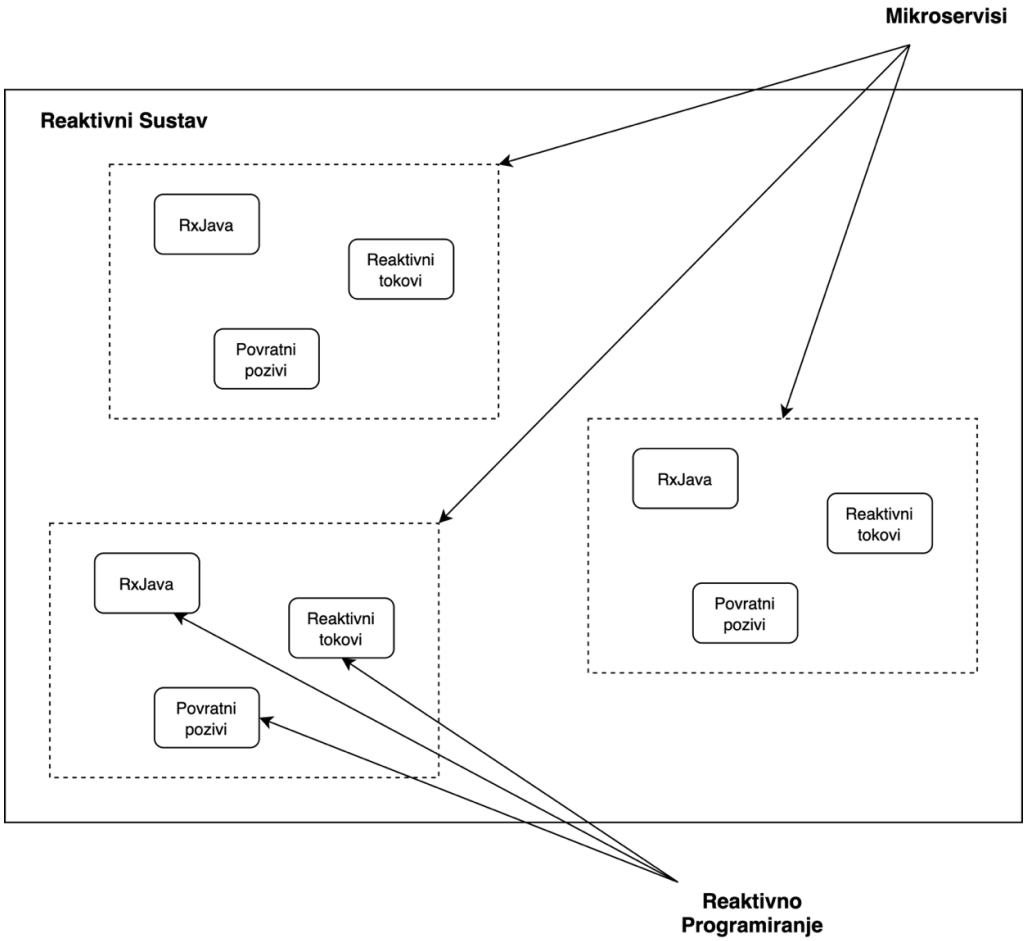
Ovo poglavlje objašnjava zašto se reaktivni sustavi ne bi smjeli poistovjećivati s reaktivnim programiranjem, odnosno zbog čega se ovo dvoje ne bi trebalo koristiti u kontekstu sinonima.

Razlike između reaktivnog programiranja i reaktivnog sustava možemo promatrati s gledišta domene njihove primjene. Reaktivno programiranje se primjenjuje u domeni interne logike programskih komponenata dok reaktivni sustavi predstavljaju koncept više razine primjenjiv u domeni izrade arhitekture sustava. Aplikacije koje implementiraju reaktivno programiranje svoj rad zasnivaju na radu s događajima što rezultira konkurentnim načinom izvršavanja koji zadovoljava svojstvo vremenske ali ne i prostorne odvojenosti. Štoviše, nedostatak lokacijske transparentnosti primjenom čistog reaktivnog programiranja čini aplikaciju iznimno teškom za skaliranje. Da bi se takva aplikacija mogla ispravno skalirati, potrebna je implementacija dodanih konstrukata. S druge strane, reaktivni sustavi

implementiraju komunikaciju baziranoj na porukama što rezultira nastankom distribuiranih sustava koji zadovoljavaju svojstvo prostorne odvojenosti ali i lokacijske transparentnosti (V. Klang & Bonér, 2016).

U sustavima baziranim na događajima, posebice sustavima koji primjenjuju reaktivno programiranje, usredotočenost je na adresirane izvore događaja dok se sustavi koji su bazirani na radu s porukama (reaktivni sustavi) usredotočuju na adresirane primatelje. S gledišta izdavača i pretplatnika to znači da su pretplatnici u sustavima baziranim na događajima spojeni, odnosno pretplaćeni na izdavača, dok u kontrastu, kod sustava baziranih na porukama, pretplatnici čekaju na poruke bez potrebe za pretplatom (*Glossary - The Reactive Manifesto*, bez dat.).

Reaktivni sustav može biti postignut primjenom reaktivnog programiranja, međutim sama primjena reaktivnog programiranja bez uzimanja u obzir i zadovoljenja ostalih potrebnih svojstva reaktivnog sustava, sustav ne čine reaktivnim. Međutim primjenom reaktivnog programiranja u sklopu reaktivnog sustava, može nam pomoći u rješavanju problema upravljanja asinkronim tokovima podataka i postizanju neblokirajućeg izvršavanja (J. B. Klang Viktor, 2016).



Slika 18: Apstraktni prikaz reaktivnog sustava (Prema: Ieo, 2018)

4.2.1. Rad s događajima naspram rada s porukama

U poglavljima u kojima smo opisivali koncepte reaktivnog programiranja, mogli smo vidjeti da se reaktivno programiranje temelji na radu s događajima koji se emitiraju i šalju nizvodnim komponentama koje te događaje osluškuju te reagiraju na njih sukladno. S druge strane, reaktivni sustavi koji se fokusiraju na otpornost, odzivnost i elastičnost se temelje na komunikaciju putem poruka s ciljem povezivanja razdvojenih komponenti sustava.

Glavna razlika između sustava temeljenog na komunikaciji putem poruka i reaktivnog programiranja koje se temelji na događajima je u tome što poruke predstavljaju usmjerene elemente dok s druge strane događaji ne predstavljaju. Drugim riječima, poruke imaju jasno definiranu komponentu sustava kojoj se šalju, kažemo da poruke sadrže odredište. Međutim, možda nije najispravnije reći da poruke eksplicitno znaju svoju odredišnu komponentnu iz razloga što se za slanje poruka primjenjuju različiti protokoli ili posrednički softveri za posredovanje porukama koje koriste konstrukte kao što su redovi, teme i slično. Nadalje, u sustavima koji se temelje na porukama, krajnji primatelji poruka čekaju na dolazak poruke s time da nemaju direktnu spoznaju od koga dobivaju poruku. Jednom kada se zaprimi poruka, komponenta reagira dok ostalo vrijeme miruje. S druge strane, u programima koji se temelje na događajima, objekti se pretplaćuju ili osluškuju druge objekte koji proizvode događaje. Objekti slušači izravno znaju od kojeg objekta mogu očekivati emitirani događaj, međutim obrnuto ne vrijedi. Objekti koji emitiraju događaje nemaju spoznaju koji sve drugi objekti slušaju. Zbog toga kažemo da su događaji ne adresirani, odnosno da se ovaj model fokusira na adresirane izvore događaja umjesto adresirana odredišta. Druga razlika je u razini njihove primjene. Poruke nam služe za komunikaciju razdvojenih komponenti sustava koji je većinom prostorno razdvojen u smislu da se različiti dijelovi sustava vrte na različitim fizičkim mašinama. To znači da se slanje poruka između komponenti odvija putem mreže. Događaji pak služe za komunikaciju na lokalnoj razini, odnosno između objekata jedne instance aplikacije.

Kod izgradnje reaktivnih sustava, uobičajeno je korištenje događaja enkapsuliranih u obliku poruka kako bi se postiglo povezivanje distribuiranog sustava temeljenog na događajima putem mreže.

4.3. Prednosti reaktivnih sustava uz primjenu RP

Već smo prije spomenuli da je korištenje reaktivnog programiranja tek jedan korak u izgradnji reaktivnih sustava i da samo njegovo korištenje ne čini sustav reaktivnim. Također, korištenje reaktivnog programiranja predstavlja opcionalan korak, reaktivni sustav može se postići i bez njegove primjene. Međutim, pretpostavimo da se ipak odlučimo na korištenje

reaktivnog programiranja u izgradnji komponenti reaktivnog sustava. Prema Posa (Posa, 2018) neke od prednosti koje možemo očekivati s ispravnom i dosljednom implementacijom jesu:

- **Samoizlječenje** – Specifikacija reaktivnih tokova tvrdi da reaktivno programiranje podržava otpornost na temelju mogućnosti prirodnijeg obrađivanja pogreške koje neće zaustaviti rad cijelog sustava, već će se kod pojave grešaka izvršiti sporedni tok. Temeljem toga moguće je izgraditi reaktivne sustave koji koriste tu tehniku oporavka od neuspjeha te nastavljaju dalje normalno raditi.
- **Visoka dostupnost** – Prema specifikaciji reaktivnih tokova, reaktivno programiranje prirodno podržava elastičnost. Elastičnost se postiže učinkovitom iskoristivošću hardverskih resursa ali i korištenjem povratnog tlaka između objekata aplikacije. Primjenom reaktivnog programiranja u izgradnji komponenti reaktivnog sustava postižemo visoku dostupnost svake pojedine komponente, a time i cijelog sustava.
- **Niska latencija** – Pošto se reaktivno programiranje temelji na asinkronim tokovima podataka te na modelu emitiranja događaja to znači da će sustav imati niže vrijeme odgovora što posljedično uzrokuje nižu latenciju između komponenti sustava te samog korisnika i sustava.
- **Visoka skalabilnost** – Visoka skalabilnost prvenstveno je postignuta izoliranošću komponenti koja omogućuje njihovu nezavisnu replikaciju jer ne dijele iste resurse. S druge strane moguće je vrtjeti više instanci na pojedinoj komponenti na koje je moguće raspodijeliti dolazni promet zahvaljujući radu s asinkronim porukama. Nadalje, korištenjem reaktivnog programiranja, svaka komponenta pa tako i sve instance te komponente iskorištavaju maksimalno hardverske i softverske resurse što u konačnici znači da će troškovi sustava što se tiče zakupljivanja resursa biti minimalni.

Korištenje reaktivnog programiranja u izgradnji reaktivnih sustava posjeduje još mnogo prednosti od kojih su većina već bila spomenuta na jedan ili drugi način, stoga se ista neće detaljno objašnjavati već ćemo ih samo spomenuti: slaba povezanost, transparentnost lokacije, bolje performanse, jednostavnost održavanja, lakoća u rješavanju kvarova, učinkovita iskoristivost hardverskih i softverskih resursa (Posa, 2018).

5. RP u programskom jeziku Java

U poglavlju koje se bavilo konceptima reaktivnog programiranja spomenuli smo da ono predstavlja zapravo reaktivnu paradigmu sa svojim određenim konceptima, karakteristikama i svojstvima. Sama implementacija reaktivne paradigme u svijetu Jave predstavlja dosta specifičan proces. Naime, sve do verzije 5 (Java 5) nije bilo moguće programiranje asinkronih aplikacija, sve aplikacije koje su napisane u prijašnjim verzijama su po prirodi bile sinkrone i blokirajuće. Tek s dolaskom verzije 5 koje je sa sobom donijela novo asinkrono sučelje `Future` je razvoj asinkronih aplikacija postala realnost *(baeldung, 2017)*. Reaktivno programiranje, odnosno sučelja potrebna za njegovo korištenje uključena su u standardni skup Java biblioteka tek u sklopu verzije 9 (Java 9). Međutim, zanimljiva je činjenica da je pisanje reaktivnih aplikacija korištenjem reaktivnog programiranja u Javi bilo moguće i mnogo ranije. Razlog zašto je to tako leži u proaktivnosti Java programerske zajednice koja nije željela čekati na udovoljavanje zahtjeva već se odlučila preuzeti stvari u svoje ruke. Ova proaktivnost rezultirala je pokretanjem inicijative pod nazivom *Reaktivni Tokovi* (eng. *Reactive Streams*) te stvaranje istoimene specifikacije za reaktivnog programiranje.

5.1. Specifikacija reaktivnih tokova

Kao što je u uvodu rečeno, programerska zajednica osnovala je inicijativu pod nazivom *Reaktivni Tokovi* s ciljem definiranja i uspostavljanja standarda za primjenu asinkronih tokova obrade uz poštivanje neblokirajućeg povratnog tlaka. Ova inicijativa bila je usmjerena na izvršna okruženja JVM (eng. Java Virtual Machine) i JavaScript ali je isto tako obuhvaćala i njenu primjenu u prijenosima putem različitih prenosivih medija koji podržavaju serijalizaciju i deserijalizaciju.

Problem koji je ova inicijativa trebala riješiti odnosio se na rukovanje tokovima podataka ili takozvanim živim podacima koje karakterizira nemogućnost predviđanja njihova volumena te koji kao takvi, zahtijevaju dodatnu pozornost kod korištenja u asinkronim sustavima. Najistaknutiji problem u radu s takvim podacima predstavlja sam čin njihovog konzumiranja koje ako nije ispravno implementirano može rezultirati preopterećenjem odredišne (nizvodne) komponente od strane izvorišta podataka. Zbog toga glavni cilj *Reaktivnih Tokova* predstavlja upravljanje razmjenom podataka toka kroz asinkronu granicu sustava istovremeno osiguravajući da primajuća strana nije prisiljena na skladištenje primljenih podataka. Povratni tlak predstavlja sastavni dio ovog rješenja jer se njegovim korištenjem omogućuje primjena ograničenih umjesto neograničenih visokorizičnih redova u samom procesu asinkrone komunikacije. Da bi primjena ove inicijative imala smisla, cjelokupno rješenje mora

funkcionirati na asinkroni i neblokirajući način kako bi se postigla maksimalna utilizacija raspoloživih resursa.

U dokumentu specifikacije (*Reactive Streams*, 2015) navodi se da je namjera ove inicijative bilo ostvarivanje jedinstvene specifikacije s minimalnim brojem sučelja, metoda i protokola koji bi opisali potrebne operacije i entitete za postizanje cilja asinkronih tokova podataka s neblokirajućim povratnim tlakom. Kako bi se olakšala implementacija reaktivne specifikacije i potakla njena primjena definirane su četiri radne grupe:

- **Osnovna semantika** – Definira kako se regulira prijenos podataka toka kroz povratni tlak. Prijenos i reprezentacija podataka kao i signaliziranje povratnog tlaka nije dio specifikacije.
- **JVM sučelja** – Primjenjuje osnovnu semantiku na skup programskih sučelja čija je glavna svrha omogućiti interoperabilnost različitih implementacija i jezičnih veza za prosljeđivanje tokova između objekata i dretvi unutar JVM izvršnog okruženja koristeći zajednički memorijski stog.
- **JavaScript sučelja** – Definira minimalni skup objektnih svojstava za promatranje toka podataka unutar JavaScript izvršnog okruženja.
- **Mrežni protokoli** – Definira skup mrežnih protokola za prosljeđivanje reaktivnih tokova. Primjeri protokola su: TCP, UDP, HTTP i mrežne utičnice (eng. *Web Sockets*).

U konačnici, stvaranjem ove specifikacije rezultiralo je pojavom velikog broja okvira, biblioteka i ekstenzija za različite jezike, okvire i platforme. U Java programskom svijetu možemo izdvojiti dvije najpopularnije implementacije: Reaktivne ekstenzije ili ReactiveX[4] te Project Reactor[5].

Prije nego li krenemo na opisivanje pojedinih reaktivnih biblioteka i okvira, navest ćemo najvažnija sučelja i njihove metode koje specifikacija reaktivnih tokova pruža za implementaciju vlastitih rješenja za rad s asinkronim tokovima podataka. To su četiri sljedeća sučelja (Thompson, 2017):

- **`Publisher<T>`** – Sučelje koje se koristi za implementaciju vlastitog rješenja izdavača potencijalno neograničenog broja sekvencijalnih elemenata odnosno vrijednosti koje emitira u skladu s potražnjom svojih pretplatnika. Izdavač može na dinamičan način u različitim trenutcima posluživati veći broj pretplatnika. Sadrži samo jednu tvorničku metodu i to je metoda `subscribe(Subscribers)` koja kao parametar prima `Subscriber` objekt, odnosno objekt pretplatnika

[4] API za asinkrono programiranje s promotrivim tokovima podataka (http://reactivex.io/).
[5] Reaktivna biblioteka četvrte generacije bazirana na specifikaciji *Reaktivnih Tokova* (https://projectreactor.io).

koji se želi pretplatiti na izdavača te koji će konzumirati signale emitirane od strane izdavača na kojeg je ujedno i pretplaćen. Ovu metodu moguće je pozvati veći broj puta te se svakim njenim pozivom kreira nova pretplata. Svaka pojedina pretplata odnosi se samo na pojedinog pretplatnika. Generični tip `T` na razini klase predstavlja tip elementa kojeg izdavač signalizira.

```
public interface Publisher<T> {
    void subscribe(Subscriber<? super T> var1);
}
```

Kôd 8: Sučelje `Publisher`

- **`Subscriber<T>`** – Ovo sučelje se koristi za implementaciju vlastitog rješenja pretplatnika te isto kao i kod sučelja izdavača, generični tip `T` na razini klase predstavlja tip signaliziranog elementa. Sučelje sadrži četiri javne metode:
  - **`onSubscribe(Subscription var1)`** – Za parametar prima objekt pretplate i poziva se onog trenutka kada se izdavaču proslijedi instanca pretplatnika kroz njegovu `subscribe` metodu `Publisher.subscribe(Subscriber)`. Proslijeđeni objekt pretplate može se iskoristiti za odjavu pretplate na događaje ali se isto koristi i za pozivanje `request(long n)` metode proslijeđenog objekta pretplate `Subscription.request(long)`. `Request` metoda aktivira `onNext(T var)` metodu `Subscriber` sučelja onoliko puta koliki je iznos parametra. Važno je naglasiti da podaci neće početi teći, odnosno da `Publisher` objekt neće početi slati nove podatke sve dok se `request` metoda ne pozove od strane `Subscriber` objekta koji je ujedno zadužen za njeno pozivanje kadgod su potrebni novi podaci.
  - **`onNext(T var1)`** - Ova metoda poziva se onoliko puta koliko je definirano `Subscription.request(long)` metodom. Za parametar prima generički tip podataka koji je jednak instanci klase koja je definirana kod kreiranja toka, odnosno `Publisher` objekta kroz generički tip. Koristi se kod `Consumer` objekta za obradu vrijednosti događaja dobivenog iz toka.
  - **`onError(Throwable var1)`** – Za parametar prima objekt klase `Throwable` koji označava pojavu greške ili iznimke. Poziva se u trenutku pojave greške unutar toka te označava prekid toka odnosno njegov neuspjeli dovršetak. Greška se signalizira svim pretplatnicima. Signaliziranjem neuspjelog dovršetka toka i njegovim prekidom, daljnji

podaci neće biti poslani kroz `onNext` metodu pa čak i ako se ponovo pozove `request` metoda.

- **`onComplete()`** – Ne prima ni jedan parametar. Označava prekid toka ali za razliku od `onError` metode pretplatnicima se signalizira uspješan dovršetak toka. Kao i kod prethodne metode, nakon signaliziranja dovršetka, tok podataka se prekida te se novi podaci više neće slati kroz `onNext` metodu ni u slučaju poziva `request` metode.

```
public interface Subscriber<T> {
    void onSubscribe(Subscription var1);
    void onNext(T var1);
    void onError(Throwable var1);
    void onComplete();
}
```

Kôd 9: Sučelje `Subscriber`

- **`Subscription<T>`** – Koristi se za implementaciju pretplate koja predstavlja poveznicu između objekta izdavača i objekta pretplatitelja. Svaka nova pretplata na izdavača stvara novu pretplatu što znači da se ista može koristiti od strane samo tog istog pretplatitelja. Koristi se u svrhe potraživanja podataka ali i za raskidanje potražnje te čišćenje resursa. Sadrži dvije javne metode:
  - **`request(long n)`** – Poziva se od strane `Subscriber` objekta i za parametar prima strogo pozitivan broj zatraženih podataka pretplatnika od izdavača. Metodu je moguće pozvati koliko god puta je potrebno ali nepodmirena kumulativna potražnja ne smije nikada prijeći maksimalnu vrijednost long tipa (`Long.MAX_VALUE`). `Publisher` objekt ima mogućnost poslati manji broj podataka nego što je zatraženo od njega u slučajevima kada se dogodi njegovo iscrpljivanje. U tim slučajevima osim podataka, izdavač emitira i signal dovršenosti.
  - **`cancel()`** – Ova metoda se koristi kada se želi zatražiti od izdavača prekid slanja novih podataka i čišćenje odnosno oslobađanje resursa. Kako je ovo asinkroni zahtjev, može se dogoditi slanje određenog broja podataka s ciljem zadovoljavanja prethodno signaliziranog zahtjeva iako se prije toga uspješno izvršio poziv `cancel` metode.

```
public interface Subscription {
    void request(long var1);
    void cancel();
}
```

Kôd 10: Sučelje `Subscription`

- **`Processor<T, R>`** - Procesori predstavljaju specijalnu vrstu izdavača koji su ujedno i pretplatnici te kao takav poštuje oba ugovora. To znači da je moguće pretplatiti se na procesor ali je isto tako moguće pozvati i metode za ručno ubacivanje vrijednosti u niz te isti prekinuti. Kako bi to bilo moguće, ovo sučelje nasljeđuje sučelje `Subscriber<T>` i sučelje `Publishe<R>` ali ne posjeduje vlastite javne metode. U većini slučajeva preporučuje se izbjegavanje korištenja procesora iz razloga što ih je iznimno teško pravilno implementirati, a time i koristiti iz razloga što sadrži specifične kutne slučajeve.

```
public interface Processor<T, R>
        extends Subscriber<T>, Publisher<R> {
}
```

Kôd 11: Sučelje `Processor`

Sve biblioteke i okviri imaju mogućnost implementirati ova četiri sučelja te se potiče da se ista implementiraju u punom opsegu kako bi se postigla univerzalnost i interoperabilnost između različitih implementacija.

5.2. Reaktivne ekstenzije (ReactiveX)

Reaktivne ekstenzije ili skraćeno ReactiveX predstavlja skup biblioteka za izradu asinkronih programa temeljenih na događajima uz korištenje nizova koji se mogu promatrati. ReactiveX proširuje uzorak promatrača kako bi bilo moguće raditi s nizovima podataka i događajima te nudi mogućnost korištenja velikog broja operatora koji omogućuju deklarativno manipuliranje tim istim nizovima istovremeno apstrahirajući rad s dretvama, sinkronizacijom, konkurentnim strukturama podataka i neblokirajućim izlazno ulaznim operacijama.

	Pojedinačni podaci	***Višestruki podaci***
Sinkrono	`T getData()`	`Iterable<T> getData()`
Asinkrono	`Future<T> getData()`	**`Observable<T> getData()`**

Tablica 2: Prikaz smještaja apstrakcije `Observable`

ReactiveX `Observable` predstavlja apstrakciju koja podržava emitiranje pojedinačnih skalarnih vrijednosti ali isto tako i emitiranje nizova podataka koji mogu biti neograničeni. Biblioteka nije pristrana prema određenoj implementaciji konkurentnosti ili asinkronosti već se

`Observable` može implementirati na različite načine kao što su koristeći bazene dretvi, petlju događaja, neblokirajuće I/O, aktore ili uz pomoć vlastite implementacije (*ReactiveX*, bez dat.).

Prednosti ove biblioteke leže u tome što podržava implementaciju za različite jezike i okvire kao što su Java, JavaScript, C#, Scala, Clojure, C++, Ruby, Dart, .NET, Swift, Android, Netty i još mnogi drugi. Međutim, reaktivne ekstenzije se neće koristiti u izradi praktičnog dijela ovog rada stoga se iste neće dalje obrađivati.

5.3. Projekt Reactor

Projekt Reactor (eng. Project Reactor) predstavlja reaktivnu biblioteku za asinkrono programiranje napravljenu od strane tvrtke Pivotal Software[6], a koja kao i ostale reaktivne biblioteke i okviri implementira specifikaciju reaktivnih tokova. Reactor se koristi za razvoj aplikacija visokih performansi za JVM, a moguće ga je koristiti s Java verzijom 8 pa nadalje. Veoma je sličan reaktivnoj ekstenziji RxJava ali s pojednostavljenim apstrakcijama za rad. Biblioteka se primarno koristi za razvoj aplikacija baziranih na Spring okviru te glavnu primjenu pronalazi u okviru za razvoj Java web aplikacija naziva Spring Webflux. Iako se primarno koristi za razvoj Spring aplikacija, postoji mogućnost njegove primjene i za razvoj Android aplikacija, ali se to ne preporučuje. U idućim potpoglavljima obradit će se glavne i specifične značajke ove biblioteke te će korištenje istih biti prikazano u kratkim programskim isječcima. U idućem potpoglavlju vidjet ćemo koji su to reaktivni tipovi podataka koje biblioteka Reactor nudi na korištenje.

5.3.1. Flux i Mono

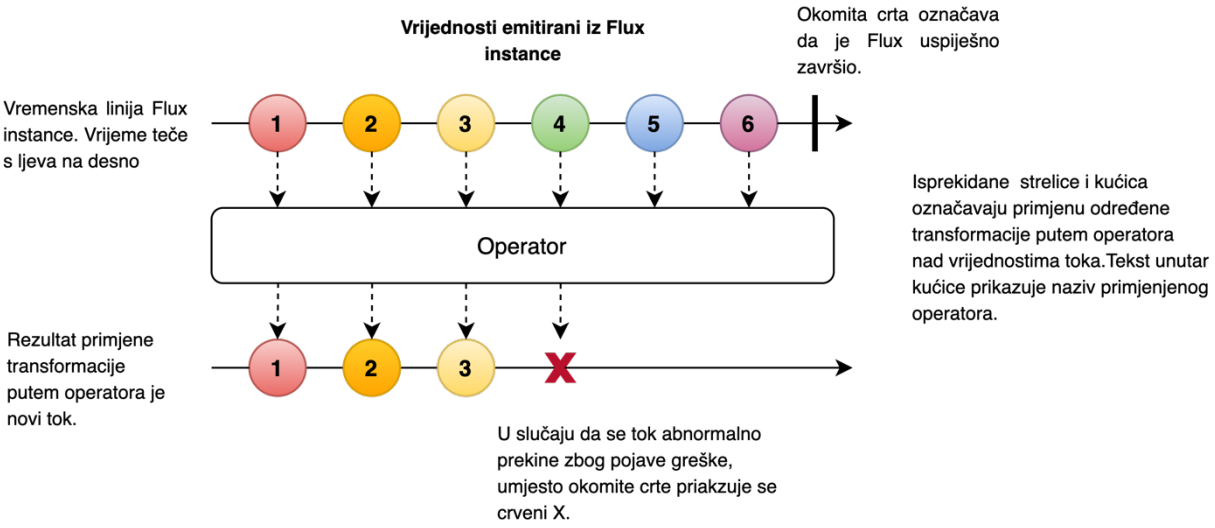
Dva glavna reaktivna tipa biblioteke Reactor jesu **`Flux`** i **`Mono`** čije se instance kreiraju putem tvorničkih metoda (eng. *Factory method*) istoimenih apstraktnih klasa. Obje klase implementiraju sučelje **`Publisher`** prema specifikaciji reaktivnih tokova. Reactor koristi `Flux` i `Mono` za emitiranje vrijednosti u obliku asinkronog niza. Razlika između njih je u tome što `Flux` predstavlja standardnu implementaciju `Publisher` sučelja te reprezentira asinkroni niz koji se može sastojati od nula, jednog pa do N emitiranih vrijednosti gdje N teži u beskonačnost. S druge strane `Mono` predstavlja specijaliziranu implementaciju sučelja `Publisher` i koristi se za emitiranje samo jedne vrijednosti s time da kao i `Flux` ne mora sadržavati ni emitirati vrijednost. U slučaju da ne emitira vrijednost, tada se na njega može gledati kao na asinkroni procesa bez vrijednosti koji sadrži samo koncept dovršetka te se koristi za obavještavanje klijenta o dovršenosti neke akcije. Usporedba se može povući sa

[6] Američka multinacionalna tvrtka sa sjedištem u San Francisco. Od prosinca 2019. Pivotal je postao dio VMware-a (https://tanzu.vmware.com/).

sučeljem `Runnable` u Javi. Oba tipa završavaju s emitiranjem novih podataka na temelju dva signala: signala dovršetka i signala pogreške. Međutim, razlika između ova dva tipa nije samo u kardinalnosti, odnosno broju sadržanih vrijednosti, već i u skupu operatora koje pružaju za rad. Kako je `Mono` ograničen na rad s nula ili jednom vrijednošću dolazimo do situacije u kojoj određeni operator, pogotovo oni koji su specifični za rad s više vrijednosti, nisu primjenjivi. Stoga takvi operator nisu dostupni za korištenje nad `Mono` instancama te posljedično ovaj tip podržava samo podskup mogućih operatora koji su inače dostupni za tip `Flux` uz dodatak sebi svojstvenih operatora koji su primjenjivi samo za njega. `Mono` i `Flux` ne predstavljaju odvojene tipove te se zbog toga mogu vrlo lako transformirati iz jednog u drugog putem određenih operatora. Nadalje, biblioteka je dovoljno pametna da optimizira određene transformacije koje ne utječu na promjenu semantike (Dokuka & Lozynskyi, 2018).

