SVEUČILIŠTE U MOSTARU
FAKULTET STROJARSTVA, RAČUNARSTVA I ELEKTROTEHNIKE

DIPLOMSKI RAD

RAZVOJ APLIKACIJE ZA PRIKUPLJANJE I SLANJE PODATAKA U REALNOM VREMENU KORISTEĆI MONGO CHANGESTREAMS

Tin Tadić

Mostar, rujan 2023.

IZJAVA

Ovim putem izjavljujem da sam diplomski rad napisao samostalno, koristeći navedenu literaturu. Zahvaljujem se svom mentoru doktoru znanosti profesoru Goranu Kraljeviću i svojim mentorima i kolegama iz NSoft-a na pruženoj pomoći tijekom izrade ovog diplomskog rada. Hvala Vam na vremenu i pomoći.

Posebno bih se zahvalio svojim roditeljima i kolegama na svoj podršci koju su mi pružili kroz život i školovanje.

Izjavu dao:

Tin Tadić

FSRE | FAKULTET STROJARSTVA, RAČUNARSTVA I ELEKTROTEHNIKE

Osnovne informacije o diplomskom radu kandidata:

Vrsta rada	Diplomski rad	
Student	Tin Tadić	
Mentor	Doc. dr. sc. Goran Kraljević	
Naslov rada na hrvatskom jeziku	Razvoj aplikacije za prikupljanje i slanje podataka u realnom vremenu koristeći Mongo ChangeStreams	
Naslov rada na engleskom jeziku	Application for gathering and sending data in real-time using Mongo ChangeStreams	
Godina izdanja	2023. god.	
Izdavač publikacije	Fakultet strojarstva, računarstva i elektrotehnike Sveučilište u Mostaru	
Adresa izdavača	Matice hrvatske b.b., 88000 Mostar, Bosna i Hercegovina	
Fizički opis rada	Broj stranica:	43
	Broj poglavlja:	8
	Broj slika:	15
	Broj tablica:	-
	Broj priloga	-
	Broj citirane literature:	7
Znanstveno polje	Strojarstvo/Računarstvo/Elektrotehnika	
Smjer/usmjerenje	Programsko Inženjerstvo i Informacijski Sustavi	
Ključne riječi rada (min. 3)	ChangeStreams, SpringBoot, Data Streaming	
Mjesto arhiviranja	Knjižnica Fakulteta strojarstva, računarstva i elektrotehnike	

SVEUČILIŠTE U MOSTARU

FAKULTET STROJARSTVA, RAČUNARSTVA I ELEKTROTEHNIKE

Diplomski studij:	Računarstvo
Smjer/usmjerenje:	Programski Inženjerstvo i Informacijski Sustavi
Ime i prezime:	Tin Tadić
Broj upisnice:	602-RM

ZADATAK DIPLOMSKOG RADA

Naslov diplomskog rada:	Razvoj aplikacije za prikupljanje i slanje podataka u realnom vremenu koristeći Mongo ChangeStreams
Zadatak diplomskog rada:	U diplomskom radu potrebno je dizajnirati i razviti aplikaciju koja će podatke stream-ati iz više MongoDB baza i kolekcija, obrađivati ih, i slati na API treće strane. Potrebno je detaljno opisati sve faze razvoja navedene aplikacije.
Prijava rada:	14. 3. 2023.
Rad predan:	14. 9. 2023.

Predsjednik Povjerenstva

Mentor

Razvoj aplikacije za prikupljanje i slanje podataka u realnom vremenu koristeći Mongo ChangeStreams

Sažetak:

U ovom diplomskom radu predstavljen je razvoj aplikacije koja stream-a podatke iz više MongoDB baza i kolekcija, njihovo formatiranje, obradu, i na kraju slanje na API treće strane. Diplomski rad prati razvoj aplikacije, od odabira tehnologije do postavljanja na server. Opisani su problemi sa kojima sam se susreo i načinima na koji su riješeni.

Application for gathering and sending data in realtime using Mongo ChangeStreams (Data Stream Application)

Abstract:

This paper presents the development of an application that streams dana from multiple MongoDB collections and databases, their formatting, processing, and ultimately their sending to a third-party API. This paper tracks the entire development process, from the choice of technologies to eventual server deployment. It describes the problems I encountered, and the solutions employed.

SADRŽAJ:

1 UVOD

Svaka industrija generira podatke, i kvalitetna obrada tih podataka može dati prednost nad konkurencijom. Njihovom analizom se mogu pronaći uzorci, anomalije, zanimljivosti koje bi ljudskom mozgu promakle u moru informacija. Obrada podataka je kompliciran postupak, i ima mnoštvo pristupa koji se mogu koristiti ovisno o potrebama i traženim rezultatima. Ali prije nego što se ti podatci obrade, moraju se poslati na aplikaciju koja će vršiti obradu. Naravno, ta aplikacija može čitati podatke direktno iz operativne baze, ali to se u praksi ne radi iz više razloga. Prvi i osnovni razlog su performanse, korištenjem druge baze podataka smanjuje se operativno opterećenje glavne (aplikacijske) baze podataka u vidu smanjenja operacija čitanja i pisanja, te smanjenja same veličine baze podataka. Manja baza znači manji broj indeksa, što omogućuje brže performanse, ali također daje opciju kupovine jeftinijeg plana baze podataka za analitičku bazu. Drugi jednako važan razlog je sigurnost, što je manje podataka izloženo svijetu to smo sigurniji od potencijalnog curenja podataka.

Ovaj diplomski rad prati razvoj aplikacije koja stream-a podatke iz jedne baze, konvertira ih u format prihvatljiv drugoj bazi, te iz te druge baze ih prosljeđuje trećoj aplikaciji na obradu.

2 RAZRADA TEME (TEORIJSKI DIO)

2.1. Cilj i svrha projekta

Zadatak projekta je razviti aplikaciju koja će pratiti podatke iz više MongoDB kolekcija, i slati ih na treću aplikaciju za obradu. Svrha ovog projekta je da se ista funkcionalnost izbaci iz već postojećeg projekta koji se bliži kraju životnog ciklusa. Dio je standardne podjele monolitskog servisa na mikroservise radi lakšeg održavanja i povećanja stabilnosti. Aplikacija se treba spojiti na MongoDB klaster, i tu pratiti sve kolekcije koje imaju određena imena, bez obzira na to u kojoj se bazi podataka nalaze. Te podatke mora dalje poslati trećoj aplikaciji na obradu.

2.2. Streamanje podataka - općenito

Streamanje podataka je proces u kojem se podatci šalju sa jednog mjesta na drugo. Izvor tih podataka može biti baza podataka, internet scrapper bot, ili pak dupliciranje podataka koji se već šalju na drugo mjesto. Odredište tih podataka je najčešće druga baza podataka, ali može biti i aplikacija koja će odmah podatke obrađivati. Bez obzira na svrhu streamanja podataka, svaka vrsta implementacije streamanja podataka mora osigurati da će se podatci poslati točno onakvi kakvi su izvorno napravljeni. Podatci se također moraju slati u formatu koji je prihvatljiv odredištu, to se može izvršiti ili predobradom podataka prije slanja, ili odredište mora znati format podataka da izvrši konvertiranje podataka prije obrade. U slučaju Data Stream aplikacije, koristi se prvi pristup i podatci se prije slanja na API treće strane obrađuju u unaprijed dogovoreni format.

2.3. Planiranje projekta

Razrada plana projekta uključuje odabir tehnologija koje će se koristiti, usporedba njihovih prednosti i mana, te njihova implementacija u već korištene tehnologije unutar organizacije. Glavni zahtjev jest da novi projekat, koji je dobio ime Data Stream, bude lakši za održavanje od prethodnog projekta. Stoga će većina projekta biti implementirana u tehnologijama koje se već koriste u organizaciji, iako na prvu možda ne izgledaju kao očit izbor.
Uz te zahtjeve, dodatno su zadani kriteriji koje aplikacija mora zadovoljiti kako bi bila poboljšanje u odnosu na prethodnu funkcionalnost.

Ti zahtjevi su:

- Brzina: Aplikacija mora moći obraditi određenu količinu podataka u sekundi
- Pouzdanost: Pošto je riječ o osjetljivim podatcima, nije dozvoljeno nikakvo odstupanje od stvarnog stanja u bazi
- Skalabilnost: Aplikacija mora biti horizontalno skalabilna u slučaju da se obujam podataka poveća
- Stabilnost: Aplikacija se mora moći sama oporaviti od većine grešaka, ili u slučaju neoporavljivih grešaka mora odmah biti očito da je došlo do problema
- Data-Guarantee: Aplikacija mora garantirati da će se svaki podatak isporučiti trećoj aplikaciji barem jednom
- Podrška za database sharding - tijekom razvoja aplikacije došlo je do promjene zahtjeva i dodan je zahtjev da Data Streaming aplikacija mora podržavati MongoDB database Sharding

3 TEHNOLOGIJE

3.1. SpringBoot

SpringBoot je open-source projekat koji omogućuje jednostavniji i efikasniji razvoj web aplikacija i mikro/makro-servisnih aplikacija koristeći Spring framework. Prednosti SpringBoot-a nad Spring-om je u tome što SpringBoot podržava autokonfiguraciju, olakšava konfiguraciju gdje je potrebna, i uostalom samo nadograđuje Spring framework i čini ga jednostavnijim za uporabu. Budući da koristi javu, kod se prvo mora kompajlirati i zapakirati da bi se mogao izvršavati. Za kompajliranje projekta, njegovo testiranje i upravljanje paketima postoji par alata. Alat koji se koristi u organizaciji je Apache Maven.

Slika 1. SpringBoot logo[1]

3.2. Maven

Maven je open-source alat za upravljanje projektom, kojeg razvija Apache Group. Koristi se za automatizirano i olakšano upravljanje projektom. Neke od stvari koje Maven podržava su upravljanja build-ovima software-a, kompajliranjem i pakiranjem koda, izrada i održavanje dokumentacije, upravljanje paketima o kojima projekt ovisi (project dependencies) i njihovim verzijama, distribuiranjem tih paketa, upravljanjem testova, itd. IntelliJ dolazi sa automatskom Maven integracijom, što uvelike olakšava razvoj sa alatom.

Slika 2. Maven logo[2]

3.3. MongoDB

Operativna baza iz koje treba stream-ati je MongoDB baza podataka, samim tim baza podataka je predodređena. MongoDB je NoSQL baza podataka koja je optimizirana za velik broj podataka koji nisu međusobno ovisni. MongoDB sprema podatke u dokumente koji imaju strukturu ključ-vrijednost parova i nalikuju strukturi JSON dokumenta, ali zapravo koristi BSON - binarni JSON. BSON ima višestruke prednosti nad JSON-om i većina njih potiče od toga što je BSON enkodiran binarno: podržava više tipova podataka od JSON-a, zauzima mnogo manje memorije pa ga je lakše pohraniti i samim time mrežno prenositi. Vizualno, kod pregleda podataka, i praktično, tijekom razvoja software-a, možemo tretirati BSON kao JSON s više tipova podataka i nećemo imati problema s njegovom uporabom.
Kao i ostale NoSQL baze, MongoDB nema određenu shemu dokumenata i svaki dokument je neovisan o drugim dokumentima unutar jedne kolekcije. To naravno ne znači da će se svi podatci trpati u jednu kolekciju, nego je uglavnom slučaj da se za svaki entitet unutar aplikacije napravi posebna kolekcija. Takva struktura uvelike olakšava izvršavanje upita nad bazom, i samim time nam omogućuje praćenje točno određenih kolekcija dokumenata kako bi se projekat mogao realizirati.

