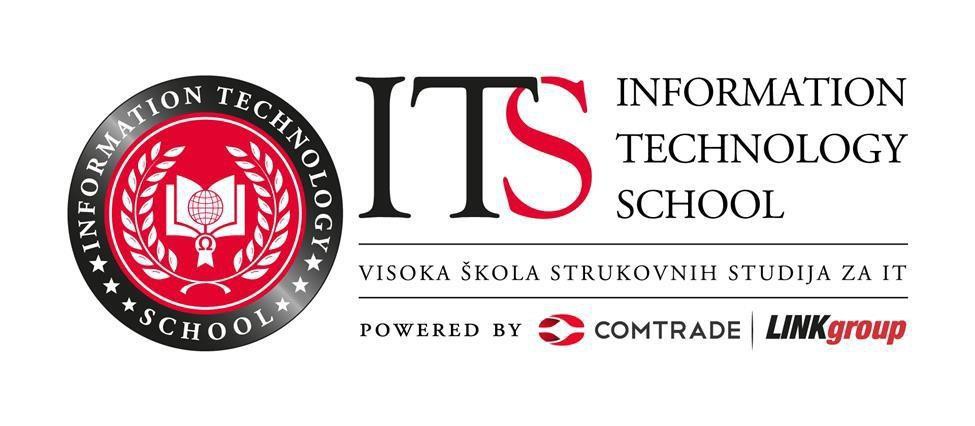
**VISOKA ŠKOLA STRUKOVNIH STUDIJA ZA INFORMACIONE TEHNOLOGIJE**



Završni rad

**Clean arhitektura sa CQRS i Mediator dizajn paternima – primeri dobre prakse**

Mentor: Student:

dr Svetlana Anđelić Radoš Perović 17/17

Beograd

Oktobar,

2022

**Sadržaj**

REZIME ............................................................................................................3

KLJUČNE REČI ...................................................................................................3

Srediti

**REZIME**

Ovim radom prikazan je razvoj i implementacija backend aplikacije po pravilima I dobrim praksama clean arhitekture, praćena kombinacijom CQRS I Mediator dizajn paterna što predstavlja jedan od najvećih nivoa arhitekture u softverskom svetu, čime se postiže brz i lak razvoj novih zahteva, maksimalna održivost i skalabilnost softvera, olakšano i kvalitetno testiranje kao i jako velika čitljivost koda.

**KLJUČNE REČI**

C#, Clean Architecture, Hexagonal Architecture, CQRS, Mediator, Entity Framework, NET 6, SSMS.

**1. UVOD**

U ovom radu prikazan je primer izrade back-end aplikacije koja prati standarde clean arhitekture i agilnog razvoja softverskog proizvoda. U pitanju je demo veb aplikacija koja simulira rad bilo koje platforme za vodjenje poslovanja, gde imamo osnovnu biznis logiku kao što je kreiranje I izmena proizvoda, kreiranje I narucivanje narudžbenica I brisanje istih. (dodati jos koju recenicu)

U prvom delu rada opisan je arhitekturalni stil, kao I tehnologije i paketi koji su korišćeni pri izradi aplikacije. Objašnjen je REST API arhitekturalni stil, a zatim su predstavljene I

navedene sledeće tehnologije: .NET 6, Microsoft Sql Server management Studio, kao i paketi: Entity Framework, MediatR, LINQ, Microsoft.DependencyInjection, SwaggerUI.

Opisana je njihova kratka istorija I po čemu su se odvojili od konkurencije.

Ovakav način izrade back-end aplikacije postaje priroretna stvar kod svakog softverskog inžinjera, a samim tim i kod svake firme. Velika skalabilnost, održivost I dobra postavka arhitekture koja kasnije donosi brze rezulate, smatraju se ključem produktivnosti inžinjera kao I zadovoljstvom klijenata

**3. TEHNOLOGIJE I PAKETI**

**3.1. Tehnologije**

Za implementaciju demonstracionog projekta je korišćen C# jezik, .NET 6 kao platforma za razvoj veb aplikacije i Microsoft SQL Server Management Studio za kreiranje skladišta podataka i samo skladištenje.