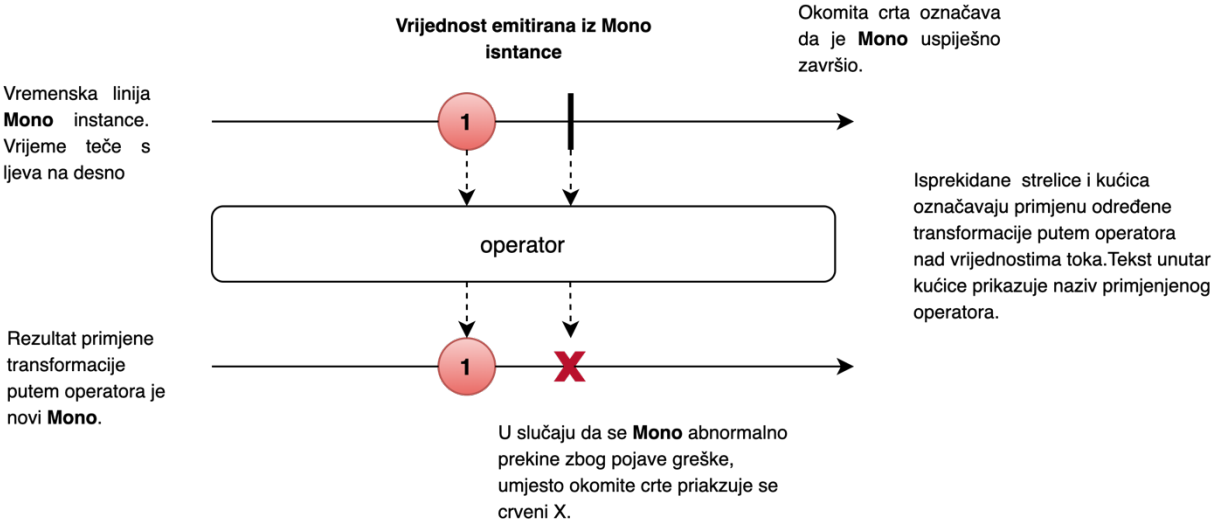
Razlog zašto je nastala ova podjela leži u tome što se uvidjelo da postoje situacije u kojima jednostavno nije potrebno, a nekad ni moguće emitirati veći broj vrijednosti od jedne. Jedan takav primjer moguće je navesti kod rada web aplikacija koje za svakih HTTP zahtjev stvaraju i vraćaju samo jedan odgovor. Stoga u ovom slučaju nema previše smisla vratiti rezultat kao tip `Flux<HttpResponse>` već se u takvim situacijama rezultat izražava kao tip `Mono<HttpResponse>`. Štoviše, na ovaj način omogućeno nam je davanja dodatnog značenja potpisima metoda ali nam isto tako i omogućava učinkovitiju implementaciju `Mono` tipa zbog uklanjanja potrebe za korištenjem suvišnih spremnika i skupih sinkronizacija. Na iduće dvije slike možemo vidjeti grafički prikaz tipova `Flux` i `Mono` u obliku mramornog dijagrama koji se koristi za vizualizaciju tokova, emitiranih vrijednosti i transformacija uzrokovanih primjenom različitih operatora.

Prva slika prikazuje vizualizaciju `Flux` reaktivnog tipa u kontekstu reaktivnih tokova. Gornja linija (strelica) predstavlja vremensku crtu na kojoj se nalaze kružići označeni brojevima koji predstavljaju emitirane vrijednosti od strane promatranog proizvođača. Na lijevom kraju gornje strelice nalazi se okomita crta koja označava kraj dovršetka emitiranja vrijednosti. To znači da poslije tog trenutka više neće biti novo emitiranih vrijednosti od proizvođača. Kućica s nazivom „Operator“ predstavlja bilo koji operator koji imamo na raspolaganju koristeći Reactor biblioteku. Operatori vrše transformaciju emitiranih vrijednosti te takve transformirane vrijednosti emitiraju dalje na novi tok podataka. Crveni znak X na donjoj strelici predstavlja abnormalni prekid toka zbog pojave greške kod transformacije vrijednosti u sklopu operatora.



Slika 19: Primjer vizualizacije `Flux` tipa u sklopu mramornog dijagrama (Prema: *Project Reactor - Documentation*, 2020)

Druga slika prikazuje slični mramorni dijagram ali za `Mono` tip. U ovom dijagramu možemo primijetiti da za razliku od prethodnog, gornja strelica ili tok podataka sadrži samo jednu emitiranu vrijednost. Razlog toga je što `Mono` može sadržavati 0 ili najviše 1 vrijednost, za razliku od tipa `Flux` koji može sadržavati beskonačni broj vrijednosti. Iako sadrži samo jednu vrijednost, sva pravila koja su primjenjiva za `Flux`, primjenjiva su i za `Mono` kao što možemo vidjeti iz slike ispod.



Slika 20: Primjer vizualizacije `Mono` tipa u sklopu mramornog dijagrama (Prema: *Project Reactor - Documentation*, 2020)

5.3.2. Pretplata

Kada pričamo o pretplaćivanju na `Flux` i `Mono`, postoji više mogućih varijanti metode `subscribe()` ali isto tako i veći broj preopterećenja metode koje primaju različiti broj parametara. Neke od tih preopterećenja uz opis rada metoda možemo vidjeti u sljedećem isječku.

```
/* Pretplati se i započni emitiranje. */
subscriber();

/* Consumer predstavlja objekt koji konzumira emitiranu vrijednost. */
subscribe(Consumer<? super T> consumer);

/* Rukuj emitiranim vrijednostima, ali reagiraj i na greške. */
subscriber(Consumer<? super T> consumer,
           Consumer<? super Throwable> errorConsumer);

/* Rukuj s vrijednostima i greškama te izvrši neki kod nakon uspješnog
dovršetka emitiranja. */
subscribe(Consumer<? super T> consumer,
          Consumer<? super Throwable> errorConsumer,
          Runnable completeConsumer);

/* Sve iznad navedeno, plus učini nešto s objektom pretplate nastalog
putem ovog pretplatničkog poziva. */
subscribe(Consumer<? super T> consumer,
          Consumer<? super Throwable> errorConsumer,
          Runnable completeConsumer,
          Consumer<? super Subscription> subscriptionConsumer);
```

Kôd 12: Primjer mogućih varijanti metode `subscribe` s različitim brojem parametara

Sve navedene metode za pretplatu vraćaju referencu na pretplatu koja se može iskoristiti za raskidanje pretplate u slučaju kada vrijednosti više nisu potrebne. Nakon raskida pretplate proizvođač prestaje s emitiranjem novih vrijednosti određenom potrošaču te izvršava čišćenje resursa koje je stvorio. Ovo ponašanje u Reactoru predstavljeno je posebnim dodanim sučeljem koje nije dio biblioteke reaktivnih tokova, a zove se **`Disposable`**.

Sljedeći programski isječak pokazat će nam korištenje zadnje navedene `subscribe` metode u prethodnom popisu s ciljem stvaranja malo jasnije slike kako pretplata radi. Ova `subscriber` metoda prima četiri parametara. Prvi parametar predstavlja potrošača koji će koristiti emitiranu vrijednost. U ovom slučaju taj potrošač predstavlja instancu `PrintStream`

klase s korištenjem metode `println` koja se koristi za ispisivanje vrijednosti na konzolu. Idući parametar također prima instancu potrošača, međutim ovaj potrošač se koristi za rukovanje s greškama koje se mogu pojaviti tijekom procesa emitiranja vrijednost. U slučaju da se pojavi pogreška, naziv iste će biti ispisan na konzolu. Treći parametar je tip `Runnable` i predstavlja akciju koja će se izvršiti nakon što je emitiranje vrijednosti gotovo. U ovom slučaju nakon završetka emitiranja na konzoli će se ispisati `String` vrijednost „Done“. I zadnji parametar predstavlja povratnu referencu pretplate koja se kreira nakon što se izvrši pretplata. U ovom primjeru uz pomoću reference pretplate signaliziramo izvoru da želimo primiti najviše pet podataka.

```
Flux<String> charFlux = Flux.just("a","b","c");
charFlux.subscribe(
        System.out::println,
        error -> System.err.println("Error: " + error),
        () -> System.out.println("Done"),
        subscription -> subscription.request(5)
);
// Ispis na konzoli:
// a
// b
// c
// Done
```

Kôd 13: Primjer i opis rada jednog od preopterećenja metode subscribe

Kao što smo mogli vidjeti iz ovog primjera, `subscribe` metoda omogućuje nam korištenje lambda izraza za kreiranje logike pretplatnika. Međutim, postoji dodatno preopterećenje metode koja prima objekt klase `Subscriber`, a koji predstavlja kompletnog pretplatitelja. Vlastitu klasa pretplatitelja moguće je kreirati uz pomoć korištenja proširive klase `BaseSubscriber` koju pruža Reactor.

5.3.3. Procesori

Implementaciju sučelja `Processor` reaktivne specifikacije u Reactor okviru moguće je grubo podijeliti u dvije zasebne kategorije. Prvu kategoriju predstavljaju **direktni procesori** koji uključuju `DirectProcessor` i `UnicastProcessor`. Ovi procesori imaju mogućnost guranja podataka samo putem izravnog djelovanja korisnika. Drugu kategoriju predstavljaju sinkroni procesi u koju se ubrajaju `EmitterProcessor` i `ReplayProcessor`. Ova dva procesora posjeduju mogućnost guranja vrijednosti kroz akcije korisnika ili putem iscrpljivanja uzvodne komponente izdavača na kojeg je procesor pretplaćen (*Project Reactor - Documentation*, 2020).

`DirectProcessor` predstavlja procesor koji može osim samog slanja vrijednosti također slati i signale svojim pretplatnicima. Ovaj procesor je najlakši za instancirati i to je moguće putem jedinstvene statičke metode `DirectProcessor.create()`. Međutim, iako sadrži prednost laganog instanciranja, također posjeduje i jedan veliki nedostatak, a to je da ne podržava rad s povratnim tlakom. Kao posljedicu nemogućnosti rada s povratnim tlakom, ovaj procesor će signalizirati `IllegalStateException` svojim pretplatnicima u slučaju kada se kroz procesor pokuša gurnuti broj vrijednosti koji je veći od onih koje su pretplaćeni potrošači zatražili. Dodatna specifičnost ovog procesora je ta što kada procesor završi s emitiranje zbog dovršenosti ili zbog pojave greške, i dalje nudi mogućnost pretplate novim potrošačima, ali im u tom slučaju ne ponavlja prethodno emitirane vrijednosti već im signalizira prekid.

`UnicastProcessor` za razliku od prethodne vrste procesora, ova vrsta posjeduje veće mogućnosti što direktno proizlazi iz postojanja većeg broja statičkih metodi za njegovo stvaranje. Primjerice, korištenjem osnovne metode za stvaranje, ovaj procesor postaje neograničen što znači da se kroz njega može gurnuti bilo koja količina vrijednosti. Sve gurnute vrijednosti bit će spremljeni u njegov interni spremnik. Ovo ponašanje moguće je promijeniti pružanjem vlastite implementacije reda koji će biti korišten kao interni spremnik procesora. U slučaju da je implementacija našeg spremnika ograničena, procesor može odbiti guranje novih vrijednosti sve dok je spremnik pun. Dodatna prednost ove vrste procesa je ta što podržava rad s povratnim tlakom što je iznimno važno kod posjedovanja spremnika koji je ograničen. Međutim, kompenzacija za rad s povratnim tlakom je ta što ova vrsta procesora može imati samo jednog pretplaćenog potrošača u bilo kojem trenutku vremena.

`EmitterProcessor` predstavlja svojevrsnu nadogradnju na `UnicastProcessor` iz razloga što se korištenjem ovog procesora mogu emitirati vrijednosti većem broju pretplaćenih potrošača uz istovremeno poštivanje njihovih povratnih tlakova. Procesor također sadrži i mogućnost primanja te spremanja određenog broja vrijednosti čiji se broj određuje postavkom veličine njegovog unutarnjeg spremnika. Specifičnost ovog procesora je u tome što on ne mora nužno imati pretplaćene potrošače da bi bio u mogućnosti spremiti zaprimljene vrijednosti. Ako se do trenutka popunjavanja unutarnjeg spremnika ne pretplati ni jedan potrošač, metoda `onNext()` se blokira sve dok se spremnik ne isprazni. U slučaju da se svi potrošači odjave, procesor čisti unutarnji spremnik i ukida mogućnost novih pretplata.

`ReplayProcessor` sprema vrijednosti koje mu se mogu direktno gurnuti kroz njegovu `sink()`metodu ili kroz uzvodnog izdavača te koje se na temelju postavke procesora ponavljaju svim pretplaćenim slušačima, naglasak je na one slušače koji su se pretplatili u kasnijem vremenskom periodu. Ovaj procesor moguće je kreirati s nekoliko različitih postavki za spremanje i ponavljanje vrijednosti. To su sljedeće moguće postavke: procesor s

postavkom spremanja posljednjeg elementa, procesor s postavkom koja omogućuje spremanje ograničenog ili neograničenog broja vrijednosti, procesor s postavkom vremenskog okvira unutar kojeg će se spremljene vrijednosti ponoviti slušačima, te procesor s postavkom koja se sadrži kombinaciju vremenskog okvira i broja povijesnih vrijednosti.

5.3.4. Operatori

U ovom potpoglavlju pozabavit ćemo se opisom i radom s operatorima Reactor biblioteke. Operatore možemo generalno podijeliti u dvije velike skupine: statički operatori i objektni operatori. Statički operatori služe nam u većini slučajeva za stvaranje tokova podataka iz izvora koji može biti `Flux` ili `Mono` te ih možemo pronaći kao statičke metode odgovarajućih klasa. Kako je kreiranje tokova prvi i glavni korak u reaktivnom programiranju, statički operatori će se detaljnije obraditi. Druga, veća skupinu operatora jesu objektni operatori. Objektne operatore koristimo za sljedeće primjene: transformacije tokova, spajanje tokova, rukovanje greškama, matematičke operacije, filtriranje i slično. Zbog velikog broja objektnih operatora i velikog broja njihovih primjena, obradit će se samo one primjene i oni operatori koji su najkorisniji te koji se najučestaliji.

5.3.4.1. Stvaranje tokova

Kao što smo već prije spomenuli u teorijskom dijelu reaktivnog programiranja, tokovi podataka predstavljaju kostur reaktivnog programiranja. Stoga, da bismo mogli započeti rad s reaktivnim programiranjem, prvi korak predstavlja stvaranje tokova podataka iz izvora podataka. Izvore podataka u Reactor biblioteci predstavljaju prethodno dva spomenuta reaktivna tipa Flux i Mono. Stvaranje njihovih instanci moguće je učiniti putem brojnih operatora koji dolaze u sklopu njihovih apstraktnih klasa. Tih operatora postoji više s većim brojem preopterećenja, ovisno iz čega i na koji način želimo stvoriti tok podataka. Prvu skupina operatora predstavljaju jednostavnije operatore za stvaranje toka podataka i oni uključuju: **`just, range, from, interval i from.`**

Prvi i najjednostavniji od njih jest operator **`just(T…`** `data)`**`.`** Operator `just` prima kao parametar jedan ili više objekata istog tipa te na temelju njih stvara tok s vrijednostima koje će se emitirati nakon što se dogodi pretplata. Vrijednosti se emitiraju redoslijedom kojim su proslijeđene, od prve prema zadnjoj.

Iduća po redu **`range(int`** `start,` **`int`** `count)` koristi se za stvaranje niza vrijednosti. Operator prima dva `Integer` parametra, prvi koji predstavlja početnu vrijednost niza, i drugi koji predstavlja krajnju vrijednost.

Treći operator **`interval(Duration`** `delay,` **`Duration`** `period)` predstavlja veoma koristan operator za stvaranje neograničenih tokova. On sadrži više preopterećenja

međutim nama je najzanimljivije preopterećenje koje za parametre prima dva objekta klase `Duration`. Objekt `Duration` klase koristi se za definiranje tipa vremenske jedinice ali i za definiranje vrijednosti vremenske periode. Prvi parametar predstavlja vremenskom trajanje odgode prije nego li izvor može započeti s emitiranjem vrijednosti. Drugi parametar označava vremensku jedinicu i njezin iznos koji će se koristiti za periodično emitiranje vrijednosti, a čiji će iznos biti jednak periodi kako vrijeme protječe. Drugim riječima, izvor stvoren na ovaj način emitirat će vrijednosti svakih nekoliko trenutaka sve dokle god se potrošač ne odjavi ili dok se ne dogodi greška koja će rezultirati nasilnim prekidom toka. To znači da ovaj operator stvara neograničeni niz vrijednosti, odnosno beskonačni tok koji je vrlo koristan u slučajevima kada treba spojiti ili grupirati više vrijednosti većeg broja toka na temelju vremenskog perioda. Ovaj operator moguće je koristiti samo na `Flux` instancama.

Četvrti operator **`from`** predstavlja skup operatora za stvaranje toka iz predefiniranog podatka ili pak iz samog izvora. Zanimljivo je uočiti da Flux i Mono sadrže samo jedan zajednički operator iz ovog skupa i to je operator **`from(Publisher<?`** `extends T> source)` koji stvara tok iz samog izvora. Svi ostali operatori iz ovog skupa su međusobno isključivi. Tako za `Flux` imamo mogućnost korištenja sljedećih operatora: `from`, `fromArray`, `fromIterable`, `fromStream`. Dok kod `Mono` imamo puno veći izbor koji se sastoji od: `from`, `fromCallable`, `fromCompletionStage`, `fromDirect`, `fromFuture`, `fromRunnable`.

Neki od prethodno obrađenih operatora za stvaranje toka prikazani su u sljedećem programskom isječku.

```
Consumer consumer = System.out::println;
Flux<Integer> publisher = Flux.range(1, 3);
Flux.just(1, 2, 3).subscribe(consumer);
Flux.range(1, 3).subscribe(consumer);
Flux.from(publisher).subscribe(consumer);
Flux.fromIterable(Arrays.asList(1, 2, 3)).subscribe(consumer);
Mono.fromCallable(() -> "1\n2\n3");
// Ispis na konzoli (ispis za svaki zasebni operator je isti):
// 1
// 2
// 3
```

Kôd 14: Primjer korištenja nekih od operatora za stvaranje tokova

Svaki od operatora navedenih u primjeru koristi se, kao što smo prethodno rekli, za stvaranje toka vrijednosti. Svaki pojedini operator prihvaća drugačiji način odnosno tip ulaznih parametra. U ovom primjeru za sve operatore ulazni parametri su koncipirani tako da svi operatori generiraju isti izlazni tok, odnosno tok koji će sadržavati tri iste emitirane vrijednosti.

To su redom brojčane vrijednosti 1, 2 i 3. Međutim da bismo mogli vidjeti na konzoli ispis emitiranih vrijednosti, potrebno je napraviti pretplatu na svaki pojedinačni tok. To radimo uz pomoć objekta `Consumer` i metode `subscribe`. Prisjetimo se da se emitirane vrijednosti počinju konzumirati tek onda kada se dogodi pretplata na tok.

Idući skup operatora predstavljaju specijalizirane operatore za stvaranje tokova. Razlog zašto za ovaj skup kažemo da predstavljaju specijalizirane slučajeve je taj što ni jedan od sljedeće navedenih operatora ne stvara tok podataka u punom smislu, već zapravo stvaraju krnje tokove. Ovaj skup operatora predstavljaju sljedeći operator: **`never`**, **`error`** i **`empty`**.

Operator `never` stvara tok koji ne emitira ni jednu vrijednost, ali isto tako ne emitira grešku niti signalizira uspješnu dovršenost toka. Kao takav, ovaj operator ima primjenu samo kod testiranja.

Operator `error` kao i prethodna `never` metoda ne emitira vrijednosti ali za razliku od prethodne emitira signal greške nakon čijeg se emitiranja zatvara tok. Ova metoda također pronalazi svoju primjenu kod testiranja.

Zadnji operator `empty` stvara prazni tok podataka što znači da ni jedna vrijednost neće biti emitirana, ali za razliku od prethodnih metoda ovakav tok se uspješno dovršava i kao takav šalje signal dovršenosti svojim pretplatiteljima, ako ih posjeduje. Kao i prethodne dvije metode, ova se također koristi uglavnom samo za testiranje.

Zadnji operator koji ćemo obraditi u ovom potpoglavlju za stvaranje tokova je **`defer`** operator. Ovaj operator kao što i samo ime operatora sugerira ne stvara tok podataka već njegovo stvaranje odgađa odnosno prepušta proslijeđenom `Publisher` objektu, što čini evaluacija izvora podataka lijenim. To znači da će se tok podataka stvoriti tek u trenutku kada se na objekt izdavača pretplati neki pretplatnik, u suprotnom tok ostaje ne kreiran. Osim toga ova metoda za svaku novu pretplatu stvara zasebni tok podataka sa zasebnim nizom vrijednosti. Na idućem primjeru možemo vidjeti korištenje `defer` metode koja pokazuje ove dvije navedene specifičnosti operatora. U primjeru vidimo da svaka nova pretplata na tok podataka kreiran operatorom `defer` rezultira novim sistemskim vremenom u milisekundama, dok s druge strane pretplate na izdavača stvorena na uobičajen način rezultira istim sistemskim vremenom iako je simuliran razmak pretplata korištenjem `Thread.sleep()` metode.

```
Mono<Long> publisher = Mono.just(System.currentTimeMillis());
Mono<Long> clock = Mono.defer(() -> Mono.just(System.currentTimeMillis()));
clock.subscribe(t -> System.out.println("Defer: " + t));
publisher.subscribe(t -> System.out.println("Publisher: " + t));

Thread.sleep(1000);

clock.subscribe(t -> System.out.println("Defer: " + t));
publisher.subscribe(t -> System.out.println("Publisher: " + t));

// Ispis na konzoli:
// Defer: 1594483869304
// Publisher: 1594483869303
// Defer: 1594483870307
// Publisher: 1594483869303
```

Kôd 15: Prikaz `defer` operatora za stvaranje toka

5.3.4.2. Programsko stvaranje toka

U ovom dijelu pokazat ćemo programsko stvaranje `Flux` i `Mono` instanci na način da ćemo im programski definirati pridružene događaje `onNext`, `onError` i `onComplete`. Zajedničko ovim metodama je to da sve one otkrivaju API koji omogućuje izazivanje događaj kojeg nazivamo *sudoper* (eng. *Sink*). Za programsko stvaranje tokova koristimo sljedeći skup operatora: **`generate`**, **`create`**, **`push`**, **`handle`** (*Project Reactor - Documentation*, 2020).

Generate operator predstavlja najjednostavniji način stvaranja `Flux` instance. Osnovna verzija ovog operator `generate(Consumer<SynchronousSink<T>> generator)` za parametar prima generator funkciju te koristi se za sinkrone i pojedinačne emisije vrijednosti. Za pojedinačne emisije koristi se `SynchronousSink` čija se metoda `next()` može pozvati najviše jedanput po invokaciji povratnog poziva. `Complete` i `error` metoda mogu biti naknadno pozvane.

```
public interface SynchronousSink<T> {
    void complete();

    Context currentContext();

    void error(Throwable var1);

    void next(T var1);
}
```

Kôd 16: `SynchronousSink` sučelje

Najkorisnija varijanta ove metode predstavlja onu koja omogućuje čuvanje stanja koje se koristi za referenciranje kod korištenja sudopera te na temelju kojeg se odlučuje koju

vrijednost ili signal emitirati. U toj varijanti generator operator prima dva parametra prvi koji predstavlja početno stanje i drugi parametar koji predstavlja generator funkciju koja sada postaje tipa `BiFunction<S, SynchronousSink<T>, S>` gdje generični tip `S` predstavlja tip stanja objekta. Korištenjem ove verzije metode generator funkcija vraća novo stanje svakim novim izvršavanjem. U sljedećem kôdu možemo vidjeti korištenje verzije `generate` metode koja čuva stanje. U ovom primjeru početno stanje postavljamo na brojčanu vrijednost 1, te nakon svakog novog izvršavanja, stanje povećavamo za 1 sve dok ne dođemo do vrijednosti koja je djeljiva s brojem 4. Nakon toga terminiramo tok sa signalom dovršenosti.

```
Flux<String> flux = Flux.generate(
        () -> 1,
        (state, sink) -> {
            if (state % 4 == 0) {
                sink.complete();
            }
            sink.next(state.toString());
            return state + 1;
        }
);
flux.subscribe(System.out::println);
// Ispis na konzoli:
// 1
// 2
// 3
```

Kôd 17: Primjer `generate` operatora

Idući operator predstavlja napredniji oblik programskog stvaranja toka, a to je operator **`create`**. Ovaj operator pogodan je za višestruke emisije po izvršavanju. Za razliku od `generate` operatora, `create` operator nema varijantu koja koristi stanje ali zato može okinuti višedretvene događaje u povratnom pozivu. Korištenjem ovog operatora moguće je napraviti poveznicu između postojećeg asinkronog ali i sinkronog API-a s reaktivnim svijetom. Važno je naglasiti da `create` iako posjeduje mogućnost aktiviranja višedretvenih događaja te mogućnost korištenja u skladu s asinkronim programskim sučeljima, on ne čini kôd asinkronim niti ga paralelizira. Operator `create` posjeduje još jednu dodatnu mogućnost, a to je rukovanje s povratnim tlakom te je ovaj operator moguće koristiti nad `Flux` ali i `Mono` instancama. U sljedećem kôdu možemo vidjeti jedan od mnogih primjera `create` operatora u kojem demonstriramo na koji način možemo rezultat blokirajućeg API-a preslikati u reaktivni tok podataka. Međutim ovo je vrlo jednostavan primjer, ondje gdje upotreba ovog operatora sjaji

je kod rada s asinkronim API-em gdje je onda korisno koristiti i strategiju upravljanja povratnim tlakom.

```
public Flux<User> findAllUsers(){
    return  Flux.create(userFluxSink -> {
        userService.findAllUsers().forEach(user -> {
           userFluxSink.next(user);
        });
        userFluxSink.complete();
    });
}
```

Kôd 18: Operator `create` kao most između sinkronog i reaktivnog svijeta

Idući operator, operator **`push`** predstavlja srednju opciju između operatora `generate` i operatora `create`. Ovaj operator omogućuje procesiranje događaja od samo jednog izdavača te je sličan operatoru `create` u smislu da također može biti asinkron i da može upravljati povratnim tlakom korištenjem određenih strategija. Razlika naspram `create` operatora je ta što ne posjeduje mogućnost višedretvenog okidanja događaja, već metode `next`, `complete` i `error` mogu biti pozvane od strane samo jedne dretve u pojedinom trenutku. Korisno je spomenuti da operator `push` ali i operator `create` zapravo prate gurni/povuci model kao i većina ostalih Reactor operatora. To znači da iako je procesuiranje većinom asinkrono što znači model guranja, i dalje postoji mali dio koji se bazira na modelu povlačenja i on se realizira u obliku `request` metode za koju znamo da signalizira izdavaču spremnost za primanje novih podataka. Oba `push` i `creator` operatore nude mogućnost korištenja `onRequest` metode kako bi mogli upravljati traženim iznosom podataka i kako bi se isti gurali kroz sudoper tek onda kada postoji zahtjev za podacima.

Zadnji operator za programsko stvaranje tokova je operator **`handle`**. Ovaj operator moguće je koristiti nad `Flux` i `Mono` tipovima podataka te predstavlja objektni operator. To znači da ovaj operator sam po sebi ne kreira tok podataka već se veže na postojeći izvor podataka kao i ostali obični operatori. Sličan je operatoru `generate` u smislu da kao i on koristi `SynchronosuSink` te da dopušta samo pojedinačne emisije. Razlika je što handle operator može biti korišten za stvaranje proizvoljnih elemenata iz svakog izvornog elementa. Korisna primjera ovog operatora je kada imamo predefiniranu metodu koju želimo koristiti za mapiranje vrijednosti, međutim predefinirana metoda ponekad vraća `null` vrijednosti. U specifikaciji reaktivnih tokova `null` vrijednosti nisu dopuštene u tokovima. Stoga korištenjem `map` operatora o kojem će više biti riječi kasnije nam u ovom slučaju ne bi pomogao. Da bi se riješio ovakav slučaju možemo koristiti operator `handle` kao što možemo vidjeti u sljedećem programskom primjeru.

```
Flux<Integer> numbers =
        Flux.just(1, 2, 3, 4)
            .handle((i, sink) -> {
                //Može vratiti null vrijednost
                Integer n = customMap(i);
                if (n != null) {
                    sink.next(n);
                }
            });
```

Kôd 19: Korištenje `handle` operatora za izbjegavanje pojave `null` vrijednosti

5.3.4.3. Transformacija toka

Transformacija tokova predstavlja vrlo čestu aktivnost koju je potrebno izvršavati nad tokovima podataka bilo to zbog promjene tipa emitiranog podatka, ekstrahiranja određene vrijednosti, grupiranja većeg broja podataka, spljoštenja većeg broja sekvencijalno emitiranih nizova podataka u jedan niz i slično. Za primjenu transformacije tokova opisat ćemo četiri važna objektna operatora: **`map`**, **`flatMap`**, **`buffer`**, **`window`**, **`scan`** i **`groupBy`** (*Project Reactor - Documentation*, 2020).

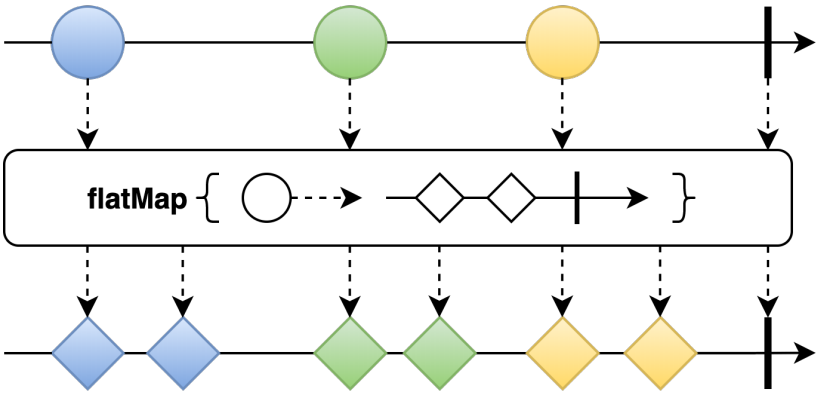
`Map` operator je najjednostavniji operator transformacije toka. Ovaj operator uzima jednu po jednu vrijednost s ulaznog toka i nad njima vrši određenu transformaciju te ih prosljeđuje na izlazni tok. Glavna primjena ovog operatora je u slučajevima kada iz kompleksnog objekta želimo dobiti samo određenu vrijednost nekog njegovog atributa ili iskoristiti neku njegovu metodu za dobivanje tražene vrijednosti. U sljedećem primjeru možemo vidjeti korištenje `map` operatora u svrhe izračunavanja ukupnog iznosa složenog objekta košarice.

```
Flux<Cart> fluxCart = Flux.just(
        cartService.getCart(userService.getCurrentUser())
);
fluxCart.map(cart -> cart.calculateTotalPrice())
        .subscribe(System.out::println);
```

Kôd 20: `map` operator

Idući operator po redu je **`flatMap`** operator. Ovaj operator veoma je sličan prethodnom u smislu da se temelji na istom principu ali se razlikuje po tome što za jednu ulaznu vrijednost može kreirati nula, jednu ili više vrijednosti u novom toku, za razliku od `map` operatora koji za jednu ulaznu vrijednost uvijek vraća samo jednu izlaznu vrijednost. Možemo reći da `flatMap` operator ulazne vrijednosti raspetljava u veći broj pojedinih vrijednosti od koje je ulazna

vrijednost sastavljena. Kako bi se lakše razumjelo djelovanje ovog operatora, njegova vizualizacija može se vidjeti na sljedećem mramornom dijagramu i u programskom kôdu 21.