Slika 3. MongoDB logo[3]

3.4. Docker

Docker je alat koji omogućuje developerima izradu, upravljanje, ažuriranje, i objavljivanje kontejnera. Unutar jednog kontejnera obično se u zapakiranom obliku nalazi aplikacija ili servis, i jedan operativni sistem. Time nam Docker omogućava laku distribuciju software-a na server-e i praktički garantira da će se kontejner koji pokrećemo na serveru ponašati isto kao i da smo ga pokrenuli na lokalnoj mašini. Docker kontejneri su minificirani kako bi zauzimali što manje prostora na disku i što manje resursa procesora i RAM-a. Time se ujedno ubrzava aplikacija i štedi na resursima. Docker nosi sa sobom i više prednosti od ovih nabrojanih, te je stoga postao industrijski standard. Nudi mnogo prednosti tijekom razvoja aplikacije jer razvijanjem aplikacije unutar Docker kontejnera ne samo da možemo vidjeti točno kako će se aplikacija ponašati na serveru kada je deploy-amo, nego lako možemo istovremeno raditi sa više verzija različitih servisa. Možemo na primjer istovremeno imati aktivnu instancu više različitih verzija neke baze podataka, što konvencionalno nije praktično. U potpunosti uklanja potrebu za lokalnom instalacijom servisa koji bi inače trošili resurse, što omogućuje laganiji i brži razvoj. Dodatna prednost Docker-a je u tome što je suradnja između developera mnogo lakša kada se preuzimanje i postavljanje projekta svodi na preuzimanje projekta sa github-a, pokretanje docker-compose komande za generiranje kontejnera, i odmah se može početi sa razvojem.

Slika 4. Docker logo[4]

3.5. Kubernetes

Kubernetes je Google-ov open-source projekt za upravljanje kontejneriziranim projektima. Glavne odlike su mu lagana skalabilnost i centralizirani dashboard za upravljanje deploy-

mentima aplikacija, ali i baza, servera, proxija... Kontejneri unutar Kubernetes klastera su slični Docker kontejnerima, ali su mnogo više povezani sa Kubernetes sustavom na kojem su deploy-ani, nazvanim namespaceom. Kubernetes automatizira deployment i rollback u slučaju propalog deploy-menta, restartira kontejner u slučaju pada, i osigurava da se kontejner ne samo podigne ispravno, nego provjerava radi li kontejner ispravno prije nego što ga označi kao ispravnim. Alat za upravljanje Kubernetes klasterom koji je korišten je Rancher. Kubernetes ne nudi usluge baze podataka, proxija, i sličnih servisa, ali omogućuje deployment tih servisa unutar Kubernetes klastera. Kubernetes također ne nudi alat za deployment koda i aplikacija, nego se za to moraju koristiti drugi alati. Alat korišten za to je Jenkins.

Slika 5. Kubernetes logo[5]

3.6. Jenkins

Jenkins je open-source alat koji se zasniva na ideji Kontinuirane Integracije/Kontinuirane Dostave - Continuous Integration/Continuous Delivery (CI/CD). To je praksa razvoja koja se temelji na ideji da više developera donosi izmjene na zajednički repozitorij nekada čak i više puta dnevno. Te se izmjene potom odmah build-aju i distribuiraju na servere/uređaje koji dalje deploy-aju programe. Tim se pristupom omogućuje brzo i automatizirano rješavanje problema, sa developera se skida teret build-a i distribucije koda, te je na kraju mnogo lakše upravljati sa više različitih servisa, mijenjati im verzije, itd. Jenkins podržava razne plugin-e koji dodatno ubrzavaju i olakšavaju taj postupak. Točan tijek programa se piše u takozvanom Jenkinsfile-u, koji definira operacije i redoslijed njihovog izvršavanja.

Slika 6. Jenkins logo[6]

3.7. MockServer

Pošto se podatci moraju stream-ati na API treće strane koji je još bio u izradi za vrijeme razvoja Dana Stream aplikacije, testiranje tog dijela aplikacije je otežano, čak i nemoguće testirati sa stvarnim API-jem dok nam treća strana to ne omogući. Naravno, priložena je dokumentacija dijela API-ja na koji će se spajati Data Stream aplikacija u obliku OAS-a, tj. OpenApi Specifikacije. To je autogenerirani JSON dokument koji se koristi za brzo dokumentiranje krajnjih točaka nekog API-ja. Ukoliko se Data Streaming aplikacija razvije po uzoru na taj dokument, onda bi ne bi trebalo biti problema. Postupak si možemo dodatno olakšati time što po uzoru na OAS konfiguriramo instancu mock server-a koristeći istoimeni MockServer servis. Time možemo testirati kako će se aplikacija ponašati kada se spoji na API treće strane. Samim činom prepisivanja OAS-a u mock server konfiguraciju možemo pronaći nejasnoće ili potencijalne probleme u dokumentaciji. Nakon što je mock server u pogonu možemo s lakoćom simulirati odgovore na sve zahtjeve koje Data Stream aplikacija šalje prema API-ju treće strane, i time simulirati ponašanje Data Stream aplikacije u slučaju grešaka ili neočekivanog ponašanja aplikacije treće strane. Još jedna prednost mock server-a je mogućnost hvatanja i analize zahtjeva koji se šalju prema API-ju treće strane bez potrebe za postavljanjem i korištenjem drugih servisa kao network proxy-ja, što olakšava rješavanje potencijalnih problema pri integraciji sa API-jem treće strane. Dodatna je prednost to što se MockServer može ubaciti u docker-compose fajl i tako njime upravljati.

Slika 7. MockServer logo[7]

3.8. Transcrypt

Već dulje vremena git se koristi kao industrijski standard za verzioniranje koda. Omogućuje nam lagano i centralizirano spašavanje verzija koda, te olakšava suradnju između developera. Jedan od problema koji proizlazi iz te centralizacije koda jeste sigurnost osjetljivih podataka. To mogu biti kredencijali za pristup bazi, API ključevi, ili pak osjetljivi algoritmi. Njihovim curenjem u javnost, bilo slučajno ili namjerno, bi se ugrozila ispravnost aplikacije, ili bi čak moglo doći do downtime-a ili druge financijske štete. Stoga se ti podatci enkriptiraju na razne načine. Transcrypt koristi git-ovu ugrađenu funkcionalnost kako bi enkriptirao fajlove koristeći šifru koju unese korisnik. Ti fajlovi se potom neenkriptirani čuvaju na mašini i moguće ih je pregledati i mijenjati po potrebi. Transcrypt vodi brigu o tome da ih ponovno enkriptira kada ih push-amo na git repozitorij. Jedina stvar o kojoj developer treba razmišljati je da će te iste podatke trebati nekako dekriptirati kada dođe na red deployment na server, ali to ne predstavlja velik problem.

4 RAZRADA PROJEKTA

4.1. Rješavanje ograničenja

Projekat kao takav ima par većih ograničenja koja se moraju zaobići kako bi se ispunili uvjeti, ranije su spomenuta u poglavlju 2.3 i sada ću objasniti kako su ta ograničenja riješena. Nakon toga ću se dotaći glavnih zanimljivosti i problema sa kojima sam se susreo radeći na aplikaciji.

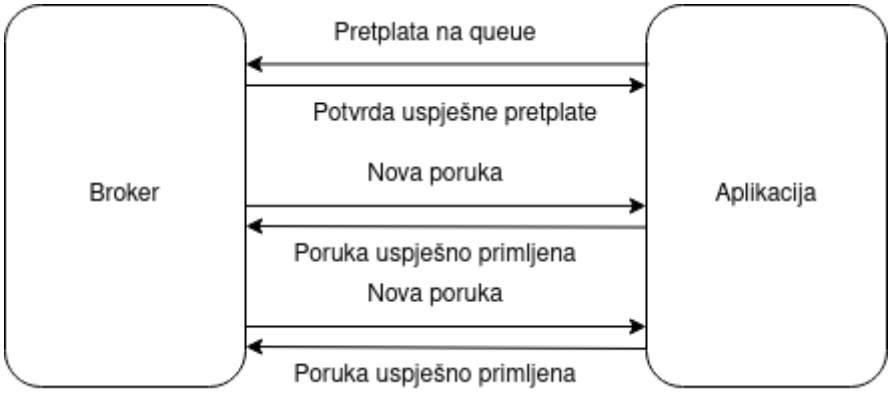
4.1.1. Skalabilnost

Kada govorimo o skalabilnosti moramo prvo spomenuti da razlikujemo dva načina skaliranja aplikacije. Vertikalno, koje postižemo povećanjem resursa na serveru (poput procesora, RAM-a, prostora na disku), i horizontalno, koje postižemo povećanjem broja aplikacija. Vertikalno skaliranje je gotovo uvijek moguće odraditi bez poteškoća, dok mogućnost horizontalnog skaliranja u potpunosti ovisi o vrsti aplikacije i njenoj realizaciji. Neke se aplikacije ne mogu uopće horizontalno skalirati, dok se druge mogu, ali se moraju kao takve od početka razvoja praviti. Data Stream aplikacija sadržava 2 submodul-a, Streamer i Consumer. Streamer submodul je nemoguće horizontalno skalirati jer on replicira oplog baze na koju je spojen, i na osnovu tih repliciranih podataka generira podatke. Ukoliko pokušamo sa više aplikacija replicirati podatke iz oplog-a i spremati ih u istu bazu, onda će doći do dupliciranja i gubljenja podataka. Consumer submodul je moguće horizontalno skalirati jer je njegova zadaća da preuzima poruka sa RMQ queue-a koje šalje Streamer submodul. Pošto RMQ garantira da će se jedna poruka moći preuzeti sa queue-a samo jednom, možemo bez dodatnih problema deploy-ati dodatne instance Consumer submodul-a ukoliko nam količina poruka na RMQ queue-u bude prevelika za jednu instancu Consumer aplikacije.

4.1.2. Ograničenje brzine na API-ju treće strane

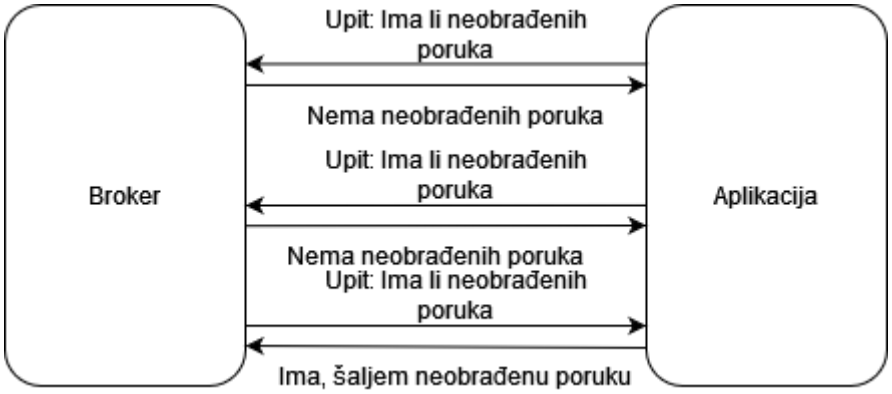
API treće strane na koji se podatci moraju slati na obradu ima ograničenje brzine i protoka podataka koje može obraditi. Završna verzija API-ja može obraditi 40 zahtjeva po minuti, sa time da shvati zahtjev sadrži skup od 100 poruka. U početku razvoja Data Stream aplikacije ovaj broj mogućih zahtjeva po minuti i broj poruka u skupu je bio mnogo manji, ali je kroz razvoj više puta povećan. Pošto nije moguće skalirati Streamer submodul, nema smisla da slanje poruka na RMQ bude pretjerano ograničeno. Stoga se ograničava brzina preuzimanja poruka na strani Consumer-a. U pravilu proces koji se spaja na RMQ queue prvo se poveže, i potom čeka da mu queue javi da ima nepreuzetih poruka, prikazano na

slici 8. Ovo smanjuje broj mrežnih poziva i ubrzava komunikaciju između RMQ queue-a i procesa koji consume-a te poruke.