**C#**

C# je programski jezik nastao 2000. Godine pod okriljem Majkrosofta, potpuno baziran na principima objektno orjentisanog programiranja. C# je nastao u želji da se kreira novi jezik koji će upotpuniti nedostatke C++ i pojednostavi složenosti navedenog jezika. Sama sintaksa C# ima puno sličnosti sa C++ jezikom [4].

Verzija koja se trenutno koristi je 10.0, u novembru 2022 očekuju se izlazak nove 11.0 verzije.

**.NET 6**

.NET 6 je okruženje za kreiranje aplikacije razvijeno od strane Majkrosofta. Trenutna aktualna verzija ja 6, očekuje se nova verzija 7 u novembru 2022 gde će se predstaviti i nova verzija C# 11.0. Verzija .NET 6 je naslednik verzije .NET 5 koja je sjedinila .NET Framework i .NET Core, što znači da sadrži alate za kreiranje multiplatformskih aplikacija različitih vrsta (veb, mobilne, desktop..). Takođe su podržani i jezici C#, F# i Visual Basic [5].

**Microsoft SQL Server Managment Studio**

SQL Server Managment Studio je integrisano okruženje razvijeno od strane Majkrosofta i koristi se za upravljanje svim SQL infrastrukturama. Pri izradi projekta je korišćena verzija SSMS 18 [6].

**3.2. Paketi**

**--ispisati**

Ispisati pakete kad se odradi aplikacija – sta je sve korisceno, kratka istorija i zasto se izdvajaju od konkurencije, zasto sam bas njih izabrao

**2. ARHITEKTURALNI STIL RAZVOJA SOFTVERA**

**2.1 API**

API je skraćenica od naziva Application Programming Interface**,** koji predstavlja skup definicija i protokola koji služe za kreiranje i integraciju softverske aplikacije. API, kao što mu i ime sadrži, predstavlja vid interfejsa – granice nečega što nam je dato na korišćenje bez da znamo kako konkretna implementacija izgleda i kako je napravljena.

**Primer API-ja u realnom svetu**

Da bi bolje razumeli primenu i primer API-ja u softverskog svetu, navešćemo jedan primer API-ja u realnom svetu. Ako posedujemo automobil, ono što bi se moglo nazvati pojmom API, je upravo volan našeg automobila. Uz automobil je isporučen volan, kao deo automobila koji je poznat po upotrebi ljudima koji ga koriste a koji nisu inženjeri, u prevodu znamo da koristimo volan bez da znamo kako je taj volan povezan sa točkovima i kako on to konkretno radi, što znači da su u prevodu inženjeri koji su kreirali taj model automobila – nama upravo kreirali API, vid interfejsa koji je nama poznat za korišćenje i koji obavlja određen posao, kako ga obavlja to nije u našem domenu i to nas ne treba da interesuje.

**Primer API-ja u softverskom svetu**

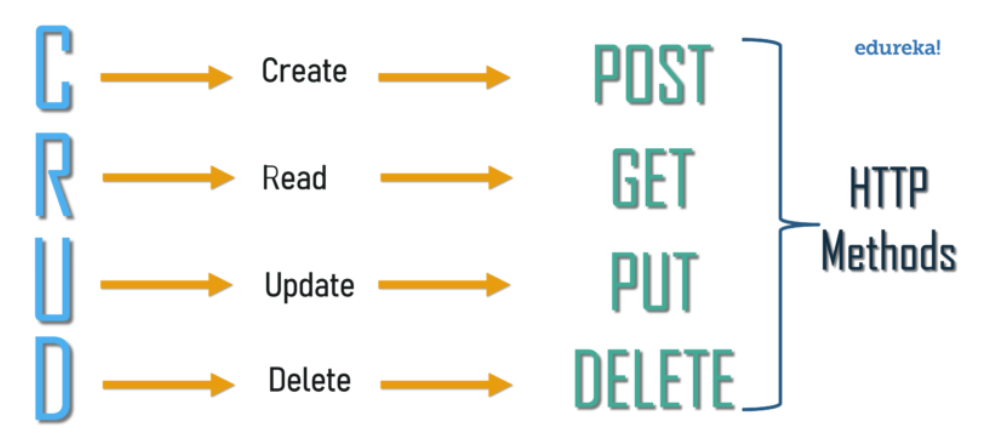
API u softverskom svetu postoji na mnogo nivoa. Funckije neke klase koje su izložene korišćenju predstavljaju API te klase, npr konkretne funckije koje nam je Entity Framework dao na korišćenje (.AddRange, Where, SaveDatabaseChanges) predstavljaju ustvari API biblioteke Entity Framework.  
Ako pogledamo konkretnu arhitekturu, tačke (funckije) u kodu koje predstavljaju ulaz u jezgro ili ti core arhitekture, predstavljaju API sloja jezgra naše arhitekture.