Slika 21: Mramorni dijagram `flatMap` operatora (Prema: Project Reactor - Documentation, 2020)

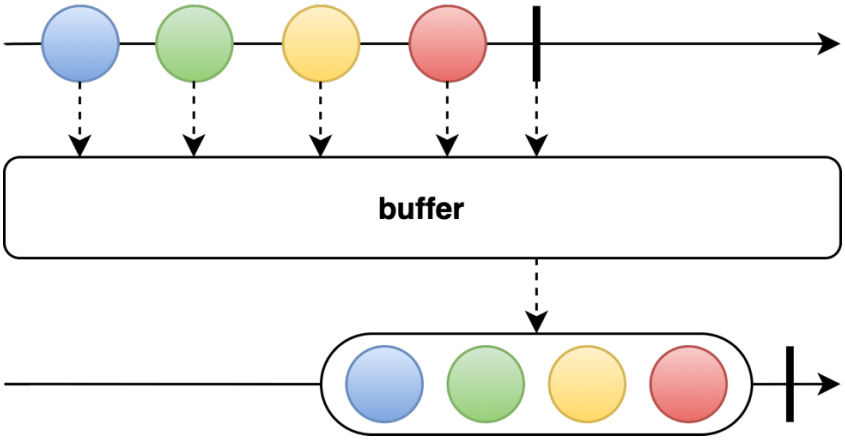
Kao što možemo vidjeti u programskom isječku, imamo stvorenu listu `String` vrijednosti. Tu listu koristimo za stvaranje toka podataka tako što ju prosljeđujemo `just` operatoru koji će proslijeđenu listu emitirati na novo kreirani tok. Nakon toga na isti `Flux` objekt vršimo dvije pretplate. Prva pretplata sadrži samo pretplatnika koji ispisuje zaprimljenu emitiranu listu. Na konzoli možemo vidjeti ispis oblika [a, b, c] što znači da je cijela lista emitirana kao jedna vrijednost. Međutim, prije iduće pretplate iskoristili smo mogućnost primjene objektno operatora `flatMap` koji nam služi da iz jedne emitirane vrijednosti emitiramo n vrijednosti gdje je n u ovom slučaju jednak broju elemenata same liste. Zbog toga ispis na konzoli kod druge pretplate razlikuje od prethodnog i u ovom slučaju ispisuje svako pojedino slovo liste u novi red.

```
List<String> chars = Arrays.asList("a", "b", "c");
Flux<List<String>> flux = Flux.just(chars);
flux.subscribe(System.out::println);
flux.flatMap(c -> {
    return Flux.fromIterable(c);
}).subscribe(System.out::println);
// Ispis na konzoli:
// [a, b, c]
// a
// b
// c
```

Kôd 21: `flatMap` operator

`Buffer` operator predstavlja operator koji periodički sakuplja emitirane vrijednosti izvora u pakete i kao takve ih emitira na tok podataka umjesto da se svaka vrijednost zasebno emitira. Postoji veliki broj varijanti `buffer` operatora i oni se razlikuju u načinima odabira

emitiranih vrijednosti koje će se sakupljati u pakete, definiranju veličine paketa, vremenskog okvira i slično. U idućem mramornom dijagramu možemo vidjeti primjer `buffer` operatora koji skuplja četiri po četiri emitirane vrijednosti u pakete i potom te pakete emitira na izlazni tok. Ovo je varijanta `buffer` operatora koji kao parametar prima veličinu paketa. U slučaju da tok ne sadrži dovoljan broj vrijednosti za punjenje paketa, operator također može emitira i ne popunjeni paket.



Slika 22: `Buffer` operator (Prema: Project Reactor - Documentation, 2020)

U idućem programskom isječku možemo vidjeti primjer korištenja `buffer` operatora na podatkovnom toku. Kreiramo tok koji sadrži tri emitirane vrijednosti 1, 2 i 3. `Buffer` operator predstavlja lančani operator te ga stoga možemo nadovezati na sam operator kreiranja. Kao parametar prosljeđujemo brojčanu vrijednost 2 koja definira da će se vrijednosti proslijediti dalje nizvodno na tok tek kada se popuni spremnik s dvije emitirane vrijednosti. U ovom primjeru emitiraju se tri vrijednosti tako da je ispis na konzoli podijeljen u dvije linije. Prva linija koja sadrži prve dvije emitirane vrijednosti [1, 2] i druga linija koja sadrži samo jednu vrijednost [3]. Kao što smo prethodno rekli u slučaju da tok ne sadrži dovoljan broj vrijednosti `buffer` operator dopušta i emitiranje manjeg broja vrijednosti.

```
Flux.just(1, 2, 3)
    .buffer(2)
    .subscribe(System.out::println);
//Ispis na konzoli:
//[1, 2]
//[3]
```

Kôd 22: Primjer `buffer` operatora

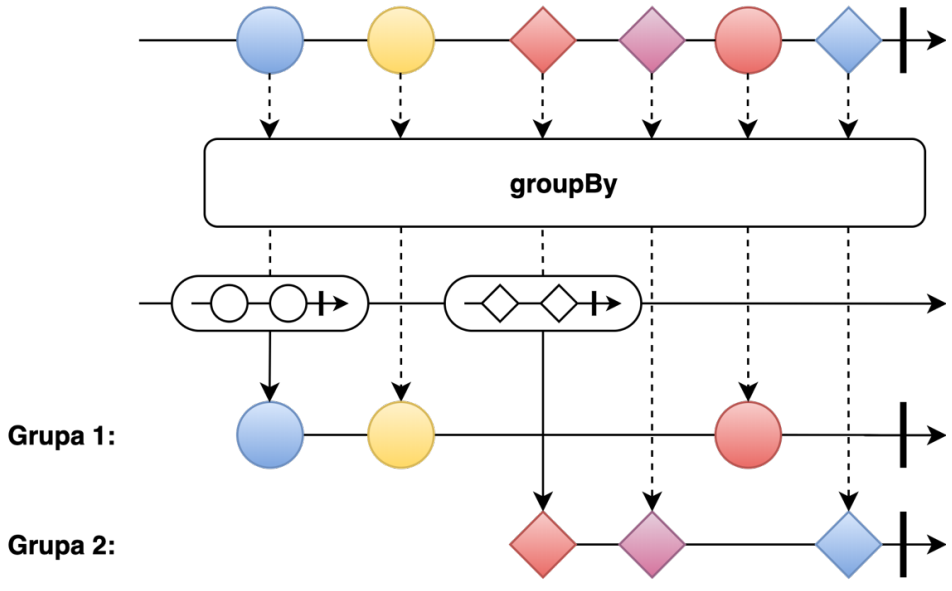
Idući po redu operator je **`scan`** operator. `Scan` operator radi na način da iz ulaznog toka podataka uzima prethodnu i trenutnu emitiranu vrijednost te nad njima primjenjuje proslijeđenu funkciju čiji rezultat vraća na izlazni tok. Rezultat te funkcije koristi se kao

prethodna vrijednost u sljedećem pozivu, proces se ponavlja dok se ne prođe kroz sve vrijednosti toka, odnosno ponavlja se n-1 puta zbog toga što prva vrijednost na toku nema prethodnika. Primjena ovog operatora korisna je kod kumulativnih izračuna vrijednosti ili kod izračuna brzine kretanja temeljene na GPS koordinatama i slično.

```
Flux.just(1, 2, 3)
    .scan((n1, n2) -> {
        System.out.printf(String.format("(%d + %d) = ", n1, n2));
        return n1 + n2;
    })
    .subscribe(System.out::println);
// Ispis na konzoli:
// 1
// (1 + 2) = 3
// (3 + 3) = 6
```

Kôd 23: Kumulativno zbrajanje primjenom `scan` operator

Predzadnji operator koji ćemo obraditi je operator **`groupBy`**. Ovaj operator koristi se za grupiranje vrijednosti po vlastitom definiranom kriteriju. Sličan kao `buffer` operator ali daleko opširniji zbog mogućnosti implementacije vlastitog kriterija grupiranja i samih grupa. Da bi grupiranje bilo moguće potrebno je osim samog kriterija funkciji proslijediti i ključeve na temelje kojih će se vršiti grupiranje. Za svaki proslijeđeni ključ kreirat će se novi tok klase `GroupedFlux` koji će sadržavati vrijednosti svrstane u pojedinu grupu. Na sljedećem mramornom dijagramu možemo vidjeti vizualizaciju ovog operatora. U ovom slučaju grupiranje se vrši po obliku. Pa tako imamo dvije grupe, grupa 1 koja sadrži samo krugove, i grupu 2 koja sadrži samo dijamante.



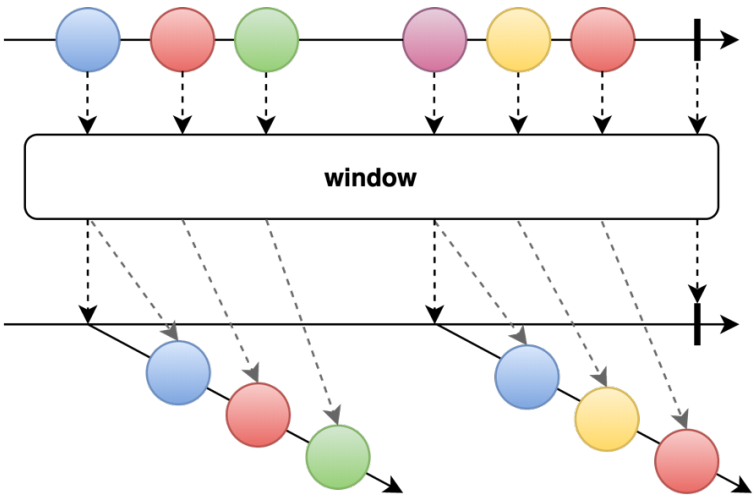
Slika 23: `groupBy` operator (Prema: Project Reactor - Documentation, 2020)

Isti princip možemo vidjeti i u programskom kôdu 24 koji pokazuje način na koji se može izvršiti grupacija niza brojeva na parne i neparne. Primjetimo korištenje `flatMap` operatora. Bez njegovog korištenja kao ispis bismo dobili naziv klase dviju instanca koje odgovaraju našim dvjema grupama, u ovom slučaju to su `UnicastGroupedFlux` klase.

```
Flux.just(3, 6, 2, 7, 10, 34)
    .groupBy(n -> n % 2 == 0 ? "Even" : "Odd")
    .flatMap(group -> group.buffer().map(v -> group.key() + " : " + v))
    .subscribe(System.out::println);
// Ispis na konzoli:
// Even : [6, 2, 10, 34]
// Odd : [3, 7]
```

Kôd 24: Primjer `groupBy` operatora za grupiranje vrijednosti

Zadnji operator koji pripada skupini ili kategoriji operatora za transformaciju tokova je operator **`window`**. Na ovaj operator možemo gledati kao na kombinaciju operatora `buffer` i operatora `groupBy`. Naime, ovaj operator kao i `buffer` operator periodički skuplja emitirane vrijednosti i emitira ih u skupinama ili prozorima radije nego da emitira pojedinačne vrijednosti. Međutim, za razliku od `buffer` operatora, skupine vrijednosti se emitiraju na zasebne izlazne tokove, slično kao što to radi i operator `groupBy`. Međutim razlika je što u `groupBy` imamo određeni broj novih tokova koji je jednak broju predefiniranih grupa. Kod `window` operatora broj novokreiranih tokova može biti beskonačan, ovisno o varijanti operatora.

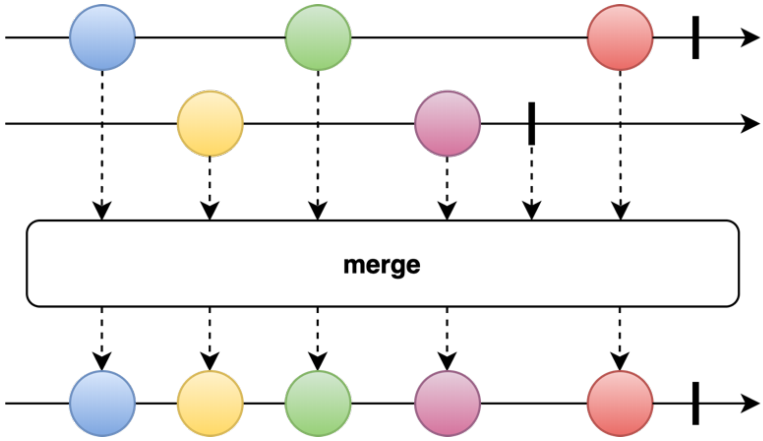


Slika 24: Mramorni dijagram `window` operatora (Prema: Project Reactor - Documentation, 2020)

5.3.4.4. Spajanje tokova

U radu s tokovima podataka u reaktivnom programiranju vrlo često ćemo se naći u situaciji da trebamo spojiti dva ili više različitih tokova podataka većinom iz potrebe spajanja emitiranih vrijednosti kako bi iz njih dobili neku za nas korisnu informaciju ili smisleno značenje. Spajanje tokova moguće je izvršiti putem brojnih statičkih ali i objektnih operatora `Flux` i `Mono` klasa. Za spajanje tokova postoji veliki broj statičkih ali isto tako i objektnih operatora `Flux` i `Mono` klasa. U ovom poglavlju obradit će se samo nekolicina njih dok će ostatak biti samo spomenut. Operator koje ćemo obraditi jesu sljedeći: **`merge`**, **`zip`**, **`concat`**, **`concatWith`** i **`combineLatest`** (*Project Reactor - Documentation*, 2020).

Prvi na redu operator **`merge`**. Ovaj operator sadrži mnogo varijanti, međutim najjednostavnija služi za spajanje dvaju ili više tokova podataka ili emitiranih nizova vrijednosti dvaju izdavača u obliku isprepletenog niza. Važno je spomenuti da je ovaj operator namijenjen za spajanje konačnih tokova, odnosno tokova koji će u jednom trenutku završiti s emitiranjem vrijednosti. U slučaju da se s ovim operatorom pokušaju spojiti tokovi od kojih je barem jedan neograničen dovesti će do blokiranja jer se spajanje nikad neće dovršiti. Interesantno je za spomenuti da je ovaj operator suprotan operatoru `concat` u smislu da se kod `merge` operatora vrijednosti spajaju željno (eng. *Eager*) za razliku od operatora `concate` koji ih spaja lijeno (eng. *Lazy*). To znači da će ovaj operator rezultirati nizom vrijednosti drugačijeg poretka ako usporimo, ubrzamo ili dodamo vremensku odgodu emitiranje vrijednosti. Na sljedećem mramornom dijagramu možemo vidjeti primjer operatora `merge`.



Slika 25: Vizualizacija `merge` operatora (Prema: Project Reactor - Documentation, 2020)

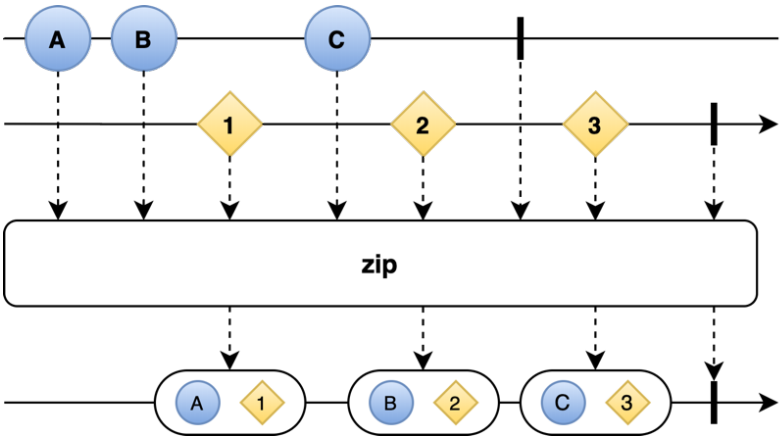
U idućem programskom isječku možemo vidjeti primjenu `merge` operatora kod spajanja dvaju tokova podataka. Na početku programskog koda inicijaliziramo `delay` varijablu koja će sadržavati `Duration` objekt s vrijednosti od 10 milisekundi. Nakon toga kreiramo dva

različita toka podataka uz pomoć korištenja `range` operatora. Prvi tok `numbers1` sadrži vrijednosti 1, 2 i 3 dok drugi tok `numbers2` sadrži vrijednosti 4 i 5. Kao što smo rekli `merge` operator spaja vrijednosti dvaju tokova u novi tok. Poredak vrijednosti u novom toku ovisi će o trenutku emitiranja vrijednosti na pojedine tokove. U ovom primjeru iskoristili smo operator `delayElements` nad prvim tokom kako bi emisija vrijednosti započela kasnije naspram drugog s ciljem simuliranja ovog slučaja. Zbog toga ispis na konzoli nije slijedan 12345, već je ispis oblika 45123. Ovaj primjer usko je povezan s primjerom spajanja tokova korištenjem operatora `cocant`. Kod `concat` operatora vidjet ćemo da korištenjem istog scenarija odgode emitiranja i dalje rezultira slijednim ispisom vrijednosti što znači da se vrijednosti tokova sekvencijalno spajaju u novi.

```
Duration delay = Duration.ofMillis(10L);
Flux<Integer> numbers1 = Flux.range(1, 3);
Flux<Integer> numbers2 = Flux.range(4, 2);
Flux.merge(
        numbers1.delayElements(delay),
        numbers2
).subscribe(System.out::print);
Thread.sleep(1000L);
//Ispis na konzoli:
//45123
```

Kôd 25: Primjer `merge` operatora

Operator **`zip`**, slično kao i operator `merge`, koristi se za spajanje vrijednosti dvaju ili više tokova, međutim njihovo spajanje ne rezultira nizom isprepletenih vrijednosti na izlaznom toku već operator čeka da svi izdavači emitiraju po jedan element te ih nakon toga grupira odnosno spaja u novi tip podataka `TupleX (Tuple2, Tuple3,..)` gdje X predstavlja broj tokova, odnosno broj grupiranih vrijednosti. Tako grupirane vrijednosti se emitiraju na izlazni tok. U slučaju da jedan ili više tokova ne posjeduje više vrijednosti za spajanje, ostatak vrijednosti ostalih tokova se ignoriraju. Jedan takav slučaj može biti da primjerice želimo izvršiti spajanje vrijednosti triju tokova, ali u nekom trenutku spajanja samo dva od tri toka još sadrže ne spojene vrijednosti, tada spajanje završava i sve ostale ne spojene vrijednosti se ispuštaju ili ignoriraju.



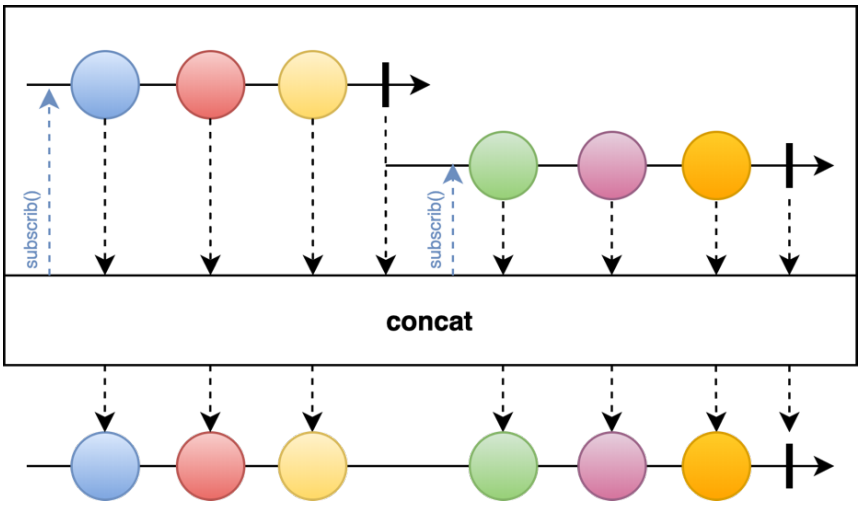
Slika 26: Vizualizacija `zip` operatora (Prema: Project Reactor - Documentation, 2020)

U sljedećem programskom isječku možemo vidjeti primjenu `zip` operatora. Kao što je i na dijagramu iznad prikazano, `zip` operator spaja vrijednosti na način da uzima po redu emitirane vrijednosti i spaja ih u novi tip podataka `Tuple` koji se može sastojati od n elemenata, ovisno koliko tokova se spaja operatorom. U ovom primjeru korištena su samo dva toka stoga u konzolnom ispisu imamo vrijednosti [1, 4] prvi `Tuple` objekt i [2, 5] drugi `Tuple` objekt.

```
Flux<Integer> numbers1 = Flux.range(1, 3);
Flux<Integer> numbers2 = Flux.range(4, 2);
Flux.zip(numbers1, numbers2)
    .subscribe(System.out::print);
//Ispis na konzoli:
//[1,4] [2,5]
```

Kôd 26: Primjer `zip` operatora

Operator **`concat`** provodi spajanje dvaju ili više tokova. Spajanje se postiže sekvencijalnim pretplatama na prethodni izvor započevši s prvim, nakon kojeg se čeka na njegovo dovršavanje prije pretplate na sljedeći i tako sve do posljednjeg izvora i njegovog dovršetka. Iz razloga što se čeka na dovršavanje trenutnog toka prije nego li se pretplati na idući, kažemo da je ovaj operator lijeni operator što znači da možemo usporiti, ubrzati ili ubaciti vremensku odgodu, poredak emitiranih vrijednosti uvijek ostaje isti. Pojava pogreške prekida niz te se ista prosljeđuje nizvodnim komponentama. Rezultat spajanja predstavlja tok sa sekvencijalno emitiranim vrijednostima izvora odnosno izdavača čiji će poredak ovisiti o poretku kojim su isti proslijeđeni operatoru kao parametri.



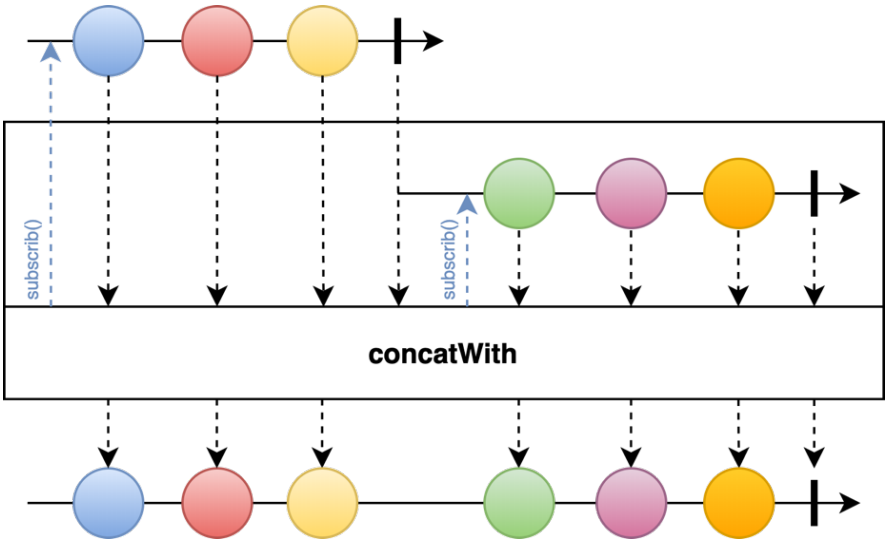
Slika 27: Vizualizacija `concat` operatora (Prema: Project Reactor - Documentation, 2020)

U sljedećem programskom isječku možemo vidjeti uporabu `concat` operatora. Kao što smo već rekli kod `merge` operatora, koristimo isti scenarij za prikaz spajanja. Primijetite da se ispis na konzoli razlikuje naspram `merge` operatora.

```
Duration delay = Duration.ofMillis(10L);
Flux<Integer> numbers1 = Flux.range(1, 3);
Flux<Integer> numbers2 = Flux.range(4, 2);
Flux.concat(
        numbers1.delayElements(delay),
        numbers2
).subscribe(System.out::print);
Thread.sleep(1000L);
//Ispis na konzoli:
//12345
```

Kôd 27: Primjer `concat` operatora

Prethodno prikazani operator predstavlja statički operator za slijedno spajanje tokova zajedno s njihovim emitiranim vrijednostima. Međutim isto se može postići i objektnim operatorom **`concatWith`**. Ova dva operatora su zapravo ekvivalentni, jedina razlika među njima je u načinu njihova pozivanja. Operator `concat` predstavlja klasni operator odnosno sadržan je unutar `Flux` klase kao javna statička metoda `Flux.concat()` dok `concatWith` s druge strane predstavlja objektni operator koji se može jedino pozvati nad objektom klase `Flux`. Za koji god pristup se odlučimo, krajnji rezultat će ostati isti. Na idućem mramornom dijagramu možemo vidjeti tu ekvivalenciju kod pružanja rezultata spajanja iako se sada prvi tok nalazi izvan pravokutnika.



Slika 28: Vizualizacija `concatWith` operatora (Prema: Project Reactor - Documentation, 2020)

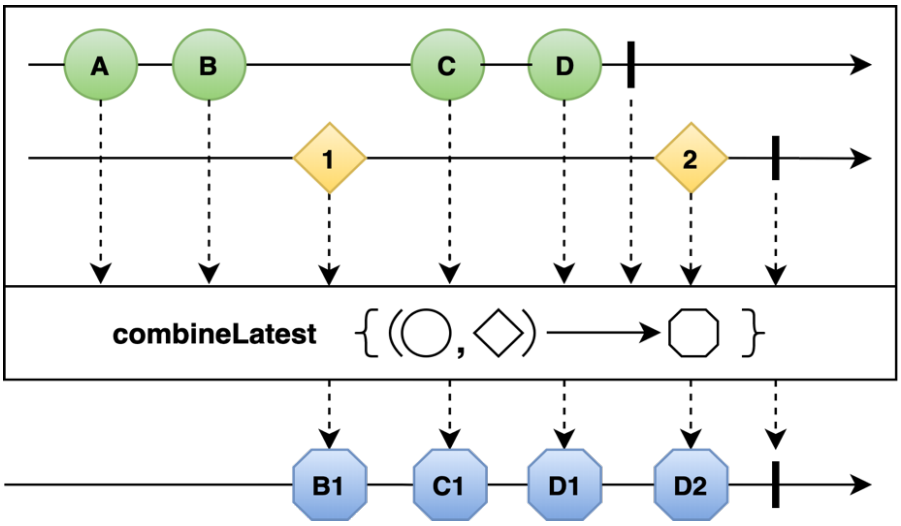
U idućem programskom isječku možemo vidjeti uporabu `concatWith` operatora. Prethodno obrađeni operatori predstavljaju klasne operatore, što znači da se mogu koristiti samo putem statičkih metodi. Za razliku od njih `concatWith` je objektni operator koji je moguće pozvati nad objektom toka podataka te mu se kao takvom prosljeđuje drugi objekt toka s kojim želimo napraviti spajanje. U ovom primjeru spajamo tok podataka `numbers2` na već postojeći tok podataka `numbers1` te na konzoli dobivamo očekivani ispis u obliku 12345.

```
Flux<Integer> numbers1 = Flux.range(1, 3);
Flux<Integer> numbers2 = Flux.range(4, 2);
numbers1.concatWith(numbers2)
        .subscribe(System.out::print);
//Ispis na konzoli:
//12345
```

Kôd 28: Primjer `concatWith` operatora

Zadnji operator za spajanje tokova koji će se obraditi je **`combineLatest`**. Ovaj operator predstavlja statički ili klasni operator te kao i ostali služi za spajanje dvaju ili više tokova. Što se tiče načina spajanja tokova, najsličniji je operatoru `zip`. Naime, ovaj operator vrši spajanje na način da uzima zadnje, odnosno najnovije emitirane vrijednosti tokova koje potom spaja u proizvoljni objekt na temelju proslijeđene funkcije. Zbog toga što spajaju samo najnovije emitirane vrijednosti, vrlo je velika vjerojatnost da će neke vrijednosti biti ispuštene odnosno ignorirane zato što je u periodu dok su ostali tokovi emitirali svoje najnovije vrijednosti prvotna vrijednost bila zamijenjena novijom. Ista takva situacija prikazana je na idućem mramornom dijagramu gdje je element A prvog toka bio zamijenjen novijim elementom B koji

je zatim bio spojen s najnovijom vrijednosti drugog toka čineći tako proizvoljni objekt B1. Ovaj operator koristan je kada želimo vršiti izračun ili spajanje podataka u realnom vremenu.



Slika 29: Vizualizacija `combineLatest` operatora (Prema: Project Reactor - Documentation, 2020)

U idućem programskom isječku možemo vidjeti uporabu `combineLatest` operatora. U primjeru je prikazano spajanje toka koji sadrži slova i toka koji sadrži brojčane vrijednosti, s time da tok koji sadrži slova počinje s emitiranjem 750 milisekundi kasnije nego tok koji sadrži brojčane vrijednosti. Drugi tok emitira novu brojčanu vrijednosti svakih 500 milisekundi. U ispisu na konzoli možemo vidjeti da su se neke brojčane vrijednosti ponovile te da su spojene s različitim brojčanim vrijednostima što ukazuje na spajanje zadnjih emitiranih vrijednosti tokova.

```
Duration noDelay = Duration.ofMillis(0);
Duration period = Duration.ofMillis(500);
Duration delay = Duration.ofMillis(750);
Flux<String> chars =
        Flux.just("a", "b", "c", "d", "e")
            .delayElements(delay);
Flux<Long> numbers =
        Flux.interval(noDelay, period);
Flux.combineLatest(
        chars,
        numbers,
        (a, b) -> a + b
).subscribe(s ->
        System.out.print(s + ", "));
Thread.sleep(3000L);
//Ispis na konzoli:
// a1, a2, a3, b3, b4, c4, c5, c6, d6
```

Kôd 29: Primjer `combineLatest` operatora

Ostali mogući operatori za spajanje tokova: **`join`**, **`startWith`**, **`mergeSequential`**, **`mergeDelayError`**, **`mergeOrdered`**, **`mergeSequentialDelayError`**, **`using`**, **`usingWhen`** i **`switchOnNext`** (*Project Reactor - Documentation*, 2020).

5.3.4.5. Rukovanje greškama

Prije nego li krenemo u objašnjavanje kako Reactor rukuje greškama i koji operatori postoje za njihovo rukovanje, važno je ponovo natuknuti da po reaktivnoj specifikaciji pogreške predstavljaju terminalni događaj. To znači da pojava pogreške u toku isti prekida onog trenutka kada se dogodi te se propagira nizvodno niz lanac do zadnjeg operatora, odnosno do `onError` metode objekta pretplatnika. Kao i u ostalim paradigmama, pogreške se i dalje obrađuju na aplikacijskoj razini što znači da svaka klasa pretplatnika treba implementirati `onError` metodu kako bi je bila u stanju obraditi. Osim samog rukovanja greškama pri samom kraju podatkovnog toka, Reactor okvir nudi i alternativna sredstva za njihovim rukovanjem u sredini lanca operatora. Međutim kako pogreška predstavlja terminalni događaj, čak i ako se koriste alternativna sredstva u obliku operatora, originalni niz će i dalje biti prekinut te će se dogoditi događaj emitiranja signala pogreške kao dio novog alternativnog podatkovnog toka. Rukovanje greškama u reaktivnom svijetu ne predstavlja koncept koji je osmišljen od nule već se može napraviti eksplicitna poveznica između imperativnog i reaktivnog načina. Tako

primjerice u imperativnom načinu za rukovanje greškama možemo koristiti neku od sljedećih akcija u sklopu *pokušaj-uhvati* (eng. *try-catch*) bloka:

- Uhvati i vrati statičku vrijednost.
- Uhvati i izvrši alternativni put uz korištenje alternativne metode.
- Uhvati i dinamički izračunaj alternativnu vrijednost.
- Uhvati, zabilježi poruku o pogrešci i ponovo ju baci.
- Koristi konačni blok za čišćenje resursa.