Slika 8. Princip rada standardne RMQ pretplate na queue

Takav princip rada nažalost nije moguće izvesti u sklopu Data Stream aplikacije zbog prethodno spomenutog ograničenja broja API poziva na API-ju treće strane. Stoga se implementira takozvani polling consumer. On ne drži konstanto otvorenu konekciju na RMQ queue, nego se pri izvršavanju funkcije izvrši jedan poziv na RMQ koji od queue-a traži jednu poruku i onda dalje rezultat šalje na obradu, prikazano na slici 9.



Slika 9. Princip rada polling RMQ consumer-a

Slijedi kod zadužen za preuzimanje poruka sa queue-a. Funkcija je asinkrona jer se poziva svako 1,5 sekundi, time je osigurano da se neće blokirati završavanje glavnog programa u slučaju da njeno izvršavanje traje dulje. Varijabla *template* je u ovom slučaju autokonfigurirani RMQ konekcijski servis, preko kojeg se poziva metoda *receiveAndConvert*, čiji je jedini parametar naziv queue-a iz kojeg se poruke trebaju preuzeti. Odmah se konvertira u Java objekat, u ovom slučaju *TypeReference* se odnosi na tip podataka unutar List-e. Pošto postoji mogućnost da u queue-u nema poruka, onda se

mora zaustaviti daljnje izvršavanje logike da se uštedi na resursima, jer nema smisla obrađivati i slati batch unutar kojeg nema poruka.

```
@Async("pollingConsumerTaskExecutor")
public void consumeSingleMessage() {
      List<CustomMessage> messages =
mapper.convertValue((template.receiveAndConvert(queueName)), new
TypeReference() {});
      if (messages != null) {
            gateway.sendMessage(messages);
      }
}
```

4.1.3. Database Sharding

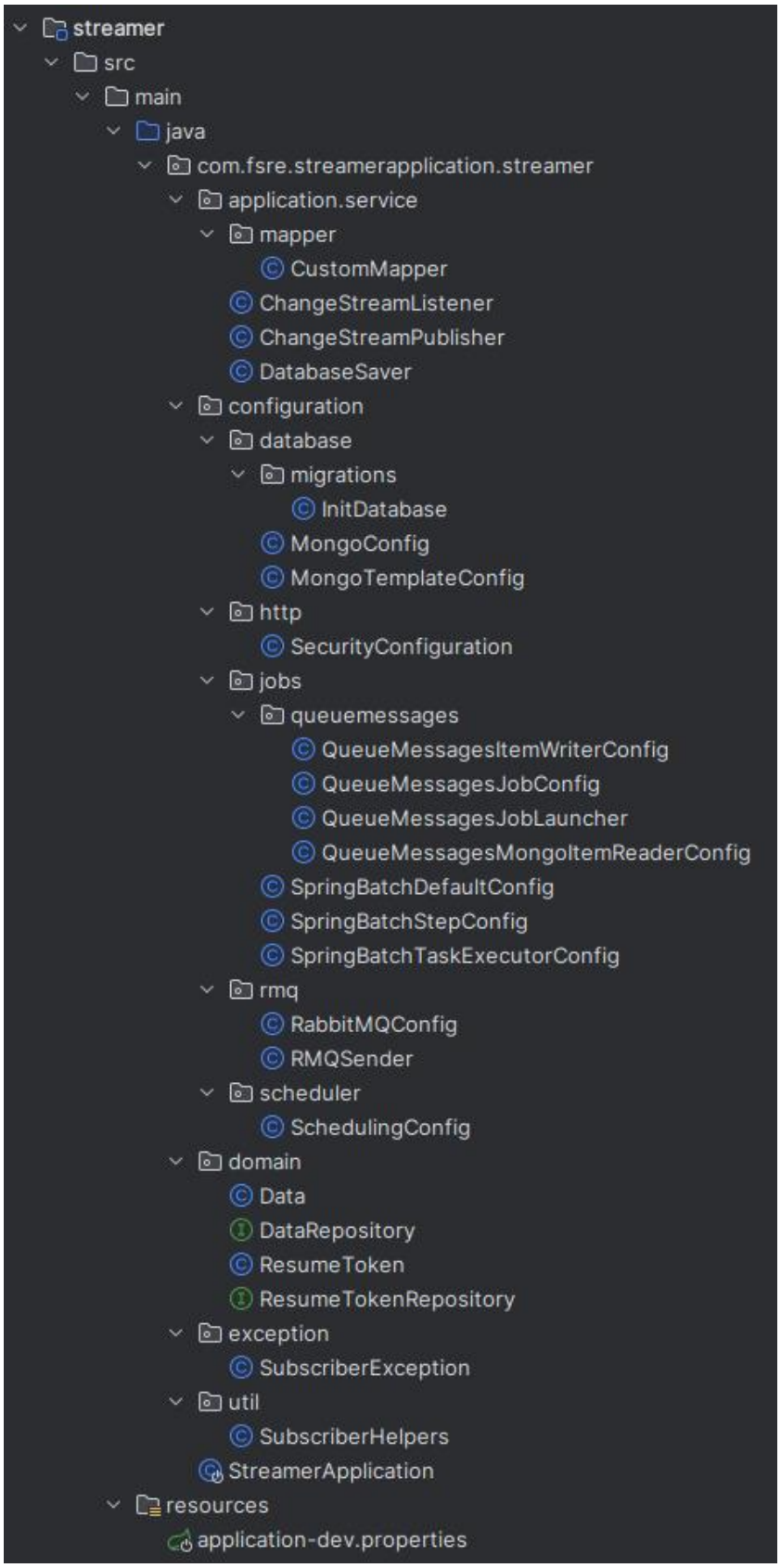
Pošto MongoDB podržava Database Sharding, aplikacija mora uzeti u obzir mogućnost da će se baza na koju se spaja shard-irati. U praksi to znači da se pozornost mora obratiti na to da se kontrolne varijable ne vežu za jednu bazu, nego da budu spremljene unutar aplikacije. Najjednostavniji način za postizanje toga jeste da se za svaki shard baze deploy-a Data Stream aplikacija. Time se postiže dijeljenje odgovornosti nad jednim shard-om baze isključivo jednoj instanci Data Stream aplikacije i ne gubi se mogućnost horizontalnog skaliranja Consumer submodula u slučaju da jedan shard bude imao veći promet podataka od drugog.

4.2. Streamer submodul

Streamer submodul ima dvije zadaće:

- da replicira podatke iz oplog-a baze, mapira ih u dokumente u sprema jednu kolekciju
- da čita podatke iz te kolekcije, mapira ih u dokumente koji su prihvatljivi API-ju treće strane, i šalje na RMQ kako bi ih Consumer submodul obradio i poslao na taj API

U ovom poglavlju ću se dotaći općenitog toka podataka unutar Streamer submodula te predstaviti par problema sa kojim sam se susreo i kako sam ih riješio.



Slika 10. Struktura Streamer submodula

4.2.1. Mongo ChangeStreams

Mongo ChangeStreams je stream podataka baze u realnom vremenu, koji teče iz baze u aplikaciju. Omogućava nam odabir kolekcija i baza koje će aplikacija pratiti, i kako će na te promjene reagirati. ChangeStreams ne čita promjene iz oplog-a baze, nego iskorištava MongoDB-ovu mogućnost replikacije podataka da te podatke šalje direktno u ChangeStreams stream podataka, koji ih dalje prosljeđuje. To sa sobom nosi mnoštvo performantskih i sigurnosnih prednosti.

Podatak se pri upisu u bazu istovremeno šalje na ChangeStreams stream podataka. ChangeStreams provjerava postoji li pretplata na tu kolekciju ili bazu, i onda dalje šalje podatak na obradu. Data Stream aplikacija preuzima taj podatak iz stream-a i potom ga mapira u objekt koji odgovara modelu kojeg će Data Stream aplikacija spremati za daljnju uporabu. Time nastali Objekt se sprema u Data Stream bazu, i nakon toga se resume token sprema u zasebnu kolekciju. Resume token se sprema sa svakim podatkom u slučaju da dođe do pada aplikacije. Ukoliko resume token postoji, onda će aplikacija pokušati nastaviti replicirati podatke iz oplog-a od tog resume token-a. Time je osiguran integritet podataka i aplikacija će moći nastaviti čitati podatke od onog podatka na kojem je stala, što zadovoljava jedan od funkcionalnih uvjeta koji su uvjet za prihvaćanje aplikacije.

4.2.2. Indeksi na kolekcijama

Kako bi osigurali optimalno ponašanje aplikacije i smanjili opterećenje baze, svaki upit nad bazom mora biti kvalitetno indeksiran. Bez indeksa, svaki upit bi pretražio cijelu kolekciju, što predstavlja problem kada se istovremeno vrši stotine upita nad kolekcijama sa po desetke milijuna dokumenata.
Jedna specifičnost MongoDB baze podataka jeste u tome što podržava takozvane TTL indekse (eng. time-to-live, vrijeme života). TTL indeksi su posebna vrsta indeksa koja omogućava automatsko brisanje podataka iz kolekcije nakon određenog vremena. Kada specificiramo TTL indeks moramo pratiti stroge konvencije zapisa podataka unutar tog polja kako bi MongoDB znao točno kada će izbrisati dokument iz kolekcije. Drugi način definiranja TTL indeksa je nad poljem u kojem je zapisan datum, u kojem slučaju će MongoDB izbrisati dokument nakon što taj datum prođe. TTL indeksi se primjenjuju nad cijelom kolekcijom, i u sklopu Data Stream aplikacije je definirano da se poslani i obrađeni podatci izbrišu nakon 30 dana. Time se smanjuje broj podataka u kolekciji i ostavlja sasvim dovoljno vremena za provjeru podataka ukoliko dođe do grešaka.

