Milena Laketić

Command Query Responsibility Segregation Pattern

Seminarski rad

Arhitekture sistema velikih skupova podataka

Novi Sad, 2022.

Sadržaj

1. Uvod

U kompleksnim aplikacijama korišćenje istog modela podataka za čitanje i pisanje može dovesti do preterano komplikovanog modela koji preuzima previše odgvornosti. Sa ovim u vidu, *Command and Query Responsibility Segregation* šablon (skraćeno CQRS) rešave neke od problema u ovim situacijama jednostavnim odvajanjem operacija čitanja i pisanja nad podacima. U ovom radu biće opisane osnove šablona i motivacija za korišćenje, kao i povezani pristupi, ukazaće se na prednosti i mane pri upotrebi CQRS, kao i preporuke kada razmotriti njegovu primenu. Neophodno je imati u vidu da CQRS nije *silver bullet*, kao i da ga ne bi trebalo koristiti kao pri dizajniranju *top – level* arhitekture sistema, već bi ga trebalo koristit u podstistemima koji imaju posebnih benefita od njegove primene.

2. Motivacija i osnovni koncepti

2.1. Problem

U tradicionalnim arhitekturama, isti model podataka se koristi za kreiranje upita i izmenu baze podataka. Ovaj jednostavni pristup se pokazao veoma kompatibilnim pri osnovnim CRUD operacijama nad podacima. Međutim, sa porastom kompleksnosti aplikacije, može postati nepovoljan. Jednostavan primer gde ovaj metod nije najpogodniji bio bi izvršavanje raznolikih upita koji vraćaju DTO (*Data transfer objects*) veoma različitih struktura. U tom slučaju, mapiranje objekata može postati veoma komplikovano. Sa druge strane, pri pisanju i izmenama podataka, model može implementirati složene validacije i biznis logiku. Kao rezultat oba slučaja, dobija se suviše kompleksan model koji obavlja previše.

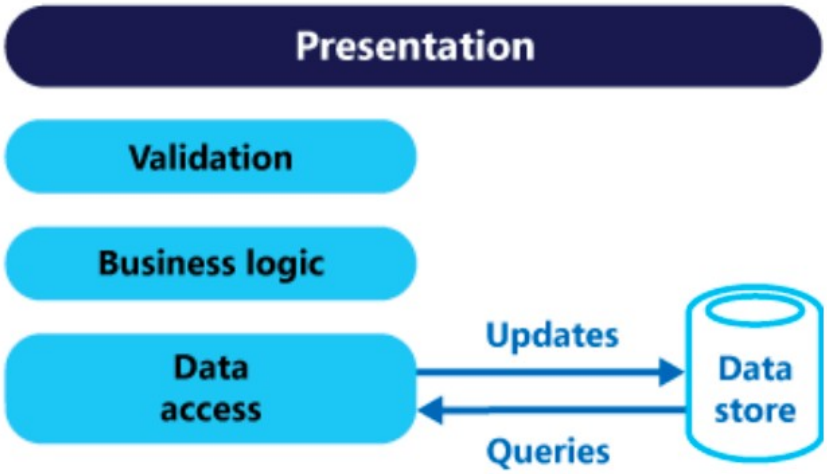

Figura 1 : Tradicionalni pristup [izvor literatura 1]

U ovakvoj arhitekturi uočavaju se sledeće nepogodnosti:

- Često postoji odstupanje u reprezentaciji podataka za čitanje i pisanje. Na primer, dodatne kolone potrebne za vizualizaciju podataka.
- Pristup istim podacima u paraleli može dovesti do takmičenja procesa
- Potencijalno negativne performanse zbog kompleksnosti upita i opterećenja skladišta podataka

- Svaki entitet je subjekat operacija čitanja i pisanja što zahteva rukovođenje permisijama i sigurnost na određenom nivou kako se ne bi izložili podaci u pogrešnom kontekstu
- Potreba za čitanjem i pisanjem podataka je često nesrazmerna, zahtevajući različite performanse i skaliranja za obe operacije

2.2. Rešenje

Razdvojiti operacije čitanja i pisanja – kao bazična ideja CQRS šablona. Osnovna tvrdnja iza ovog koncepta je :

- Metoda se može posmatrati kao komanda koja izvršava neku akciju ili kao upit koji vraća podatke, ali **ne oba**.

Drugim rečima, postavljanje pitanja ne bi trebalo da promeni odgovor. Formalnije, metode bi trebalo da vraćaju neku vrednost bez bočnih efekata.

Sa ovim u vidu, trebalo bi razdvojiti metode objekta u dve strogo disjunktne kategorije:

1. Upiti (eng. q*ueries*) : imaju povratnu vrednost ali ne menjaju stanje sistema
2. Komande (eng. *commands*) : menjaju stanje sistema, ali nemaju povratnu vrednost

Primer prikazan na Figuri 1 je orijentisan ka podacima (eng. *data-centric*) i pretpostavlja da korisnik obavlja operacije koje su zapravo CRUD nad podacima. Međutim, u nekim aplikacijam korisnije je da korisnik šalje komande kroz korisnički interfejs (eng. *UI*) umesto DTO i da na taj način zahteva izmene podataka od aplikacije. Komande bi trebalo biti orijentisane ka zadacima (eng. *behavior-centric*) umesto ka podacima (eng. *data-centric*). Na primer, komanda bi trebalo da bude *bookRoom* umesto *setReservationStatus.* Ovako definisane komande direktno oslikavaju operacije u domenu, mogu biti intuitivnije korisniku i efikasnije oslikavaju nameru korisnika. U tipičnim implementacijama CQRS šablona model za čitanje komunicira sa UI preko standardnog pristupa – kroz DTO, dok UI šalje komande modelu za pisanje. Sa ovim pristupom, gore prikazani dijagram (Figura 1) može dobiti novi oblik :