Za sve ove navedene imperativne akcije postoje ekvivalentne akcije u Reactor biblioteci u obliku posebnih operatora i metoda za rukovanje greškama. Pa tako primjerice ekvivalent za sam koncept *try-catch* bloka u reaktivnom svijetu predstavlja već spomenuta `onError` metoda sučelja Subscriber koja se nalazi na kraju samog lanca implementirana od strane pretplatnika. U idućem programskom isječku možemo vidjeti primjer njenog korištenja.

```
Flux<Integer> numberFlux = Flux.range(1, 10);
numberFlux
        // Mogućnost pojave iznimke
        .map(n -> transformationThatCanThrowException(n))
        // 2. map operator se izvršava samo ako nije bilo pojave iznimke
        .map(n -> safeTransformation(n))
        .subscribe(
                n -> System.out.println("Value: " + n),
                error -> System.err.println("Caught Exception: " + error)
        );
```

Kôd 30: `onError` metoda

Kao što možemo vidjeti nad kreiranim tokom vrijednosti, koristimo dva `map` operatora kako bismo izvršili transformaciju nad vrijednostima. Prvi operator koristi metodu čije izvršavanje može rezultirati generiranjem pogreške odnosno iznimke. Drugi `map` operator koristi sigurnu metodu čije izvršavanje neće nikad rezultirati iznimkom. U slučaju da se prva transformacija izvrši uspješno, prelazi se na izvršavanje druge nakon čijeg se završetka transformirana vrijednost emitira objektu pretplatnika koji će ju ispisati na konzolu. U slučaju da izvršavanje prve transformacija rezultira pojavom iznimke, tok se prekida, što rezultira ne izvršavanjem drugog operatora i druge transformacije te emitiranjem signala pogreške objektu pretplatnika umjesto vrijednosti. Pretplatnik signal pogreške obrađuje u svojoj `onError` metodi čija je implementacija u ovom primjeru prikazana putem lambde koja ispisuje sadržaj greške na konzolu u obliku pogreške.

U slučaju da pojavu greške ne želimo propagirati sve do objekta pretplatnika, moguće je korištenje specifičnih operatora u sredini lanca koji će rezultirati alternativnim načinom izvršavanja toka u slučaju pojave pogreške. Jedan od tih operatora predstavlja operator **onErrorReturn** koji u slučaju pojave iznimke vraća neku statičku vrijednost kao alternativnu vrijednost, a čije emitiranje sprječava neuspješno završavanje toka. Drugim riječima, njegovim korištenjem tok se neće nasilno prekinuti i neće se dogoditi emitiranje signala pogreške pretplatniku. U slučaju da želimo emitirati nešto više osim obične statičke vrijednosti te ako imamo sigurniji alternativni način za procesuiranja podataka, možemo koristiti **onErrorResume** operator koji kao parametar prima funkciju koja će se izvršiti u slučaju pojave iznimke. Oba ova operatora imaju varijante koje omogućavaju dodatno filtriranje iznimka kako bi se moglo definirati u kojim slučajevima i na koji način treba reagirati kako bi se postigao zadovoljavajući oporavak aplikacije. U idućem programskom isječku prikazana je primjena ovih dvaju operatora. Kod prvog operatora vrlo je jednostavno zaključiti što se događa, kod pojave iznimke, pretplatniku se emitira predefinirana statička vrijednost. Drugi slučaj je malo zanimljiviji jer sadrži primjer koji je dosta čest u primjeni kod stvarnih sustava. U ovom primjeru cilj nam je dohvatiti proizvod na temelju njegovog identifikatora. Kako bismo izbjegli skupi poziv udaljenog servisa za dohvaćanje proizvoda, pretpostavljamo da isti proizvod možemo pronaći spremljenog u našoj `cache` memoriji. U slučaju da se proizvod ne nalazi spremljen u memoriji, metoda `getFromCache` generirat će iznimku koju će `onErrorResume` operator uhvatiti i u tom slučaju izvršiti poziv udaljenog servisa s ciljem dohvaćanje svježeg proizvoda. Takav dohvaćen proizvod će se potom emitirani pretplatniku.

```
// onErrorReturn
Flux.range(1, 10)
    .map(n -> transformationThatCanThrowException(n))
    .onErrorReturn(STATIC_VALUE)
    .subscribe(consumer);
// onErrorResume
Flux.just(productID)
    .flatMap(id -> getFromCache(id))
    .onErrorResume(e -> getProfuctFromSAP(e))
    .subscribe(consumer);
```

Kôd 31: `onErrorReturn` i `onErrorResume` operatori

Još jedan od mogućih načina kako se možemo oporaviti u slučaju pojave greške je uz pomoć ponavljanja (eng. *Retry*). Za ovaj način oporavka postoji poseban skup objektnih operatora u koje ubrajamo: **retry** i **retryWhen**. `Retry` operator jest jednostavniji operator te kao što i samo ime kaže omogućuje ponavljanje niza koji je uzrokovao pogrešku. To radi na način da se ponovo pretplati na uzvodni `Flux`, odnosno izdavača. Na taj način zapravo se

dobiva drugačiji niz odnosno novi tok, dok se prethodno originalni terminira. `Retry` operator kao parametar prima brojčanu vrijednost koja predstavlja broj ponavljanja pretplate prije nego li se dopusti propagiranje pogreške nizvodno kroz lanac. Operator `retryWhen` predstavlja naprednije verziju u smislu da sadrži prateći `Flux` uz pomoć kojeg određuje da li se neki određeni neuspjeh treba ponoviti ili ne. Prateći `Flux` kreira se od strane operatora ali programer ima mogućnost njegovog ukrašavanja proizvoljnom funkcijom te time predstavlja robusniju verziju ovog operatora.

5.3.4.6. Upravljanje dretvama

Reactor kao i ostali reaktivni okviri i biblioteke sadrži ugrađenu podršku za korištenje višedretvenosti, međutim ono ne nameće određeni model već odabir ostavlja programerima na izbor. Stoga, kreiranje `Flux` ili `Mono` tokova ne znači da će se isti izvršavati na posebno dediciranim dretvama. Većina operatora nastavlja svoj rad na dretvi na kojoj se izvršio prethodni operator pripadajućeg lanca. Štoviše, ako nije eksplicitno drugačije definirano, prvi operator u lancu (izvor) će se izvršiti na dretvi na kojoj se dogodio poziv metode `suscribe()`, što u većini slučajeva znači da će se izvršavanje odvijati na glavnoj dretvi. Ako želimo eksplicitno definirati dretvu na kojoj će se određeni tok izvršavati to možemo učiniti uz pomoć korištenja dodatne apstrakcije naziva **`Schedulers`**. `Schedulers` klasu možemo usporediti s imperativnom apstrakcijom `ExecutorService` iz razloga što posjeduju slične odgovornost. `Schedulers` klasa sadrži brojne statičke metode koje omogućuju pristup sljedećim kontekstima izvršavanja:

- **`paralle()`** – Optimiziran za brza i ne-blokirajuća izvršavanja.
- **`single()`** – Optimiziran za jednokratno izvršavanje s niskim vremenom kašnjenja.
- **`elastic()`** – Optimizirano za duga izvršavanja, predstavlja alternativu za blokirajuće zadatke gdje broj zadataka i dretvi može neograničeno rasti.
- **`boundedElastic()`** - Optimiziran za duga izvršavanja blokirajućih zadataka čiji je broj ograničen.
- **`immediate()`** – Ne zakazuje izvršavanje proslijeđenog Runnable objekta već njegovo izvršavanje izvodi momentalno.
- **`fromExecutorService(`**`ExecutorService`**`)`** – Kreira novu `Scheduler` instancu oko proslijeđenog `ExecutorService` objekta.

Za prebacivanje konteksta izvršavanja reaktivnog lanca u Reactoru koriste se dva operatora: `publishOn` i `subscribeOn`. Oba operatora za parametar primaju `Scheduler` objekt na kojeg se prebacuje kontekst izvršavanja. Operator **`publishOn`** primjenjujemo kao i svaki drugi objektni operator, što znači da ga je moguće primijeniti u sredinu pretplatničkog

lanca te je njegov zadatak da promjeni kontekst izvršavanja odnosno dretvu na kojoj će se izvršavati preostali operatori koji slijede nakon njega u lancu. Kontekst izvršavanja ostaje isti za sve iduće operatore lanca osim ako se ponovnim korištenjem operatora `publishOn` eksplicitno ne definira drugačije. Zbog toga što ovaj operator utječe na izvršavanje konteksta izvršavanja operatora koji su u lancu poslije njega, njegov smještaj u samom lancu igra važnu ulogu. S druge strane, operator **`subscribeOn`** koristi se za prebacivanje konteksta izvršavanja samog procesa pretplate te time utječe na mijenjanje konteksta izvršavanja cijelog lanca odnosno toka. Smještaj ovog operatora u lancu nije važan jer ima globalan utjecan na tok ali se može nadjačati operatorima kao što je `publishOn` koji može i dalje prebaciti kontekst izvršavanja dijelova lanca koji slijede nakon njega. U idućem programskom primjeru vidimo upotrebu oba operatora nad jednim lancem.

```
Consumer beforePublishOn = v -> System.out.println("Before publishOn: (Value"
        + "= " + v + ") (Thread: " + Thread.currentThread().getName() + ")");
Consumer afterPublishOn = v -> System.out.println("After publishOn: (Value"
        + "= " + v + ") (Thread: " + Thread.currentThread().getName() + ")");
Consumer recieved = v -> System.out.println("Received\t\t: (Value"
        + "= " + v + ") (Thread: " + Thread.currentThread().getName() + ")");

Flux.just("A", "B")
    .doOnNext(beforePublishOn)
    .publishOn(Schedulers.elastic())
    .doOnNext(afterPublishOn)
    .subscribeOn(Schedulers.parallel())
    .subscribe(recieved);
// Before publishOn: (Value = A) (Thread: parallel-1)
// Before publishOn: (Value = B) (Thread: parallel-1)
// After publishOn : (Value = A) (Thread: elastic-2)
// Received        : (Value = A) (Thread: elastic-2)
// After publishOn : (Value = B) (Thread: elastic-2)
// Received        : (Value = B) (Thread: elastic-2)
```

Kôd 32: Primjer `publishOn` i `subscribeOn` operatora za prebacivanje konteksta izvršavanja

Kao što smo rekli, pozicija `subscribeOn` operatora nije važna te se ona stoga nalazi pred kraj samog lanca. Pozicija operatora `publishOn` igra ulogu te je smještena nakon poziva prvog operatora `doOnNext`. Operator `doOnNext` koristi se kako bi mogli prikazati koji dio lanca se izvodio unutar kojeg konteksta (dretve). Ovaj operator prima `Consumer` objekt koji na konzolu ispisuje primljenu emitiranu vrijednost i naziv same dretve na kojoj se izvršio. Tako na

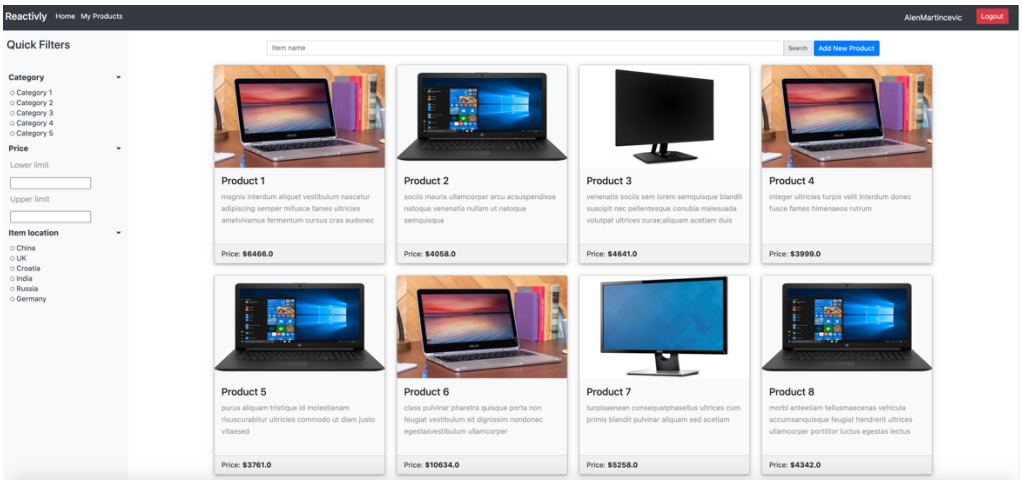
ispisu u konzoli možemo vidjeti da se prije prvog poziva `publishOn` operatori izvršavaju na dretvi naziva “parralel-1” koju smo definirali u pozivu operatora `suscribeOn`. Time vidimo da ovaj operator utječe na sam izvor emisije ali isto tako i na kontekst izvršavanja cijelog lanca. Međutim, čim se dogodi prvi poziv `publishOn` operatora, kontekst izvršavanja se mijenja na dretvu naziva “elastic-2” i ovaj kontekst ostaje isti do kraja izvršavanja lanca. To možemo vidjeti iz toga što se drugi poziv operatora `doOnNext` ali i poziv `subscribe` operatora izvršio unutar istog novog dodijeljenog konteksta.

6. Izrada reaktivne web aplikacije

6.1. Ideja aplikacije

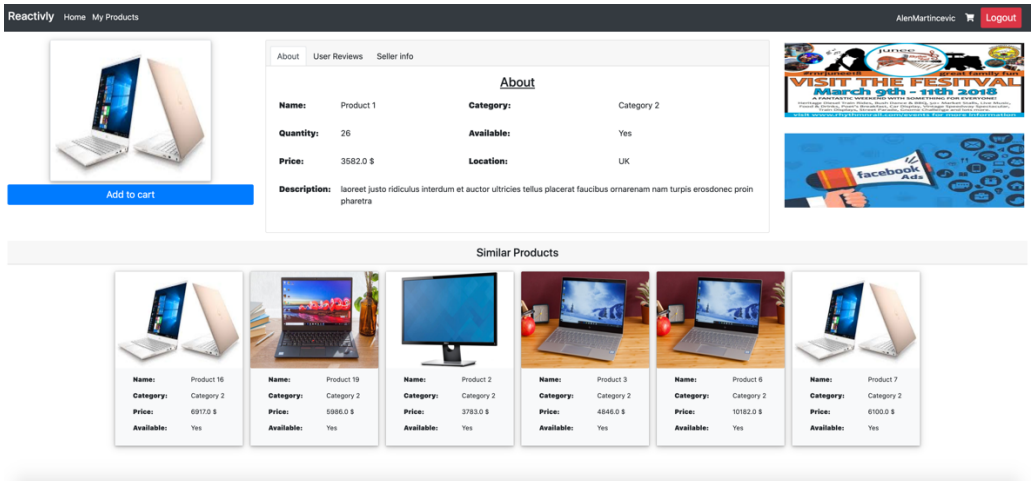
Kako bi se prikazalo reaktivno programiranje u punom opsegu i sjaju odlučeno je da ideja aplikacije bude realizacija jednostavne web aplikacije za e-trgovinu (eng. *E-commerce*). Aplikacije za e-trgovinu predstavljaju jedne od najkorištenijih aplikacija na internetu što se najviše može vidjeti u blagdanskim periodima kao što su Božić, Uskrs i slično. Iz razloga što ove aplikacije mogu doživjeti vrtoglavi skok korisnika u vrlo kratkom vremenu i zato što ovakve aplikacije direktno generiraju prihode iznimno je važno da svi podsustavi aplikacije budu neovisni jedan od drugoga kako se ne bi desilo da pad jednog uzrokuje rušenje cijele aplikacije jer bi u suprotnom to značilo izravne novčane gubitke. Zbog prethodno navedenih razloga osim same primjene reaktivnog programiranja u izradi aplikacije, ista će biti koncipirana na način da zadovolji određeni podskup zahtjeva reaktivnog sustava kako bi se minimizirala opasnost rušenja cijele aplikacije. Važno je napomenuti da izrađeni sustav ne predstavlja reaktivni sustav u punom smislu te da isti zahtjeva dodatne nadogradnje kako bi se uspješno zadovoljili svi zahtjevi koje reaktivni sustav mora ispunjavati. Ova aplikacija ne predstavlja standardnu implementaciju e-trgovine iz razloga što nedostaju određene ključne funkcionalnosti kao što je proces provedbe plaćanja ali isto tako i zato što sadrži neke dodatne neuobičajene funkcionalnosti za ovakvu vrstu aplikacije. U nastavku poglavlja navest ćemo i objasniti glavne funksionalnosti aplikacije. Osim samog objašnjenja prikazat će se i ekrani sučelja izrađene aplikacije u sklopu ovog rada. Detaljno objašnjenje pojedinih funkcionalnosti s aspekta programskog koda obradit će se kasnije u poglavlju. Glavne funkcionalnosti aplikacije su sljedeće:

- **Prijava/Registracija korisnika** – Standardna funkcionalnost prijave i registracije korisnika u svrhu mogućnosti korištenja sustava i kupovine artikala. Svaki novo registrirani korisnik ima mogućnost korištenja sustava kao prodavač i kupac što znači da može kupovati i prodavati proizvode.
- **Prikaz dostupnih artikala -** Prikaz dostupnih artikala web trgovine u obliku mreže kartica koje sadrže naslov, kratki opis, sliku i cijenu artikla. Klikom na pojedinu karticu artikla otvara se nova stranica s detaljnim prikazom informacija o artiklu**.** Pretraživanje artikala moguće je vršiti putem unošenja ključnih riječi u traku pretraživanja ili označavanjem ponuđenih filter opcija koje se sastoje od vrste kategorije, raspona cijene i lokacije artikla.



Slika 30: Prikaz dostupnih artikala

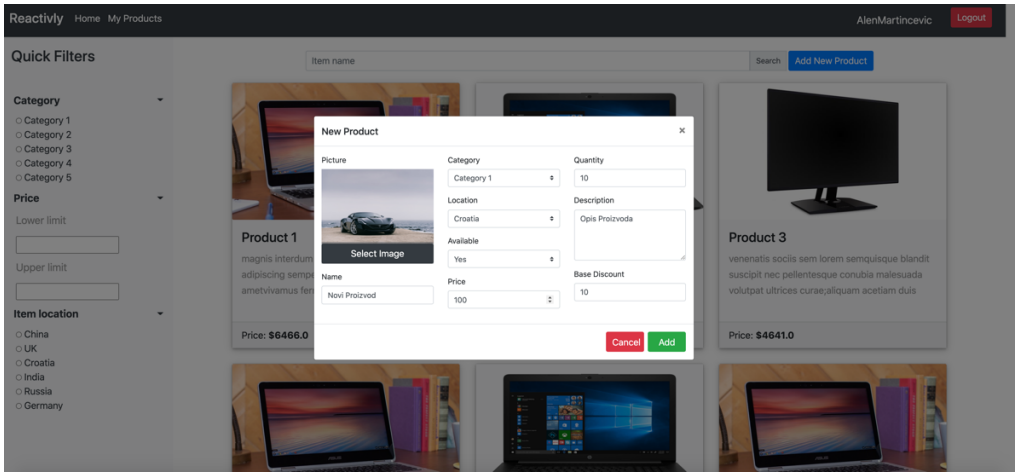
- **Detaljni prikaza informacija o artiklu -** Prikaz detaljnog opisa artikla s osnovnim podacima kao što su ime, dostupna količina, cijena, opis, kategorija, dostupnost, lokacija, informacije o prodavaču i slično. Ova stranica također prikazuje i korisničke osvrte te nudi mogućnost ostavljanja istog. Na dnu stranice nalazi se i sekcija s popisom sličnih proizvoda iz iste kategorije. Klikom na proizvod korisnika se prebacuje na stranicu detaljnog prikaza određenog artikla. Putem ove stranice otvoreni artikl moguće je dodati u košaricu na gumb koji se nalazi ispod slike artikla.



Slika 31: Detaljni prikaz informacija o proizvodu

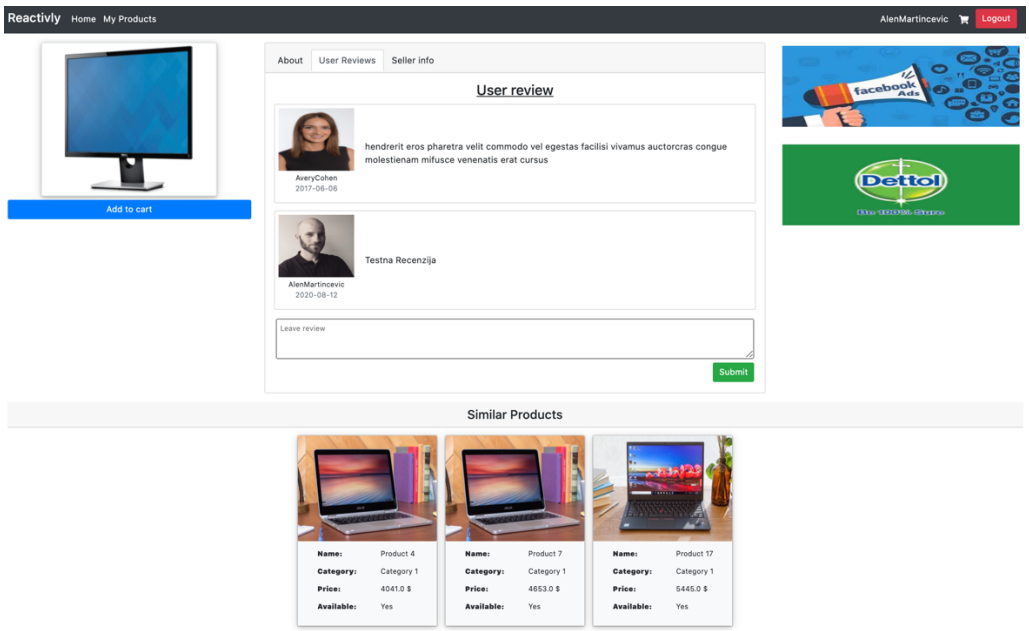
- **Prikaz vlastitih artikla –** Zasebna stranica koja prikazuje artikle kreirane od strane prijavljenog korisnika. Služi za jednostavniji pregled objavljenih artikala i za vođenje njihove evidencije te njihovog lakšeg upravljanja.

- **Dodaj/Ažuriraj artikl –** Funkcionalnost dostupna u sklopu uloge prodavača. U sustav je moguće dodati novi artikl ili ažurirati postojeći. Artiklu se moraju definirati sljedeći atributi: naziv, kategorija, dostupnost, cijena, količina, opis, popust i slika. Nakon što je artikl kreiran, isti je moguće i ažurirati. Ažuriranje je moguće samo za artikle koji su kreirani od strane korisnika.



Slika 32: Funkcionalnost dodavanja novog proizvoda

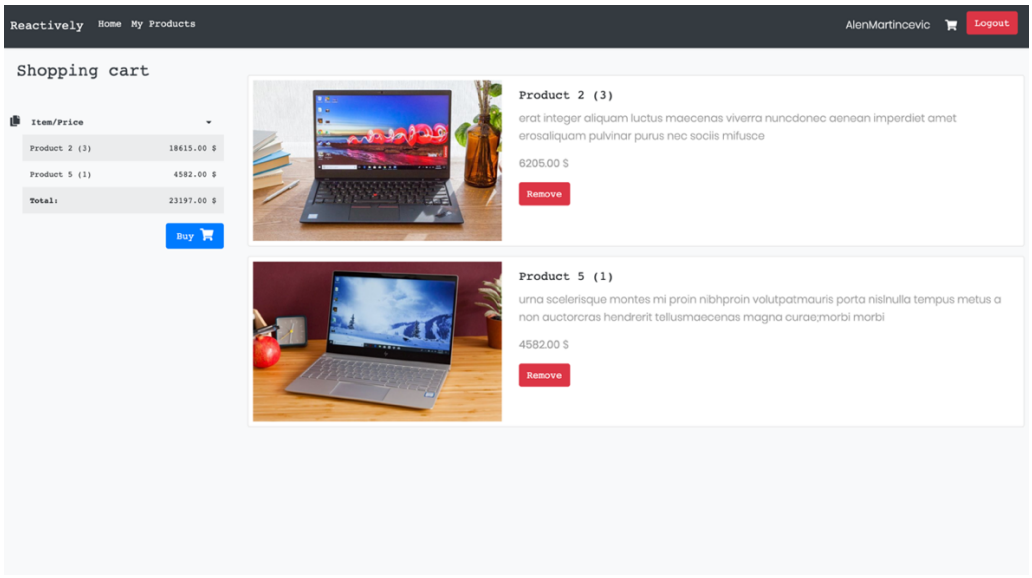
- **Recenziranje artikla –** Svaki korisnik može ostaviti recenziju za artikl. Recenziju je moguće dodati pristupanjem stranici detaljnog prikaza informacija o artiklu pod sekcijom „Korisničke recenzije“ (eng. *User Reviews*). Nakon što je recenzija podnesena, ista je dostupna svim korisnicima.



Slika 33: Funkcionalnost recenziranja artikla

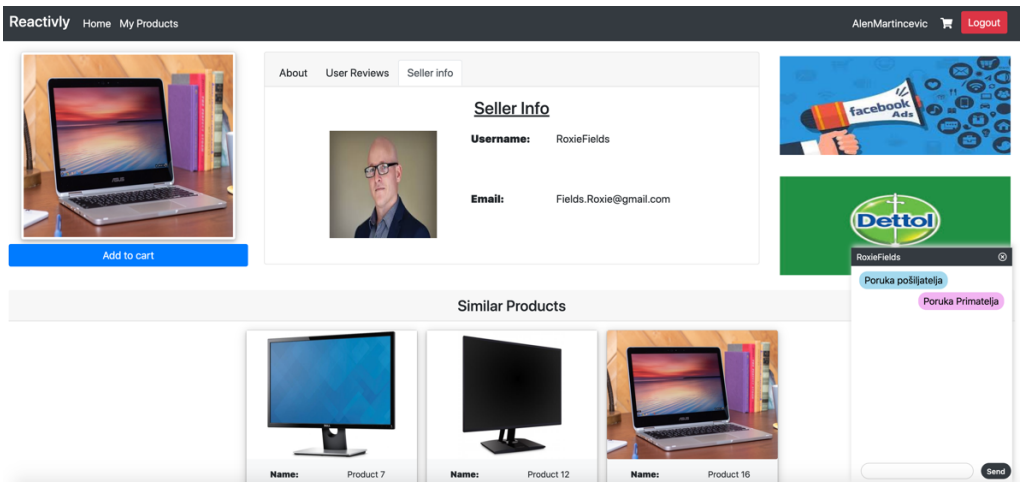
- **Košarica –** Nakon dodavanja artikala u košaricu iste je moguće vidjeti klikom na gumb košarice u navigacijskoj traci. Stranica košarice sadrži listu kartica

dodanih artikla sa sljedećim informacijama: naziv, opis, cijena, slike i dodane količine pojedinog artikla. Artikle je moguće ukloniti iz košarice pritiskom na gumb za uklanjanje artikla. Osim liste artikla, na stranici je prikazana i lista izračuna cijene te gumb za obavljanje simulirane kupovine.



Slika 34: Prikaz košarice

- **Razgovor –** Ova funkcionalnost nije standardna funkcionalnost aplikacija za e-trgovinu, a predstavlja mogućnost kontaktiranja prodavača određenog artikla. Ako kupac odluči zatražiti više informacija o pojedinom artiklu to je moguće napraviti putem stranice za detaljni prikaz artikla čijim se otvaranjem otvara skočni prozor u kojem kupac može napisati i poslati poruku prodavaču. Poslanu poruku prodavač će primiti jednom kada se ulogira u sustav te potom na istu može odgovoriti.

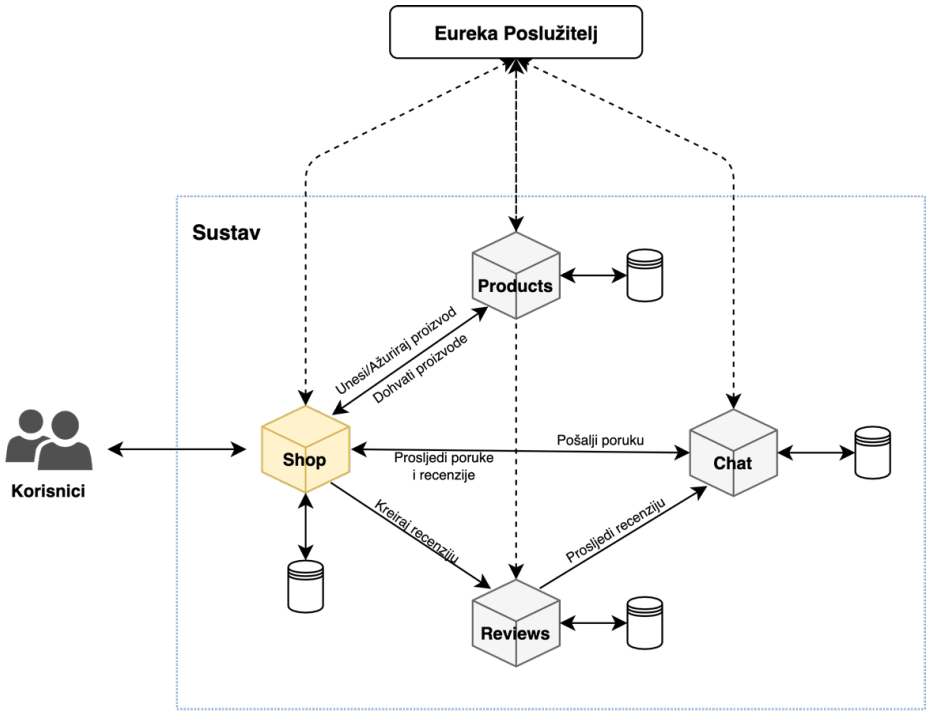


Slika 35: Funkcionalnost razgovora prodavača i kupca

6.2. Tehnički aspekti aplikacije

6.2.1. Arhitektura sustava

Arhitektura aplikacije ili bolje rečeno sustava za e-trgovinu temelji se na mikroservisnoj arhitekturi. Mikroservisna arhitektura predstavlja sustav koji je razbijen na manje zasebne, autonomne cjeline (komponente) koje međusobno surađuju. Svaka zasebna cjelina trebala bi se fokusirati na rješavanje specifičnog problema što u konačnici znači da cjelina smije imati samo jednu odgovornost i jedan razlog da se promjeni ili zamjeni. Svrha primjene mikroservisne arhitekture u izradi aplikacije bilo je zadovoljavanje kriterija vremenske i prostorne odvojenosti komponenti kao jednog od kriterija reaktivnih sustava ali isto tako i činjenje sustava otpornijim na ispade. U arhitekturi sustava možemo izdvojiti jednu glavnu aplikaciju naziva Shop koja predstavlja korisničku aplikaciju koju korisnici vide i koriste te tri mikroservisa: Products, Reviews i Chat.



Slika 36: Arhitektura sustava s posebno označenom glavnom aplikacijom (žuto) i eureka poslužiteljem

Mikroservis **Products** predstavlja komponentu čija se odgovornost sastoji od manipulacije artiklima. To znači njihovim spremanje, brisanjem, ažuriranjem, dohvaćanjem iz baze podataka. Svi zahtjevi za rad s artiklima dolaze od strane korisničke aplikacije Shop.

Mikroservis **Reviews** predstavlja „prolaznu“ komponentu koja je zadužena za primanje novo objavljenih korisničkih recenzija određenog artikla, njihovog spremanja u bazu podataka te njihovog prosljeđivanja trećem mikroservisu Chat nakon uspješnog spremanja.