4.2.3. MongoDB ChangeStreams i UPDATE operacije

Zbog načina na koji radi, MongoDB ChangeStreams ima jednu kritičnu manu kod replikacije podataka. Kada se dokument prvi put spašava u bazu, spašava se sa tipom operacije INSERT. Svaki idući put kada se dokument sa tim id-om spašava u bazu, spašava se sa UPDATE operacijom. ChangeStreams je sposoban razaznati između ove dvije operacije, ali ukoliko se iz stream-a podataka iščita INSERT događaj, a taj dokument već ima UPDATE događaj, ChangeStreams će vratiti novo stanje dokumenta, a ne ono izvorno stanje koje je bilo nakon INSERT operacije. To se može dogiti jedino u slučaju kada se izvršava previše operacija nad bazom, i Data Stream aplikacija ne stigne obraditi podatak prije nego što se on promijeni. Ovo ponašanje je promijenjeno u novijim MongoDB verzijama, 6 pa na dalje, ali je verzija MongoDB-a na koju se Data Stream aplikacija spaja ispod te verzije. Uzevši u obzir ovo ograničenje, postavlja se pitanje je li isplativo podići verziju MongoDB-a na jednu koja podržava ovakvo ponašanje, ili se može problem riješiti aplikativno.
Na kraju je odlučeno da se problem može riješiti aplikativno, jer je jedino polje koje se mijenja polje „version". Njegovo početno stanje kod svake INSERT operacije jeste 0, pa se onda u slučaju INSERT događaja ono postavlja na 0. U slučaju UPDATE operacije bi se podatci trebali u potpunosti ignorirati. Blok koda koji je odgovoran za postavljanje ispravnog „version" polja je poprilično jednostavan:

```
try {
      if (document.getOperationType().equals(OperationType.INSERT))
{
            doc.put("version", 0);
      } else {
            return;
      }
} catch (NullPointerException exception) {
      logger.warn("Null pointer exception {}", document);
}
```

Ovime je osigurano da je vrijednost „version" polja uvijek 0, čime je zadovoljen uvjet točnosti podataka i nije bilo potrebe za riskantnim podizanjem verzije MongoDB-a.

4.2.4. Slanje podataka na RMQ – Spring Batch Job

Podatci iz svih kolekcija se na kraju spremaju u jedinstvenu kolekciju, i zadaća streaming dijela aplikacije je gotova. Dalje se podatci trebaju slati na RMQ, gdje će ih Consumer submodul pokupiti. Razlog ove podjele je u prethodno spomenutom skaliranju. Dio aplikacije koji podatke čita iz te jedinstvene kolekcije nije moguće skalirati, pa je stoga dio Streamer submodula.
Slanje podataka na RMQ je postignuto pomoću Spring Batch Job-a koji se pokreće pomoću SpringBoot-ove *@Scheduled* anotacije koja u određenom intervalu izvršava funkciju koja pokreće Job. Spring Batch Job se sastoji od par komponenti: Launcher, Configuration. Reader, i Writer. Launcher komponenta je zadužena za pokretanje Job-a i prosljeđivanje potencijalnih parametara Job-u. Za potrebe Data Stream aplikacije Launcher prosljeđuje vremenski interval, na osnovu kojeg će se vršiti upiti nad bazom. Job čita konfiguraciju iz Config-a, unutar kojeg su konfigurirani koraci, TaskExecutor, i tok izvršavanja Spring Batch Job-a. Reader čita podatke u ovisnosti od njegove konfiguracije, a u slučaju Data Stream aplikacije podatci se čitaju iz baze u ovisnosti od triju parametara. Prvi parametar je vremenski interval unutar kojeg su kreirani podatci, time se ograničava količina i starost podataka koji se šalju na obradu. Drugi kontrolni parametar je polje „errorInfo”, unutar kojeg Consumer submodul upisuje grešku za olakšano debugiranje u slučaju da dođe do greške pri slanju na API treće strane. Treći parametar je polje „publishedAt”, koje bilježi vrijeme slanja podatka na RMQ. Reader komponenta neće kupiti podatke koje sadrže te dva polja. Ovaj pristup također omogućava prilagodbu shard-iranom MongoDB klasteru, čime se aplikacija pridržava uvjetima. Podatci se potom šalju u Chunker, kojem je zadaća podatke podijeliti na grupe preddefinirane veličine, u slučaju Data Stream aplikacije to su grupe od po 200 dokumenata. Potom se podatci šalju na obradu u Processor komponentu, koja je zadužena za formatiranje i predobradu podataka prije slanja u Writer komponentu. Processor komponenta nije obavezna ukoliko se podatci mogu prosljeđivati direktno iz Reader u Writer komponentu. Na kraju je Writer komponenta zadužena za završnu obradu podataka i njihovo slanje u RMQ queue. Nakon što se podatci uspješno pošalju u queue u bazi im se dodaje polje „sentAt”, čime ih Reader komponenta više neće iščitavati. Time je završen tok procesa Streamer submodul-a.

Kod za JobLauncher ne spada nužno pod konfiguraciju SpringBoot Batch Job-a jer je Job-ove moguće pokrenuti preko HTTP zahtjeva, ruta, ili čak automatski sa pokretanjem aplikacije. Automatsko pokretanje je preddefinirana vrijednost i ukoliko se ne promijeni u *.properties* konfiguraciji aplikacije može doći do pokretanja Job-ova prije nego je aplikacija spremna, pa se stoga automatsko pokretanje onemogućuje konfiguracijom

spring.batch.job.enabled=false. Kod za JobLauncher je ubiti jednostavna @Scheduled funkcija koja uzima trenutno vrijeme i vrijeme od prije 24 sata, te sa tim parametrima pokreće Job. Implementirana je i @SchedulerLock anotacija koja onemogućava istovremeno izvršavanje više instanci Batch Job-ova ukoliko prvi nije završio sa obradom.

```
@Scheduled(fixedDelay = 10000, initialDelay = 10000)
@SchedulerLock(name = "Task_Send_Messages_To_Queue-
#{'${app.env}'}")
protected void runJob() {
	JobParametersBuilder jobParametersBuilder = new
JobParametersBuilder();

	// From 24 hours ago
	Calendar calFrom = Calendar.getInstance();
	calFrom.add(Calendar.HOUR, -24);
	jobParametersBuilder.addDate("from", calFrom.getTime());

	// up to now
	Calendar calTo = Calendar.getInstance();
	jobParametersBuilder.addDate("to", calTo.getTime());

	try {
		jobLauncher.run(sendMessages,
jobParametersBuilder.toJobParameters());
	} catch (JobExecutionAlreadyRunningException |
JobRestartException | JobInstanceAlreadyCompleteException |
JobParametersInvalidException e) {
		logger.error("Queue messages job error: {}",
e.getMessage());
	}
}
```

Job je definiran unutar JobConfig klase, gdje se specificiraju koraci Job-a, TaskExecutor koji će biti odgovoran za upravljanje resursima dostupnim Job-u, i konfiguracije Job-a koja određuje koji će se ItemReader i ItemWriter izvršavati unutar kojeg TaskExecutor-a. Job može imati više Step-ova, ali za potrebe Data Stream aplikacije je dovoljan jedan Step.

```
@Bean(name = "queueMessagesTaskExecutor")
public ThreadPoolTaskExecutor threadPoolTaskExecutor(){
      ThreadPoolTaskExecutor executor = new
ThreadPoolTaskExecutor();
      executor.setCorePoolSize(10);
      executor.setMaxPoolSize(10);
      executor.initialize();

      return executor;
}

@Bean(name = "queueMessagesJob")
public Job queueMessages(@Qualifier("queueMessagesStep") Step
step1) {
      return jobBuilderFactory
                  .get(JOB_NAME)
                  .start(step1)
                  .build();
}

@Bean(name = "queueMessagesStep")
public Step step1(
            @Qualifier("queueMessagesReader")
MongoItemReader<CustomMessage> queueMessagesReader,
            @Qualifier("queueMessagesWriter") ItemWriter<? super
CustomMessage> queueMessagesWriter,
            @Qualifier("queueMessagesTaskExecutor")
ThreadPoolTaskExecutor queueMeessagesTaskExecutor
) {
      return stepBuilderFactory.get("step1")
                  .allowStartIfComplete(true)
                  .<CustomMessage, CustomMessage>chunk(100)
                  .reader(queueMessagesReader)
                  .writer(queueMessagesWriter)
                  .taskExecutor(queueMeessagesTaskExecutor)
                  .throttleLimit(10)
                  .build();
```

```
}
```

Prva stvar koja se pokreće kada se Job pokrene je ItemReader, koji na osnovu parametara koji su proslijeđeni Job-u mora dohvatiti podatke za daljnju obradu. Kod za ItemReader se sastoji od definiranja kriterija po kojem će se podatci čitati iz baze podataka, specificiranjem dodatnih parametara upita nad bazom, i pripremanja tog upita za izvršavanje. ItemReader ne izvršava upit, nego ga samo priprema.

```
@Bean(name = "queueMessagesReader")
@StepScope
public MongoItemReader<CustomMessage> reader(
           @Value("#{jobParameters['from']}") Date from,
           @Value("#{jobParameters['to']}") Date to
) {
      Query query = new Query();

      query.addCriteria(Criteria
                  .where("createdAt")
                  .gte(from)
                  .lte(to)
      );
      query.addCriteria(Criteria.where("errorInfo").exists(false));
      query.addCriteria(Criteria.where("publishedAt").exists(false)
);

      query.noCursorTimeout();
      query.cursorBatchSize(1000);

      MongoItemReader<CustomMessage> reader = new
MongoItemReader<>();
      reader.setTemplate(mongoTemplate);
      reader.setMaxItemCount(1000);
      reader.setSort(Map.of("createdAt", Sort.Direction.ASC));
      reader.setTargetType(CustomMessage.class);
      reader.setSaveState(false);
      reader.setQuery(query);
```

```
      return reader;
}
```

ItemWriter se sastoji od samo jedne funkcije koja je zadužena da podatke pošalje na RMQ queue.

```
@Bean(name = "queueMessagesWriter")
public ItemWriter<CustomMessage> writer() {
      return new QueueMessagesItemWriterConfig(rmqSender);
}

@Override
public void write(List<? extends CustomMessage> messages) {
      rmqSender.sendToQueue((List<CustomMessage>) messages);
}
```

Premda se funkcija ne poziva na više mjesta, izvučena je u dodatnu klasu kako bi se očuvala jednostavnost koda i omogućilo korištenje te funkcije van konteksta ItemWriter-a u budućnosti. Funkcija prvo provjerava ima li uopće poruka unutar liste, jer se nekad znalo dogoditi da ItemWriter dobije praznu listu, što bi uzrokovalo pad aplikacije. Također se može primijetiti da se podatci iz List objekta prebacuju u Set objekt. To je zato što je kroz razvoj uočeno dupliciranje poruka unutar jednog batch-a. Problem je bio dovoljno rijedak da ga je teško debug-irati, i nije došlo do gubljenja podataka, ali su duplicirani podatci uzrokovali probleme na API-ju treće strane. Zbog vremenskih ograničenja i bezopasne prirode problema odlučeno je da se podatci prvo prebace u Set objekt, koji ne podržava duplicirane vrijednosti, i kao takav se prvo šalje na RMQ queue, i onda sprema u bazu. Spremanje u bazu se vrši nakon slanja, jer u slučaju da dođe do greške sa slanjem podataka, ItemReader će ih moći opet pročitati i poslati na slanje.

```
public void sendToQueue(List<CustomMessage> messages) {
      // Null check because BulkOps throws an error if it gets an
empty list
      if (messages.size() == 0) {
            return;
      }

      // Some batches contain duplicate messages. This is less a
proper fix, and more a workaround
```

```
      Set<CustomMessage> messageSet = new HashSet<>(messages);

      rabbitTemplate.convertAndSend(exchange, routingKey,
messageSet);
      saveMessagesAsPublished(messageSet);
}

private void saveMessagesAsPublished(Set<CustomMessage> messages)
{
      BulkOperations bulkOperations =
mongoTemplate.bulkOps(BulkOperations.BulkMode.UNORDERED,
CustomMessage.class);

      for (CustomMessage message : messages) {
            message.setAsPublished();
            bulkOperations.updateMulti(

      Query.query(Criteria.where("_id").is(message.getId())),
                        new Update().set("publishedAt",
message.getPublishedAt())
            );
      }

      bulkOperations.execute();
}
```