Pored svih nivoa postojanja pojma API u softverskom svetu, najzastupljeniji je pojam koji se odnosi na celokupnu aplikaciju. Aplikacija se posmatra kao „black box“ [1] koja prima, procesuira i vraća neke podatke. Programer je zadužen da napravi ulazne tačke u aplikaciji, gde će ona primati podatke, koje kasnije može procesuirati i vratiti u nekom formatu. Kako se podaci obrađuju, upisuju i vraćaju je samo problem programera, ne i krajnjeg korisnika koji će koristiti tu aplikaciju – što je savršen opis pojma API.

**2.2 REST API**

REST API poznat i kao RESTful API je arhitekturarni stil koji se koristi za kreiranje API-ja koji koriste HTTP protokol za komunikaciju [2]. HTTP protokol sadrži osnovne metode koje predstavljaju CRUD operacije[3] kao što su :

* GET (Read) za čitanje podataka
* POST (Create) za kreiranje novih podataka
* PUT (Update) za izmenu postojećih podataka
* DELETE (Delete) za brisanje podataka



Slika 1. CRUD operacije i HTTP metode

REST API je skup pravila koji se koristi pri pisanju API (aplikacije), kako bi se postigla uniformnost različitih sistema, što dovodi do lakšeg i bržeg međusobnog komuniciranja.

REST API rešava problem:

* Skalabilnosti, sistemi koji implementiraju REST API arhitekruralni stil, mogu efikasno da se skaliraju jer ovaj arhitekturalni stil optimizuje interakciju izmedju klijenta i servera tako što uklanja zahvete iz prošlosti jer i ne mora da ih poseduje, sa druge strane dobro organizovano keširanje podataka će smanjiti čestu i ponavljanu interkaciju između klijenta i servera, što dovodi do podržavanja skalabilnosti tj uklanjanja uskih grla u komunikaciji koja smanjuju performanse.
* Fleksibilnosti, potpuno razdvajanje klijent – server usluga. Pojednostavljanje i razdvajanje serverskih komponenti tako da svaki deo može da se razvija nezavisno. Promene platforme ili tehnologije na serverskoj strani ne utiču na klijentsku aplikaciju.
* Nezavisnosti, ovakvi servisi su nezavisni od tehnologije koja se koristi. Moguće je pisati i klijentske i serverske aplikacije na različitim programskim jezicima bez uticaja na dizajn API-ja. Takođe, moguće je promeniti tehnologiju na bilo kojoj strani bez uticaja na komunikaciju.

Diagram, website

Description automatically generated

Slika 2. REST API model

**4. ARHITEKTURA SOFTVERA**

Arhitektura softvera opisuje organizaciju ili strukturu sistema i njegovo ponašanje. Predstavlja vid projektovanog nacrta kako će nas sistem izgledati često je upoređujući sa građevinskim nacrtima i arhitekturom.

Često se postavlja pitanje zašto je postavka arhitekture bitan, ako ne i najbitniji korak kreiranja softverskog rešenja, gde mnogi programeri i menadžeri ne vide njen uticaj i dovode u pitanje potrebno vreme za njeno dobro projektovanje.

**4.1 “Čista” arhitektura**

“Čista” arhitektura je pojam koji se u softverskom inžinjerstvu odnosi na arhitekturu koju karakteriše odvajanje biznis logike (ponašanja aplikacije) od ostalih komponenti kao što su tip aplikacije (dekstop, veb..), izvor čuvanja podataka (različite vrste baza podataka, fajl..), vrsta pozivnih resursa.