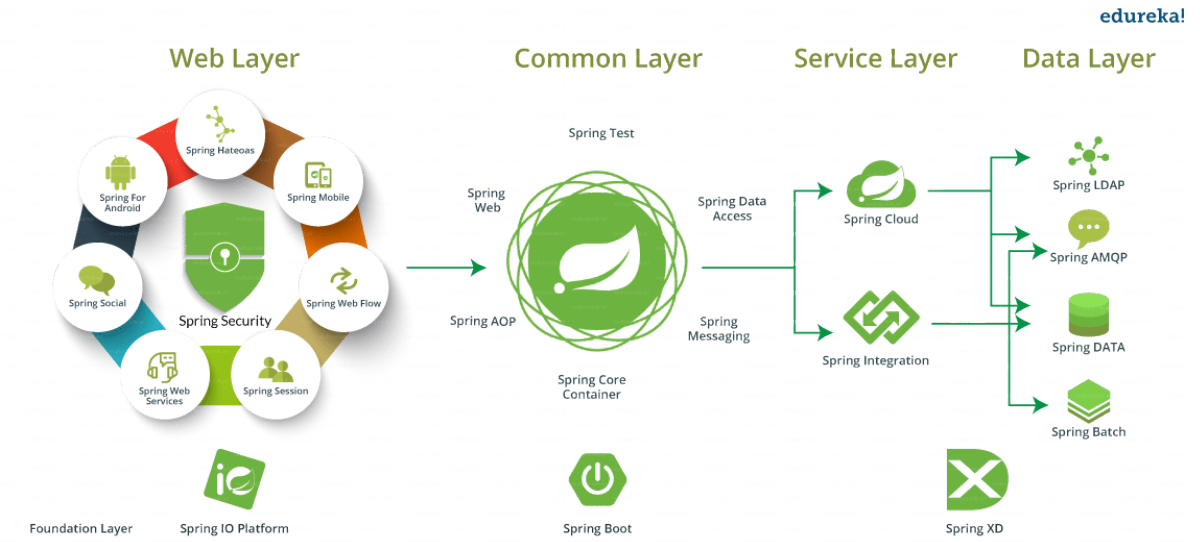
Mikroservis **Chat** predstavlja komponentu čija domena obuhvaća sveukupnu komunikaciju za pojedinačni artikl. U tu domenu komunikacije spadaju prethodno spomenute recenzije artikala ali isto tako i komunikacija između kupca i prodavača kao dio funkcionalnosti razgovora. Zaduženja ovog mikroservisa ogledaju se u primanju recenzija, primanju poruka kao dio procesa komunikacije između dva korisnika, njihovog spremanja, dohvaćanja te prosljeđivanja oba načina komunikacije nazad korisničkoj aplikaciji.

Korisnička aplikacija **Shop** predstavlja glavnu aplikaciju koju korisnici vide i koriste. Ova aplikacija sadrži sve funkcionalnosti koje su blisko vezane uz samog korisnika kao što su prijava, registracija, ažuriranje profila, košarica, prikaz proizvoda, ostavljanje recenzije i slično. Kako ova aplikacija predstavlja grafičko sučelje, ona sadrži sve potrebne predloške html i css datoteke, dodatne resurse u obliku slika i slično.

Eureka poslužitelj još poznat i kao poslužitelj za otkrivanje (eng. *Discovery Server*) predstavlja aplikaciju koja sadrži podatke o svim ostalim klijentskim aplikacijama i mikroservisima kao što su IP adresa i port na kojem se vrte te dostupnost samog servisa. Svaki servis bila to aplikacija ili mikroservis mora se registrirati na Eureka poslužitelj kako bi ih ostali korisnički servisi mogli dinamički otkriti i dohvatiti. Eureka poslužitelj se koristi primarno za dinamičko otkrivanje servisa ali i za balansiranje opterećenja (eng. *Load Balancing*) što također predstavlja jednu od karakteristika i zahtjeva reaktivnog sustava.

6.2.2. Korišteni projekti, moduli, okviri i biblioteke

Za razvoj reaktivne web aplikacije koristit će se Spring ekosustav sa svim potrebnim reaktivnim okvirima koje ovaj ekosustav nudi na raspolaganje. Spring ekosustav možemo podijeliti na sljedeće grupacije: projekti, potprojekti, moduli i slojevi. Moduli u Spring ekosustavu predstavljaju najmanju zasebnu i neovisnu jedinicu rada. S druge strane potprojekti i projekti predstavljaju kompletne cjeline koje u sebi mogu sadržavati jedan ili više modula. Moguće je identificirati pet slojeva u Spring ekosustavu i to su sljedeći: Web Sloj (eng. *Web Layer*), Zajednički Sloj (eng. *Common Layer*), Servisni Sloj (eng. *Service Layer*), Podatkovni Sloj (eng. *Data Layer*) i Temeljni Sloj (eng. *Foundation Layer*). Svi projekti, potprojekti ali i moduli mogu se svrstati u jedan od ovih pet slojeva što možemo vidjeti na sljedećem dijagramu ekosustava (dijagram prikazuje samo jedinice visoke razine).



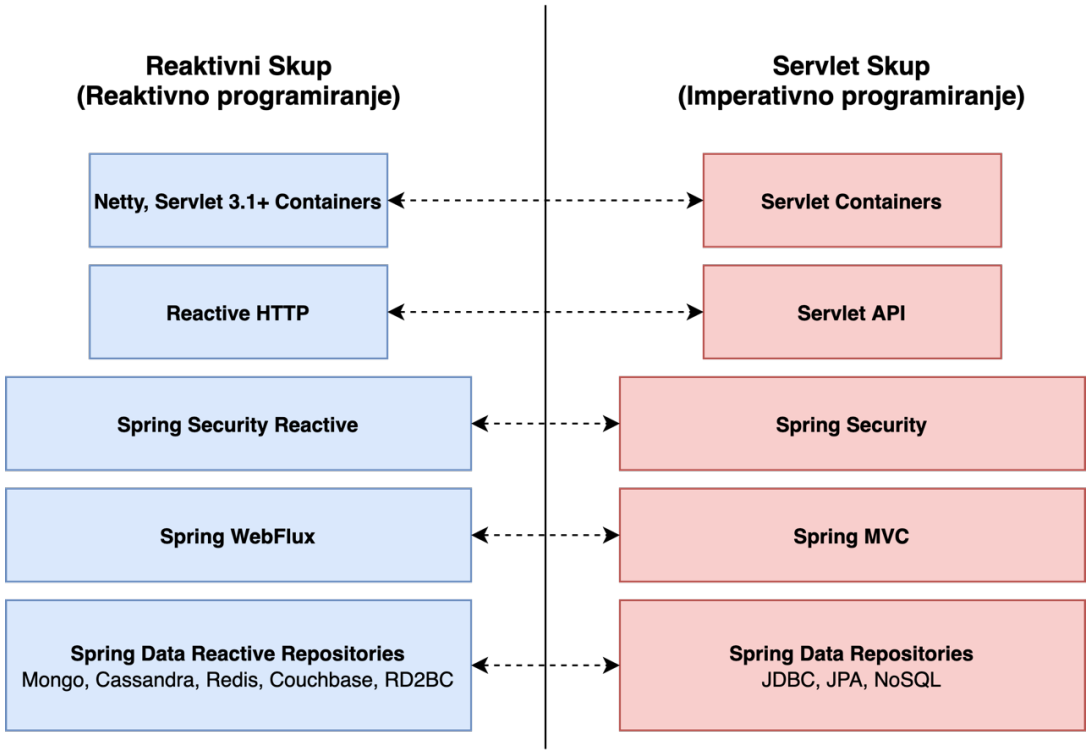
Slika 37: Spring ekosustav (Izvor: Chand, 2017)

Pa tako primjerice u **web sloj** možemo navesti sljedeće projekte: Spring HATEOAS, Spring Mobile, Spring Web Flow, Spring Session, Spring WebServices, Spring Social, Spring for Android, Spring Security. Razlog zašto je Spring Security projekt prikazan u sredini ovih projekta nije zbog toga što svi projekti koji su navedeni oko njega ovise o njemu već je prikazan s ciljem da se pokaže da je Spring Security projekt moguće koristiti u kombinaciji s bilo koji drugim projektom. Idući sloj predstavlja **zajednički sloj** i on se sastoji od modula, kojih ima oko dvadesetak, a koji su podijeljeni u jednu od sljedećih 6 kategorija: Spring Web, Spring Test, Spring Dana Access/Integration, Spring AOP and Instrumentation, Spring Messaging, Spring Core Container. Zajednička stavka svih ovih grupa i modula zajedničkog sloja je ta što svi Spring projekti koriste barem jedan ili više modula iz ovog sloja. Idući sloj je **servisni sloj** i on sadrži samo dva projekta Spring Cloud i Spring Integration. Predzadnji sloj predstavlja **sloj podataka** i on je zadužen za rukovanje i manipuliranje podacima te nudi mogućnosti korištenja univerzalnih API sučelja za rad s različitim vrstama baza podataka. U ovaj sloj ubrajamo sljedeće projekte: Spring LDAP, Spring AMQP, Spring Data i Spring Batch. Zadnji sloj predstavlja **temeljni sloj** i ovdje možemo pronaći projekte od kojih neki ujedno predstavljaju i kompletne platforme. Ovi projekti ne pripadaju ni jednom drugom sloju već radije služe kao temelj za sve ostale projekte. U ovaj sloj ubrajamo sljedeće projekte/platforme: Spring IO Platform poznatu pod imenom Spring Initializr, Spring Boot i Spring XD.

Kao što možemo vidjeti Spring ekosustav predstavlja podosta robusno i opširno rješenje. Za razvoj reaktivne web aplikacije u Spring ekosustavu koristit će se samo određeni podskup svih navedenih projekata/platforma i modula. Korišteni skup sastoji se od: Spring Initializr platforme za generiranje početnog projekta, Spring Boot okvira, reaktivnog Spring Security projekta, Spring WebFlux (koji koristi Reactor biblioteku), Spring Data (Reaktivni

repozitoriji), Spring Cloud, adapteri za reaktivni tok, servlet kontejneri i ugrađeni Netty server. Ovaj podskup uz izuzetak Spring Cloud projekta predstavljaju minimalni skup potreban za izradu funkcionalne reaktivne web aplikacije. Kako bi se dobio malo bolji kontekst projekata i modula koji će se koristiti, na idućoj slici možemo vidjeti usporedbu odnosno presliku jedan na prema jedan navedenih projekata i modula dvaju različitih skupova: reaktivni skup koji se temelji na reaktivnom programiranju i servlet skup koji predstavlja standardni skup za kreiranje MVC (eng. Model-View-Controller) web aplikacija temeljenom na imperativnom programiranju.

U izradi implementacijskog dijela ovog rada koristi će se lijevi skup „Reaktivni skup“ koji se sastoji od modula i projekata koji su prvenstveno namijenjeni za izradu Spring web aplikacija baziranih na reaktivnom programiranju. Rad s navedenim modulima bit će detaljnije objašnjeni u narednim poglavljima koja će se specifično baviti njihovom primjenom i korištenjem u implementaciji specifičnih funkcionalnosti aplikacije.

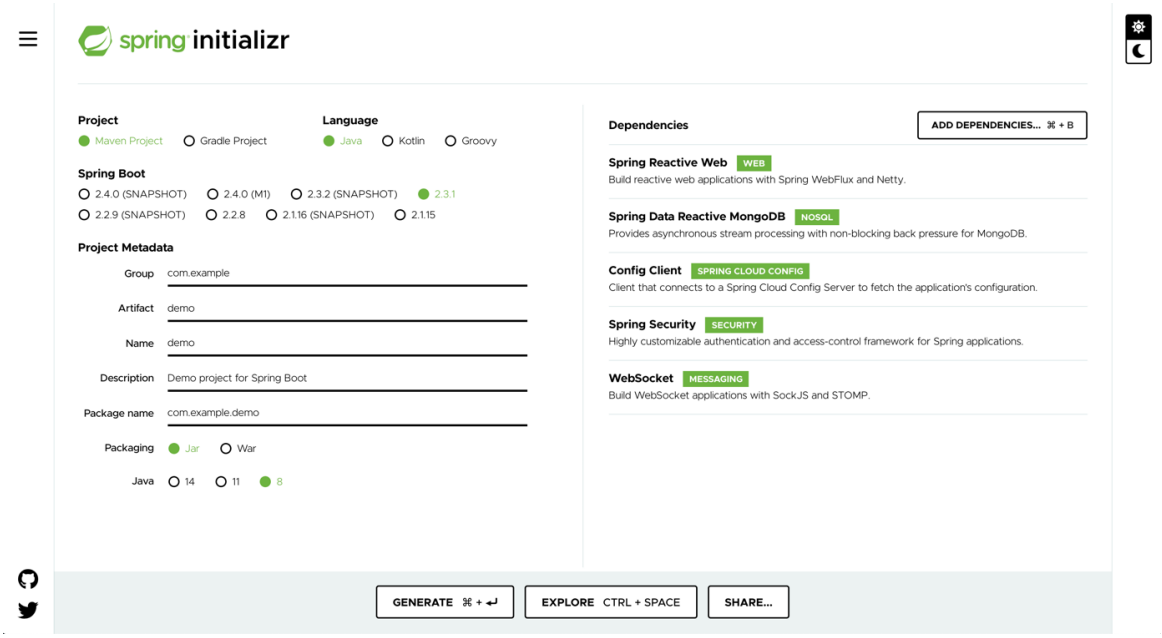


Slika 38: Razlika između reaktivnog skupa i servlet skupa okvira u Spring ekosustavu (Prema: *Project Reactor and the Spring Portfolio*, 2020)

6.2.2.1. Spring boot okvir i Spring Initializr platforma

Spring Boot predstavlja okvir koji je svojevrsna nadogradnja na bazni Spring okvir koji se koristi za izradu aplikacija baziranih na Spring-u. Spring Boot stvoren je sa svrhom olakšavanja početnog postavljanja ali i konfiguriranja aplikacija u daljnjem razvoju s ciljem da se izbjegne opterećivanje programera s konfiguriranjem brojnih XML datoteka čije konfiguriranje može biti komplicirano, dugotrajno te koje smanjuje produktivnost u razvoju

aplikacije. Spring Boot automatski kreira sve potrebne konfiguracijske datoteke aplikacije na temelju jedne glavne xml ili gradle konfiguracijske datoteke (ovisno koji alat za automatiziranu izgradnju projekta se koristi: Gradle ili Maven) u kojoj programer definira sve potrebne ovisnosti koje želi koristiti u projektu. Međutim programer je oslobođen i samog procesa ručnog postavljanja ovisnosti projekta na način da je Spring kreirao posebnu platformu po nazivom Spring Initializr[7] koja služi kao generator početnog projekta. Korištenjem ove platforme potrebno je samo navesti određeni skup podataka kao što su vrsta projekta (Maven ili Gradle), jezik koji će se koristi za programiranje (Java, Kotlin ili Groovy), verzija Spring Boot-a te metapodaci projekta (grupa, naziv artefakta, naziv projekta, korijen paketa, verzija jave) i ono najvažnije ovisnosti (eng. *Dependencies*) koje želimo koristiti u našem projektu. Nakon što je sve definirano, klikom na gumb generiranje dobivamo zip datoteku koja sadrži početni Spring Boot projekt s postavljenom projektnom strukturom i sa stvorenim konfiguracijskim datotekama. Ovako generirani projekt moguće je instantno pokrenuti jer dolazi s ugrađenim internim aplikacijskim serverom.



Slika 39: Primjer korištenja Spring Initializr platforme

U sljedećoj tablici možemo vidjeti koje su to sve glavne ovisnosti korištene u izradi pojedinog projekta u konačnom sustavu. Tablica sadrži nazive modula (ovisnosti) zajedno s pripadajućim artefaktom (eng. Artifact) koje je moguće pronaći uz pomoć korištenja Spring Initializr platforme. Iz tablice možemo vidjeti da mikroservisi Products, Chat, Reviews imaju isti skup ovisnosti dok glavna aplikacija Shop ima par dodatnih kao što su Spring Security i Thymeleaf pošto igra ulogu korisničke aplikacije. Eureka poslužitelj sadrži samo jednu istoimenu ovisnost iz razloga što ona sadrži samo informacije o svim klijentskim aplikacijama.

[7] Platforma za generiranje Spring Boot projekta (https://start.spring.io/)

Projekt	***Ovisnosti i artefakti*** (eng. *Dependencies*)
Shop	▪ **Spring Reactive Web** (spring-boot-starter-webflux) ▪ **Spring Security** (spring-boot-starter-security) ▪ **Spring Data Reactive MongoDB** (spring-boot-starter-data-mongodb-reactive) ▪ **Reactive Cloud Stream** (spring-cloud-stream-reactive) ▪ **Eureka Discovery Client** (spring-cloud-starter-netflix-eureka-client) ▪ **Thymeleaf** (spring-boot-starter-thymeleaf)
Products	▪ **Spring Reactive Web** (spring-boot-starter-webflux) ▪ **Spring Data Reactive MongoDB** (spring-boot-starter-data-mongodb-reactive) ▪ **Reactive Cloud Stream** (spring-cloud-stream-reactive) ▪ **Eureka Discovery Client** (spring-cloud-starter-netflix-eureka-client)
Chat	▪ **Spring Reactive Web** (spring-boot-starter-webflux) ▪ **Spring Data Reactive MongoDB** (spring-boot-starter-data-mongodb-reactive) ▪ **Reactive Cloud Stream** (spring-cloud-stream-reactive) ▪ **Eureka Discovery Client** (spring-cloud-starter-netflix-eureka-client)
Reviews	▪ **Spring Reactive Web** (spring-boot-starter-webflux) ▪ **Spring Data Reactive MongoDB** (spring-boot-starter-data-mongodb-reactive) ▪ **Reactive Cloud Stream** (spring-cloud-stream-reactive) ▪ **Eureka Discovery Client** (spring-cloud-starter-netflix-eureka-client)
Eureka Server	▪ **Eureka server** (spring-cloud-starter-netflix-eureka-server)

Tablica 3: Prikaz ovisnosti i artefakata korištenih u pojedinom projektu

6.2.2.2. Spring Reactive Web modul

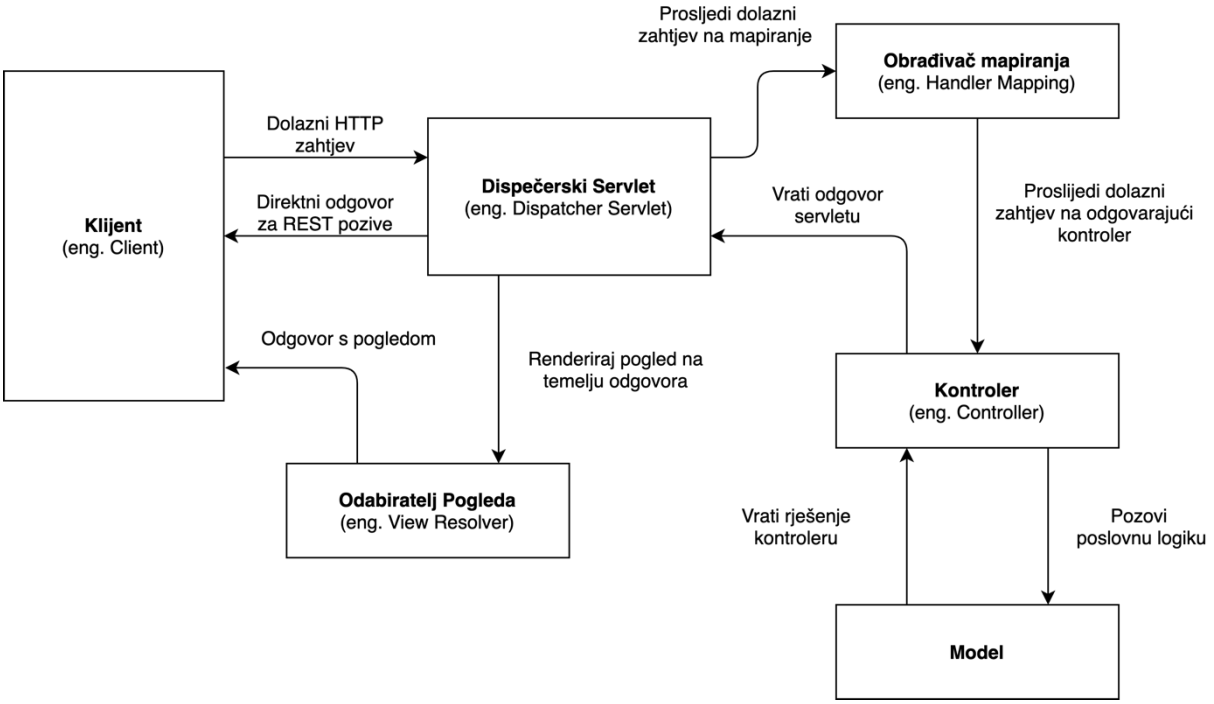
Spring Reactive Web modul predstavlja modul koji pripada Spring Web grupaciji modula koje svrstavamo u zajednički sloj (eng. *Common Layer)* kao što smo mogli vidjeti na dijagramu Spring ekosustava. Ovaj modul došao je u sklopu nove verzije Spring okvira, verzije 5. Reaktivni modul podržava iste modele programiranja kao i njegov dvojnik Spring MVC koji se koristi za standardnu imperativnu implementaciju serverske aplikacije, ali s novom značajkom izvršavanja na reaktivan i neblokirajući način. Ovaj modul sastoji se od WebFlux i Reactor Netty okvira čiji su princip rada od iznimne važnosti za ovaj rad iz razloga što implementacijski dio, odnosno projekt, koristi WebFlux i Netty u pozadini za svoj rad. Stoga kako bi se mogle lakše i bolje objasniti funkcionalnosti aplikacije dati ćemo kratko objašnjenje ovih tehnologija te prikazati jednostavan primjer reaktivnog servisa korištenjem WebFlux okvira.

Reactor Netty, kao što ime govori, baziran je na Netty okviru, te predstavlja asinkroni mrežni neblokirajući programski okvir baziran na radu s događajima. Cilj ovog okvira je pružiti asinkrono i neblokirajuće izvršno okruženje visoke propusnosti s podrškom za rad s povratnim tlakom reaktivnih tokova za klijentske ali i poslužiteljske aplikacije s mogućnošću korištenja niza protokola kao što su TCP, UDP i HTTP.

WebFlux predstavlja reaktivnu verziju web okvira za razvoj web aplikacija baziranih na Spring okviru. Osim što pruža podršku za razvoj reaktivnih poslužiteljskih aplikacija, također nudi i podršku u obliku klijentskih biblioteka za pozivanje udaljenih REST servisa. WebFlux je potpuno neblokirajući okvir koji podržava specifikaciju reaktivnih tokova i povratnog tlaka. Ovaj okvir moguće je izvršavati na serverima kao što je Netty, Undertow i Servlet 3.1+ kontejnerima. Razlog zašto nije poželjno za izvršavanje koristiti standardni Tomcat server koji dolazi uz Spring MVC okvir je taj što Tomcat funkcionira na drugačijem modelu arhitekture jedna dretva jedan zahtjev dok Netty i Undertow koriste arhitekturu petlje događaja koja izvršavanje čini neblokirajućim. Korištenjem WebFlux okvira u sklopu Spring Boot aplikacije, Spring Boot automatski postavlja u konfiguracijskim datotekama Reactor Netty kao standardni server za korištenje.

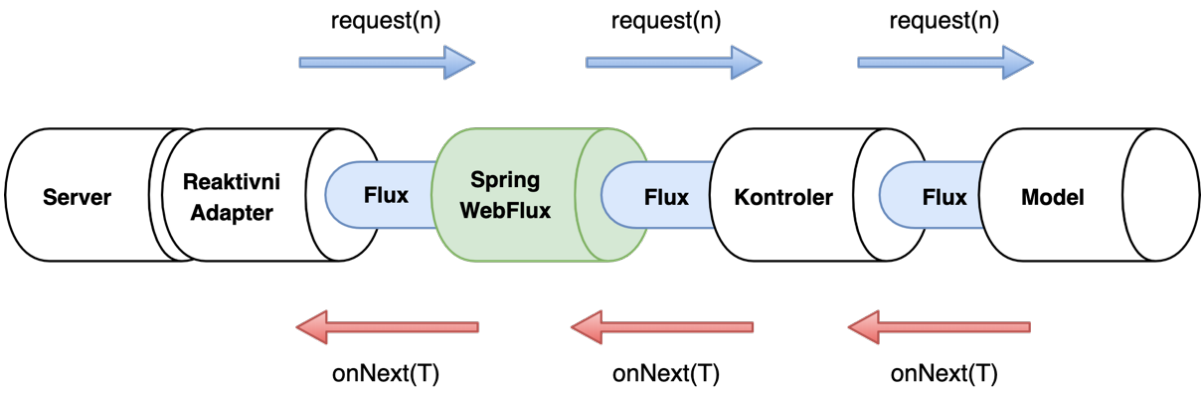
U nastavku ovog poglavlja prikazat ćemo implementaciju jednog jednostavnog primjera REST kontrolera na reaktivan način uz pomoć korištenja WebFlux okvira. Međutim prije nego li krenemo na sami programski primjer prvo ćemo objasniti što je to kontroler u Spring okviru, zašto se koriste i na koji način. Koncept kontrolera koriste se u više svrha: za presretanje dolaznih HTTP zahtjeva, konvertiranje korisnog tereta (eng. *Payload*) HTTP zahtjeva u interne strukture podataka, prosljeđivanje konvertiranih struktura sloju modela u kojem se strukture dalje procesuiraju, te za dohvaćanje i prosljeđivanje procesuiranih vrijednosti iz sloja modela

u sloj pogleda u kojem se događa renderiranje prikaza. Kako bismo lakše shvatili proces na idućem dijagramu možemo vidjeti prikaz arhitekture visoke prethodno opisanog procesa.



Slika 40: Prikaz Spring MVC arhitekture visoke razine (Prema: Dutta, 2016)

Prikazani dijagram prikazuje proces prosljeđivanja HTTP zahtjeva u Spring MVC arhitekturi. Spring MVC predstavlja još jedan od mogućih okvira za razvoj web aplikacija baziranih na Spring-u. Međutim ovaj okvir se prvenstveno temelji na principu blokirajućih ulazno/izlaznih operacija ali nudi i mogućnost korištenja reaktivnih značajki do određene razine. Ovaj okvir moguće je izvršavati na poslužitelju kao što je Tomcat koji rade na principu jedna dretva jedan zahtjev. Ovaj dijagram prikazan je zato što MVC i WebFlux zapravo koriste isti model programiranja stoga iste koncepte kao što su anotacije, kontroler i model možemo pronaći u jednom i u drugom. Ako isti dijagram toka zahtjeva želimo prikazati s gledišta u kontekstu WebFlux okvira to možemo učiniti na sljedeći način.



Slika 41: Prikaz WebFlux arhitekture visoke razine (Prema: Stoyanchev, 2018)

Sljedeći programski primjer prikazuje vrlo jednostavni reaktivni REST kontroler koji je tipičan za Spring web okvire. Naime ovaj kontroler sadrži određene anotacije kao što su `@RestController` koja je potrebna kako bi se implementacijska klasa mogla otkriti u procesu skeniranja klasa te `@RequestMapping` koja se koristi za mapiranje dolaznih zahtjeva na odgovarajući kontroler. Klasa kontrolera implementira četiri metode `get`, `getAll`, `save`, `update` i `delete` koje respektivno dohvaćaju, spremaju, ažuriraju i brišu određeni proizvod ili proizvode. Ove četiri metode predstavljaju osnovne metode koje je moguće pronaći u REST kontroler implementaciji. Jedini reaktivni konstrukti koji se mogu primijetiti u primjeru jesu reaktivni povratni tipovi `Flux` i `Mono` pojedinih REST metoda te korištenje operatora `map` i `defaultIfEmpty`. Naravno u ovom primjeru pretpostavljamo da je i korišteni servis implementiran na reaktivan način, odnosno da vraća `Flux` i `Mono` tipove podataka.

```
@RestController
@RequestMapping("/products")
public class ProductController {
    ...
    @GetMapping(path = "/all")
    public Flux<Product> getAll(){
        return this.productService.findAll();
    }
    @GetMapping(path = "/{id}")
    public Mono<Product> get(@PathVariable("id") UUID uuid) {
        return this.productService.findOne(uuid);
    }
    @PostMapping
    public Mono<ResponseEntity<Product>> save(@RequestBody Product prod) {
        return this.productService
                .save(prod)
                .map(savedProduct ->
                        new ResponseEntity<>(savedProduct, HttpStatus.CREATED));
    }
    @PutMapping
    public Mono<ResponseEntity<Product>> update(@RequestBody Product prod) {
        return this.productService
                .update(prod)
                .map(savedProduct ->
                        new ResponseEntity<>(savedProduct, HttpStatus.CREATED))
                .defaultIfEmpty(new ResponseEntity<>(HttpStatus.NOT_FOUND));
    }
    @DeleteMapping(path = "/{id}")
    public Mono<ResponseEntity<String>> delete(
            @PathVariable("id") UUID uuid) {
        return this.productService
                .delete(uuid)
                .map((Boolean status) ->
                        new ResponseEntity<>("Deleted", HttpStatus.ACCEPTED));
    }
}
```

Kôd 33: Reaktivni REST servis u okviru WebFlux

6.2.2.3. Spring Security

Spring Security je vrlo moćan i prilagodljiv okvir za provjeru autentičnosti i kontrolu pristupa. Predstavlja de-facto standard za osiguravanje Spring aplikacije. Okvir se fokusira na pružanje i provjeru autentičnosti i autorizacija (*Spring Security*, 2020). Ovaj okvir najčešće se koristi u integraciji sa Spring Web MVC okvirom, ali podržava i integraciju s WebFlux okvirom. Za omogućavanje WebFlux podrške u Spring Security potrebno je samo navesti anotaciju `@EnableWebFluxSecurity` na klasnoj razini u sklopu konfiguracijske klase za sigurnost. Nakon što smo anotirali klasu u njenoj implementaciji možemo iskoristiti prednosti koje nam nudi klasa `ServerHttpSecurity`. Ova klasa koristi se specifično za izgradnju sigurnosnog lanca filtera za reaktivne aplikacije.

```
@Configuration
@EnableWebFluxSecurity
public class WebSecurityConfig {

    @Bean
    public SecurityWebFilterChain securityWebFilterChain(
            ServerHttpSecurity http){
        return http
                .authorizeExchange()
                .pathMatchers("/registration*", "/login*")
                .permitAll()
                .anyExchange().authenticated()
                .and()
                    .formLogin()
                    .loginPage("/login")
                .and()
                    .logout()
                    .logoutUrl("/logout")
                    .logoutSuccessHandler(logoutSuccessHandler("/"))
                .and()
                    .csrf().disable()
                .build();
    }
    ...
}
```

Kôd 34: Primjer implementacije klase sigurnosne konfiguracije za osiguravanja aplikacije izgrađene korištenjem WebFlux okvira.