4.2.5. Problem sa kontrolom odabira podataka za obradu

U jednoj od iteracija razvoja, plan za odabir podataka za slanje je funkcionirao drugačije. Sa ciljem optimizacije upisa i čitanja nad bazom, korištena je samo jedna kontrolna varijabla za čitanje „processedAt". Reader komponenta je iz baze iščitavala sve dokumente koji nemaju to polje, i potom ih slala na obradu u Writer koji ih je označavao kao obrađene i spremao u istu kolekciju. Pri testiranju sa većom količinom podataka došlo je do problema. Reader komponenta bi svaki put pokupila točno pola preostalih dokumenata umjesto svih. Ukoliko je početno stanje 30 000 neobrađenih dokumenata, prva iteracija bi obradila 15 000, druga 7500, treća 3750, i tako sve dok broj neobrađenih dokumenata ne bude ispod 1 000. U tom slučaju je broj preostalih dokumenata manji od maksimalnog broja dokumenata

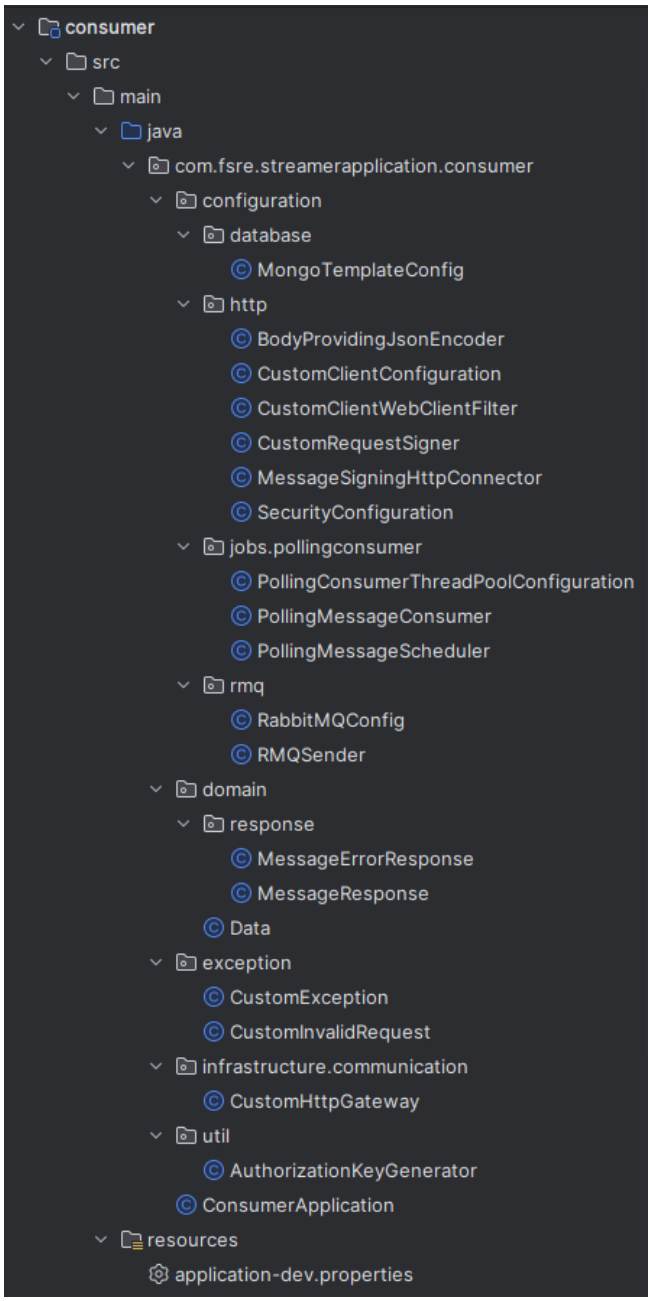
kojeg Reader može odjednom zatražiti od baze, i baza proslijedi sve preostale dokumente. Kroz dosta debugiranja, eliminirani su problemi sa memorijom i konkurentnosti kao potencijalni uzroci problema. Problem sa memorijom je elimininiran je jer bi Reader, kada se postavi da umjesto 1 000 dokumenata vrati sve dokumente, ispravno obradio ogromnu količinu podataka. Problem konkurentnosti i potencijalnih utrka uvjeta je eliminiran ograničavanjem TaskExecutor-a na samo 1 Thread. Aplikativno testiranje i čitanje izvornog koda *MongoItemReader* klase nije ukazalo na potencijalne probleme, pa sam na kraju pažnju obratio na upite koji se izvršavaju nad bazom. Upit koji logira SpringBoot-ov logger je izgledao sasvim normalno i uredno formatirano, i nije se ništa indikativno o problemu moglo iz njega iščitati. Potom, sam uključio detaljno logiranje upita na samoj bazi, i nakon detaljne pretrage logova uočio sljedeću liniju:

```
{ find: "message", filter: { processedAt: { $exists: false } },
skip: 8190, limit: 10, batchSize: 1000, noCursorTimeout: true, $db:
```

Negdje između aplikacije i MongoDB servera se upit koji od baze traži 1 000 dokumenata razdijeli na 100 upita sa limit i skip parametrima. Pošto se stanje u bazi mijenja konstanto, to jest dokumenti se istovremeno i čitaju i spremaju u kolekciju, onda se sljedeći upit nad bazom vraća različit rezultat. Samim time skip dio upita preskače neobrađene dokumente. To objašnjava zašto je aplikacija svaki put obrađivala pola od ukupnih dokumenata – za svaki pročitani i obrađeni dokument preskoče se efektivno dva. Nije bilo potrebe ni vremena da se riješi problem koji onemogućuje ovakvu implementaciju, pa je stoga odlučeno da se implementira trenutni sistem, opisan u prethodnom poglavlju 4.2.4, koji nema ovih problema i performansama odgovara potrebama Dana Stream aplikacije.

4.3. Consumer submodul

Consumer submodul ima jednu zadaću: da kupi podatke sa RMQ-a, formatira ih u oblik prihvatljiv API-ju treće strane, i pošalje ga tom API-ju. Prethodno je objašnjeno da je Consumer submodul odvojen jer je njega jedino moguće i smisleno horizontalno skalirati. Ovo poglavlje objašnjava princip implementacije i tok razvoja funkcionalnosti Consumer submodula. Pošto je princip polling RMQ queue consumer-a objašnjen u poglavlju 4.1.2, neću se dotaći te teme u ovom poglavlju.



Slika 11. Struktura Consumer submodula

4.3.1. Problem sa skaliranjem

Aplikacijski polling RMQ consumer je jednostavna funkcija koja preuzima jednu poruku sa RMQ queue-a. Pokreće se pomoću SpringBoot-ove *@Scheduled* anotacije, koja svako 1,5 sekundi preuzima jednu poruku. Ovime se ispoštuje ograničenje brzine slanja poruka na API treće strane i omogućuje horizontalno skaliranje. Jedina stvar na koju treba obratiti pozornost jeste u tome što se podizanjem dodatnih instanci Consumer-a neće promijeniti ograničenje brzine, pa će odgovor servera na svaku poruku preko tog ograničenja biti obavijest da smo prekoračili ograničenje. Stoga se prvo moraju od API-ja treće strane

nabaviti novi pristupni kredencijali kako bi API mogao razlikovati između dviju ili više instanci Consumer aplikacije.

4.3.2. OAS i MockServer

Open API Specification je JSON dokument unutar kojeg su definirane sve krajnje točke nekog API-ja. Unutar tog dokumenta su ispisane sve rute, objekti DTO-ovi, mogući odgovori i formati odgovora i njihovi pripadajući HTTP kodovi. Uz pomoć OAS-a moguće je u potpunosti razviti aplikaciju koja će se spajati na API kojeg OAS predstavlja. OAS na osnovu kojega je rađena Data Stream aplikacija se mijenjao više puta kroz razvoj aplikacije, pa se i aplikacija morala prilagođavati tim izmjenama. Dodatno, API treće strane se razvijao istovremeno sa Data Stream aplikacijom, pa stoga kroz većinu razvoja nije se bilo moguće spojiti na API i testirati ispravnost Data Stream implementacije API-ja.
Premda je svrha prikaza podataka u JSON formatu da bude lagano ljudski čitljivo, ta čitljivost se izgubi kada je OAS dug preko tisuću linija, od kojih se većina ne odnosi na rad Data Stream aplikacije. Uz pomoć MockServer-a podignut je lokalni mock server koji omogućuje testiranje implementacije API-ja u svakom trenutku razvoja. Razvoj dijela Consumer submodul-a koji je odgovoran za spajanje na API treće strane je uvelike olakšan mijenjanjem odgovora na zahtjeve aplikacije i analizom zahtjeva na MockServer dashboard-u. Čak je tijekom prepisivanja OAS specifikacije u MockServer konfiguraciju uočeno par nejasnoća I nedosljednosti u API-ju treće strane prije nego što bi se inače otkrili.

4.3.3. Implementacija Netty web klijenta

Slanje podataka prema API-ju treće strane je izvršeno implementacijom Netty WebClient library-ja. Slanje se vrši direktno iz polling RMQ consumer-a, koji nakon što preuzme poruku odmah je prosljeđuje prema servisu koji je odgovoran za slanje prema API-ju. *WebClient* je instanciran kao SpringBoot Bean sa nekim preddefiniranim postavkama. Te postavke se svode na prilagođenu implementaciju logiranja zahtjeva i odgovora, te izdvojeno umetanje autorizacijskih parametara u svaki zahtjev radi poboljšanja preglednosti koda. Dodatna prednost tog pristupa jeste u tome što se eliminira potreba za konstruiranjem i dekonstruiranjem WebClient-a pri svakom zahtjevu, čime se ubrzava rad aplikacije. Sam čin slanja koristi asinkronu komunikaciju sa serverom treće strane. Asinkrona komunikacija je neblokirajuća komunikacija, to znači da će *WebClient* poslati zahtjev i neće čekati odgovor od API-ja prije nego što omogući slanje sljedećeg zahtjeva, nego će se pretplatiti na odgovor ovog prethodnog i slušati ima li odgovora dok šalje naredne zahtjeve. Dok koristimo asinkronu implementaciju *WebClient*-a moramo voditi računa o preddefiniranim osnovnim postavkama i ograničenjima Netty *WebClient*-a. Maksimalni broj istovremenih

pretplata na odgovore kod asinkronih HTTP zahtjeva iznosi 500. Svaki dodatni zahtjev se ne izvršava odmah, nego je privremeno spremljen u interni Netty-jev queue zahtjeva na čekanju, i izvršava se tek nakon što završi jedan od trenutno aktivnih 500 zahtjeva. Ni taj queue zahtjeva na čekanju nije beskonačno velik, nego po preddefiniranim Netty postavkama iznosi 1 000 zahtjeva. Kada se taj queue napuni, svaki dodatni zahtjev će automatski biti odbijen od strane WebClient-a jer mu fali resursa. Tek nakon što jedan zahtjev pređe iz queue-a za čekanje u izvršavanje će se zahtjev ponovno moći ubaciti u queue za čekanje.