Ključ i cilj ovakvog tipa arhitekture je da se biznis logika same aplikacije totalno odvoji od svega što je prati, što u prevodu znači da je upravo biznis logika tj ponašanje naše aplikacije ulazna tačka kreiranja ovakvog sistema.   
Najbitnija nam je informacija šta će naš sistem da radi, a koji će tip aplikacija biti i gde ćemo i kako upisivati podatke to nam je informacija koja će nam trebati na samom kraju izrade našeg rešenja kada je već sva logika rada sistema proverena i ispisana.

Ovakav vid gledanja na softverski proizvod, dovodi do puno pozitivnih i neophodnih posledica koje su ključne pri kreiranju sistema, među kojima su najbitnije:

* Lako dodavanje novih zahteva i funkcionalnosti - Jedna od jako bitnih karakteristika koje nam donosi “čista” arhitektura, želimo da naš sistem bude lako nadogradiv i to nam mora biti jako bitno. Retko kada ćemo imati situaciju da nam klijent pre projektovanja samog sistema izjasni sve svoje želje i da ćemo se pridržavati toga, projektovanje softvera je poznato po svojoj raznovrsnosti, i naš sistem mora biti lako nadogradiv, a ne fiksiran, kako bismo ispunili sve klijentove želje i uspešno odradili svoj posao.
* Lako održavanje i izmena postojećih funkcionalnosti - želimo da naš sistem bude elastičan i dobro organizovan. Kao što je dodavanje novih zahteva učestala stvar u softverskom projektovanju, tako je i izmena postojećih funckionalnosti. Ne želimo doći u situaciju da naš sistem bude robustan i fiksiran, gde bi svaka potencijalna promena dovela do kraha sistema ili oduzimanja velike količine vremena kako bismo ga osposobili. Sistem mora biti elastičan.
* Skalabilan - naš sistem treba biti tako projektovan, da ni u jednom trenutku veličina sistema ne utiče na brzinu i kompleksnost isporuke novih zahteva. Svaka nova funckionalnost se treba smatrati kao jedan detalj koji će se lako priključiti jezgru našeg sistema.
* Testiranje sistema – sistem sa dobrom i “čistom” arhitekturom je pogodan za testiranje, što je jako bitno uspostaviti na samom početku projektovanja proizvoda. Pored toga što testiranje otkriva nesmotrene i nepokrivene slučajeve korišćenja sistema, takođe nam je upravo sposobnost testiranja softvera garancija da je sve ono što smo prethodno napisali ispravno, testirano i stabilno. Kada dodajemo novu funckionalnost ili menjamo trenutnu, testovi su nam garancija da nismo pokvarili ranije napisanu funkcionalnost sistema dodavanjem ili menjanjem postojećeg koda.

Postoje tri trenutno prihvaćene “čiste” arhitekture:

* Hexagonal Architecture (Ports and Adapters) by Alistair Cockburn 2005 [7]
* Onion Architecture by Jeffrez Palermo 2008 [8]
* Clean Architecture by Robert C. Martin 2012 [9]

Ova tri pristupa “čistoj” arhitekturi razlikuju nazivi pojedinih stvari i sama organizacija foldera, ali im je cilj zajednički I zasniva se na istim principima.

**Principi “čiste arhitekture”**

Kao što već znamo, jedan od ciljeva “čiste” arhitekture je da odvoji ponašanje aplikacije od spoljašnjih faktora, a to ćemo postići razdvajanjem sistema u više slojeva, gde svaki sloj ima svoj zadatak I ulogu u sistemu. Često se slojevi aplikacije dele u zasebne projekte (Class Library-je) kako bi se postigla nezavisnost i laka promena okruženja.

Sistem se deli na 4 osnovna sloja, a to su :

* Domenski sloj (Domain layer)
* Aplikativni sloj (Application layer)
* Infrastrukturni sloj (Infrastructure layer)
* Prezentacioni sloj (Presentation layer)

Diagram

Description automatically generated

**Domenski sloj**

Ovaj sloj je srce samog sistema. U njemu se nalazi kod koji objašnjava ponašanje aplikacije, njenu biznis logiku i to je ujedno i najkompleksniji i najbitniji sloj sistema.

Kreiranje nekog sistema bi trebalo početi od ovog sloja, uglavnom oduzima najviše vremena- sa jedne strane jer je potrebno utvrditi i najsitnije detalje ponašanja sistema od klijenta, dok sa druge strane uglavnom predstavlja komplesne algoritme, izračunavanja i logiku koja zahveta puno vremena i rada.