6.2.2.4. Spring Data Reactive MongoDB

Spring Data predstavlja rješenje čiji je cilj pružiti poznati i dosljedni model programiranja pristupa podacima pritom zadržavajući posebne osobine spremišta podataka koje se koriste u pozadini. Omogućava jednostavno korištenje tehnologija pristupa podacima relacijskih ali i ne-relacijskih baza podataka. Predstavlja takozvani kišobran projekt (eng. *Umbrella project*) što sugerira da se sastoji od većeg broja potprojekta specifičnih za pojedine baze podataka (*Spring Data*, 2020).

Spring Data Reactive MongoDB predstavlja takav jedan potprojekt koji pruža karakteristike asinkronog toka procesuiranja s neblokirajućim povratnim tlakom za nerelacijsku bazu podataka MongoDB (*Spring Data MongoDB - Reference Documentation*, bez dat.). Razlog zašto je izabrana nerelacijska baza podataka, specifično MongoDB je taj što je u vrijeme izrade diplomskog rada bila jedan od rijetkih baza podataka koja je podržavala rad s Java reaktivnim upravljačkim programima ili drajverima (eng. *Java Reactive Drivers*).

Pošto se u izradi diplomskog rada ciljalo na izradu cjelokupnog reaktivnog programskog rješenja, korištenje baze za koju je bilo moguće implementirati reaktivnu komunikaciju predstavljalo je važan kriteriji kako se ne bi dogodilo blokiranje izvršavanja programa u procesu komunikacije sa samom bazom. Međutim što to točno znači za samo izvršavanje programa naspram korištenja standardnih sinkronih upravljačkih program za komunikaciju. Recimo da želimo pomoću sljedećeg SQL upita `SELECT TOP 100 * FROM Products` dohvatiti prvih 100 redova iz tablice Products. U ne reaktivnom svijetu, izvršavanjem ovog upita značilo bi da se iz bazena dretvi za komunikaciju treba utilizirati dretva koja bi bila zadužena za rukovanje s konekcijom na bazu te koja bi se koristila samo za tu jednu specifičnu konekciju. Za ovakav jednostavan upit problem ovog pristupa nije toliko vidljiv, međutim zamislimo da se radi o nekom složenijem upitu s ugniježđenim podupitima koju u konačnici može vratiti preko 10 000 redova rezultata. Ovakav upit bi se vjerojatno vrlo dugo izvodio što bi posljedično blokiralo našu komunikacijsku dretvu, prvo čekanjem na odgovor, a onda zatim i obradom velike količine podataka. Ovim načinom uzrokovali bismo pojavu uskog grla u podatkovnom sloju. Međutim, korištenjem reaktivnog drajvera, koja je u MongoDB slučaju nadogradnja na njegov već postojeći asinkroni, izbjegavamo pojavu ovakve situacije. Nadalje, ova poveznica implementirana je na način da reaktivni API zapravo zrcali asinkroni API drajvera na način da sve metode koje uzrokuju mrežnu I/O operaciju zapravo vraćaju objekt tipa `Publisher<T>` gdje `T` predstavlja vrstu odgovora operacije.

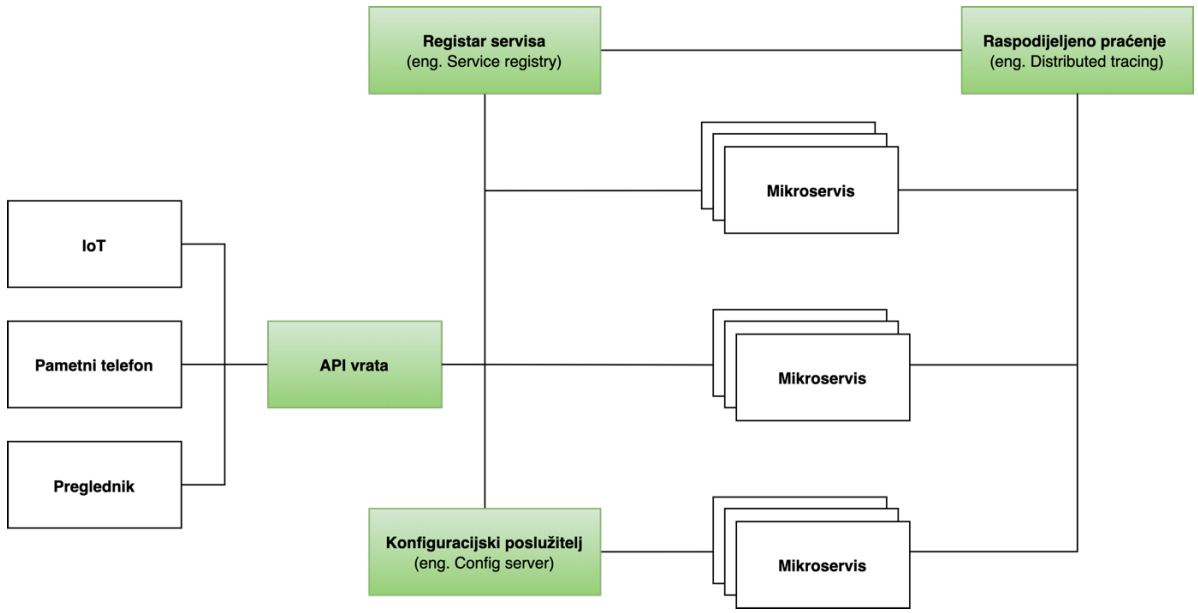
U sljedećem programskom isječku možemo vidjeti primjer jednog tipičnog sučelja u Spring aplikacijama koje se koristi u podatkovnom sloju za izvršavanje upita nad bazom podataka. Važna činjenica za primijetiti u ovom sučelju je da ovo sučelje nasljeđuje `ReactiveCrudRepository<T,S>` koje nudi na korištenje već neke predefinirane metode kao što su `findAll()`, `findById(long)`, `count()` i tako dalje. Također možemo vidjeti da sve ove metode vraćaju reaktivne tipove podataka `Flux<T>` i `Mono<T>`.

```
@Repository
public interface ProductRepository
        extends ReactiveCrudRepository<Product, Long> {
    Mono<Product> findByName(String name);
    Mono<Product> findById(long productId);
    Flux<Product> findAll();
    Mono<Product> findTopByOrderByIdDesc();
}
```

Kôd 35: Primjer sučelja reaktivnog repozitorija

6.2.2.5. Spring Cloud i Cloud Stream

Spring Cloud predstavlja skup projekata i alata koje je moguće koristiti za izradu nekih od uobičajenih obrazaca u distribuiranim sustavima kao što su obrasci za otkrivanje servisa, upravljanje konfiguracijama, inteligentno usmjeravanje, prekidača i slično. U većini slučajeva, proces izrade distribuiranih sustava, njihova koordinacija i konfiguracija uzrokuje potrebu korištenja velikog broja predložaka poznatiji pod nazivom “boilerplate”. Korištenjem Spring Cloud-a olakšava razvoj distribuiranih sustava na način da se konfiguracijski predlošci mogu na jednostavan način ukomponirati u samu aplikaciju ali isto tako nudi i na korištenje veliki broj servisa koji olakšavaju samu izradu distribuiranog sustava (*The Spring Cloud*, 2020).



Slika 42: Spring Cloud okvirna arhitektura (Izvor: The Spring Cloud, bez dat.)

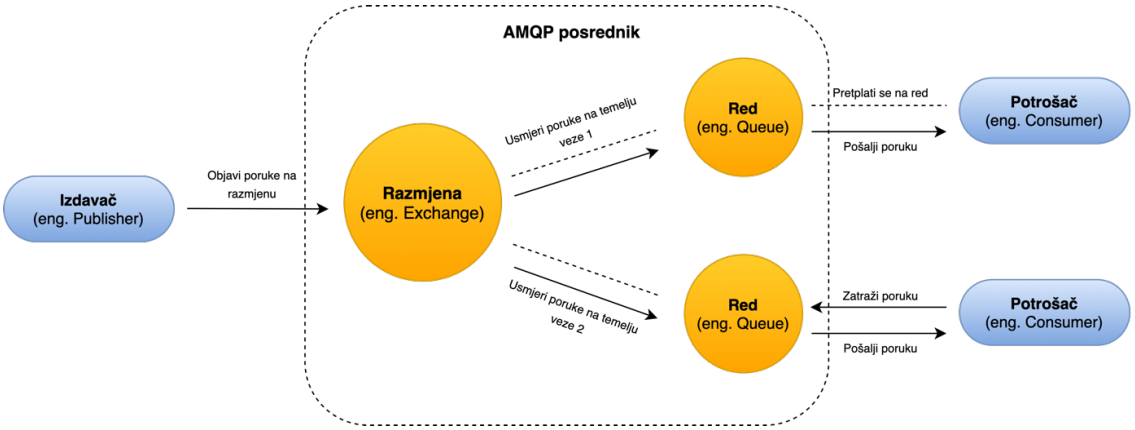
Spring Cloud projekt u ovom radu korišten je iz više razloga. Jedan od razloga bila je podjele poslovne logike aplikacije u više pojedinih servisa. Drugi razlog korištenja i ujedno onaj važniji bio je za prikazivanje mogućnosti i načina primjene reaktivnog programiranja u distribuiranim sustavima s ciljem postizanje i zadovoljavanja specifikacija reaktivnih sustava. Međutim sama implementacija mikroservisne arhitekture uz pomoć Spring Cloud projekta nije dovoljna za postizanje specifikacija reaktivnih sustava, koja ako se prisjetimo navedenih

zahtjeva, nalaže da komunikacija između mikroservisa aplikacije te samih podsustava i sustava bude temeljena na porukama. Zbog toga je za razvoj sustava korištena još jedan dodatan okvir naziva **Spring Cloud Stream** koji podržava rad na reaktivan način. Ovaj okvir se koristi primarno za izgradnju vrlo skalabilnih mikroservisa usmjerenih na rad s događajima, a koji su međusobno povezani zajedničkim sustavom razmjene poruka. Glavni koncepti Cloud Stream okvira jesu (*Spring Cloud Stream*, 2020):

- **Odredišna veziva** (eng. *Destination binders*) – Komponente odgovorne za integraciju s vanjskim sustavima za razmjenu poruka.
- **Odredišne veze** (eng. *Destination bindings*) – Predstavlja most između vanjskih sustava za razmjenu poruka te aplikacijskih proizvođača i potrošača poruka (stvorenih od strane odredišnih veziva).
- **Poruke** – Kanonska struktura podataka koju proizvođači i potrošači koriste za komunikaciju s odredišnim vezivima te s ostalim aplikacijama putem vanjskih sustava za razmjenu poruka.

Cloud Stream okvir podržava rad s nekolicinom implementacija odredišnih veziva pruženih od trećih strana. U ovom radu korištena je projekt imena RabbitMQ[8] kao implementacija odredišnog veziva, a koji igra ulogu posrednika za poruke. RabbitMQ izravno implementira napredni protokol čekanja poruka (eng. *AMQP - Advance Message Queuing Protocol)* koji predstavlja protokol otvorenog standarda za pružanje interoperabilnosti poruka između različitih sustava na aplikacijskom sloju. Na sljedećoj slici možemo vidjeti proces razmjene poruka između komponenti izdavača i potrošača putem AMQP posrednika te njegova tri glavna tipa entiteta. Ti entiteti su: **razmjena** (eng. *Exchange*), **poveznica** (eng. *Binding*) te **red** (eng. *Queue*). Komponente izdavača svoje poruke objavljuju na entitet razmjene. Nakon što entitet razmjene zaprimi poruke iste usmjerava na jedan od svojih povezanih redova koji su vezani za razmjenu na temelju posebnih pravila koja se nazivaju poveznicama. Svaka razmjena može imati nula ili više povezanih redova. Komponente potrošača mogu primiti poruku iz reda na dva načina. Prvi način je da eksplicitno odnosno ručno povuku poruke iz određenog reda, drugi način je automatski način primanja poruka gdje se poruke guraju iz reda ka komponenti potrošača na temelju pretplate potrošača na pojedini red. Kao što možemo primijetiti iz slike, za opis razmjene poruka koristi se ista terminologija izdavača, potrošača i pretplate koja se također koristila i za opisivanje principa reaktivnog programiranja.

[8] Softver otvorenog kôda za posredovanje poruka (https://www.rabbitmq.com)



Slika 43: Klijent-server uzorak poruka (Izvor: AMQP Testing | ReadyAPI Documentation, 2020)

Međutim, ponovo naglašavamo da cilj ovog rada nije izrada reaktivnog sustava u punom smislu, stoga iako ćemo koristiti komuniciranje putem poruka između servisa, njezina primjene neće se zahtijevati u svim dijelovima programskog rješenja iz razloga kako bismo mogli prikazati i mogućnost implementacije poziva udaljenih servisa (koji ne primjenjuju AMQP protokol) na reaktivan način.

Dijelovi sustava koji koriste komunikaciju baziranu na porukama jesu mikroservisi: Reviews i Chat. Prvi mikroservis je jednostavniji iz razloga što ima samo dva zaduženja, spremanje zaprimljenih recenzija od glavne aplikacije u bazu podataka te njihovo prosljeđivanje drugom mikroservisu. U idućem programskom isječku možemo vidjeti jedan primjer jednostavne servisne klase unutar mikroservisa Reviews. U sklopu klase možemo vidjeti korištenje određenih anotacija na klasnoj razini ali i na razini metoda. Prva anotacija `@Service` predstavlja standardnu anotaciju Spring okvira koja klasu označava kao klasu koja pripada servisnom sloju što znači da sadrži samo poslovnu logiku. Iduća anotacija `@EnableBinding(Processor.class)` predstavlja anotaciju Cloud Stream okvira koja se koristi za vezanje aplikacija koje sadrže ulazne i izlazne anotacije kanala s nekom od implementacija posrednika (u našem slučaju RabbitMQ) koje se izvodi na temelju popisa sučelja koja se prosljeđuju anotaciji. Korištenjem ove anotacije Cloud Stream okvir zapravo signalizira kreiranje novog kanala RabbitMQ posredniku koji potom u pozadini preslikava stvorene kanale u entitete razmjene i redove. U ovom primjeru anotaciji prosljeđujemo predefinirano sučelje `Processor` koje za nas definira ulazne i izlazne kanale s pripadajućim anotacijama i nazivima kanala. U slučaju da u sklopu jedne aplikacije koristimo više ulazni i izlaznih kanala tada moramo implementirati vlastita sučelja koje će sadržavati jedinstvena imena ulaznih i izlaznih kanala. Kako `Processor` sučelje implementira oba kanala (ulazni i izlazni) ovaj servis, a tako i cijela aplikacija (mikroservis) ima mogućnost primanja i slanja poruka.

```
public interface Processor extends Source, Sink {}

public interface Source {
    String OUTPUT = "output";

    @Output("output")
    MessageChannel output();
}

public interface Sink {
    String INPUT = "input";

    @Input("input")
    SubscribableChannel input();
}
```

Kôd 36: Standardna sučelja `Processor`, `Source` i `Sink` za stvaranje komunikacijskih kanala

Prethodnom anotacijom definirali smo na razini aplikacije ulazne i izlazne kanale, odnosno ispravnije bi bilo reći ulazno i izlazne točke. Međutim, da bismo s ulaznih točaka mogli pročitati podatke ili na izlazne točke proslijediti podatke potrebno je korištenje dodatnih anotacija na samoj razini metode. Jedna od tih anotacija je `@StreamListener` koja označava anotiranu metodu kao slušača ulaznih vrijednosti deklariranih korištenjem prethodno opisane anotacije i definicije kanala. Ova anotacija prima parametar ulaznog sučelja što bi u našem slučaju bilo `Processor.INPUT`, međutim u našem primjeru anotacija ulaznog kanala implementirana je na način da omogućava fleksibilnije potpise metoda koje se pozivaju pa tako možemo vidjeti anotaciju `@Input(Processor.INPUT)` koja anotira sam ulazni parametar metode. Ovisno koliku fleksibilnost želimo i jedan i drugi predstavlja validni način. Zadnja anotacija koju koristimo na razini ove metode je anotacija `@Output(Processor.OUTPUT)` koja označava na koji kanal se prosljeđuju podaci. Osim samog korištenja konstrukata Cloud Stream okvira za definiranje procesa razmjena poruka između većeg broja aplikacija, također možemo vidjeti i vrlo jednostavnu primjenu reaktivnog programiranja čije se korištenje ogleda u korištenju tokova podataka `Flux` za ulazni parametar, korištenju reaktivnog repozitorija za spremanje recenzija te korištenju operatora `map`. Važno je napomenuti da ovaj mikroservis iako prima tok podataka kao ulaznu vrijednost ne predstavlja reaktivni izvor iz razloga što ne prosljeđuje podatke u obliku toka na izlazni kanal. Da bi se običan izvor podataka pretvorio u reaktivni potrebno je koristiti dodatnu anotaciju `@StreamEmitter` na razini metode. Za ispravnu implementaciju reaktivnog slanja poruka potrebno je definirati `FluxSender` objekt kao izlazni parametar metode koja se koristi za emitiranje podataka. Ova anotacija koristi se u glavnoj aplikaciji za reaktivno slanje kreiranih recenzija mikroservisu Reviews.

```
@StreamEmitter
public void emit(@Output(Source.OUTPUT) FluxSender output){
    output.send(this.flux);
}
```

Kôd 37: Reaktivna metoda za emitiranje poruka u sklopu korištenja Cloud Stream okvira

Sve ovo prikazuje mogućnost zajedničkog korištenja ovih dvaju koncepata u procesu izgradnje ali i rada aplikacija i sustava. Ako se prisjetimo poglavlja koje se bavilo opisom reaktivnih sustava, spomenuli smo da ova kombinacija predstavlja jednu od mogućih kombinacija za implementaciju takvih sustava u kojoj se reaktivno programiranje koristi na razini aplikacije, a komuniciranje putem poruka na razini cijelog sustava. U idućem programskom isječku možemo vidjeti primjenu prethodno opisanih anotacija i reaktivnog programiranja.

```
@Service
@EnableBinding(Processor.class)
public class ReviewService {

    private final static Logger log =
            LoggerFactory.getLogger(ReviewService.class);
    private ReviewRepository reviewRepository;

    public ReviewService(ReviewRepository reviewRepository) {
        this.reviewRepository = reviewRepository;
    }

    @StreamListener
    @Output(Processor.OUTPUT)
    @StreamEmitter
    public Flux<Review> save(
            @Input(Processor.INPUT) Flux<Review> newComments)
    {
        return reviewRepository
                .saveAll(newComments)
                .log("reviewService-save")
        .map(review ->{
            log.info("Saving new review " + review);
            return review;
        });
    }
}
```

Kôd 38: Implementacija servisa za spremanje i prosljeđivanje recenzija

6.2.3. Komunikacija s udaljenim servisima

Kao što smo prethodno rekli, ova aplikacija ne predstavlja reaktivni sustav u punom smislu te iako se koristi koncept slanja poruka između mikroservisa ona nije nužno podržana u svim dijelovima aplikacije. Stoga u samoj aplikaciji postoji korištenje i obične komunikacije s udaljenim servisima ali na reaktivan način. U Spring Web Reactive modulu za komunikaciju s udaljenim servisima koja podržava reaktivnu paradigmu koristi se specifično sučelje

`WebClient` koje predstavlja glavnu ulaznu točku. Ovo sučelje će u budućim verzijama Spring okvira u potpunosti zamijeniti korištenje starog `RestTemplate` sučelja koje predstavlja sinkrono i blokirajuće rješenje.

U iduća dva programska isječka možemo vidjeti korištenje `WebClient` sučelja za izvršavanje `HTTP GET` i `HTTP DELETE` zahtjeva. U prvom primjeru koristimo `HTTP GET` kako bismo dohvatili recenzije nekog artikla. Kako određeni artikla može imati više recenzija to znači da ćemo odgovor morati preslikati u `Flux`. Međutim prije nego što odgovor preslikamo moramo pozvati metodu `exchange()` koja će kreirati tok *`Mono<ClientResponse>`* iz razloga što odgovor sadržan kao dio tijela `HTTP` odgovora predstavlja jednu cjelinu unutar koje se možemo nalaziti veći broj različitih vrijednosti. Tek nakon toga možemo iskoristiti operator *`flatMapMany`* unutar kojeg ćemo iz sadržaja tijela odgovora kreirati novi tok koji će sadržavati niz vrijednosti sadržane u tijelu dobivenog odgovora.

```
@Component
public class CommentHelper {
    private final WebClient webClient;

    public CommentHelper() {
        this.webClient = WebClient.create("http://localhost:9000");
    }

    @HystrixCommand(fallbackMethod = "defaultReviews")
    public Flux<Review> getReviews(long productId){
        return webClient.get()
                .uri("/reviews/{productId}", productId)
                .exchange()
                .flatMapMany(clientResponse -> {
                   return clientResponse.bodyToFlux(Review.class);
                });
    }

    public Flux<Review> defaultReviews(long productId){
        return Flux.empty();
    }
}
```

Kôd 39: Primjer korištenja poziva udaljenog servisa `HTTP GET`

Drugi primjer prikazuje korištenje `WebClient` sučelja za izvršavanje `HTTP DELETE` zahtjeva. Ovdje je proces malo jednostavniji jer nakon što je zahtjev poslan ne očekujemo nikakav odgovor stoga kreiramo samo prazan tok tipa `Mono<Void>`. Isti princip primjenjuje se i za sve ostale vrste `HTTP` zahtjeva.

```
@Component
public class ProductHelper {
    private final WebClient webClient;

    public ProductHelper() {
        this.webClient = WebClient.create("http://localhost:9090");
    }

    ...
    public Mono<Void> deleteProduct(long productId){
        return webClient.delete()
                .uri("/product/{productId}",productId)
                .exchange()
                .flatMap(clientResponse -> {
                    return clientResponse.bodyToMono(Void.class);
                });
    }
    ...
}
```

Kôd 40: Primjer korištenja poziva udaljenog servisa `HTTP DELETE`

6.3. Implementacija specifičnih funkcionalnosti

Kako je aplikacija podosta bogata funkcionalnostima u ovom poglavlju bit će prikazane samo one koje ne pripadaju uobičajenom skupu funkcionalnosti, a koje se odnose na dohvaćanje, kreiranje, ažuriranje, prikaza podataka i slično te će se veća pažnja dati onim funkcionalnostima koje koriste reaktivno programiranje za rješavanje nekog specifičnog problema kao što je osvježavanje prikaza u realnom vremenu bez korištenja AJAX poziva.

6.3.1. Reaktivna registracija

Iako funkcionalnost registracije sama po sebi predstavlja uobičajenu funkcionalnost aplikacije bilo ona web, desktop ili mobilna aplikacija ipak ćemo prikazati kako se ova funkcionalnost može implementirati na reaktivan način iz razloga što predstavlja savršenu funkcionalnost za objašnjavanje školskog primjera tokova podataka i alternativnih puteva. Također zbog korištenje NoSQL baze podataka MongoDB postoji jedna specifična prepreka kod registracije i kreiranja novih korisnika, a to je da NoSQL baze, koncept poznat pod nazivom primarni ključ malo drugačije implementiraju i koriste u odnosnu na standardne relacijske baze. Stoga kako bismo olakšali uvid u povezanost između različitih entiteta dokumenata u bazi koristi ćemo vlastito definirano polje za ID koje će nam predstavljati primarni ključ određenog zapisa. Krenimo prvo od opisivanja klase kontrolera za registraciju. Kao što već od prije znamo, u Spring okviru entiteti kontrolera čine glavnu ulaznu točku za HTTP zahtjeve korisnika. Pa tako za registraciju imamo sljedeću klasu kontrolera.

```
@Controller
public class RegistrationController {
    @Autowired UserRegistrationService userRegistrationService;
    @GetMapping("/registration")
    public Mono<String> registration(){
        return Mono.just("registration");
    }
    @RequestMapping(value = "/registration", method = RequestMethod.POST)
    public Mono<String> registerNewUserAccount(
            @ModelAttribute("user") UserDto userDto,
            Model model) {
        return userRegistrationService.registerNewUserAccount(userDto)
                .map(user -> redirectToLoginPage())
                .switchIfEmpty(returnRegistrationPageWithErrorMsg(model));
    }
    private String redirectToLoginPage(){
        return "redirect:/login";
    }
    private Mono<String> returnRegistrationPageWithErrorMsg(Model model){
        model.addAttribute("message", "Unsuccessful registration!");
        return Mono.just("registration");
    }
}
```

Kôd 41: Klasa registracijskog kontrolera

Kao što možemo vidjeti, klasa se sastoji od standardnih anotacija specifičnih za Spring web aplikacije. Prva metoda `registration()` koja je anotirana `@GetMapping` anotacijom te koja je mapirana na URL putanju <korijenski_url>/registration bit će pozvana onda kada korisnik napravi `HTTP GET` zahtjev, odnosno kada klikne na gumb, upiše u tražilicu ili na neki treći način zatraži prikaz stranice za registraciju. Ako se prisjetimo načina na koji okvir obrađuje ovakve zahtjeve, server odnosno aplikacija vraća samo ime predloška na temelju kojeg se generira pogled ili stranica koja će se prikazati korisniku. Iz razloga što je potrebno generirati pogled na temelju samo jednog predloška, ova metoda vraća `Mono` tip podataka koji sadrži samo jednu `String` vrijednost, a to je sam naziv predloška. Druga metoda ove klase `registerNewUserAccount` mapirana je na isti URL kao i prethodna ali se koristi za drugačiji tip `HTTP` zahtjeva, a to je `POST` zahtjev. Ova metoda poziva se kada korisnik ispuni registracijsku formu i klikne na gumb za predaju forme. Ova akcija predstavlja trenutak u kojem se može vidjeti potencijal primjene reaktivnog programiranja. Naime, ovisno o unesenim podacima korisnika te o već postojanju korisnika s istim korisničkim imenom registracija može biti uspješna ili neuspješna što znači da ova metoda može vratiti dva različita odgovora. Jedan u obliku redirekcijskog URL-a koji korisnika preusmjerava na ekran za prijavu i drugi koji vraća ime predloška zajedno s ažuriranim objektom modela za pogled, a koji sadrži poruku pogreške. Kako je u reaktivnom programiranju sve tok podataka, ova funkcionalnost može se napisati kao jedan slijed događaja koje uz primjenu reaktivnog programiranja postižemo uz pomoć slaganja i korištenja operatora koji će se izvršiti nad danim tokom. Proces implementiran na ovaj način izgleda puno prirodnije i čitljivije nego što je to slučaj kod imperativnog programiranja. Glavni tok predstavlja pozivanje injektiranog reaktivnog servisa `userRegistrationService` koji služi za obavljanje registracije korisnika. U slučaju da je korisnik uspješno registriran, servis će nam vratiti objekt korisnika što znači da će se `map` operator uspješno izvršiti jer servis emitira vrijednost na tok. U sklopu map operatora, pozivamo metodu za transformaciju koja nam vraća redirekcijski URL na stranicu prijave. U slučaju da registracija korisnika nije prošla uspješno, servis neće emitirati vrijednost već će emitirati samo signal dovršetka. Kako se nije dogodilo emitiranje vrijednosti, `map` operator se neće izvršiti ali će se zato aktivirati operator `switchIfEmpty` koji kao što smo već prethodno rekli kreira novi alternativni tok izvršavanja. U ovom alternativnom toku izvršavanja emitirat će se predložak za stranicu registracije s dodatnim vrijednostima u sklopu objekta modela koji će sadržavati poruku pogreške. U idućem programskom primjeru možemo vidjeti implementaciju reaktivnog servisa za registraciju korisnika. Vidimo da ovaj servis injektira dva zrna (eng. *Beans*) preko konstruktora. Prvo je zrno `UserRepository` koje nasljeđuje `ReactiveCrudRepository` sučelje koja nam pruža mogućnost reaktivnog rada s bazom podataka. Drugo zrno predstavlja klasu `ChatHelper` i ova klasa se koristi za početno kreiranje

dokumenta u bazi koje će sadržavati korisnikove razgovore s drugim korisnicima. Metoda `registerNewUserAccount` kao parametar prima `UserDto` objekt koji sadrži korisnikove podatke unesene preko obrasca na stranici registracije. U metodi možemo vidjeti definiranje dvaju tokova podataka. Prvi tok podataka bavi se dohvaćanjem zadnjeg spremljenog korisnika u bazi. Međutim nas zanima samo ID zadnje spremljenog korisnika, stoga uz pomoć operatora vršimo transformaciju nad ovim objektom i emitiramo samo vrijednost njegovog ID polja. U slučaju da je baza prazna, emitira se vrijednost 0. Drugi tok bavi se provjerom postojanja korisnika s istim korisničkim imenom u bazi. U slučaju da korisnik ne postoji emitira se `boolean` vrijednost `false` te se u tom slučaju utilizira pomoćna klasa `ChatHelper` čijoj se instanci prosljeđuje korisničko ime koja ga zatim prosljeđuje kao dio tijela `HTTP POST` zahtjeva mikroservisu zaduženom za komunikaciju. Na kraju se kreira novi objekt korisnika na temelju unesenih podataka. U suprotnom, ako korisnik već postoji, emitira se `boolean` vrijednost `true` koja potom ulazi u `else` dio bloka (zbog negacije vrijednosti) te se u tom slučaju ne emitira vrijednost već samo signal dovršenosti `return Mono.empty()`. Na kraju koristimo operator `Mono.zip()` uz pomoć kojeg vršimo spajanje ovih dvaju tokova. Razlog razdvajanja ovih operacija u dva zasebna toka leži u tome što oba sadrže operacije čije izvršavanje ponekad može biti dugotrajnije. Tako primjerice prva operacija, ovisno koliko korisnika imamo spremljenih u bazi, pretraživanje, sortiranje i dohvaćanje zadnjeg registriranog korisnika može potrajati (pogotovo iz razloga što ne koristimo standardno indeksirano polje za primarni ključ). Nadalje, druga operacija sadrži korištenje vanjskog servisa koji su sami po sebi podložni latenciji zbog potrebe za slanjem podataka putem interneta. Stoga kako bi se ove dvije operacije mogle izvoditi konkurentno razdvojene su u dva toka koje potom spajamo u jedan tok na temelju `zip` operatora. `Zip` operator spojit će tokove tek onda kada oba toka emitiraju vrijednosti te će se tek u tom trenutku izvršiti spremanje novo kreiranog korisnika u bazu.