Kada se uzme u obzir ograničenje brzine slanja poruka na API-ju treće strane, ovo Netty-jevo ograničenje ne predstavlja problem u radu aplikacije, ali samo ukoliko se postavi dovoljno mala "timeout" vrijednost za svaki zahtjev.

4.3.4. Logika upravljanja odgovorima na zahtjeve

Jedna od specifičnosti API-ja treće strane jeste u tome što na svaki uspješno procesiran zahtjev vraća HTTP kod 207. Uspješno procesiran zahtjev je definiran kao onaj koji ne uzrokuje logičke greške, i čak i ako svaka poruka unutar poslanog batch-a bude odbijena, zahtjev će smatrati uspješno obrađenim i vratiti HTTP kod 207. Za bilo kakve druge greške moguće je implementirati jednostavno upravljanje, to jest u slučaju HTTP koda 500 ili timeout greške možemo odmah grešku logirati i vratiti poruku nazad u RMQ queue kako bi se podatci opet poslali na obradu. API također ima i par specifičnih grešaka za koje je izravno rečeno da se mogu ignorirati, te da bi se podatci trebali vratiti u RMQ queue, pa je i za njih potrebno implementirati posebni način upravljanja odgovorima.

Pravi izazov je implementirati upravljanje odgovorima u slučaju odgovora HTTP koda 207, koji može ali ne mora sadržavati greške. Po dokumentaciji unutar OAS-a, u odgovoru na zahtjev prema API-ju server će vratiti HTTP kod, broj poruka koje su odbačene, i listu poruka sa razlozima odbacivanja. Broj odbačenih poruka odgovara veličini te liste. Ukoliko je broj odbačenih poruka 0, onda pretpostavljamo da su sve poruke ispravno procesirane i spremamo ih u bazu sa pripadajućim “sentAt” poljem. Ovo polje je indeksirano kao TTL indeks, i 30 dana od slanja poruka će se izbrisati iz kolekcije sa ciljem očuvanja prostora. Ukoliko nam je broj odbačenih poruka jednak broju poslanih poruka, onda provjeravamo tip greške. Ako je greška već poznata kao dogovorena greška API-ja treće strane, onda ih možemo vratiti u queue za ponovno slanje. Ukoliko je greška nepoznata, onda će se svaka poruka spremiti u bazu sa pripadajućom porukom o grešci i neće se pokušati ponovno slanje tih podataka. U ovom slučaju je potrebna izravna intervencija developera kako bi se problem identificirao i greška ispravila. Ukoliko je broj odbačenih poruka manji od ukupnog broja poslanih poruka, onda se iz liste poslanih poruka moraju izdvojiti poruke koje se ne nalaze u listi odbačenih poruka, tj. Uspješno su obrađene, i kao takve spremiti kao ispravno

obrađene sa “sentAt” poljem. Potom se sa listom odbačenih poruka postupa kao u prethodnom koraku, sve se spremaju zajedno sa pripadajućom greškom i potrebna je intervencija developera. Ovakav pristup kontroli grešaka na prvu izgleda kompliciran, ali je ponašanje definirano od API-ja treće strane, pa se Data Stream aplikacija mora prilagoditi njemu.

Kod odgovoran za slanje podataka i upravljanje odgovorima sastoji se iz više komponenata, od kojih je glavni dio zadužen za postavljanje parametara, definiranje ponašanja i na kraju samo slanje poruke.

```
public void sendMessage(List<CustomMessage> messages) {
      try {
            webClient.post()
                  .contentType(MediaType.APPLICATION_JSON)
                  .header("MY-CUSTOM-HEADER-KEY", specialKey)
                  .body(BodyInserters.fromValue(messages))
                  .exchangeToMono(r ->
r.bodyToMono(MessageResponse.class))
                  .timeout(Duration.ofMillis(15000L))
                  .doOnSuccess(response -> doOnSuccess(response,
messages))
                  .doOnError(response -> doOnError(response,
messages))
                  .onErrorResume(ex -> Mono.empty())
                  .subscribe();
      } catch (WebClientResponseException we) {
            throw new CustomException(we.getMessage(),
we.getRawStatusCode());
      }
}
```

Dio koda odgovoran za upravljanje greškama je jednostavniji, pa ćemo se prvo dotaći njega. Da se smanji broj upisa nad bazom, koristit će se *BulkOperatios*, koji omogućuje izvršavanje više izmjena u jednom upitu nad bazom. Zbog kompliciranosti koda, komentari su korišteni jako često kako bi bilo lakše pratiti kod.

```
private void doOnError(Throwable response, List<CustomMessage>
messages) {

      logger.error("ERROR sending messages: {}",
response.toString());



      if (response instanceof CustomException) {

            logger.error("Received code: {}", ((CustomException)
response).getCode());

      }



      // Resend to queue in case of timeout or rate limiter error

      if (isTimeoutError(response)) {

            logger.error("Timeout or rate limiter error, requeueing
messages: {}", response.getMessage());

            rmqSender.returnToQueue(messages);

      } else if (is500Error(response)) {

            logger.error("Received 500 from destination API,
retrying send: {}", messages);

            rmqSender.returnToQueue(messages);

      } else {

            logger.error(

                        "Critical error sending messages! HttpCode:
{} Message: {} Messages: {}",

                        ((CustomException) response).getCode(),

                        response.getMessage(),

                        messages

            );

            BulkOperations bulkOperations =
mongoTemplate.bulkOps(BulkOperations.BulkMode.UNORDERED,
CustomMessage.class);



            // Prevent resending of messages for other errors
```

```
            for (CustomMessage message : messages) {
                  message.setErrorInfo(response.getMessage());
                  bulkOperations.updateMulti(

      Query.query(Criteria.where("_id").is(message.getId())),
                              new Update().set("errorInfo",
message.getErrorInfo())
                  );
            }
            bulkOperations.execute();
      }
}
```

Iz ovog dijela je izdvojeno par pomoćnih funkcija koje su veoma jednostavne, ali njihovim izdvajanjem se uvelike olakšava čitanje i razumijevanje koda.

```
private static boolean isTimeoutError(Throwable response) {
      return response instanceof TimeoutException
                  || (response instanceof CustomException &&
((CustomException) response).getCode() == 429);
}

private static boolean is500Error(Throwable response) {
      return response instanceof CustomException
                  && ((CustomException) response).getCode() == 500;
}

private boolean isErrorRetriable(MessageErrorResponse
errorResponse) {
      if ("Connection error, please
retry!".equals(errorResponse.getMessage())) {
```

```
            logger.error("Third party server connection error,
requeueing messages: {}", errorResponse.getMessage());

            return true;

      }


      return false;

}
```

Za kraj, najkompliciraniji blokovi koda, prvo dio koji upravlja odgovorima HTTP koda 207, koji sam je poprilično jednostavan ukoliko nema grešaka, ali se znatno zakomplicira ukoliko ima grešaka.

```
private void doOnSuccess(MessageResponse response,
List<CustomMessage> messages) {

      BulkOperations bulkOperations =
mongoTemplate.bulkOps(BulkOperations.BulkMode.UNORDERED,
CustomMessage.class);



      // A partial success (or even one where all messages fail)
still returns 207, so we have to check for errors

      if (response.getErrorCount() != 0) {

            logger.error("There were issues sending some messages:
{}", response);



            // Handle all messages that failed to send case by case

            List<CustomMessage> requeueableMessages = new
ArrayList<>();

            for (MessageErrorResponse returnedMessageErrorResponse
: response.getErrors()) {

                  handleFailedMessage(messages, requeueableMessages,
returnedMessageErrorResponse);

            }
```

```
            // Requeue the ones with "safe" errors
            rmqSender.returnToQueue(requeueableMessages);
      }

      // Save all successfully sent messages as such
      for (CustomMessage message : messages) {
            message.setAsSent();
            bulkOperations.updateMulti(

      Query.query(Criteria.where("_id").is(message.getId())),
                        new Update().set("sentAt",
message.getSentAt())
            );
      }

      bulkOperations.execute();
}
```

Funkcija *handleFailedMessage* koja se poziva iz programa je zadužena za provjeravanje tipa greške jer je moguće da je greška koju dobijemo u odgovoru različita za svaku poruku, i neke od tih poruka nas mogu upućivati da ih ponovno pošaljemo na obradu.

```
private void handleFailedMessage(
            List<CustomMessage> messages,
            List<CustomMessage> requeueableMessages,
            MessageErrorResponse returnedErroredMessage
) {

      logger.error(
                  "Error for message {} is {}, httpCode: {}",
```

```
                returnedErroredMessage.getId(),
                returnedErroredMessage.getMessage(),
                returnedErroredMessage.getHttpCode()
        );

        // Find the message that failed to send
        CustomMessage failedMessage = messages.stream()
                .filter(message ->
message.getId().equals(returnedErroredMessage.getId()))
                .findAny()
                .get();

        if (isErrorRetriable(returnedErroredMessage)) {
            requeueableMessages.add(failedMessage);
        } else {
            // Remove it from the list of sent messages, making
that list only contain successfully sent messages at the end of
iteration
            messages.remove(failedMessage);

            // Save failed messages to db so we don't try to resend
it

        failedMessage.setErrorInfo(returnedErroredMessage.getMessage(
));
            mongoTemplate.save(failedMessage);
        }
}
```

Detaljno ponašanje koda je objašnjeno u poglavlju broj 6, i mnogo ga je lakše pratiti i razumjeti uz grafove koji su prikazani u tom poglavlju.

5 POSTAVLJANJE APLIKACIJE NA SERVER

5.1. Preduvjeti

Prije postavljanja aplikacije na server treba se uvjeriti da aplikacija radi ispravno na lokalnoj mašini, no nakon toga šta dalje? Treba napisati skriptu koja sadrži upute kako će se projekt kompajlirati, koje će šifre koristiti za dekriptiranje podataka, na kojim će se serverima vršiti kompajliranje, gdje će se kompajlirani kod izvršavati, i tako dalje. Jedna od velikih prednosti rada u organizaciji jeste u tome što postoje različiti timovi koji su specijalizirani za pojedinačne stvari, pa tako i NSoft ima tim zadužen za upravljanje Jenkins serverom i Kubernetes klasterom. I dalje treba prilagoditi aplikaciju za postavljanje na server, pa treba kontaktirati odgovorne timove kako bi se dobili naputci za daljnji razvoj aplikacije.