Domenski sloj nema zavisnost (referencu) ni na jedan drugi sloj.

U njemu se nalaze entiteti “entitites”, klase koje predstavljaju fektorije “factories” i interfejsi ka repozitorijumima “repositories”.

**Aplikativni sloj**

Sloj koji se zasniva na domenskom sloju, i zavisi od njega. Koristi i diriguje ponašanjem domenskog sloja. Sadrži u sobe interfejse koji će predstavljati kuku na koju će se ostala 2 sloja kačiti svojom implementacijom. Aplikativni sloj u celini ima zavisnost od svojih interfejsa i domenskog sloja.

Domenski i aplikativni sloj zajedno čine “core” ili ti jezgro aplikacije.

U njemu se nalaze hendleri “handlers“ i njihovi interfejsi, servisi “services” i njihovi interfejsi, DTO – ovi [5], interfejsi koji će eksterni servisi u druga dva sloja koristiti za implementaciju.

**Infrastrukturni sloj**

Sloj koji služi za sve eksterne servise. U aplikativnom sloju se nalazi interfejsi koji se koriste u aplikativnim hendlerima, a u ovom sloju je konkretna implementacija tih interfejsa – čime se postiže veoma laka promena u celom sistemu.

Uzmimo za primer promenu baze podataka, ako aplikativni sloj koristi domenski interfejs koji nam predstavlja upis i ispis u bazu podatka, za konkretnu promenu koja će rezultirati promenom baze uopšte nije neophodno dirati deo koja koji se nalazi u „core“ ili ti jezgru sistema. Takva promena će se izvršiti u infrastrukturnom sloju gde se konkretna implementacija povezivanja sa bazom i nalazi.

Zavisi od aplikativnog i domenskog sloja.

U njemu se nalaze konkretne implementacije za:

* povezivanje sa bazom
* povezivanje na eksterni servis
* logovanje
* upisivanje i čitanje fajlova
* povezivanje na servis bas „service bus“

**Prezentacioni sloj**

Sloj koji predstavlja GUI (Graphical User Interface) [6]. Sloj koji se kači na aplikativni sloj, koristi njegove servise i od njih, u zavisnosti od slučaja, dobija neki rezultat.

Zavisi od aplikativnog i domenskog sloja.

Predstavlja različite vidove aplikacija koje su najbliži klijentu:

* veb
* desktop
* konzolne

Diagram

Description automatically generated

Chart, diagram

Description automatically generated

Putanja kontrole bi trebala da izgleda tako da :

1. Prezentacioni sloj zove aplikativni

2. Aplikativni zove domenski

3. Aplikativni zove infrastrukturni

4. Aplikativni vrati podatak prezentacionom

5. Prezentacioni ispise podataka

**Chart, line chart

Description automatically generated**

**AKO BUDE BILO POTREBE – opisati svaku arhitekturu pojedicno. Razlike su samo u imenovanju foldera i drugim nazivima za iste stvari.**

**4.2. Clean Architecture – Klin Arhitektura**

**4.2 Hexagonal Architecture**

**4.3 Onion Architecture**

**6. DIZAJN PATERNI**

Dizajn paterni su ponovna iskoristiva rešenja koja se mogu primeniti na problem koji se često ponavlja u razvoju softvera.

Kada želimo prevesti englesku reč “pattern” (patern), dobijamo prevod šablon, uzorak, obrazac. To nam govori da je dizajn patern nešto što možemo “štancati” i koristiti više puta, što nam daje sledeće prednosti:

* Paterni su uglavnom dokazana rešenja – koristićemo isproban, testiran i optimizovan pristup rešavanju problema u razvoju softverskog proizvoda, baziran na iskustvu i radu programera koji su doprineli razvoju tog paterna.
* Lako ih je ponovo iskoristiti – paterni su uglavnom pojednostavljena rešenja nekog čestog problema, a činjenica da možemo da ih prilagodimo svojim potrebama čini ovaj koncept moćnim.
* Paterni su ekspresivni – za svaki od njih se uglavnom vezuje i određena struktura i specifičan rečnik, što čini saradnju I komunikaciju među programerima lakšom i efikasnijom.

