Sadržaj

[Domain Driven Design 2](#_Toc156129843)

[Uvod u Domain Driven Design 2](#_Toc156129844)

[Bounded context 3](#_Toc156129845)

[Ubiquitous language 4](#_Toc156129846)

[Anemičan i bogat domenski model 5](#_Toc156129847)

[Razumevanje domena 5](#_Toc156129848)

[Entitet i value object pattern-i 5](#_Toc156129849)

[Aggregate pattern 7](#_Toc156129850)

[Aggregate root pattern 7](#_Toc156129851)

[Repository pattern 7](#_Toc156129852)

[Domain event (domenski događaj) 8](#_Toc156129853)

[Integration event 9](#_Toc156129854)

[CQRS 9](#_Toc156129855)

[CQRS i DDD 11](#_Toc156129856)

[Mikroservisi 12](#_Toc156129857)

[Message queue 14](#_Toc156129858)

[RabbitMQ 15](#_Toc156129859)

[Outbox pattern 16](#_Toc156129860)

[Literatura 16](#_Toc156129861)

# Domain Driven Design

## Uvod u Domain Driven Design

**Domain Driven Design (DDD)** predstavlja pristup u razvoju softvera koji se fokusira na razvoj **modela domena nekog problema.** Pritom se teži razumevanju procesa i pravila samog domena.

DDD pruža principe i šablone za rešavanje složenih problema. Tako se dobija **jasan, čist, testabilan kod** koji predstavlja neki domen. Međutim, glavni cilj nije pisanje koda, već **rešavanje problema**. To zahteva puno rada sa klijentima kako bi se shvatile njihove potrebe i razumeo domen problema. Neophodan je neprestani rad tima za razvoj softvera sa **domen ekspertima**-ljudima koji su stručnjaci za određeni domen.

Jedan od glavnih principa prilikom rada sa DDD-em je: **“Divide and conquer“** – “Zavadi pa vladaj“. Ovaj princip je takođe poznat i kao: “**Separation of Concerns**“ – razdvajanje odgovornosti. Odnosno, potrebno je podeliti domen na manje delova i fokusirati se na deo po deo. Tako se dobijaju takozvani subdomeni (**subdomain**). Svaki od njih je potrebno **modelovati** i rešiti problem sa kojim se on susreće.

**Benefiti DDD-a** su:

* fleksibilan softver (lako dodavanje novih funkcionalnosti)
* jasna putanja prilikom rešavanja kompleksnog problema
* čist kod
* kod lak za održavanje
* biznis logika je na jednom mestu
* korisni design pattern-i

Iako Domain Driven Design pruža navedene benefite, treba se jako pažljivo koristiti. Nije uvek najbolje rešenje koristiti DDD. Kao što i sam **Eric Evans**, začetnik Domain Driven Desgn-a kaže, DDD se treba koristiti samo kod kompleksnih problema gde se i mogu uvideti pravi benefiti DDD-a. U suprotnom, rad sa DDD-em može postati **overengineering**-razvoj proizvoda koji je bespotrebno ukomplikovan. Osim ove **mane**, kao manu definitivno treba navesti to da sa korišćenjem DDD-a se dobija jedno “kompleksnije“ rešenje koje je teže i **vremenski zahtevnije za testiranje**. Takođe, korišćenje svih procesa DDD-a zahteva veće i konstantno angažovanje stake holder-a i domain expert-a.

S obzirom da je DDD jedan jako kompleksan pristup, postoji nešto što se zove **DDD Mind Map**. On prikazuje koncepte i pattern-e DDD-a, njihovu interakciju i položaj. Mind Map-u je moguće videti na sledećoj slici:

A diagram of a company

Description automatically generated

## Bounded context

Bounded context predstavlja centralni pattern Domain Driven Design-a. Prilikom rešavanja kompleksnih problema, model podataka može postati jako veliki. Zbog toga DDD deli veliki model na više različitih delova, a svaki deo se zove **Bounded context**. To je zapravo granica (boundary) u okviru domena gde se konkretni domenski model primenjuje. Bounded context sprečava širenje nekih koncepta modela tamo gde im nije mesto (u druge bounded context-e).

Najpravilnije korišćenje DDD-a bi značilo da svaki Bounded context ima različite podatke, kod, kao i tim koji radi na razvoju modela jednog Bounded contexta. Međutim, u praksi se jako retko dešava da postoji baš ovoliki nivo odvajanja.