5.2. Jenkins

Za Jenkins je potrebno prvo napraviti pipeline koji definira korake kroz koje će aplikacija proći prije nego što dođe do servera. Ti su koraci opisani unutar posebne skripte zvane Jenkinsfile. Ali prije nego što se može početi pisati skripta, mora se napraviti novi Jenkins Build Job na Jenkins serveru. Unutar Job konfiguracije definira se par važnih parametara:

- Onemogućuju se istovremeni build-ovi. Ovime se prekidaju build-ovi koji su u tijeku ukoliko se pokrene novi build job
- Specificira se adresa git repozitorija sa kojeg će se preuzimati kod
- Specificira se način preuzimanja koda. Za development i staging servere je moguće na server deploy-ati kod sa bilo kojeg branch-a, dok je na produkcijskom serveru moguće deploy-ati samo kod koji je povezan sa tag-om.
- Postavlja se token za pristup Rancher-u
- Definira se lokacija k8s-values fajlova

Uz pomoć ovih parametara Jenkins Build Job je konfiguriran. Sljedeći korak je pisanje Jenkinsfile-a. Jenkinsfile je u suštini skripta koja se sastoji od pomoćnih funkcija, i jedne glavne funkcije unutar koje se te pomoćne funkcije pozivaju. Glavna funkcija se zove pipeline (eng. cijev), i pomoćne funkcije se zovu steps (eng. koraci). Prva stvar koja se mora definirati unutar pipeline-a jeste timeout, i potom varijable okruženja koje su potrebne skripti za rad. Tek onda može krenuti izvršavanje glavnog dijela Jenkinsfile skripte, unutar kojeg se repozitorij prvo mora dekriptirati. Nakon dekripcije, pošto je riječ o Java projektu, program se mora kompajlirati. Testovi se izvode zasebno radi lakšeg pronalaženja potencijalnih grešaka. Ukoliko svi testovi uspješno prođu, kompajlirana aplikacija će se tag-irati i poslati

na Docker repozitorij. Nakon toga će Jenkins poslati Rancheru upute kako dalje. Rancher će povući aplikaciju i na osnovu dodatnih parametara na Kubernetes klasteru deploy-ati aplikaciju.

5.3. Kubernetes

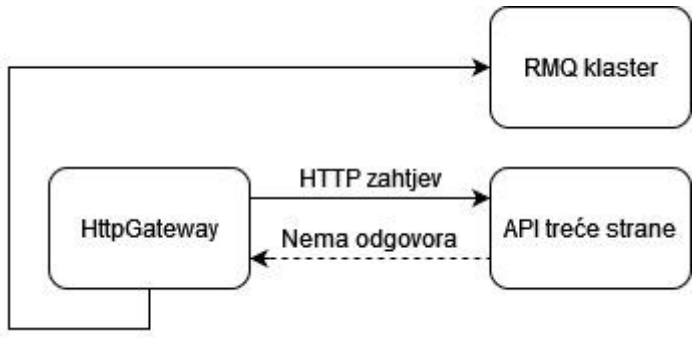
Kubernetes je više u nadležnosti tima koji njime upravlja, pa ju stoga tu bilo potrebno znatno manje posla nego na Jenkinsu. Trebalo je napraviti namespace (eng. prostor) na Kubernetes klasteru gdje će se aplikacija vrtiti. RMQ server koji Streamer i Consumer dijelovi aplikacije trebaju za rad bi bilo neisplativo podizati zasebno na Kubernetes, pa je odlučeno da se promet preusmjeri na već postojeći RMQ klaster koji je postavljen na server van Kubernetes-a. Pošto nije bilo potrebe podizati dodatne instance drugih servisa. Jedina stvar za koju nas Kubernetes obvezuje jeste postavljanje dvaju ruta: Liveness i Readiness ruta. Pošto Kubernetes drži aplikacije u kontejnerima, te rute koristi kako bi provjerio stanje kontejnera. Readiness ruta služi za provjeru statusa kontejnera, tj. je li se aplikacija uspješno pokrenula. Liveness ruta služi da Kubernetesu kaže da je aplikacija spremna za rad. Pošto je Data Stream aplikacija pisana u Spring Boot-u, obe rute se mogu istovremeno implementirati ukoliko *implementiramo spring-boot-starter-actuator* paket, koji uz te funkcionalnosti nudi dodatnu analitiku za analiziranja ponašanja aplikacije, prometa, itd. Pošto Data Stream aplikacija treba podržavati database sharding, potrebno je za svaki shard definirati vlastiti k8s-values file i postaviti ingress DNS host vrijednost.

6 DIJAGRAM TOKA ZA UPRAVLJANJE ODGOVORIMA API-JA TREĆE STRANE

Pošto je logika upravljanja odgovorima na zahtjeve koji idu na API treće strane dosta kompliciranа, slikovito objašnjenje uz pomoć grafova uvelike olakšava razumijevanje logike iza toka podataka.

6.1. Prvi slučaj: Nema odgovora

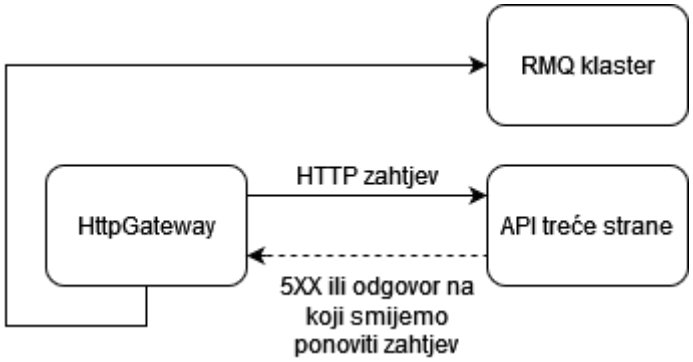
Najjednostavniji slučaj do kojeg može doći jeste da Dana Stream aplikacija uopće ne dobije odgovor od API-ja treće strane. Uzrok može biti gubitak konekcije, bilo na strani Dana Stream aplikacije ili API-ju, greške sa DNS-om, timeout-a, ili nečeg drugog. U svakom slučaju, točan uzrok nije bitan, i poruke se mogu vratiti u RMQ queue, odakle će se opet preuzeti i poslati na API. Ukoliko API nije dostupan predugo, onda će se nagomilati poruke na RMQ queue-u, na što treba obratiti pozornost. Zbog toga se pali alarm na internom alatu za monitoriranje u slučaju da broj poruka pređe 10 000.



Slika 12. Dijagram toka u slučaju da nema odgovora

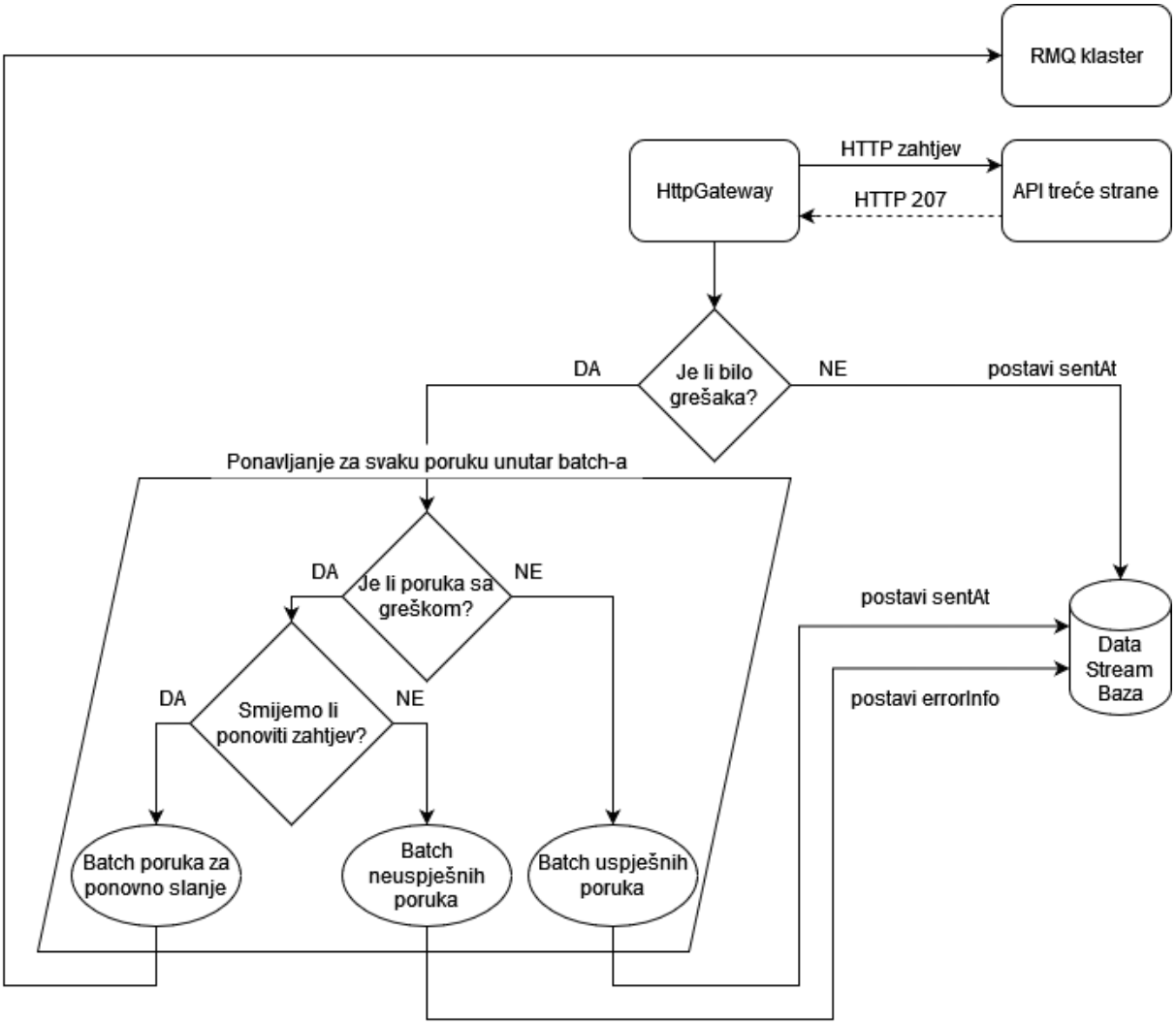
6.2. Drugi slučaj: Ponovi zahtjev

Drugi slučaj do kojeg može doći jeste da API treće strane vrati odgovor, ali da taj odgovor nije HTTP status 207. Prethodno je spomenuto kako taj HTTP kod označava uspješno obrađen zahtjev, stoga u slučaju HTTP koda 5XX zahtjev se može tretirati isto kao da odgovor nije primljen, to jest vratiti batch nazad u queue. Jedna stvar na koju treba pripaziti jeste da se ne zaguši API treće strane. Ukoliko API vraća 5XX zbog toga što je preopterećen, onda se situacija neće promijeniti ako se zahtjevi ne zaustave. Da je API u našoj nadležnosti ovakva kontrola bi se mogla uspostaviti, ali pošto to nije slučaj onda to nažalost to nije moguće implementirati. I u ovom slučaju je moguće da dođe do gomilanja poruka na RMQ queue-u, na što će se također upaliti alarm u internom alatu za monitoriranje.



Slika 13. Dijagram toka u slučaju 5XX greške

6.3. Treći slučaj: HTTP kod 207



Slika 14. Dijagram toka u slučaju HTTP koda 207

Zadnji slučaj koji treba objasniti je ujedno i najkompliciraniji. HTTP kod 207 je HTTP kod koji označava da je moguće da je server vratio više odgovora. U praksi se web preglednik neće nikada susresti sa ovim kodom, ali je pri razvoju API-ja treće strane imalo smisla iskoristiti ga jer će se ruta koja ovaj status može vratiti gađati isključivo iz aplikacija koje implementiraju ovu rutu.