Paterni predstavljaju neki vid već iskorišćenog rečenja za ponavljani problem, logika koju možemo koristiti nebrojano puta.

Jako bitna stavka koja je ujedno I najteža za savladati kada se govori o dizajn paternima je kada I kako prepoznati pravi trenutak I mesto gde iskoristiti patern. U suprotnom, kompleksnost projekta nepotrebno raste, samim tim I njegovo održavanje I testabilnost što naravno dovodi do povećanih troškova.

Pored poznavanja samog koncepta određenih dizajn paterna, ekvivalentno je bitno I iskoristiti ih na pravom mestu u pravo vreme.

Dizajn paterni se mogu kategorizovati u 3 osnovne vrste:

* Kreacioni dizajn paterni
* Strukturalni dizajn paterni
* Bihevioralni dizajn paterni

**6.1 Kreacioni paterni**

Ovi paterni se bave kreacijom objekata, na način prilagođen određenoj primeni.

Posebno su važni u situacijama u kojima bi uobičajen pristup kreiranju objekata doveo do povećanja kompleksnosti projekta.

Neki od paterna koji spadaju u ovu grupu su:

* Constructor,
* Factory Method,
* Abstract Factory,
* Prototype
* Singleton
* Builder

**6.2 Strukturalni paterni**

Strukturalni paterni se bave kompozicijom i obično predstavljaju različite načine za definisanje odnosa među objektima. Oni obezbeđuju da kada je neophona promena u jednom delu sistema, ostatak sistema ne mora da se menja. Takođe pomažu da svaki deo sisstema radi ono čemu je najbolje prilagođen.

Neki od paterna koji spadaju u ovu grupu su:

* Decorator,
* Facade,
* Flyweight,
* Adapter,
* Proxy

**6.3 Bihevioralni paterni**

Bihevioralni paterni su paterni koji poboljšavaju komunikaciju između različitih objekata u sistemu. Akcenat je na ponašanju konkretnog objekta i lakoj promeni njegovog ponašanja kao i dodavanju novog, bez rušenja starog koda. Svaka izmena je samo dodatak na osnovu.

Neki od paterna koji spadaju u ovu grupu su:

* Mediator,
* Strategy,
* Observer,
* Iterator,
* Visitor

Timeline

Description automatically generated

**6.3.1 Mediator dizajn patern**

Mediator dizajn patern je patern koji spada pod grupom Biheviralnih paterna.  
Njegovo postojanje omogućuje da više objekata komunicira između sebe bez da su upoznati sa strukturom onog drugog. Definiše objekat koji enkapsulira način na koji se objekti ponašaju. Postoji kako bi smanjio kompleksnost komunikacije između više objekata.

Mediator objekat je centar zbivanja radnje, objekat koji sadrži listu svih svojih članova i koji ih povezuje na elegantan način praveći između njih „labavu vezi“ [10] ili ti nezavisnost.

Uglavnom se koristi u vidu logike čet sobe, gde jedan član čet sobe šalje poruku celoj grupi (obaveštava) ili konkretnom drugom članu.

Diagram

Description automatically generated

Svoju primenu u C# je našao u obliku MediatR paketa, koji se pokazao kao najelegantniji i najprostiji način upotrebe Mediator dizajn paterna.

Konkretna implementacija Mediator centralnog objekta se nalazi u samom paketu, ali logika funckioniše po principu da se na osnovu Request<T> (T je konkretan tip klase Response-a) kreira instanca klase RequestHandler<T> i pokreće njegovu funciju Handle().

Konkretan RequestHandler<T> predstavlja jedan Use Case [11].  
Funkcija Handle() sadrži sekvence koda koje predstavljaju biznis logiku konkretnog UseCase.

Diagram

Description automatically generated

**6.4 Arhitekturalni paterni**

Pored osnovnih dizajn paterna koji su vezani za rešavanje konkretnog problema u kodu ili ponašanju nekog dela aplikacije, postoje i paterni koji se koriste u višem nivou apstrakcije i vezani su za arhitekturu.

Koriste se u rešavanju problema koji se nalaze na većem sloju infrastrukture samog sistema.