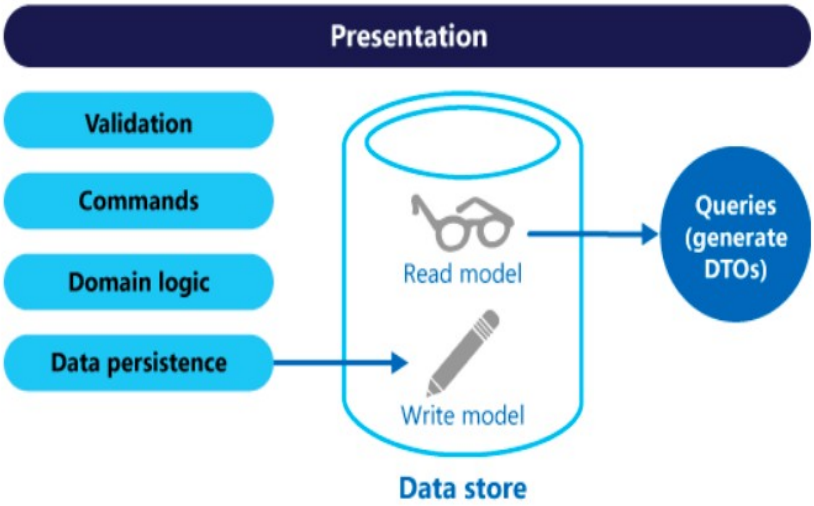

Figura 2: Arhitektura uz CQRS [izvor literatura 1]

Prvi, najočigleniji benefit ovakve podele je pojednostavljenje koda kao i ograničavanje odgovornosti svakog modela. Objekti su odgovorni ili za izmenu podataka ili za njihovo dobavljanje.

Moguće je i potpuno razdvajanje podataka u dva fizički odvojena skladišta podataka. U tom slučaju, baza podataka za čitanje može imati svoju šemu i biti optimizovana za upite. Čuvanjem materijalizovanih pogleda mogu se izbeći kompleksna spajanja (eng. *join*) ili *O/RM* mapiranja podataka. Takođe, skladišta mogu biti drugačijeg tipa kako bi bila što prikladnija za operaciju čitanja ili pisanja. Pogodno je koristiti normalizovani oblik za skladištenje modela za pisanje, odnosno denormalizovani za čitanje. Na dijagramu ispod prikazana je arhitektura sa fizički razdvojenim skladištima.

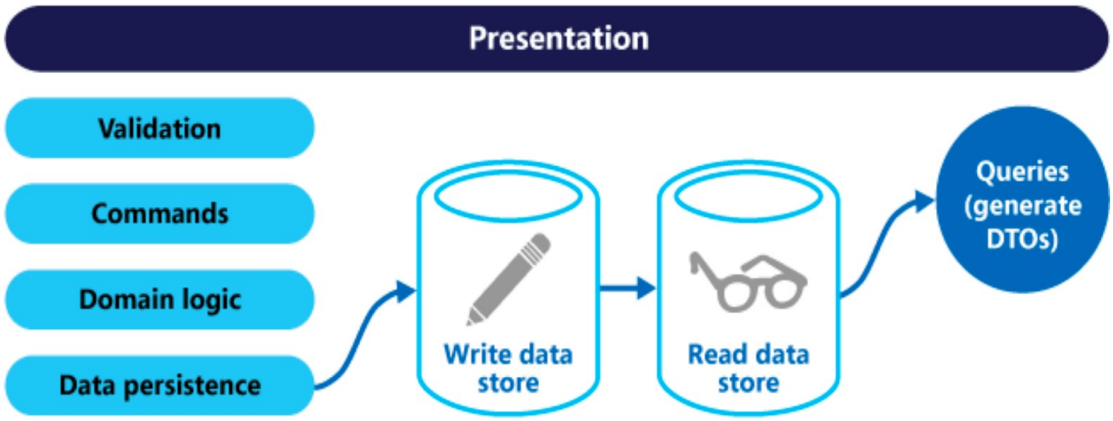

Figura 3: Fizički odvojena skladišta za pisanje i čitanje podataka [izvor literatura 1]

U ovom slučaju potrebno je sinhronizovati skladišta za pisanje i čitanje podataka. Obično, model za pisanje kreira događaj (eng. *event*) svaki put kada se izmeni baza podataka. Ovaj pristup detaljnije je obrađen u poglavlju 4. Izmena podataka i kreiranje događaja moraju se odigrati u okviru jedne transakcije. Neki od benefiti koji se postižu razdvajanjem su:

- Skladište za čitanje podatka može biti *read-only* replika skladišta za pisanje podataka. Korišćenjem više *read-only* replika može povećati performanse upita, posebno u distribuiranim sistemima, gde se ove replike mogu nalaziti blizu instanci aplikacija.
- Skladišta za čitanje i pisanje podataka mogu imati sasvim drugačiju strukturu (npr. baza podaka za pisanje može biti relaciona, dok se za čitanje koristi dokument orijentisano skladište).
- Moguće je različito skaliranje obe strane (npr. skladišta za čitanje podataka obično su više opterećena od skladišta za pisanje).
- Jednostavnije šeme za zaključavanje podataka. Nije potrebno brinuti kako zaključavanje utiče na upite i prikaz podataka

Neke od implementacija CQRS šablona koriste *Event Sourcing* šablon što je i tema poglavlja 4.

Iako figura 3 prikazuje dva odvojena skladišta podataka, ovakva podela kao i odabir tehnologije za skladištenje (relaciona baza, *NoSQL*, *event sourcing*...) nije uslov da bi se primenio CQRS. Trebalo bi ga posmatrati kao šablon koji olakšava podelu podataka i podržava širok spektar mehanizama za skladištenje. Osim izbora tehnologije, potrebno je i odrediti na koji način će podaci biti particionisani (horizontalno, vertikalno i funkcionalno) kako bi se primenom CQRS postiglo poboljšavanje performansi, a ne nepotrebno usložnjavanje celog sistema.