Prva stvar koju treba provjeriti jeste je li uopće bilo grešaka. Ranije je spomenut format odgovora, i on se sastoji iz HTTP koda, ukupnog broja grešaka, i liste koja sadrži id-ove poruka koje su neuspješno obrađene zajedno sa razlogom neuspješne obrade. Ukoliko grešaka uopće nije bilo, onda će broj grešaka biti nula, i možemo sve poslane poruke iz

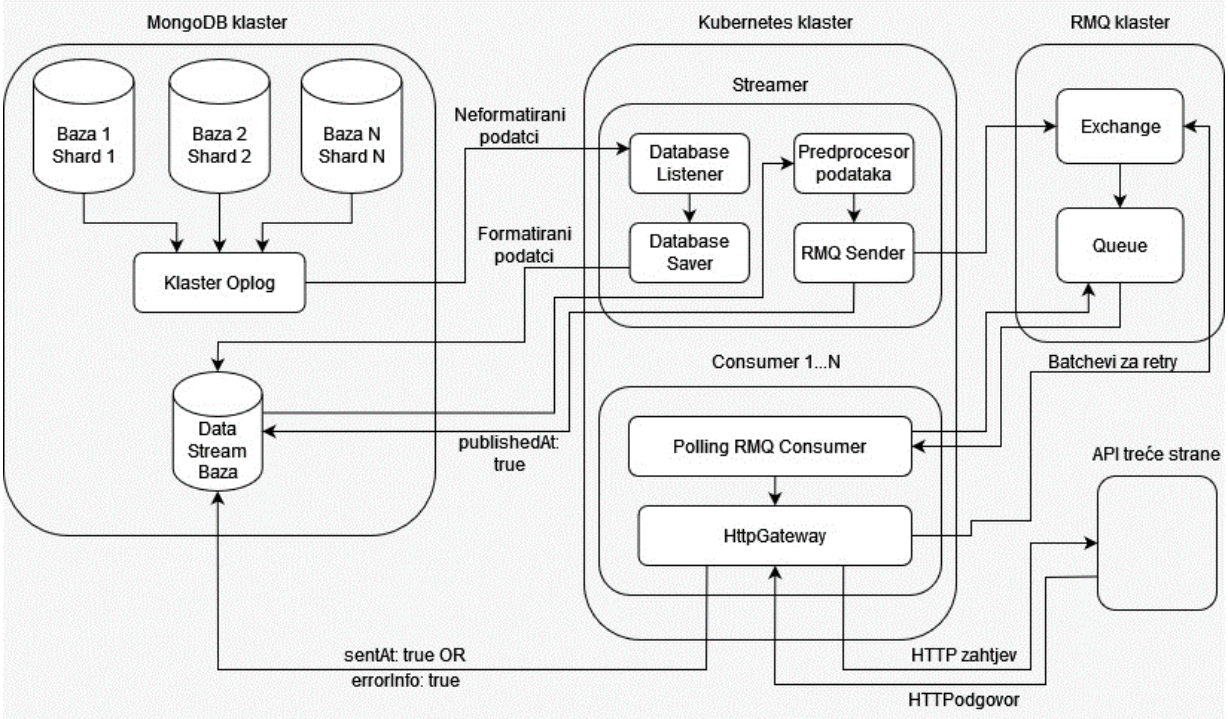
batch-a označiti kao uspješno obrađene i spremiti ih u bazu. To je najjednostavniji slučaj, i idealno najčešće do njega dolazi.

Drugi mogući slučaj jeste da broj grešaka ne bude nula. U tom slučaju se svaka poruka sa greškom mora ispitati. Za svaku poslanu poruku iz batch-a se prvo provjerava je li poruka obrađena uspješno ili neuspješno. Ukoliko je poruka obrađena uspješno, dodaje se u listu poruka koje su obrađene uspješno. Ukoliko je poruka obrađena neuspješno, onda se provjerava tip greške. Neki tipovi grešaka su takozvani sigurni tipovi, to jest, dogovoreno je sa API-jem treće strane da se poruka ponovno pošalje na obradu ukoliko se vrati sa tom greškom. U tom slučaju se poruka dodaje u listu poruka za ponovno slanje na RMQ queue. Zadnja mogućnost jeste da se poruka neuspješno obradila sa greškom koja nije sigurna. U tom slučaju je došlo do neke logičke greške, i poruka se ne smije ponovno slati na obradu. Poruka se sprema u listu poruka sa greškom, zajedno sa podatcima o greški koje vraća API. To može biti do Data Stream aplikacije, ili pak neka greška na API-ju treće strane, ali za svaku takvu poruku potrebna je direktna developerska intervencija kako bi se uklonio problem i implementiralo rješenje kako ne bi opet došlo do problema. Nakon što se provjere sve poruke, onda se liste poruka dalje šalju na završetak obrade. Uspješno obrađene poruke se spremaju u bazu, poruke koje se trebaju vratiti u RMQ se šalju, dok poruke koje su neuspješno obrađene se spremaju u bazu sa detaljima o grešci.

7 DIJAGRAM TOKA PODATAKA

Dijagram toka podataka se koristi kako bi se vizualizirao način prijenosa, obrade, i protoka podataka kroz aplikaciju i njene komponente. Grafičkim prikazom različitih aplikacijskih i neaplikacijskih komponenti, servera, servisa, i načina na koji međusobno komuniciraju moguće je predstaviti cjelokupni sustav na mnogo ljudski razumljiviji način.

Dijagram toka podataka koristi dosta pojednostavljenu verziju toka podataka, svjesno izostavlja međuservise, ali to je sve sa ciljem postizanja dovoljne jednostavnosti da bude lako razumljiv, a da ipak bude dovoljno precizan da će se čitanjem dokumentacije ili proučavanjem koda moći lakše razumjeti kako se različiti dijelovi aplikacije sklapaju u širu sliku.



Slika 15. Dijagram toka podataka

Jedan MongoDB klaster može sadržavati N shardova i N kolekcija. Podatci iz tih kolekcija se mogu iščitati iz oplog-a, i dobivaju se kao BSON dokumenti. Ti dokumenti mogu biti raznih formata, pa ih je potrebno formatirati u neki zajednički model kako bi se mogli dalje obrađivati. Svi dokumenti se spremaju u zajedničku kolekciju iz koje ih kupi predprocesor podataka. Predprocesor ih grupira u batch-eve od po 100 poruka i prosljeđuje ih procesu

koji je zadužen za njihovo slanje na RMQ. Isti proces po slanju ih sprema kao poslane u bazu kako ih predprocesor ne bi opet čitao. Kada poruka dođe na RMQ exchange, onda se po ključu šalje u odgovarajući queue. U pravilu bi queue odmah pretplaćenim consumer-ima proslijedio poruku, ali pošto Data Stream aplikacija koristi polling consumer pristup preuzimanja poruka, onda queue čeka da Polling RMQ Consumer komponenta Data Stream aplikacije zatraži poruku. Ukoliko poruka postoji, onda se prosljeđuje dalje u HttpGateway komponentu. Tu se podatci pakiraju u unaprijed dogovoreni format prihvatljiv API-ju treće strane, i potom se šalju na njega. Ukoliko dođe do greške za koju je dozvoljeno ponovno pokušati zahtjev, podatci se vraćaju nazad na RMQ klaster kako bi opet došli na red za slanje. Ukoliko API vrati informaciju o tome jesu li podatci uspješno obrađeni ili pak odbačeni, onda se ta informacija sprema nazad u glavnu bazu. Ta informacija je vrijeme uspješne obrade, ili vrsta greške. Sa time je gotova zadaća Data Stream aplikacije.

8 ZAKLJUČAK

U ovom diplomskom radu predstavljen je projekt kojemu je glavni zadatak prosljeđivanje podataka iz grupacije MongoDB baza i kolekcija na API treće strane. Problem sam od sebe se ne doima složeno, ali kada se u obzir uzmu razna ograničenja, tehnološki i funkcionalni zahtjevi, onda projekt jako brzo postaje kompliciran. Projekat je realiziran koristeći SpringBoot i njegove ugrađene funkcije poput Spring Batch Job-ova za predobradu podataka i automatskog Scheduler-a za pokretanje tih procesa. Iskorištene su sve mogućnosti MongoDB ChangeStreams funkcionalnosti kako bi se što više smanjilo dodatno operativno opterećenje nad već opterećenom bazom. Podjelom Data Stream projekta na submodule omogućeno je skaliranje određenih dijelova, što u suprotnom ne bi bilo moguće. Implementacija RabbitMQ-a kao medija za prijenos poruka između dvaju submodula je posljedica razdvajanja aplikacije na submodule, što je direktna posljedica potrebe za omogućavanjem skaliranja projekta. Kroz razne code snippet-e je detaljno objašnjen način implementacije raznih dijelova funkcionalnosti u odabranim programskim jezicima. Posebno su istaknuti i obrazloženi razvojni alati koji, premda nisu nužni za rad aplikacije, su ipak imali kritičnu ulogu u poboljšanju kvalitete i brzine razvoja.

Nakon završetka faze razvoja i testiranja, kontaktirani su timovi zaduženi za održavanje Jenkins i Kubernetes servera organizacije. Uz njihovu pomoć kreiran je Jenkins pipeline i pripadajuća Jenkinsfile skripta koja kompajlira program. Kompajlirani kod se potom šalje sa uputama za pokretanje na Kubernetes klaster, gdje se aplikacija pokreće. Nakon uspješnog spajanja na servise koji se nalaze van Kubernetes klastera (MongoDB baza i RMQ server), aplikacija je proglašena kao uspjeh.

Uz aplikaciju raspisana je i detaljna dokumentacija o servisima o kojima aplikaciji ovisi, problemima sa kojima sam se susreo tijekom razvoja aplikacije, i načina na koji su ti problemi riješeni. Pošto su neki od principa rada aplikacije dosta složeni i na prvu nerazumljivi, grafičkim prikazom uz pomoć više detaljnih Dijagrama Toka Podataka je uvelike pojednostavljeno razumijevanje načina na koji se različiti dijelovi aplikacije sklapaju u jednu cjelinu koja zajedno postiže cilj.

Smatram da ću znanje stečeno izradom ovog projekta na praksi moći primijeniti na bilo koji projekt u budućnosti. Pri tome ne mislim na znanje o tehnologijama, nego pristupima radu, razvoju, i dokumentiranju aplikacije na industrijskoj razini, te na načine rada u organizaciji i suradnji sa drugim developerima.

LITERATURA

[1] Službena SpringBoot web stranica – Dokumentacija
https://spring.io/projects/spring-boot/

[2] Službena Apache Maven web stranica – Dokumentacija
https://maven.apache.org/

[3] Službena MongoDB web stranica – Dokumentacija
https://www.mongodb.com/brand-resources

[4] Službena Docker web stranica – Dokumentacija
https://www.docker.com/

[5] Službena Kubernetes web stranica – Dokumentacija
https://kubernetes.io/

[6] Službena Jenkins web stranica – Dokumentacija
https://www.jenkins.io/

[7] Službena MockServer web stranica – Dokumentacija
https://www.mock-server.com/

POPIS SLIKA

SKRAĆENICE

API	*Application Programming Interface*	aplikacijsko programsko sučelje
JSON	JavaScript Object Notation	JavaScript notacija objekta
BSON	Binary JSON	binanri JSON
RAM	Random Access Memory	memorija slučajnog odabira
OAS	Open API Specification	Open API specifikacija
DTO	Dana Transfer Object	objekat za prijenos podataka
RMQ	Rabbit Messaging Queue	Rabbit red za izmjenu poruka
HTTP	HyperText Transfer Protocol	hipertekst transfer protoklol
DNS	Domain Name System	sustav domenskih naziva
ID	Identifier	identifikator