Dok smo opisivali proces registracije od klase kontrolera pa do klase servisa i pomoćne klase za komuniciranje s vanjskim servisom mogli smo primijetiti da nigdje nismo koristili operator `subscribe()`. U prethodnim poglavljima izričito se naglasilo da kreirani odnosno definirani tok ne počinje s emitiranjem vrijednosti sve dokle god se netko ne pretplati korištenjem metode `subscribe()`. Međutim, ako pokrenemo aplikaciju, aplikacija će se uredno pokrenuti i ako se pokušamo registrirati, registracija će uspješno proći. Razlog zašto se ovaj kôd izvršava leži u tome što se akcija pretplate doista događa iako ju mu nismo eksplicitno inicirali. Ovisno koji se server koristi, i da li podržava reaktivne tokove, ova pretplata se može inicirati od strane samog servera dok se kod ostalih servlet baziranih servera pretplata inicira u internim klasama Spring okvira konkretno u klasi `ServletHttpHandlerAdapter` u metodi `service(ServletRequest, ServletResponse)` koja predstavlja most između

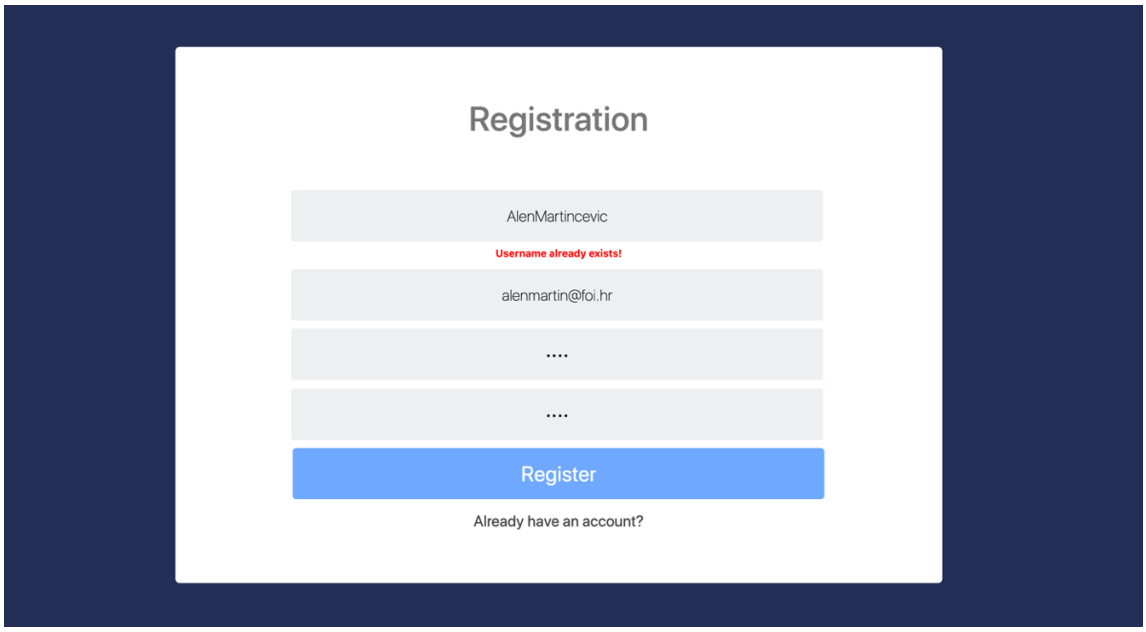
reaktivnih tokova i Servleta 3.1. Stoga u većini slučajeva eksplicitno pozivanje `subscribe` operatora nije potrebno jer to za nas u pozadini radi Spring okvir.

```
@Service
public class UserRegistrationServiceImpl implements UserRegistrationService {
    private final UserRepository userRepository;
    private final ChatHelper chatHelper;

    public UserRegistrationServiceImpl(
            UserRepository userRepository,
            ChatHelper chatHelper) {
        this.userRepository = userRepository;
        this.chatHelper = chatHelper;
    }

    @Override
    public Mono<User> registerNewUserAccount(UserDto userDto) {
        Mono<Long> lastInsertedUser = userRepository
                .findTopByOrderByIdDesc()
                .map(User::getId)
                .switchIfEmpty(Mono.just(0L))
                .subscribeOn(Schedulers.parallel());

        Mono<User> createNewUser =
                userRepository.existsByUsername(userDto.getUsername())
                .flatMap(exists -> {
                    if(!exists){
                        return chatHelper
                                .createUserChatStore(userDto.getUsername())
                                .then(Mono.just(createNewUser(userDto)));
                    }else{
                        return Mono.empty();
                    }
                })
                .subscribeOn(Schedulers.parallel());

        return Mono.zip(createNewUser, lastInsertedUser)
                   .flatMap(tuple2 ->{
                       long newId = tuple2.getT2() + 1;
                       User newUser = tuple2.getT1();
                       newUser.setId(newId);
                       return userRepository.save(newUser);
        });
    }

    private User createNewUser(UserDto userDto){
        return new User(userDto.getEmail(),
                userDto.getUsername(),
                userDto.getPassword());
    }
}
```

Kôd 42: Implementacija `UserRegistrationService` sučelja

Na sljedećoj slici možemo vidjeti stranicu registracije koja sadrži jednostavnu formu koja traži od korisnika unos korisničkog imena, email adrese i lozinke. Nakon što su podaci uneseni, radi se provjera da li korisnik već postoji u sustavu, u slučaju da ne postoji kreira se

novi korisnik i radi se redirekcija na glavnu stranicu aplikacije. U slučaju da već postoji korisniku se ispisuje poruka pogreške te se onemogućuje gumb za registraciju.



Slika 44: Forma registracije s ispisom poruke o postojanju korisnika

6.3.2. Ažuriranje prikaza na reaktivan način

Prije nego li krenemo na objašnjavanje implementacije ove funkcionalnosti, prvo moramo objasniti što je to Thymeleaf[9] te za čega ga koristimo u projektu. Thymeleaf predstavlja moderni Java poslužiteljski orijentirani mehanizam za generiranje XML/XHTML/HTML5 predložaka za internet preglednike, a koji je moguće koristiti u web i ne-web okruženjima. Kao primjer sličnih mehanizama možemo spomenuti JSP (eng. *Java Server Pages*), Facelets, FreeMaker i još mnogo drugih. Razlog zašto se koristi Thymeleaf je zato što Thymeleaf nudi najužu integraciju sa samim Spring okvirom što se može vidjeti iz činjenice da nudi na raspolaganje poveći skup Spring integracija za korištenje. Većina ovih integracija koristi se kod izrade Spring MVC aplikacija te za zadovoljavanje sigurnosnog aspekta i korištenja mogućnosti Spring Security projekta. Međutim, Thymeleaf nudi i jednu veoma zanimljivu integraciju za Spring WebFlux okvir, a ona se ogleda u obliku `ReactiveDataDriverContextVariable` klase koja dolazi kao dio thymleaf paketa za webflux `org.thymeleaf.spring5.context.webflux`. Reaktivni tok podataka obavijen ovom klasom poprimit će oblik implementacije `Publisher` sučelja kao što su `Flux` i `Mono`. Osim `Flux` i `Mono` tipova podržan je i rad s ostalim reaktivnim artefaktima zahvaljujući `ReactiveAdapterRegistry` Spring mehanizmu. Kao što možemo vidjeti primjenom Thymeleaf-a i reaktivne integracije nudi nam se mogućnost da na reaktivan način spojimo

[9] Moderni poslužiteljski mehanizam za generiranje Java predložaka (https://www.thymeleaf.org/).

poslužiteljski i korisnički dio (prikaz) bez potrebe za dodanim komplikacijama kao što su korištenje zasebnog korisničkog (eng. *Frontend*) okvira ili pak potrebe za implementacijom dodatnog mehanizma kao što je rad s utičnicama (eng. *Websockets*), AJAX poziva i slično.

```
@GetMapping("/")
public Mono<String> index(Model model) {
    IReactiveDataDriverContextVariable reactiveDataDrivenMode =
            new ReactiveDataDriverContextVariable(productHelper.getAllProducts()
                    .map(product-> new HashMap<String, Object>(){{
                        log.info(product.toString());
                        put("id", product.getId());
                        put("name", product.getName());
                        put("description", product.getDescription());
                        put("imageName", product.getImageName());
                        put("category", product.getCategory());
                        put("price", product.getPrice());
                    }}), 1);
    ...
    return Mono.just("index");
}
```

Kôd 43: Primjer korištenja `ReactiveDataDriverContextVariable`

U prethodnom programskom isječku možemo vidjeti primjenu ovog mehanizma na poslužiteljskoj strani u klasi kontrolera unutar metode ili krajnje točke (eng. *Endpoint*) kojoj se prosljeđuju `HTTP GET` zahtjevi koji su upućeni na korijenski URL aplikacije. Nakon što pristigne zahtjev i nakon što se pozove ova metoda na temelju pozadinskog mapiranja zahtjeva, kreiramo novu instancu klase `ReactiveDataDriverContextVariable` kojoj prosljeđujemo dva parametra. Prvi parametar predstavlja tok podataka kojeg u ovom slučaju dobivamo ili proizvodimo korištenjem `ProductHelper` klase i njene metode `getAllProducts()`. Metoda `productHelper.getAllProducts()` radi poziv na mikroservis Products uz pomoć korištenja reaktivnog sučelja WebClient koji nam vraća odgovor u obliku `Mono<ClienteResponse>`, a koji zatim preslikavamo u tok podataka tipa `Flux<Product>`. Nad takvim dobivenim tokom podataka primjenjujemo `map` operator koji svaki emitirani artikl transformira u novi `HashMap<String, Object>` koji će sadržavati skup mapiranih vrijednosti u kojem prva vrijednost predstavlja ključ odnosno identifikator koja će se koristiti za dohvaćanje podatka u predlošku dok druga vrijednost predstavlja konkretnu vrijednost nekog atributa artikla. Drugi parametar koji prosljeđujemo iskazan je brojčano i predstavlja veličinu međuspremnika. Ovom vrijednošću definiramo koliko će se vrijednosti emitirati odjednom. Ako se prisjetimo principa i mehanizma povratnog tlaka, ovo predstavlja jednu od mogućih strategija koju koristimo za reguliranje emitiranja, a to je strategija

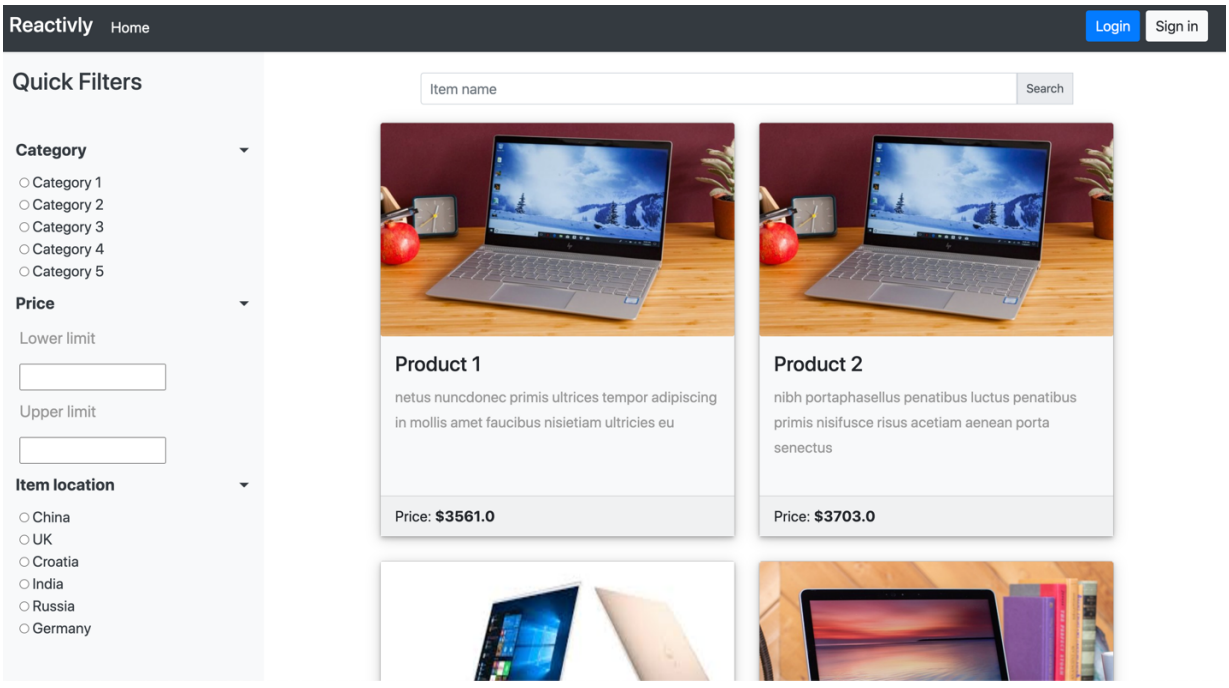
akumuliranjem putem spremnika (eng. *Buffering*). U ovom primjeru, veličina spremnika postavljena je na 1, što znači da čim se dogoditi emitiranje artikla na tok podataka isti će biti automatski proslijeđen predlošku koji će se asinkrono ažurirati s dobivenom vrijednošću odnosno vrijednosti kontekst varijable. U idućem programskom isječku možemo vidjeti dio indeks.html predloška, konkretno dio koji se odnosi na asinkrono popunjavanje putem spomenutog reaktivnog drajvera. Kao što možemo vidjeti iz HTML predloška korišteni su samo `th` modifikatori atributa za dinamično popunjavanje ili ažuriranje predloška i to su korišteni modifikatori kao što je `th:href` za generiranje poveznice, `th:src` za definiranje URL-ova koji su bazirani na protokolu, `th:text` za pretvaranja dohvaćene vrijednosti u tekst i `th:each` koji nam omogućava iteriranje kroz vrijednosti kontekst varijabli (eng. *Context Variable*). Primjena zadnjeg operatora **`th:each`** ne bi nas smjela zabuniti u mišljenju da njegovom primjenom poništavamo prednosti prethodno prikazanog reaktivnog načina. Naime, ovaj operator potrebno je koristiti jer `ReactiveDataDriverContextVariable` zapravo predstavlja reaktivnu kontekst varijablu, a ove varijable su osmišljene da sadrže više vrijednosti. Međutim to ne znači da one moraju nužno vratiti više od jedne vrijednosti, dapače mogu vratiti i samo jednu vrijednost. Nadalje, `Flux` tip smatra se događajem s više vrijednosti pa čak i ako ne emitira vrijednost ili emitira samo jednu vrijednost. S druge strane `Mono` tip se smatra događajem sa samo jednom vrijednošću.

```
<body>
<html>
…
<div id="productArea" class="row align-items-start" style="…">
    <div class="card m-3" th:each="product : ${products}" style="…">
        <a th:href="@{'/product/' + ${product.id}}"
           style="…">
            <div class="row no-gutters" style="…">
                <div class="col-md-4 my-auto">
                    <img th:src="@{'/products/' + ${product.imageName} + '/raw'}"
                         class="card-img p-2"
                         style="…">
                </div>
                <div class="col-md-8">
                    <div class="card-body" style="…">
                        <h5 class="card-title" th:text="${product.name}"></h5>
                        <p class="card-text" th:text="${product.description}"></p>
                        <p class="card-text">
                            <strong th:text="${product.price}"></strong> $
                        </p>
                    </div>
                </div>
            </div>
        </a>
    </div>
</div>
…
</body>
</html>
```

Kôd 44: HTML predložak s korištenjem Thymeleaf operatora

Koje su točno prednosti ovog načina generiranja/ažuriranja predloška. Velika prednost ovog načina leži u tome što se predložak asinkrono ažurira u realnom vremenu. To znači da nije potrebno čekati na cjelokupni odgovor nekog servisa, odgovor iz baze podataka niti samo cjelokupno generiranje predloška da bi se isti prikazao korisnicima. Već se sve to može raditi u letu što znači da kako se neka vrijednost emitira na tok, ista će biti prikazana na korisničkom sučelju. Ovime se izbjegava potreba čekanja za prikaz podataka korisnicima, kako je sve reaktivno korisnička dretva nije blokirana, drugim riječima korisnik može i dalje neometano koristiti aplikaciju, klikati na učitane elemente, navigirati na ostale dijelove stranice i slično i to već kod prvog učitavanja stranice što inače inače moguće ako se ne koriste AJAX pozivi ili neku drugi asinkroni način za dinamičko popunjavanje komponenti prikaza.

Funkcionalnost reaktivnog ažuriranja koristi se za prikaz proizvoda na glavnoj stranici aplikacije. Ovim načinom postižemo vrlo brzo učitavanje početne stranice jer kao što smo već ranije rekli nije potrebno čekati na dohvaćanje svih proizvoda iz baze podataka da bi se stranica mogla generirati sa svim potrebnim komponentama već se dohvaćanje i popunjavanje komponenti odvija asinkrono i neovisno jedno o drugom. Stoga ako odlučimo prikazati veliki broj proizvoda na početnoj stranici recimo 1000+ korisnik neće osjetiti sporije učitavanje jer će se proizvodi dohvaćati i popunjavati komponente asinkrono.



Slika 45: Glavna stranica s elementima kartice koji se generiraju na reaktivan način

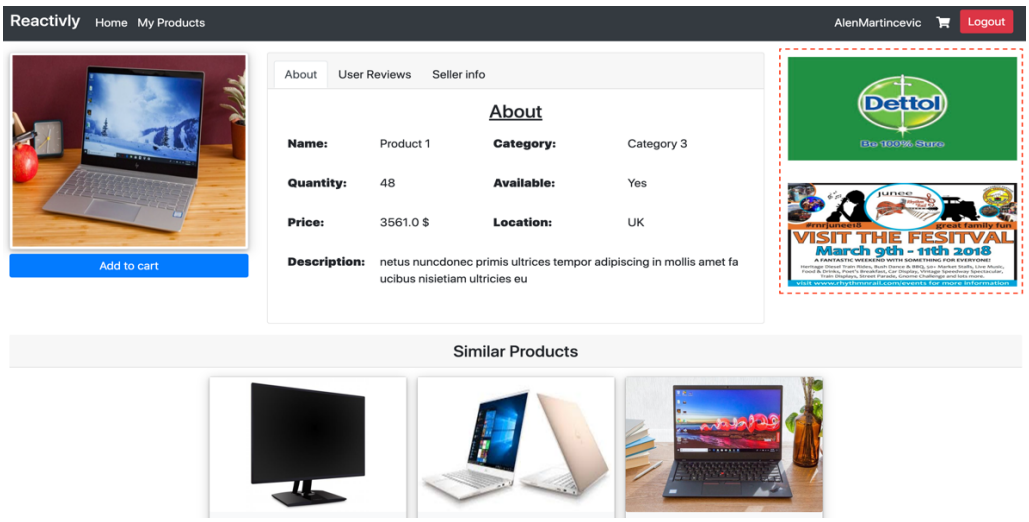
6.3.3. Funkcionalnost oglašavanja i razgovora

Iduća funkcionalnost zapravo prikazuje implementaciju i korištenje poslužiteljskih događaja ili skraćeno SSE (eng. *Server Side Events*) u Spring okviru korištenjem reaktivnog programiranja. SSE predstavlja HTTP standard koji omogućuje web stranicama rukovanje jednosmjernim tokom događaja odnosno tokom podataka gdje podatke dobiva automatski od poslužitelja čim ih poslužitelj emitira. Naglasak je na pojmu jednosmjerni tok što znači da iako se donekle slična funkcionalnost može postići korištenjem web utičnica (eng. *Web sockets*) ova dva načina ne smiju se poistovjećivati jer se razlikuju u nekoliko pojedinosti. Naime, web utičnica nije HTTP protokol, ne nudi standard za rukovanje pogreškama te nudi dvostruki smjer komunikacije (klijent -> poslužitelj, poslužitelj -> klijent) dok poslužiteljski događaji nude samo jedan smjer komunikacije (poslužitelj -> klijent).

Funkcionalnost s kojom ćemo prikazati utilizaciju reaktivnog programiranja u korištenju s poslužiteljskim događajima je funkcionalnost oglašavanja ili reklamiranja. Ideja iza funkcionalnosti je sljedeća. Na poslužiteljskoj strani u resursima imamo definiran skup slika (reklama), u ovom slučaju taj skup se sastoji od tri različite reklame. Kako bismo pojednostavili implementaciju slike su dodane kao statički resursi aplikacije što znači da same slike nisu spremljene u bazu podataka ali zato njihovi nazivi odnosno odgovarajući nazivi datoteka jesu. To znači da slike dohvaćamo iz određene datoteke poslužitelja na temelju korijenske putanje (koja je definirana u aplikaciji) plus naziva datoteke koji su spremljeni u bazi. Ove tri slike (reklame) će se periodički prikazivati na stranici detaljnih informacija o artiklu. Stoga svaki put kada korisnik otvori stranicu detalja artikla u desnom gornjem kutu prikazat će se 2 različite reklame. Od tog trenutka pa nadalje ove reklame će se izmjenjivati novima svake dvije sekunde. Time dobivamo jedan kontinuirani periodički tok promjena reklama. Način na koji ovo možemo implementirati je sljedeći. Prvo nam treba posebna REST kontroler klasa koja sadrži dvije metode `getAdvertisement()` i `getRawImage()` kao što je prikazano u kôdu 45. Prva metoda nam vraća tok podataka tipa `Flux<Tuple2<Advertisement, Advertisement>>`. To znači da će svaki emitirani podatak biti tipa `Tuple2` koji će unutar sebe sadržavati dva objekta tipa `Advertisement` koji sadržavaju vrijednosti id-a i putanju slike. Osim toga u anotaciji koja anotira ovu metodu možemo vidjeti da je vrijednost parametra `produce` definirana kao `MediaType.TEXT_EVENT_STREAM_VALUE` što znači da s ovom krajnjom točkom (eng. *Endpoint*) vraćamo tok podataka koji će sadržavati samo tekstualne vrijednosti.

Druga metoda ili druga krajnja točka ove kontroler klase vraća `Mono<ResponseEntity<?>>` te definira u anotaciji da proizvodi događaje tipa `MediaType.IMAGE_JPEG_VALUE` što označava da će ova krajnja točka vratiti resurs koji se

potom koristi za učitavanje jpeg slike na stranici. Obje ove metode koriste `AdvertisementService` sučelje odnosno njenu implementaciju ostvarenu u obliku `AdvertisementServiceImpl` klase. Implementaciju sučelja moguće je vidjeti u priloženom kôdu 46. Ova klasa nam prikazuje primjenu `Flux.zip()` operatora ali isto tako i kako kreirati novi tok podataka korištenjem operatora `Mono.fromSupplier()` na temelju proslijeđenog Supplier objekta. Prva metoda `findOneAdvertisement(filename)` za parametar prima ime datoteke koje se potom koristi za slaganje putanje do spremljene slike te se ista potom dohvaća i emitira na tok podataka. Druga metoda kreira intervalni tok podataka čije se intervalne vrijednosti koriste za određivanja koji zapis reklama treba dohvatiti iz baze podataka. Ova dva zasebna toka potom se spajaju u objekt tipa `Tuple2` korištenjem `zip` operatora te se kao takvi emitiraju. Ovime smo riješili poslužiteljski dio, međutim za cjelokupno rješenje treba nam još i implementacijski dio na korisničkoj strani. Korisnički dio implementacije realiziran je u JavaScript datoteci, a potrebnu implementaciju moguće je vidjeti u isječku kôda 47. Da bi korisnički dio mogao zaprimiti događaje emitirane od strane poslužitelja, potrebno je koristiti `EventSource` sučelje koje koristimo za povezivanje poslužiteljske i klijentske strane te za stvaranje jednosmjernog kanala od poslužitelja prema klijentu. Ovo sučelje kao parametar prima putanju do krajnje točke našeg prethodno opisanog i definiranog REST servisa. Za zaprimanje emitiranih događaja koristi se `onmessage` događaj koji poziva našu funkciju svaki put kada se dogodi emitiranje. Unutar ove metode radimo određenu transformaciju podataka na način da ekstrahiramo potrebne podatke iz zaprimljene vrijednosti koja je JSON oblika, a koji sadrži naš `Tuple2` objekt. Na temelju ekstraktiranih vrijednosti koje se sastoje od imena spremljenih slika, pozivamo drugu krajnju točku koja nam vraća te zatražene slike te na kraju ažuriramo stranicu s novo dohvaćenim slikama. Ovu funkcionalnost moguće je pronaći na stranici detalja proizvoda na krajnjoj desnoj strani u gornjem dijelu (označena crvenom isprekidanom linijom) vidljivo na sljedećoj slici.



Slika 46: Reaktivno oglašavanje

```
@RestController
public class AdvertisementController
{
    private static final String BASE_PATH = "/advertisement/";
    private static final String FILENAME = "{filename:.+}";
    private final AdvertisementService advertisementService;

    public AdvertisementController(AdvertisementService advertisementService)
    {
        this.advertisementService = advertisementService;
    }

    @GetMapping(
            value = BASE_PATH,
            produces = MediaType.TEXT_EVENT_STREAM_VALUE)
    public Flux<Tuple2<Advertisement, Advertisement>> getAdvertisement()
    {
        return advertisementService.getPeriodicallyAdvertisements(2);
    }

    @GetMapping(
            value = BASE_PATH + FILENAME + "/raw",
            produces = MediaType.IMAGE_JPEG_VALUE)
    @ResponseBody
    public Mono<ResponseEntity<?>> getRawImage(@PathVariable String filename)
    {
        return advertisementService
                .findOneAdvertisement(filename)
                .map(resource -> {
                    try
                    {
                        return ResponseEntity
                                .ok()
                                .contentLength(resource.contentLength())
                                .body(new InputStreamResource(
                                        resource.getInputStream()
                                ));
                    }
                    catch (IOException e)
                    {
                        return ResponseEntity
                                .badRequest()
                                .body("Couldn't find " + filename + " => "
                                        + e.getMessage()
                                );
                    }
                });
    }
}
```

Kôd 45: `AdvertisementController` klasa

```
@Service
public class AdvertisementServiceImpl implements AdvertisementService
{
    ...
    @Override
    public Mono<Resource> findOneAdvertisement(String filename)
    {
        return Mono.fromSupplier(() ->
                resourceLoader.getResource("file:" + UPLOAD_ROOT + "/" + filename)
        );
    }

    @Override
    public Flux<Tuple2<Advertisement, Advertisement>>
    getPeriodicallyAdvertisements(long period)
    {
        Flux<Long> fluxInterval = Flux.interval(Duration.ofSeconds(period));

        Flux<Advertisement> fluxAdvertisementPrimary =
                fluxInterval.flatMap(interval -> {
                    Mono<Long> advertisementCount =
                            advertisementRepository.count();
                    Mono<Advertisement> advertisement = advertisementCount
                            .map(count -> {
                                return (interval % count) + 1;
                            }).flatMap(advertisementId -> {
                                return advertisementRepository
                                        .findById(advertisementId);
                            });
                    return advertisement;
                });
        Flux<Advertisement> fluxAdvertisementSecondary =
                fluxInterval.flatMap(interval -> {
                    Mono<Long> advertisementCount =
                            advertisementRepository.count();
                    Mono<Advertisement> advertisement = advertisementCount
                            .map(count -> {
                                return ((interval % count) + 2 > count)
                                        ? 1 : (interval % count) + 2;
                            }).flatMap(advertisementId -> {
                                return advertisementRepository
                                        .findById(advertisementId);
                            });
                    return advertisement;
                });
        return Flux.zip(fluxAdvertisementPrimary, fluxAdvertisementSecondary);
    }
    ...
}
```

Kôd 46: Isječak klase `AdvertisementServiceImpl`

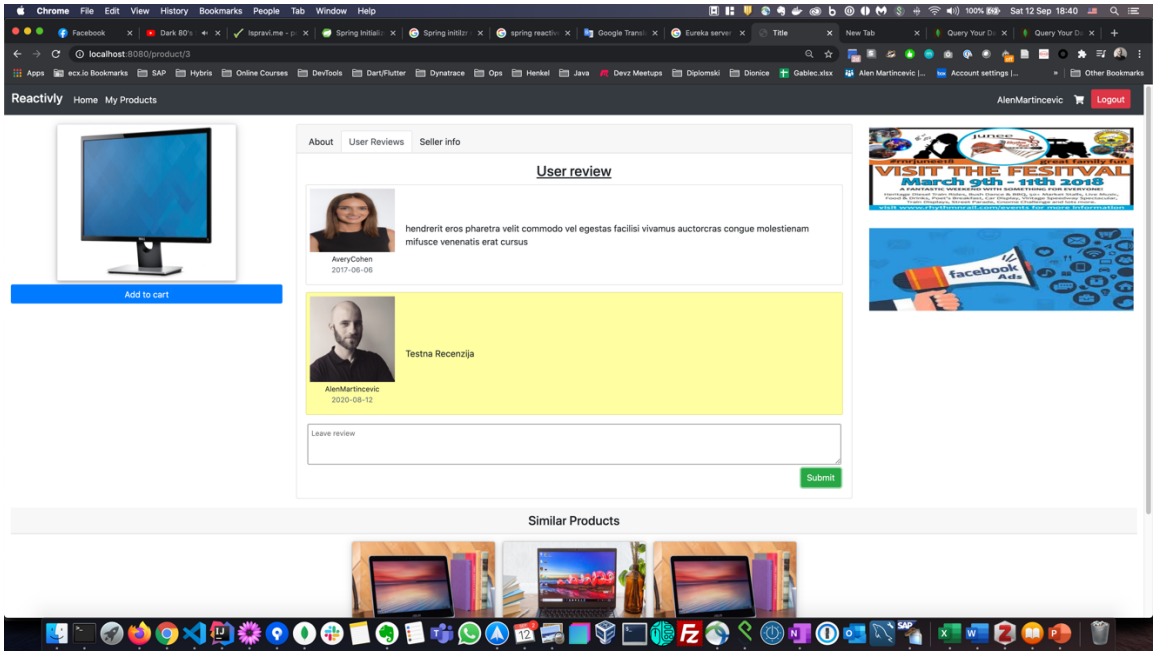
```
$(document).ready(function(){
    ...
    var source = new EventSource("http://localhost:8080/advertisement/");
    source.onmessage = function (evt) {
        var dataObj  = evt.data;
        var tupleObject = JSON.parse(dataObj);
        var imageNamePrimary = tupleObject.t1.imageName;
        var imageNameSecondary = tupleObject.t2.imageName;

        $("#advertisementImagePrimary").attr('src',  "/advertisement/"
            + imageNamePrimary + "/raw");
        $("#advertisementImageSecondary").attr('src',  "/advertisement/"
            + imageNameSecondary + "/raw");
    }
    ...
}
```

Kôd 47: Isječak JavaScript kôda za rad s poslužiteljskim događajima

6.3.4. Funkcionalnost recenziranja

Zadnju funkcionalnost koju ćemo prikazati je funkcionalnost ostavljanja recenzija. Međutim kako se ova funkcionalnost proteže kroz glavnu aplikaciju i dva mikroservisa neće se prikazati u potpunosti. Štoviše jedan dio je već prikazan i objašnjen u poglavlju Spring Cloud i Cloud Stream gdje smo opisali sve koncepte koji će se koristiti za rad s mikroservisima od kojih je jedan rad s redovima poruka za čiju implementaciju koristimo RabbitMQ. U tom poglavlju smo pokazali i neke pojedinosti kako možemo čitati ali i emitirati ili slati recenzije u izlazne kanale odnosno redove u sklopu mikroservisa Reviews. Međutim ako se prisjetimo dijagrama arhitekture aplikacije možemo vidjeti da se recenzije iz glavne aplikacije prosljeđuju mikroservisu Reviews u kojem se radi spremanje recenzija te potom njihovo prosljeđivanje drugom mikroservisu Chat koji se brine da se recenzije putem web utičnica (eng. *Sockets*) proslijede glavnoj aplikaciji kako bi se izvršilo ažuriranje klijentskog dijela aplikacije u realnom vremenu. Prvi dio komunikacije od glavne aplikacije do mikroservisa Reviews prikazan je u programskom isječku 49. U ovom isječku prikazan je primjer `ReviwController` klase koja se nalazi u glavnoj aplikaciji. U ovoj klasi možemo primijetiti sve već prethodno opisane koncepte od Spring antoacija za mapiranje zahtjeva pa do Cloud Stream anotacija za rad s redovima poruka. Kao što možemo vidjeti iz klasne anotacije `@EnableBiding(Source.class)` ova klasa predstavlja izvor poruka. Također možemo vidjeti i anotaciju `@StreamEmiter` na razini metode te anotaciju `@Output` koje se u kombinaciji koriste za slanje poruka na određeni izlazni kanal. Jedina novost koja ova klasa uvodi je primjena `FluxSink` sučelja, toplog izdavača (eng. *Hot Observable*) i strategije prelijevanja (eng. *Overflow Strategy*). U konstruktoru klase možemo vidjeti malo drugačiji način kreiranja toka podatak nego što smo to radili do sada. Vidimo da i dalje koristimo `create` operator za kreiranje toka, međutim također vidimo i primjenu dvaju novih operatora koji se

nadovezuju na operator `create()`, a to su `publish()` i `autoConnect()`. Prvi operator `publish()` omogućuje nam kreiranje toplog izdavača ili toplog toka podataka (eng. *Hot stream*) iz hladnog toka. Ovaj operator vraća `ConnectableFlux` objekt. Ako se sjetimo, topli izdavači su specifični po tome što počinju s emitiranjem podataka istog trena kada su kreirani, drugim riječima, izdavač ne čeka na pretplatu. Iz tog razloga, kako emitirane podatke ne bi izgubili čekajući na pretplatu koristimo operator `autoConnect()` koji automatski vrši pretplatu na izdavača. Nadalje, kako bi mogli povezati događaje koji su proizvedeni od korisnika, primjerice klik mišom ili u našem slučaju ostavljanja recenzije, s našim toplim tokom podataka potrebna je određena vrsta mosta. Taj most realiziran je u obliku `FluxSink` sučelja. Ova vrsta sučelja koristi se za prosljeđivanje emitiranih vrijednosti u obrnutom smjeru, s nizvodnog toka u uzvodni. Stoga, nakon što korisnik uzrokuje događaj kreiranja recenzije, na temelju mapiranja krajnjih točaka kontrolera pozvat će se `addReview()` metoda. U ovoj metodi rade se određene transformacije, odnosno spajanje recenzije s korisnikom koji ju je napisao te se uz pomoć `MessageBuilder` klase pretvaraju u objekt poruke. Naposljetku takav objekt se gura na našu instancu `FluxSink` te se korisniku vraća objekt `ResponseEntity` koji sadrži HTTP odgovor bez tijela. Onog trenutka kada se na `FluxSink` gurne emitirana vrijednost, ista se emitira na uzvodni `Flux` tok koji smo kreirali u konstruktoru klase. Ovaj tok se potom koristi u `emit()` metodi koja emitira poruke na kanal. Ovime smo uspješno postigli spajanje reaktivnog načina rada temeljenog na tokovima podataka s komuniciranjem putem poruka između aplikacija odnosno aplikacije i mikroservisa.