3. Implementacija

Sama ideja je veoma jednostavna, ipak neke od stavki koje bi trebalo razmotriti pre usvajanja ovog šablona su :

- **Konzistentnost**: u slučaju odvojenih skladišta za pisanje i čitanje, potrebno je obezbediti da podaci u bazi za čitanje budu sveži
- **Slanje poruka**: iako CQRS ne zahteva sistem za razmenu poruka, uobičajeno je da se on koristi pri obaveštavanju o izmenama i komandama

- **Kompleksnost**: može dovesti do kompleksnog dizajna aplikacije, pogotovo ako uključuje *Event Sourcing* šablon (poglavlje 4)

3.1. Konzistentnost

Bitna karakteristika podataka unutar WEB aplikacija i servisa je da precizno oslikavaju najskorije dostupne informacije kao i da su konzistentni međusobno. Druga osobina implicira da sve instance aplikacije sadrže iste podatke sve vreme. Održavati podatke u ovom obliku u distribuiranim sistemima može biti veliki izazov. Uočavaju se dva tipa konzistentnosti:

- Jaka konzistentnost (eng. *strong data consistency*)
- Eventualna konzistentnost (eng. *eventual data consistency*)

Prvi pristup implicira da se sve promene nad podacima posmatraju kao atomične operacije. Ako neki proces menja više podataka, postoje dva moguća ishoda: sve promene su izvršene uspešno i nijedna nije izvršena. Odnosno, u slučaju da jedan bilo koji deo izmena nije uspeo, sve ostale se moraju poništiti. U periodu od početka transakcije do kraja, drugim konkurentnim transakcijama nije dozvoljeno da pristupaju podacima nad kojima se izvršavaju izmene. Njihovo dalje izvršavanje je blokirano. U slučaju da su podaci replicirani, transakcija koja podržava jaku konzistentnost podataka, ne sme završiti sve dok sve postojeće kopije nisu uspešno ažurirane. Cilj ovakvog modela je da se minimizuje šansa da neka instanca aplikacije sadrži neadekvatne (ustajale, eng. s*tale*) podatke. Cena implementacije je uticaj na dostupnost, performanse i skaliranje ovakvih rešenja. U distribuiranim okruženjima gde skladišta podataka mogu biti geografski udaljena, postoje opasnosti predugog vremena potrebnog za komunikaciju, kao i blokiranja podataka na duži period.

Ako bi se primenila jaka konzistentnost na sistem koji implementira CQRS šablon pod jednu transakciju spadalo bi: ažuriranje skladišta za pisanje podataka i ažuriranje skladišta za čitanje podataka (Figura 4). Mogući problemi tiču se performansi i dostupnosti. U slučaju da se skladište za čitanje horizontalno skalira dodavanjem novih instanci, da bi se transakcija završila neophodno je ažuriranje **svih** instanci. Najkraće vreme izvršavanja jednako je potrebnom vremenu da najspori učesnik primeni izmene. U slučaju da neki čvor (deo) sistema otkaže ili iz nekog razloga ne izvrši izmene, transakcija se ne može završiti. Garantovanjem konzistentnosti ne može se garantovati i dostupnost sistema.

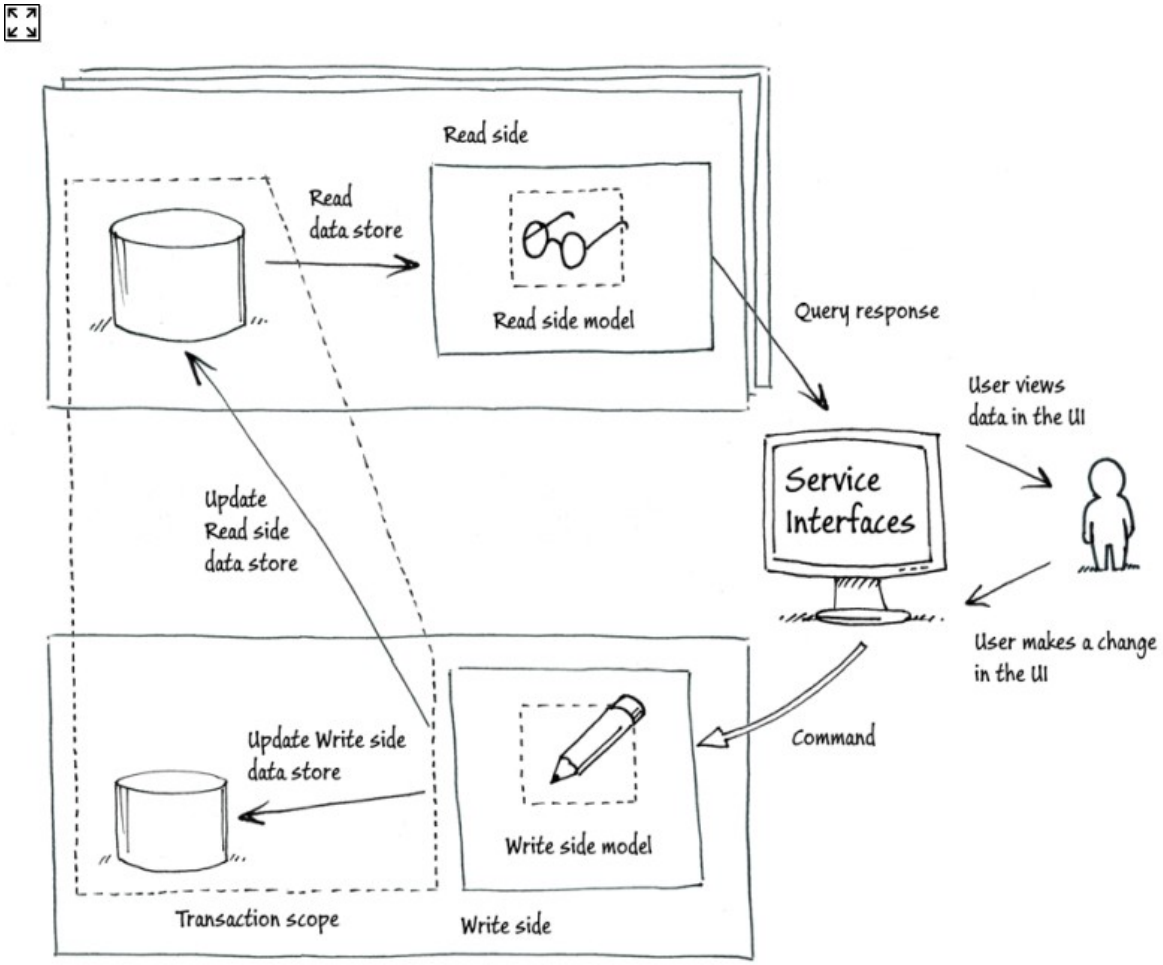

Figura 4: Strong data consistency i CQRS [izvor literatura 6]

U mnogim slučajevima jaka konzistentnost između instanci sistema nije neophodna, sve dok se sve izmene primene ili ponište u nekom trenutku. Eventualna konzistentnost implicira da procesi koji menjaju podatke mogu izvršiti svoje poslove nezavisno, bez blokiranja i zaključvanja resursa. CAP (skraćeno eng. *Consistency, Availability, and Partition Tolerance*) teorema veoma ide u korist ovom pristupu.