Neki od paterna koji se nalaze u grupi arhitekturalnih paterna su:

* CQRS
* Client-server
* MVC
* REST
* Peer to peer

**6.4.1. CQRS arhitekturalni patern**

Pojam CQRS (Command and Query Responsibility Segregation) je smišljen u cilju razdvajanja dva osnovna tipa operacija:

* Čitanje podakata nekog sistema (Query)
* Izmena podataka sistema (Command)

CQRS je arhitekturalni patern koji govori o razdvajanju operacija koje predstavljaju neko čitanje i operacije koje predstavljaju neke izmene sistema (kreiranje, menjanje ili brisanje podataka).

Ovakvo podela dve vrste operacija nam donosi

* Mogućnost da različiti timovi mogu raditi na implementaciji sistema. Tim koji je bolje upoznat sa biznis logikom radiće na implementaciji komandi, a drugi tim bi bio zadužen za čitanje podataka.
* Svaka od dve operacije se može skalirati po svojim potrebama. Operacija čitanja iz sistema je uglavnom više korišćena nego operacije izmene sistema.
* Svaka od dve operacije može imati svoj nivo zaštite. Posmatraju se kao dve razdvojene celine projekta.

Diagram

Description automatically generated

Operacije koje predstavljaju neke izmene sistema se drugačije zovu komande, i definišu ih sledeće stvari:

* Gotovo uvek koriste domenski sloj aplikacije (domensku logiku)
* Uglavnom ne vraćaju nikakav podatak nakon uspešnog izvršavanja
* Rade neku izmenu sistema, brisanje, kreiranje ili izmena postojećih podataka, što znači da je ovakav vid logike pogodan za detaljno testiranje koristeći unit testove.
* Zahtevaju tim koji dobro poznaje biznis logiku, operacije izmene sistema su kompleksnije i teže za implementaciju nego operacije čitanja, i zato uglavnom oduzimaju više vremena.
* Uglavnom se za upis, izmenu i brisanje podataka koriste relacione baze, zbog svoje lake povezanosti tabela i olakšane manipulacije podataka.

Operacije koje predstavljaju čitanje podataka iz sistema se drugacije zovu kveriji, i definišu ih sledeće stvari:

* Ne koriste domenski sloj aplikacije, što znači da se ne koristi biznis logika, implementacija je jako jednostavna.
* Uvek moraju vratiti neku vrednost, to im je poenta.
* Nema potrebe testirati ih interno, ne postoji biznis logika koja bi se mogla testirati unit testovima.
* Potreban je tim koji je tehnički potkovan, često se koriste razne metode optimizacije i keširanja podataka u cilju što bržeg i boljeg čitanja podataka.
* Za ovaj vid operacija koriste se nerelacione baze podataka, zbog svoje jednostavnosti i jako velike razlike u čitanju podataka poredeći sa relacionim bazama podataka. Upravo je u njima optimizacija.

Kompletna primena CQRS arhitekturalnog paterna zahveta upotrebu dve različite baze podataka. Prilikom izvršavanje operacija komandi koristi se relaciona baza podataka (npr SQL Server), dok se prilikom čitanja podataka koriste nerelacione baze podataka (npr Cosmos Db). Ključan deo ove ideje je držati ove dve baze u apsolutnoj sinhronizacij, što je upravo i njen glavni problem.

Savet je koristiti Event Store principe kako bi dve baze bile konzistentne, što može dovesti do nepotrebne kompleksnosti sistema.

Treba biti pažljiv prilikom modelovanja sistema, jer ako nam sistem nije dovoljno kompleksan i gabaritan, CQRS arhitekturalni patern bi u tom slučaju bio pogrešan izbor.

**Diagram

Description automatically generated**

**7. KONKRETNA APLIKACIJA**

**6.1. Baza podataka**

**6.2. Organizovanje foldera**

**6.3. Slike koda i swaggera..**

**8. ZAKLJUČAK**

Na primer: Ovakav vid organizovanja I modelovanja back-end aplikacije pokupio je za sobom razne načina tumačenja nivoa kompleksnosti ove arhitekture I načina izrade softverskog proizvoda, ali je neosporna činjenica da donosi veliku skalabilnost, održivost I stabilnost softvera, što su glavne odlike kvalitetnog softverskog proizvoda.

**9. LITERATURA**