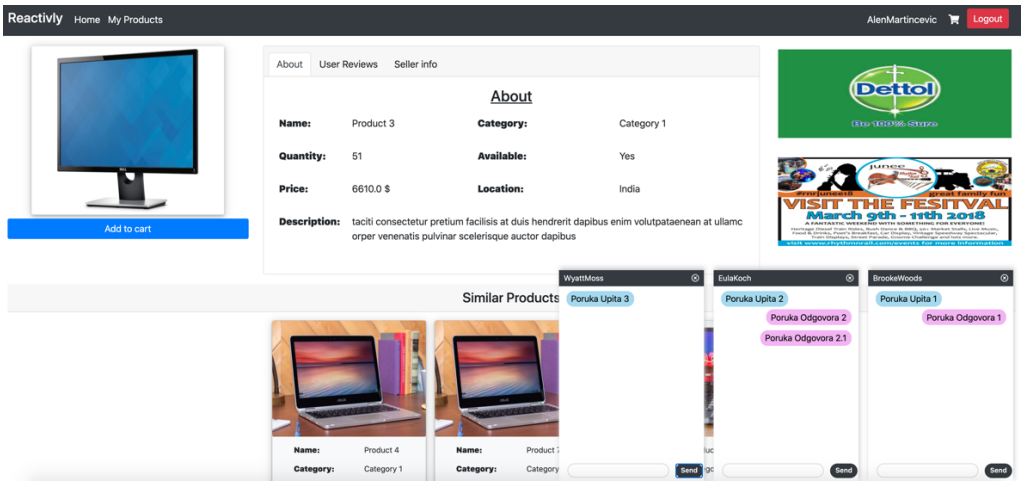
Slika 47: Funkcionalnost recenziranja proizvoda

Isti princip primijenjen je i u mikroservisu Chat koji predstavlja drugu točku u lancu razmjene poruka. Jedina različitost u odnosu na aplikaciju i mikroservis Reviews je taj što mikroservis Chat ne koristi poruke niti redove poruka za slanje podataka već se za tu svrhu primjenjuju web utičnice (eng. *Web Sockets*) vidljivo u sljede. Štoviše, korisnička strana također izgleda identično te također koristi WebSocket sučelje koje također sadrži `onmessage` događaj koji poziva našu funkciju kada god pristigne nova poruka. Primjena web utičnica na korisničkoj strani vidljiva je u sljedećem programskom isječku.

```
$(document).ready(function(){
    var socket = new WebSocket(
        'ws://localhost:8200/topic/reviews.new'
    );
    socket.onopen = function (ev) {
        console.log('--- Connected to chat service ---');
    }

    // Receive new comments from CHAT SERVICE
    // Update comment UI
    socket.onmessage = function(event){
        jsonDataObject = event.data;
        reviewData = JSON.parse(jsonDataObject);
        userId = reviewData.userId;
        date = reviewData.date;
        comment = reviewData.comment;
   }
}
```

Kôd 48: Prikaz rada s web utičnicama

Na sljedećoj slici možemo vidjeti primjer funkcionalnosti razgovora između prodavača proizvoda i kupca. Da bi korisnik mogao započeti komunikaciju s prodavačem dovoljno je otići na stranicu proizvoda i automatski će se otvoriti prozor razgovora te zainteresirani kupac može poslati poruku prodavaču. Nakon što se prodavač prijavu u aplikaciju, ako mu je netko poslao poruku, automatski će mu se otvoriti prozor s pristiglom porukom. Prodavač na tu poruku potom može poslati odgovor.



Slika 48: Prikaz funkcionalnosti razgovora

```
@RestController
@EnableBinding(Source.class)
public class ReviewController {
    private FluxSink<Message<Review>> reviewSink;
    private Flux<Message<Review>> flux;

    public ReviewController() {
        this.flux = Flux.<Message<Review>>create(
                emitter -> this.reviewSink = emitter,
                FluxSink.OverflowStrategy.IGNORE
        ).publish().autoConnect();
    }

    @PostMapping("/reviews")
    public Mono<ResponseEntity<?>> addReview(ReviewDto newReviewDto){
        if(reviewSink != null){
            return ReactiveSecurityContextHolder.getContext()
                    .map(context ->{
                        User user = (User) context
                                .getAuthentication()
                                .getPrincipal();
                        return user.getId();
                    })
                    .map(userId ->{
                        return new Review(userId, newReviewDto);
                    }).map(review -> {
                        reviewSink.next(
                            MessageBuilder
                                .withPayload(review)
                                .setHeader(
                                        MessageHeaders.CONTENT_TYPE,
                                        MediaType.APPLICATION_JSON_VALUE
                                )
                                .build());
                        return review;
                    }).flatMap(review -> {
                        return Mono.just(ResponseEntity.noContent().build());
                    });
        }else{
            return Mono.just(ResponseEntity.noContent().build());
        }
    }

    @StreamEmitter
    public void emit(@Output(Source.OUTPUT) FluxSender output){
        output.send(this.flux);
    }
}
```

Kôd 49: Primjer `ReviewController` klase kao dio glavne aplikacije

```
@Service
@EnableBinding(Sink.class)
public class CommentService implements WebSocketHandler
{
    private ObjectMapper mapper;
    private Flux<Review> flux;
    private FluxSink<Review> webSocketCommentSink;

    public CommentService(ObjectMapper mapper)
    {
        this.mapper = mapper;
        this.flux = Flux.<Review>create(
                emitter -> this.webSocketCommentSink = emitter,
                FluxSink.OverflowStrategy.IGNORE
        ).publish().autoConnect();
    }

    @StreamListener(Sink.INPUT)
    public void broadcast(Review review)
    {
        log.info("Review Recieved: " + review.toString());
        if (webSocketCommentSink != null)
        {
            log.info("Publishing " + review.toString() + " to websocket...");
            webSocketCommentSink.next(review);
        }
    }

    @Override
    public Mono<Void> handle(WebSocketSession session)
    {
        return session.send(this.flux
                .map(review -> {
                    try
                    {
                        return mapper.writeValueAsString(review);
                    }
                    catch (JsonProcessingException e)
                    {
                        throw new RuntimeException(e);
                    }
                })
                .log("encode-as-json")
                .map(session::textMessage)
                .log("wrap-as-websocket-message")
        ).log("publish-to-websocket");
    }
}
```

Kôd 50: Primjer `CommentService` klase kao dio Chat mikroservisa

U ovom poglavlju mogli smo vidjeti način na koji možemo implementirati neke od standardnih funkcionalnosti web aplikacije ali isto tako i neke malo specifičnije kod kojih nam primjena reaktivnog programiranja uvelike olakšava posao te rezultira puno boljim performansama nego što je to slučaj kod primjene standardnog blokirajućeg rješenja.

7. Zaključak

U ovom radu detaljno se obradio koncept reaktivne paradigme odnosno reaktivnog programiranja. Opisani su i detaljno objašnjeni glavni koncepti te gradivni blokovi na kojima se temelji. Nakon opisa glavnih i temeljnih koncepata navedene su i objašnjenje prednosti i nedostaci s gledišta rada aplikacije ali i samog razvoja aplikacije u globalu te u razvoju pojedinih koncepata kao što su rukovanje iznimkama i pogreškama te implementacije rada s povratnim tlakom. Pri samom kraju teorijskog opis reaktivne paradigme, opisali smo i koje posljedice ima njena primjena na dizajn arhitekture klasa, odnosno kako njena primjena utječe na strukturu i međuovisnost klasa na primjeru skupa klasa koje se mogu pronaći u svim aplikacijama e-trgovine. Na samom kraju naveli smo i opisali nekolicinu situacija u kojim nam primjena reaktivnog programiranja može olakšati posao izgradnje skalabilnog i responzivnog sustava.

U idućem poglavlju reaktivni sustavi otišli smo jednu razinu više. U njemu smo se bavili opisom i karakteristikama reaktivnih sustava danim u dokumentu naziva reaktivni manifest koji je nastao s ciljem prijedloga nove arhitekture sustava koji bi zadovoljio današnja očekivanja korisnika. Spomenuli smo da reaktivni sustavi predstavljaju složenije sustave koji se sastoje od većeg broja podsustava i/ili mikroservisa, a koji svoju međusobnu komunikaciju temelje na porukama. Obrada reaktivnih sustava bila je važna za ovaj rad iz dva razloga. Prvi razlog je bio taj što se vrlo često događa poistovjećivanje reaktivnog programiranja s konceptom reaktivnih sustava iako to dvoje ne predstavlja niti se mogu koristiti kao sinonimi. Kako bi se u budućnosti to izbjeglo detaljno je objašnjena fundamentalna razlika između ova dva pojma gdje smo mogli vidjeti da zapravo reaktivno programiranje primjenjujemo na razini programa dok koncepte reaktivnog sustava primjenjujemo na arhitekturalnoj razini. Također smo mogli vidjeti da reaktivno programiranje može biti korišteno u razvoju reaktivnog sustava s ciljem zadovoljavanja određenih karakteristika takvog sustava ali smo također mogli vidjeti i da sama primjena reaktivnog programiranja sustav ne čini reaktivnim. Nadalje, objasnili smo razliku između rada s događajima karakteristično za reaktivno programiranje i rada s porukama karakteristično za reaktivne sustave. Drugi razlog zašto smo u rad uveli koncept reaktivnih sustava je bio taj što primjena reaktivnog programiranja u izradi reaktivnih sustava donosi određene prednosti koje u određenim drugim paradigmama nisu podržane u sklopu same paradigme već se moraju naknadno implementirati te zato što se u praktičnom djelu htjelo prikazati mogućnost primjene reaktivnog programiranja u sklopu komunikacije između mikroservisa temeljenoj na porukama.

Nakon što smo obradili reaktivne sustave prešlo se na semi-praktični dio u kojem smo opisali na primjeru jednog jezika (Java) i jednog okvira/biblioteke (Reactor) primjenu reaktivnog

programiranja. U ovom poglavlju opisana su glavna sučelja, klase i operatori potrebni za rad s reaktivnim programiranjem u kontekstu kreiranja, transformacije, spajanja tokova, rukovanje greškama i upravljanje dretvama koji predstavljaju glavninu rada s reaktivnim programiranje. Za svaki kontekst dan je kratki programski primjer korištenja potrebnih sučelja, klasa i operatora.

U zadnjem dijelu rada, koji predstavlja praktični dio iznesena je ideja aplikacije koja sadrži popis, opis te slike prikaza korisničkih funkcionalnosti aplikacije. Nakon ideje, u potpoglavlju tehničkih aspekata aplikacije prikazan je dijagram sustava te komunikacija između poddijelova sustava. Spomenuti su i ukratko opisani projekti, moduli, okviri i rješenja trećih strana kao što su Spring, Spring Cloud, MongoDB, MQRabbit korištenih za razvoj i rad aplikacije. Za rad s određenim okvirima prikazani su i kratki programski isječci koji prikazuju i opisuju potrebne koncepte na razini programskog kôda. Na kraju praktičnog dijela, iz razloga što je cjelokupno rješenje preopširno, prikazane su samo određene specifične funkcionalnosti koje najviše imaju prednosti od korištenja reaktivnog programiranja. To su bile sljedeće funkcionalnosti: registracija, recenzije, reklamiranje i ažuriranje prikaza na reaktivan način.

Cilj ovog rada bio je dati teorijsku i praktičnu pozadinu reaktivnog programiranja ali isto tako i prikazati prednosti i nedostatke njenog korištenja te pokazati kakav utjecaj njena primjena ima na arhitekturu samog sustava u kontekstu strukture klasa ali i samih dijelova sustava koji mogu biti mikroservisi, baze podataka, poslužitelj i slično. Kao što smo mogli vidjeti reaktivno programiranje ima daleko bolje performanse od imperativne paradigme što se najbolje moglo vidjeti u poglavlju *3.2.2 Bolje performanse* u kojem smo na temelju testnih primjera opterećenja sustava prikazali razliku u performansama između Spring MVC okvira koji je po prirodi blokirajući i Spring WebFlux koji u pozadini koristi reaktivno programiranje, a koje predstavlja neblokirajuće rješenje. Međutim, ako želimo iskoristiti svu moć reaktivnog programiranja i doista napraviti potpuno neblokirajuće rješenje, moramo nužno koncipirati sustav na takav način da svi njegovi dijelovi podržavaju rad s događajima. To znači da je potrebno koristiti posebne okvire za razvoj same aplikacije, posebne reaktivne drajvere za komuniciranje s bazom podataka te ako se radi o web aplikaciji posebnu vrstu poslužitelja koji također podržava i utilizira rad s događajima, primjer jednog takvog poslužitelja je Netty server koji radi na principu petlje događaja. Međutim, iako reaktivno programiranje posjeduje određene prednosti i pokazuje bolje performanse u radu za situacije kada se sustav nalazi pod opterećenjem i dalje primjena reaktivnog programiranja ne predstavlja srebrni metak. Kao i svako drugo rješenje, ono nije primjenjivo za sve situacije stoga pravilan odabir i korištenje određenog rješenja i dalje ostaje na samom čovjeku te na njegovoj sposobnosti da izabere, a zatim izabrano rješenje ispravno i što bolje implementira te prilagodi određenoj situaciji.

Popis literature

[1] *AMQP Testing | ReadyAPI Documentation*. (2020). https://support.smartbear.com/readyapi/docs/testing/amqp.html

[2] Ari, D. (2019, rujan 2). *What are operators? — Learn Reactive Programming Series*. Medium. https://medium.com/@dleroari/what-are-operators-learn-reactive-programming-series-254d2f8d5358

[3] baeldung. (2017, veljača 1). *Guide to java.util.concurrent.Future*. Baeldung. https://www.baeldung.com/java-future

[4] Bagwala, M. (2019, lipanj 10). *Reactive Programming: A Step Ahead of Functional Programming – Webonise*. https://www.webonise.com/reactive-programming-a-step-ahead-of-functional-programming/

[5] Chand, S. (2017, lipanj 9). Spring Tutorial | Getting Started With Spring Framework. *Edureka*. https://www.edureka.co/blog/spring-tutorial/

[6] Christensen, B., & Nurkiewicz, T. (2016). *Reactive Programming with RxJava_ Creating Asynchronous, Event-Based Applications*. O'Reilly Media. str 726-761

[7] Dmytro Melnychuk. (2019, listopad 27). *What are the benefits of reactive programming in java* [Technology]. https://www.slideshare.net/DmytroMelnychuk/what-are-the-benefits-of-reactive-programming-in-java

[8] Dokuka, O., & Lozynskyi, I. (2018). *Hands-On Reactive Programming in Spring 5*. Packt Publishing. https://learning.oreilly.com/library/view/hands-on-reactive-programming/9781787284951/

[9] Dutta, P. (2016, kolovoz 2). *Quick Guide to Spring Controllers*. Baeldung. https://www.baeldung.com/spring-controllers

[10] Elliott, C., & Hudak, P. (bez dat.). *Functional Reactive Animation*. 11.

[11] Escoffier, C. (2017, lipanj 30). 5 Things to Know About Reactive Programming. *Red Hat Developer*. https://developers.redhat.com/blog/2017/06/30/5-things-to-know-about-reactive-programming/

[12] Event Driven and Reactive Architecture. (2015, rujan 29). *Deepak Pol's Blog*. https://deepakpol.wordpress.com/2015/09/29/event-driven-and-reactive-architecture/

[13] Gallagher, M. (2016, studeni 28). *What is reactive programming and why should I use it?* https://www.cocoawithlove.com/blog/reactive-programming-what-and-why.html

[14] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson India Education Services, ISBN 978-93-325-5540-2 str. 281-291

[15] *Glossary—The Reactive Manifesto*. (bez dat.). Preuzeto 09. rujan 2020., od https://www.reactivemanifesto.org/glossary#Message-Driven

[16] Kambona, K., Boix, E. G., & De Meuter, W. (2013). An evaluation of reactive programming and promises for structuring collaborative web applications. *Proceedings of the 7th Workshop on Dynamic Languages and Applications - DYLA '13*, 1–9. https://doi.org/10.1145/2489798.2489802

[17] Klang, J. B., Viktor. (2016, prosinac 2). *Reactive programming vs. Reactive systems*. O'Reilly Media. https://www.oreilly.com/radar/reactive-programming-vs-reactive-systems/

[18] Klang, V., & Bonér, J. (2016, prosinac 2). *Reactive programming vs. Reactive systems*. O'Reilly Media. https://www.oreilly.com/radar/reactive-programming-vs-reactive-systems/

[19] Latcu, O. (2017, prosinac 29). *Why you should learn Reactive Programming*. Medium. https://medium.com/corebuild-software/why-you-should-learn-reactive-programming-51b6ffc31425

[20] leo. (2018, srpanj 29). *Reactive System vs Reactive Programming*. Medium. https://medium.com/@jeemyeong/reactive-system-vs-reactive-programming-f9dc6b24571f

[21] Mandar jog, T. (2017). *The behavior of the Publisher-Subscriber system—Reactive Programming With Java 9*. https://learning.oreilly.com/library/view/reactive-programming-with/9781787124233/7f79852d-98ed-4523-a44b-52dca5452c9c.xhtml

[22] outsource. (2018, rujan 26). Event Loop In Node.js|Hire Top Node.js Developers|Freelance Node.js jobs. *Hire Top Talents|Freelance Jobs|Designers,Developers,SEO Experts,Content Writers*. https://blog.outsource.com/2018/09/26/understanding-the-event-loop-in-node-js-outsource/

[23] Paul, J. (2017, lipanj 2). *What is blocking methods in Java and how do deal with it?* https://javarevisited.blogspot.com/2012/02/what-is-blocking-methods-in-java-and.html

[24] Phelps, J. (2020, siječanj 4). *Backpressure explained—The flow of data through software*. Medium. https://medium.com/@jayphelps/backpressure-explained-the-flow-of-data-through-software-2350b3e77ce7

[25] Posa, R. (2018). *Benefits of Reactive systems with RP - Scala Reactive Programming*. Packt. https://learning.oreilly.com/library/view/scala-reactive-programming/9781787288645/72001d0d-b22b-4137-a815-eafcd1a6eb0c.xhtml

[26] *Project Reactor and the Spring portfolio*. (2020). https://spring.io/reactive

[27] *Project Reactor—Documentation*. (2020). https://projectreactor.io/docs

[28] *Reactive Streams*. (2015, svibanj 15). https://www.reactive-streams.org/

[29] *ReactiveX*. (bez dat.). Preuzeto 03. srpanj 2020., od http://reactivex.io/

[30] Senanayake, K. (2019, kolovoz 24). *Error handling with reactive streams.* Medium. https://medium.com/@kalpads/error-handling-with-reactive-streams-77b6ec7231ff

[31] *Spring Cloud Stream*. (2020). https://spring.io/projects/spring-cloud-stream

[32] *Spring Data*. (2020). https://spring.io/projects/spring-data

[33] *Spring Data MongoDB - Reference Documentation*. (bez dat.). Preuzeto 11. rujan 2020., od https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/#reference

[34] *Spring Security*. (2020). https://spring.io/projects/spring-security

[35] Staltz, A. (2017, studeni 25). *Reactive Programming: Why It Matters*. https://www.youtube.com/watch?v=49dMGC1hM1o

[36] Stoyanchev, R. (2018). *Servlet and Reactive Stacks in Spring Framework 5*. InfoQ. https://www.infoq.com/articles/Servlet-and-Reactive-Stacks-Spring-Framework-5/

[37] Subramaniam, V. (2017, ožujak 13). *Reactive Programming in Java by Venkat Subramaniam*. https://www.youtube.com/watch?v=f3acAsSZPhU

[38] Syer, D. (2016, srpanj 6). *Notes on Reactive Programming Part I: The Reactive Landscape*. https://spring.io/blog/2016/06/07/notes-on-reactive-programming-part-i-the-reactive-landscape

[39] *The Reactive Manifesto*. (2014). https://www.reactivemanifesto.org/

[40] *The Spring Cloud*. (2020). https://spring.io/cloud

[41] Thompson, J. (2017, Srpanj). *What Are Reactive Streams in Java? - DZone Java*. Dzone.Com. https://dzone.com/articles/what-are-reactive-streams-in-java

[42] Trincao Cunha, G. (2018, veljača 21). *Reactive vs. Synchronous Performance Test with Spring Boot 2.0—DZone Performance*. Dzone.Com. https://dzone.com/articles/spring-boot-20-webflux-reactive-performance-test

[43] Tuđan, M. (2017). *PRIMJENA REAKTIVNOG PROGRAMIRANJA PRI RAZVOJU APLIKACIJA ZA ANDROID*. Sveučilište u Zagrebu Fakultet Organizacije i Informatike Varaždin, str. 5

[44] Webber, K., & Harrington, D. (2019, veljača 12). Reactive in practice: Concurrency, parallelism, asynchrony. *IBM Developer*. https://developer.ibm.com/technologies/java/tutorials/reactive-in-practice-4/

Popis slika

SLIKA 1: DIJAGRAM KLASA UZORKA PONAŠANJA PROMATRAČ (PREMA: GAMMA ET AL., 1994) 8

SLIKA 2: UZORKA DIZAJNA - PROMATRAČ 8

SLIKA 3: VIZUALIZACIJA TRIJU KANALA (PREMA: BAGWALA, 2019) 10

SLIKA 4: PRIKAZ RADA SUSTAVA IZDAVAČA I PRETPLATNIKA U REAKTIVNOM PROGRAMIRANJU (PREMA: MANDAR JOG, 2017) 12

SLIKA 5: VIZUALIZACIJA RADA OPERATORA (PREMA: ARI, 2019) 13

SLIKA 6: PRIKAZ TOKOVA I REAGIRANJA NA EMITIRANE DOGAĐAJE (PREMA: ESCOFFIER, 2017) 15

SLIKA 7: PRIKAZ RAZLIKE SINKRONOG I ASINKRONOG IZVRŠAVANJA S POGLEDA GLAVNE I RADNE DRETVE . 16

SLIKA 8: PRIKAZ VREMENA TRAJANJA SEKVENCIJALNOG IZVRŠAVANJA ZADATAKA (PREMA: WEBBER & HARRINGTON, 2019) 17

SLIKA 9: PRIKAZ VREMENA KONKURENTNOG IZVRŠAVANJA ZADATAKA (PREMA: WEBBER & HARRINGTON, 2019) 18

SLIKA 10: PRIKAZ BLOKIRAJUĆEG IZVRŠAVANJA S GLEDIŠTA DRETVI (PREMA: WEBBER & HARRINGTON, 2019) 18

SLIKA 11: PRIKAZ KONCEPTA PETLJE (PREMA: OUTSOURCE, 2018) 19

SLIKA 12: PRIKAZ BAZENA DRETVI TEMELJEN NA DOGAĐAJIMA (PREMA: „EVENT DRIVEN AND REACTIVE ARCHITECTURE“, 2015) 20

SLIKA 13: MODEL DRETVA PO ZAHTJEVU (PREMA: DMYTRO MELNYCHUK, 2019) 23

SLIKA 14: POJAVA SILE OTPORA U KOMUNIKACIJI IZMEĐU VIŠE POSLUŽITELJA 26

SLIKA 15: PRIKAZ RADA MEHANIZMA POVRATNOG TLAKA U REAKTIVNOM PROGRAMIRANJU (PREMA: MANDAR JOG, 2017) 27

SLIKA 16: PRIKAZ OVISNOST I MEĐUSOBNOG UTJECAJA MODULA POSLOVNE LOGIKE E-TRGOVINE (PREMA: STALTZ, 2017) 32

SLIKA 17: ČETIRI KARAKTERISTIKE REAKTIVNOG SUSTAVA I NJIHOV MEĐUODNOS (PREMA: THE REACTIVE MANIFESTO, 2014) 41

SLIKA 18: APSTRAKTNI PRIKAZ REAKTIVNOG SUSTAVA (PREMA: LEO, 2018) 44

SLIKA 19: PRIMJER VIZUALIZACIJE `FLUX` TIPA U SKLOPU MRAMORNOG DIJAGRAMA (PREMA: *PROJECT REACTOR - DOCUMENTATION*, 2020) 54

SLIKA 20: PRIMJER VIZUALIZACIJE `MONO` TIPA U SKLOPU MRAMORNOG DIJAGRAMA (PREMA: *PROJECT REACTOR - DOCUMENTATION*, 2020) 54

SLIKA 21: MRAMORNI DIJAGRAM `FLATMAP` OPERATORA (PREMA: PROJECT REACTOR - DOCUMENTATION, 2020) 65

SLIKA 22: `BUFFER` OPERATOR (PREMA: PROJECT REACTOR - DOCUMENTATION, 2020) 66

SLIKA 23: `GROUPBY` OPERATOR (PREMA: PROJECT REACTOR - DOCUMENTATION, 2020) 67

SLIKA 24: MRAMORNI DIJAGRAM `WINDOW` OPERATORA (PREMA: PROJECT REACTOR - DOCUMENTATION, 2020) 68

SLIKA 25: VIZUALIZACIJA `MERGE` OPERATORA (PREMA: PROJECT REACTOR - DOCUMENTATION, 2020) 69

SLIKA 26: VIZUALIZACIJA `ZIP` OPERATORA (PREMA: PROJECT REACTOR - DOCUMENTATION, 2020) 71

SLIKA 27: VIZUALIZACIJA `CONCAT` OPERATORA (PREMA: PROJECT REACTOR - DOCUMENTATION, 2020) 72

SLIKA 28: VIZUALIZACIJA `CONCATWITH` OPERATORA (PREMA: PROJECT REACTOR - DOCUMENTATION, 2020) 73

SLIKA 29: VIZUALIZACIJA `COMBINELATEST` OPERATORA (PREMA: PROJECT REACTOR - DOCUMENTATION, 2020)..... 74

SLIKA 30: PRIKAZ DOSTUPNIH ARTIKALA 82

SLIKA 31: DETALJNI PRIKAZ INFORMACIJA O PROIZVODU..... 82

SLIKA 32: FUNKCIONALNOST DODAVANJA NOVOG PROIZVODA..... 83

SLIKA 33: FUNKCIONALNOST RECENZIRANJA ARTIKLA..... 83

SLIKA 34: PRIKAZ KOŠARICE..... 84

SLIKA 35: FUNKCIONALNOST RAZGOVORA PRODAVAČA I KUPCA..... 84

SLIKA 36: ARHITEKTURA SUSTAVA S POSEBNO OZNAČENOM GLAVNOM APLIKACIJOM (ŽUTO) I EUREKA POSLUŽITELJEM..... 85

SLIKA 37: SPRING EKOSUSTAV (IZVOR: CHAND, 2017) 87

SLIKA 38: RAZLIKA IZMEĐU REAKTIVNOG SKUPA I SERVLET SKUPA OKVIRA U SPRING EKOSUSTAVU (PREMA: *PROJECT REACTOR AND THE SPRING PORTFOLIO*, 2020) 88

SLIKA 39: PRIMJER KORIŠTENJA SPRING INITIALIZR PLATFORME 89

SLIKA 40: PRIKAZ SPRING MVC ARHITEKTURE VISOKE RAZINE (PREMA: DUTTA, 2016) 92

SLIKA 41: PRIKAZ WEBFLUX ARHITEKTURE VISOKE RAZINE (PREMA: STOYANCHEV, 2018)..... 93

SLIKA 42: SPRING CLOUD OKVIRNA ARHITEKTURA (IZVOR: THE SPRING CLOUD, BEZ DAT.) 96

SLIKA 43: KLIJENT-SERVER UZORAK PORUKA (IZVOR: AMQP TESTING | READYAPI DOCUMENTATION, 2020)98

SLIKA 44: FORMA REGISTRACIJE S ISPISOM PORUKE O POSTOJANJU KORISNIKA.....107

SLIKA 45: GLAVNA STRANICA S ELEMENTIMA KARTICE KOJI SE GENERIRAJU NA REAKTIVAN NAČIN.....110

SLIKA 46: REAKTIVNO OGLAŠAVANJE 112

SLIKA 47: FUNKCIONALNOST RECENZIRANJA PROIZVODA 116

SLIKA 48: PRIKAZ FUNKCIONALNOSTI RAZGOVORA.....117

Popis tablica

TABLICA 1: PASIVNO RJEŠENJE NASPRAM REAKTIVNOG (PREMA: STALTZ, 2017)..... 35
TABLICA 2: PRIKAZ SMJEŠTAJA APSTRAKCIJE `OBSERVABLE`..... 51
TABLICA 3: PRIKAZ OVISNOSTI I ARTEFAKATA KORIŠTENIH U POJEDINOM PROJEKTU 90