Sa tim u vidu, pod jednu transakciju spadalo bi: ažuriranje skladišta za pisanje podataka i slanje poruke o promeni (Figura 5). Oslanjanjem na pouzdani sistem za razmenu poruka obezbeđuje se da će model za čitanje biti konzistentant u nekom momentu sa model za pisanje. Pod pretpostavkom da infrastruktura za slanje poruka omogućava brzo skladištenje novih poruka, ovo rešenje ne pati od potencialnih problema sa performansama. Takođe, dostupnost sistema ne zavisi od dostupnosti i broja instanci na strani čitanja.

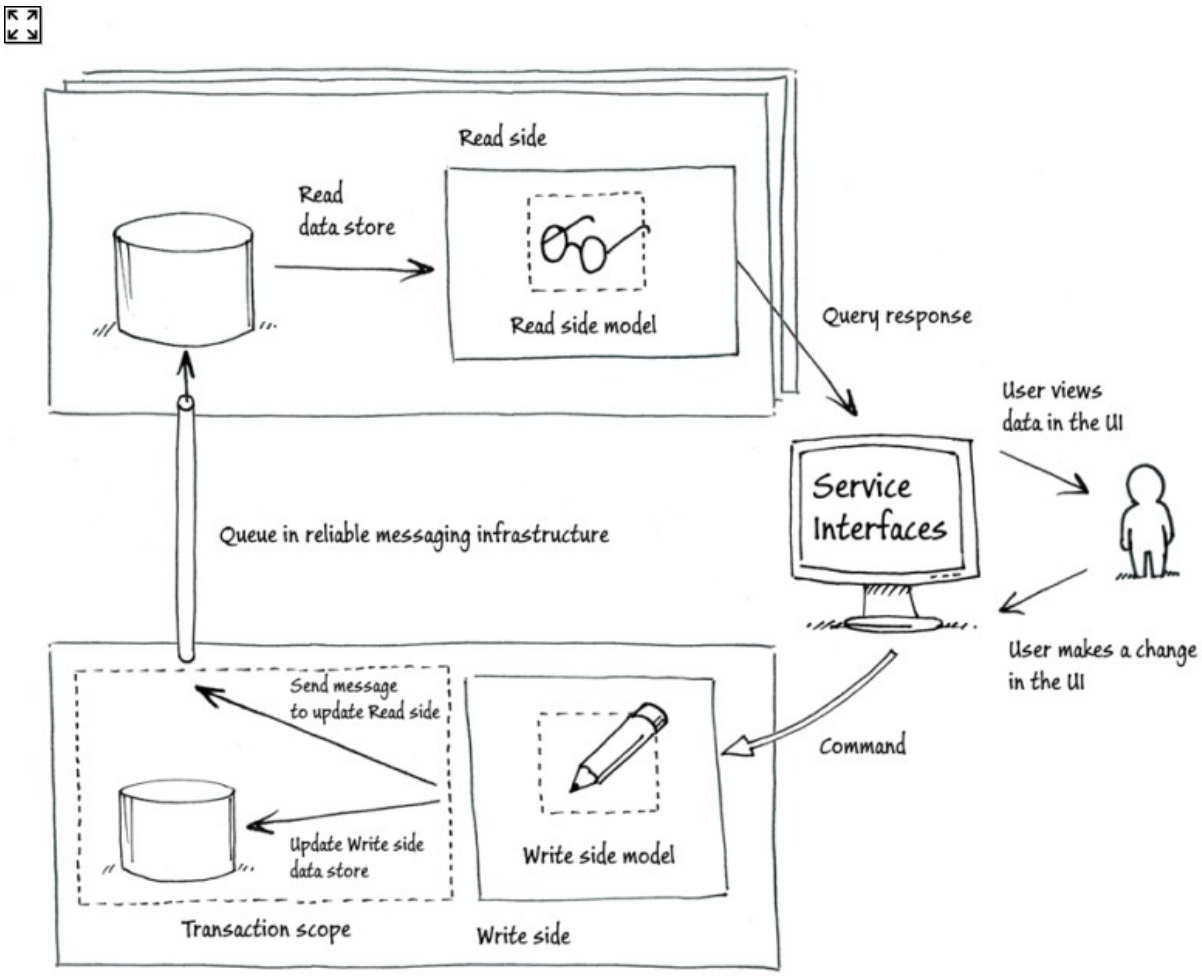

Figura 5: Eventual consistency i CQRS [izvor literatura 6]

Treći primer (Figura 6) prikazuje način da se izbegne distribuirana transakcija, ali zavisi od funkcionalnosti koje pruža skladište za pisanje podataka. U ovom slučaju, pod jednu transakciju spada samo ažuriranje podataka na strani pisanja. Međutim, potrebno je da skladište podržava slanje poruka nakon svake izmene podataka. Ovakav pristup prirodno vodi ka kombinovanju CQRS i Event Sourcing šablona koji je tema narednog poglavlja.