**Bounded context i subdomain** su dosta slični, ali opet dosta različiti koncepti. Subdomain predstavlja **prostor problema**, odnosno kako je odlučeno da se podeli poslovna logika, ili neka domenska aktivnost. Sa druge strane bounded context je **prostor rešenja**, odnosno kako je softver i razvoj tog softvera organizovan za rešavanje problema. Uglavnom se ova dve prostora poklapaju, ali ne mora nužno da znači uvek. Eric Evans daje jedan konkretan primer. Neka se posmatra soba čiji je pod potrebno prektiti tepihom. Soba je prostor problema, dakle, predstavlja subdomain, a tepih je prostor rešenja. Problem je moguće rešiti na više načina. Prvi način je prekriti sobu sa tepihom identičnog oblika kao i pod sobe i onda se subdomain i Bounded context odnose na istu stvar. Drugi način bi bio prekriti sobu sa više različitih tepiha čija površina ne pokriva čitavu površinu sobe. U ovom slučaju se subdomain i Bounded context ne poklapaju. [1]

Uglavnom postoje više različitih Bounded contexta. **Context map** demonstrira kako su povezani Bounded context-i međusobno kroz komunikaciju između timova koji rade na posebnim Bounded context-ima.

Pravilo je da svaki Bounded context ima svoju bazu podataka. Objašnjenje za to daje Eric Evans u svojoj knjizi Domain Driven Design:

“Ako ste u kompaniji koja deli bazu podataka i tu bazu podataka ažurira na hiljade procesa, veoma je teško kreirati model po DDD-u i pisati softver koji radi nešto konkretno sa ovim modelom“. [3]

Kada pričamo o mikroservisnoj arhitekturi takođe postoji jedno (nepisano) pravilo da svaki servis ima svoju bazu podataka. Kao što će biti kasnije objašnjeno, ovde se uviđa i paralela između Bounded context-a i mikroservisa. **Uglavnom** je deljenje odrađeno na sledeći način, **ali** **nije pravilo**: **1 Bounded context - 1 mikroservis - 1 baza**.

**Shared kernel** (zajednička osnova) predstavlja još jedan jako bitan koncept kod DDD-a. Odnosi ne na način interakcije između različitih bounded context-a. Sastoji se od skupa domenskog modela koji je zajednički za više različitih bounded context-a. Na ovaj način se taj zajednički domenski model deli između više različitih bounded context-a. Zove se “kernel“ zato što predstavlja osnovne/kernel elemente između context-a.

## Ubiquitous language

Ubiquitous language je ključni pojam u DDD-u. On predstavlja **zajednički jezik** između tehničkih i netehničkih osoba koje rade na rešavanju nekog problema putem DDD-a. Dakle, koriste ga svi učesnici razvojnog procesa nekog proizvoda, domain eksperti, developeri i ostale zainteresovane strane za razvoj proizvoda. Pomaže prilikom razumevanja domena problema.

Pomoću ubiquitous language-a se dobija konzistentna terminologija. Termini koji se koriste u kodu se poklapaju sa terminima koje koriste domen eksperti tokom diskusije o domenu problema. Klase, imena, metode i ostali bitni termini dolaze od domenskih eksperta i ubiquitous language-a.

Domenski eksperti igraju ključnu ulogu u razvojnom procesu kao što je već pomenuto. Ubiquitous language im pomaže prilikom komunikacije da developeri razumeju sam problem. Na kraju se dobija bolji softver.

Neophodno je pomenuti da ubiquitous language nije statičan i konačan. Razvija se tokom vremena kako celokupni tim uči i razume problem. Međusobna komunikacija članova tima i konstantne povratne informacije treba da budu prisutne kako bi se menjao i poboljšavao ubiquitous language.

Različiti bounded context-i u sistemu mogu da imaju svoj poseban ubiquitous language. Takođe, neki termini iz jednog bounded context-a mogu da imaju totalno neko drugo značenje u drugom bounded context-u. Baš zbog toga je bounded context i jako bitan kod DDD-a, zato što on omogućava postojanje i međusobno funkcionisanje različitih modela.

Tehnički aspekti prilikom modelovanja bounded contex-a su isti kao i pattern-i prilikom programiranja problema.