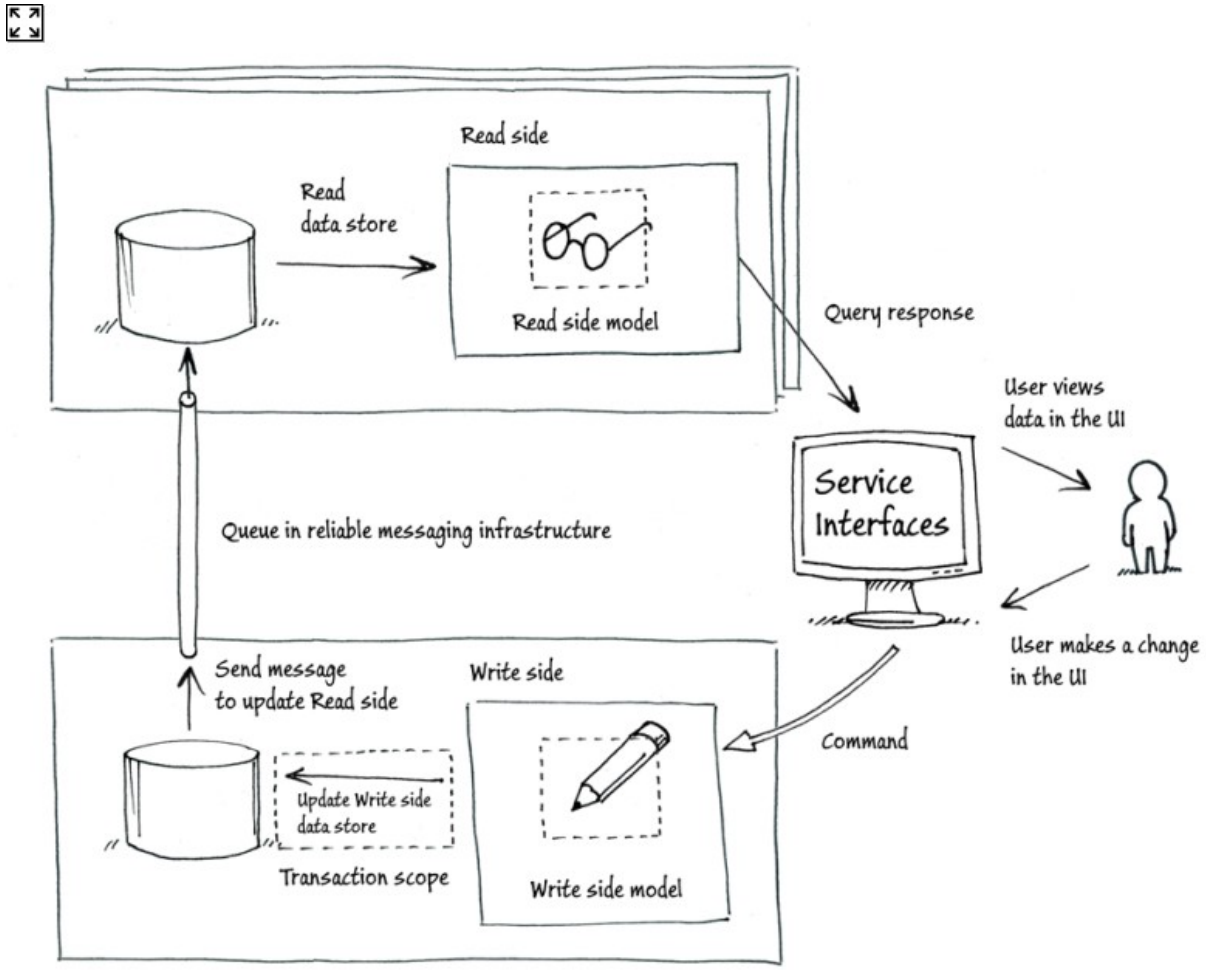

Figura 6: Nedistribuirana transakcija [izvor literatura 6]

3.2. Slanje poruka

U kontekstu primene CQRS-a postoje dva tipa poruka: komande i događaji. Tipično, sistemi koji implementiraju ovaj šablon su veoma distribuirani sistemi i neophodno je postojanje pouzdane infrastrukture za transport poruka. Za komande koje imaju samo jednog primaoca obično se koristi topologija reda. Za događaje koji mogu imati nekoliko primalaca obično se koristi topologija *publish/subscribe*.

Prvi od nekoliko mogućih problema je **višestruko slanje istih poruka**. Moguća rešenja su : dizajniranje idempotentnih poruka (odnosi se na komande) tako da duplikati ne utiču na konzistentnost, implementiranje detekcije duplikata. Neke infrastrukture za slanje poruka podržavaju ovu funkcionalnost.

Izgubljene poruke takođe mogu dovesti sistem u nekonzistentno stanje, a prouzrokovane su greškom u infrastrukturi za slanje poruka. Moguća rešenja obuhvataju odabir onih infrastruktura koje garantuju dostavljanje

barem jednom, kao i implementiranje detekcije da li je poruka pristigla na odredište (potvrada od primaoca ili brojno označavanje poruka).

Moguće je da poruke **ne pristignu u redosledu u kom su kreirane**. U nekim slučajevima, ovo ne predstavlja problem ili izabrana infrastruktura garantuje slanje poruka u odgovarajućem redosledu. U suprotnom, potrebno je brojno označavanje poruka ili implementiranje procesa koji će čuvati ovakve poruke sve dok ne bude u stanju da ih poređa u korektnom redosledu.

Neobrađene poruke: moguće je da klijent primi poruku, ali da se dogodi greška pri obradi, nakon čega je poruka izgubljena. Kao i u prethodno navedenim primerima, potrebno je odabrati infrastrukturu koja uspešno rukuje ovakvim slučajevima ili samostalno implementirati rešenje.

4. CQRS i Event Soucing

Upotrebom relacionih baza podataka na obe strane (čitanje i pisanje) i dalje se primenjuju CRUD operacije nad podacima za pisanje i neophodan je mehanizam koji bi primenio nastale promene iz normalizovanih tabela (strana pisanja) u denormalizovane (strana čitanja). Ukoliko bi se umesto konkretnih podataka u skladištu za pisanje čuvale promene koje su nastale (događaji, eng. *events*) moguće je sačuvati sve podatke jednostavno dodavanjem novih event-ova na postojeće. Odnosno korišćenjem samo operacija dodavanja (eng. *append*). Takođe, isti događaji mogli bi da se koriste za obaveštavanje skladišta za čitanje da je došlo do izmena.

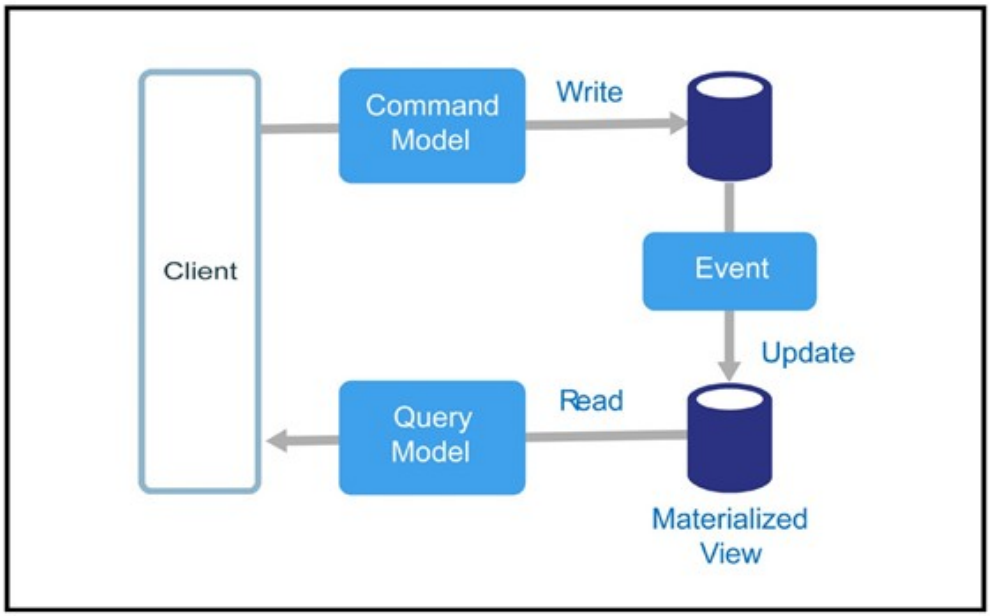

Figura 4: Arhitektura sa CQRS i Event Sourcing [https://awesomeopensource.com/project/charlessolar/TodoMVC-DDD-CQRS-EventSourcing]

S obzirom da pri primeni CQRS šablona treba uočiti i dobro razdvojiti logičke celine u sistemu (ekvivalentno agregatima u kontekstu *Domain Driven Design-a*), još jedna uloga događaja mogla bi biti komuniciranje između pomenutih celina.

U slučaju primene ova dva modela u kombinaciji, skladište za događaje je skladište za pisanje podataka, ali ujedno i izvor novih događaja. Model za čitanje podataka obezbeđuje materijalizovane poglede obično u veoma denormalizovanom obliku. Ovi pogledi su usko povezani sa korisničkim interfejsom, zadovoljavajući na taj način potrebe prikaza podataka kao i performanse upita. Moguće je asinhrono ažuriranje pogleda u bazi za čitanje. Nakon izmena u podacima, pogledi se brišu i kreiraju novi. Mogu se posmatrati i kao *read-only* keš podataka.

Čuvanjem toka događaja umesto konkretnih podataka u skladištu za pisanje omogućava lakše izbeganje konflikata pri izmeni podataka. Drugi, ne manje važan, benefit ovakvog pristupa je mogućnost "hvatanja" korisnikove namere. Na primer, umesto da se čuva podatak o korisnikovoj poslednjoj adresi, čuvala bi se istorija adresa koje je korisnik imao. Moglo bi biti korisno otkriti **zašto** je došlo do promene. Ukoliko se propusti neka informacija prilikom obrade i analize podataka, moguće je kasnije iskoristiti sve informacije u slučaju da se čuva istorija događaja. Međutim, neophodno je voditi računa o tome koje informacije eventualno mogu biti korisne.

Pri upotrebi ova dva pristupa zajedno trebalo bi voditi računa o konzistentnosti (nakon kreiranja njovog događaja potrebno je određeno vreme dok se ne ažuriraju pogledi), kao i o kompleksnosti izgradnje celog sistema. Osim vremena potrebnog da se skladište za čitanje obavesti o novokreiranom događaju, zahtevaju se i određene karakteristike sistema kako bi se promene primenile u nekom prihvatljivom vremenskom periodu. Pogotovo u slučajevima kada je neophodno izvršiti analizu nad podacima koja obuhvata prethodna stanja sistema.

5. Primena

CQRS šablon, iako baziran na jednostavnoj ideji, treba primenjivati jedino u situacijama u kojima se vidi jasan benefit. U drugim konteksima moguć je kontraefekat i nedostizanje očekivanih performansi. Sledi nekoliko sugestija kada koristiti CQRS, kao i kada to nije neophodno.

Razmotriti primenu CQRS-a u sledećim scenarijima:

- Domeni u kojima veliki broj korisnicima pristupa podacima u paraleli. CQRS omogućava definisanje komandi na tom nivou granularnosti kako bi da se što bolje izbegli konflikti.
- Korisnički interfejsi su orijentisani ka zadacima (npr. korisnik bi trebalo biti vođen kroz kompleksan proces odnosno obavljanje određenih koraka). U tom slučaju, korisno je da model za pisanje podataka sadrži sve neophodne informacije tokom tog procesa, obogaćene validacijama i biznis logikom. Takođe, model bi mogao posmatrati grupu (logički) povezanih objekata kao celinu, vodeći računa da su pomenuti objekti u konzistentnom stanju. Sa druge strane, model za čitanje mogao bi da sadrži samo informacije neophodne za prikaz.
- Slučajevi u kojima je brzina pristupa podacima radi čitanja od izuzetne važnosti. CQRS, kao što je već pomenuto, omogućava odvojeno skaliranje skladišta za čitanje.
- Očekivane su promene u modelima podataka, kao i u biznis pravilima unutar sistema.
- Integracija sa drugih sistemima

Neki od slučajeva u kojima nije preporučena primena ovog šablona su:

- Domen sistema i/ili biznis logika nije previše kompleksna
- Sistem podržava samo jednostavne CRUD operacije

Iako granice nisu jasno definisane, korisno je imati u vidu neke smernice pri razmatranju ovog šablona. CQRS je veoma pogodan za izdvajanje individualnih delova sistema. Korisna heuristika za identifikovanje mesta gde je njegova primena korisna bila bi pretraga kompleksnih komponenti koje rade u paraleli i iziskuju striktnu biznis logiku.

Na nekim mestima moguće je uočiti primenu ovog šablona, iako nije striktno naglašena. U slučaju jezera podataka sa tehnologijama kao što su Hadoop i njemu slične, nije potrebno definisati model podataka koji će se čuvati. Shema je diktirina od izvora podataka. Različiti izvori podataka kao i njihova struktura pri skladištenju pripadaju command delu CQRS šablona. Query deo obuhvata spajanje i optimizovanje podataka za različite upite pri analizi.

5.1. Primer implementacije

Kreiranje i manipulisanje korisničkim profilom je tipičan zahtev u mnogim aplikacijama. Definisaćemo jednostavan domen prikazan na figuri 5. Model je

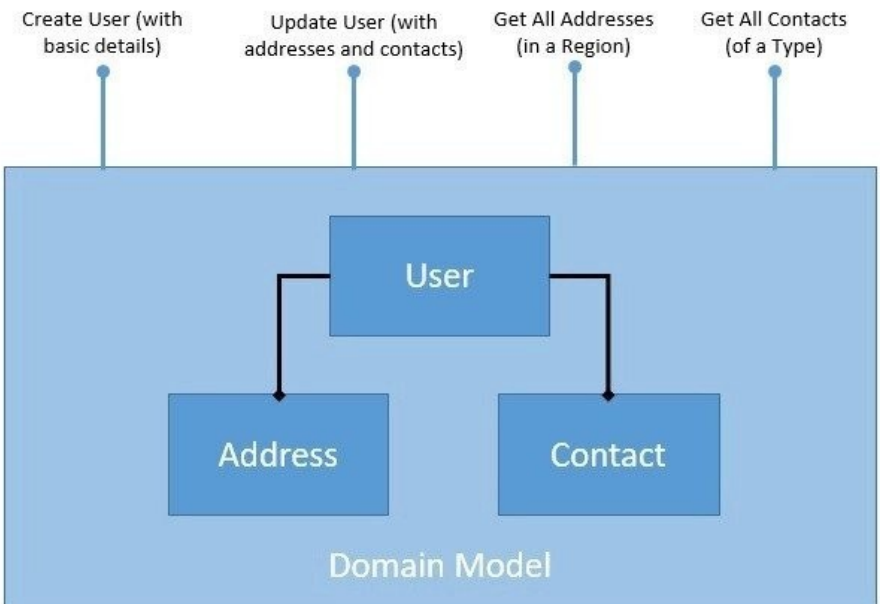

Figura 5: Jednostavan model domena [izvor literatura 8]

normalizovan i podržava nekoliko CRUD operacija : kreiranje novog korisnika, izmena korisnika, dobavljanje svih adresa u regionu i dobavljanje svih kontakata određenog tipa.

```
public class User {
private String userid;
    private String firstName;
    private String lastName;
    private Set<Contact> contacts;
    private Set<Address> addresses;
    // getters and setters
}

public class Contact {
    private String type;
    private String detail;
    // getters and setters
}

public class Address {
    private String city;
    private String state;
    private String postcode;
    // getters and setters
}
```

Figura 6: Modeli u tradicionalnom pristupu

Nakon definisanja modela, potrebno je implementirati servis koji izvršava tipične CRUD operacije nad posmatranim modelom.

```
public class UserService {
	private UserRepository repository;

	public UserService(UserRepository repository) {
		this.repository = repository;
		}

	public void createUser(String userId, String firstName, String
	lastName){
		User user = new User(userId, firstName, lastName);
		repository.addUser(userId, user);
	}

	public void updateUser(String userId, Set<Contact> contacts,
	Set<Address> addresses) {
		User user = repository.getUser(userId);
		user.setContacts(contacts);
		user.setAddresses(addresses);
		repository.addUser(userId, user);
		}

	public Set<Contact> getContactByType(String userId, String
	contactType) {
		User user = repository.getUser(userId);
		Set<Contact> contacts = user.getContacts();

		return contacts.stream() .filter(c ->
		c.getType().equals(contactType)) .collect(Collectors.toSet());
		}

	public Set<Address> getAddressByRegion(String userId, String state) {
		User user = repository.getUser(userId);
		Set<Address> addresses = user.getAddresses();

		return addresses.stream() .filter(a ->
		a.getState().equals(state)) .collect(Collectors.toSet());
		}
	}
```

Figura 7: Servis za određene CRUD operacije

U ovako definisanom klasičnom pristupu, jedan isti model koristi se i za operacije čitanja kao i za operacije pisanja. Iako ovo ne predstavlja problem za jednostavne domene kao što je prikazani, može dovesti do mnogih nepogodnosti u složenijim situacijama (optimizacija, kompleksnot i ostale mane razmotrene u poglavlju 2.1). Moguće je primeniti CQRS šablon tako da posmatrani sistem dobije sledeći oblik :

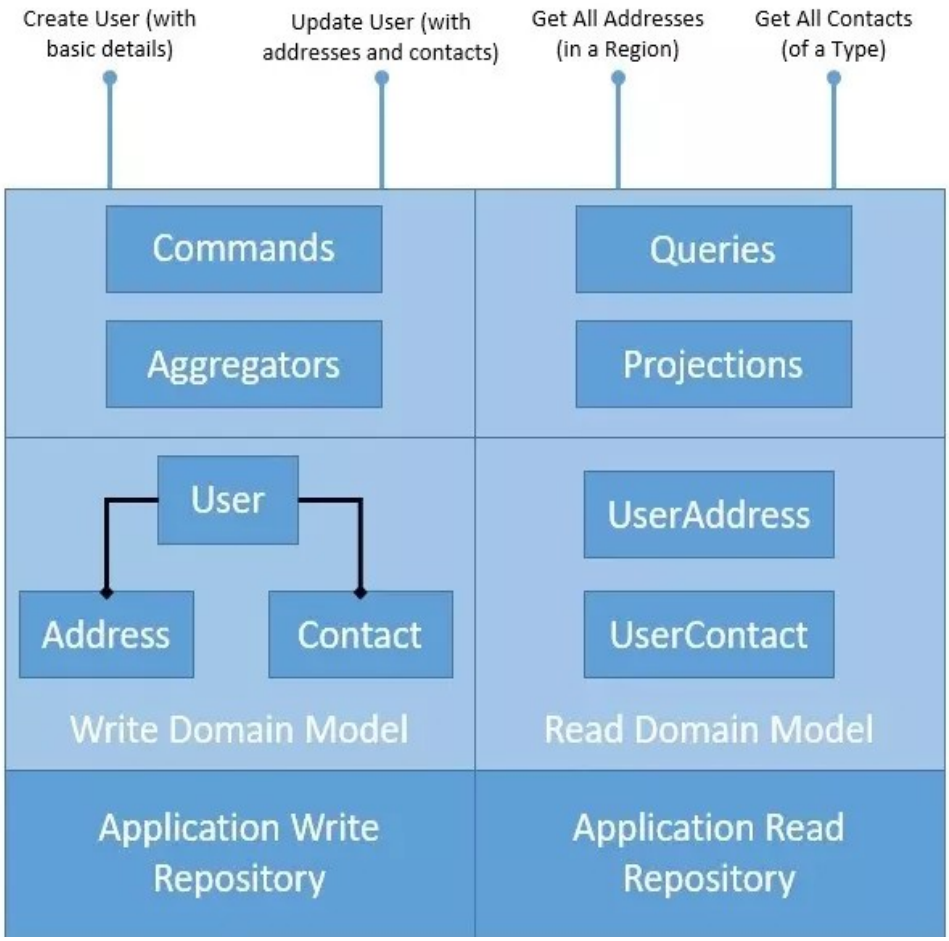

Figura 8: Model domena sa CQRS [izvor literatura 8]

Na Figuri 8 arhitektura aplikacije je striktno podeljena na deo zadužen za prikaz i čitanje informacija (*query*) i deo zadužen za pisanje i izmenu podataka (*write*). Implementacija strane za pisanje podataka podrazumeva definisanje komandi za kreiranje korisnika kao i za izmenu tih podataka.

```
public class CreateUserCommand {
    private String userId;
    private String firstName;
    private String lastName;
}

public class UpdateUserCommand {
    private String userId;
    private Set<Address> addresses;
    private Set<Contact> contacts;
}
```

Figura 9: Komande unutar write strane

Osim toga, potrebno je definisati i klasu koja igra ulogu agregata i objedinjuje pomenute metode. Klasa *UserAggregate* koristi repozitorijim da dobavi trenutno stanje i da sačuva izmene u podacima.

```
public class UserAggregate {

    private UserWriteRepository writeRepository;

    public UserAggregate(UserWriteRepository repository) {
        this.writeRepository = repository;
    }

    public User handleCreateUserCommand(CreateUserCommand command) {
        User user = new User(command.getUserId(),
      command.getFirstName(), command.getLastName());
        writeRepository.addUser(user.getUserid(), user);

        return user;
    }

    public User handleUpdateUserCommand(UpdateUserCommand command) {
        User user = writeRepository.getUser(command.getUserId());
        user.setAddresses(command.getAddresses());
        user.setContacts(command.getContacts());
        writeRepository.addUser(user.getUserid(), user);

        return user;
    }
}
```

Figura 10: Agregat koji objedinjuje komande

Implementacija strane za čitanje podataka podrazumeva definisanje modela koji sadrže informacije relevantne za upite, samih upita i klase koja će reprezentovati objekte u različitim strukturama (*UserProjection*).

```
public class UserAddress {
    private Map<String, Set<Address>> addressByRegion = new HashMap<>();
}

public class UserContact {
    private Map<String, Set<Contact>> contactByType = new HashMap<>();
}
```

Figura 11: Model orijentisan ka upitima

U domenu postoje dva upita : dobavljanje svih korisničkih adresa u nekom regionu i dobavljanje svih kontakata korisnika određenog tipa. U skladu sa upitima klase *UserAddres* i *UserContract* nose odgovarajući naziv i atribute. Na Figuri 12 prikazane su jednostavne Java klase koje sadrže informacije neophodne za izvršavanje upita.

```
public class ContactByTypeQuery {
    private String userId;
    private String contactType;
}

public class AddressByRegionQuery {
    private String userId;
    private String state;
}
```

Figura 12: Klase koje reprezentuju upite u sistemu

Na Figuri 13 prikazana je klasa koja prihvata upite kao parametre metoda i vraća tražene podatke u odgovarajućoj strukturi (kao objekat klase *UserAddres* ili klase *UserContract*).

```
public class UserProjection {

    private UserReadRepository readRepository;

    public UserProjection(UserReadRepository readRepository) {
        this.readRepository = readRepository;
    }

    public Set<Contact> handle(ContactByTypeQuery query) {
        UserContact userContact =
                readRepository.getUserContact(query.getUserId());

        return userContact.getContactByType()
          .get(query.getContactType());
    }

    public Set<Address> handle(AddressByRegionQuery query) {
        UserAddress userAddress =
                readRepository.getUserAddress(query.getUserId());

        return userAddress.getAddressByRegion()
          .get(query.getState());
    }
}
```

Figura 13: Klase koje objedinjuje upite i modele

Primenom CQRS na predstavljeni domen, dobijaju se dva disjunktna interfejsa domena (za čitanje i pisanje) koji ne zavise od osobina skladišta koje se koristi i mogu biti optimizovani i izmenjeni nezavisno jedan od drugog.

6. Zaključak

U ovom radu predstavljeni su osnovni koncepti Command Query Responsability Segregation šablona. Objašnjeni su benefiti koji se mogu postići njegovom primenom, kao što su skalabilnost, fleksibilnost, pojednostavljenje modela, bolji fokus na biznis i dizajniranje korisničkog interfejsa orijentisanog ka zadacima. Takođe, ukazano je na bitne koncepte koje bi trebalo razmotriti prilikom primene ovog šablona, a koji se tiču kompleksnosti, konzistentnosti i infrastrukture za slanje poruka. Uočene su kompatibilnosti sa drugim šablonima, kao i eventualna korist njihovog kombinovanja. Na kraju, čitaocu su date okvirne smernice za identifikovanje okolnosti pogodne za primenu CQRS, kao i onih koje to nisu. Zasnovan na jednostavnoj ideji, CQRS, otvara vrata mnogim drugim izmenama u aplikaciji koje doprinose određenim benefitima. Podstiče na detaljniju analizu samog domena sistema, uočavanje logičkih celina i posmatranje korisničkog interfejsa na drugačiji, netradicionalni, način. Zahteva proučavanje povezanih pristupa, benefita i mana koje dolaze sa njihovim kombinovanjem.

Literatura

1. https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs
2. https://martinfowler.com/bliki/CommandQuerySeparation.html
3. https://docs.microsoft.com/en-us/azure/architecture/best-practices/data-partitioning
4. https://docs.microsoft.com/en-us/azure/architecture/patterns/event-sourcing
5. https://docs.microsoft.com/en-us/previous-versions/msp-n-p/jj591568(v=pandp.10)
6. https://docs.microsoft.com/en-us/previous-versions/msp-n-p/jj591577(v=pandp.10)
7. http://dekarlab.de/wp/?p=817
8. https://www.baeldung.com/cqrs-event-sourcing-java