## Anemičan i bogat domenski model

Anemičan (anemic) i bogat (rich) domenski model predstavljaju 2 jako često korišćena termina kod DDD-a.

**Anemičan** domenski model je domenski model koji se fokusira na **stanje objekata**. Generalno gledano, nema ničega pogrešnog prilikom korišćenja anemičnog domenskog modela kada je neophodno implementirati sistem sa najosnovnijim CRUD operacijama. Ali, ukoliko je odlučeno da će se raditi po DDD-u, domen je već previše kompleksan za najobičnije CRUD operacije. Tako da anemički domenski mode **anti patern u DDD-u**.

**Bogat** domenski model se sastoji od ponašanja i poslovne logike u domenu. Dakle, prevedeno na implementaciju softvera, domenski modeli (klase), osim čuvanja stanja objekata, implementiraju kroz funkcije i metode svu potrebnu poslovnu logiku. Prilikom rada sa DDD-em, definitivno treba težiti bogatom domenskom modelu. [1]

## Razumevanje domena

Da bi se razumeo domen nekog problema, najbolje je posmatrati događaje koji se dešavaju kod tog problema. Postoji jedan jako poznat metod razumevanja domena putem posmatranja događaja. On se zove **event storming**. Ovaj metod generalno nije deo DDD-a, ali se jako često koristi uz DDD kako bi se razumeo model nekog kompleksnog sistema.

Event storming se može definisati kao **radionica** za istraživanje domena i dizajniranje sistema. Ideja je povezati ljude sa različitih pozicija, domain experte, developere i ostale zainteresovane strane. Oni zajedno modeluju tok događaja koji se javljaju u nekom procesu-domenu problema.

Event storming je **event-centric model**. Fokusira se na događaje koji se javljaju, a događaj se može definisati kao promena stanja u sistemu.

Event storming uglavnom kreće sa sesijama koje imaju za cilj posmatranje šire slike sistema-big picture, pa se zatim prelazi na detalje.

Događaji se vizualizuju po vremenskoj osnovi (liniji), a kako radionica napreduje učesnici pronalaze agregate (entitete) i komande-akcije koje izazivaju događaje.

Ovaj metod u originalu ne zahteva ni računar. Naime, učesnici ove radionice lepe papire (**sticky notes**) po zidu. [4]

## Entitet i value object pattern-i

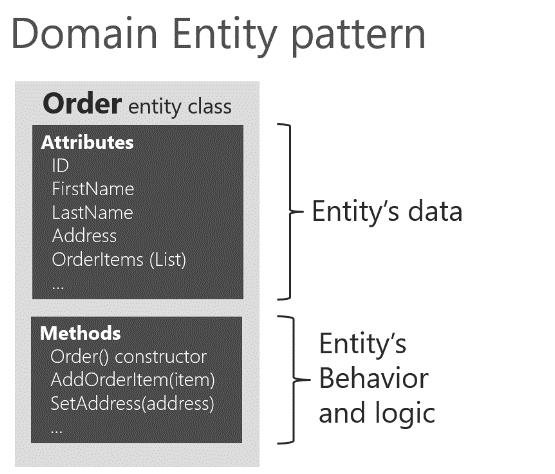
Postoje 2 glavna gradivna elementa u DDD-u koji su maltene osnova svih ostalih. To su entiteti i value object-i. Njihova glavna razlika je to po cemu se definišu. **Entiteti se definišu putem identiteta, a value object-i putem svojih vrednosti** (pa odatle i naziv value object, value=vrednost).

Entitet predstavlja objekat DDD-a koji je moguće pratiti, locirati, čuvati i pribaviti. Kako bi se entitet pratio kroz vreme neophodno je dobiti odgovor na pitanje: **“Da li je ovaj entitet isti?“**. Postoje različiti atributi koji mogu da ukažu na odgovor. Na primer, ukoliko se posmatra atribut nekog entiteta *status* i neka njegove vrednosti budu *aktivni, neaktivni, na čekanju.* Može se dobiti poprilično puno informacija o entietima koji su aktivni ako se raspolaže domenskim znanjem. Međutim, ovakav jedan atribut ne odgovara do kraja na pitanje “Da li je ovaj entitet isti?”. [5] Neophodno je razlikovati sve entitete jednog istog statusa, a za to je najpogodniji uvek **ključ-jedinstveni identifikator**. Takođe, ostali atributi entiteta mogu da se promene, tako da se ni oni ne mogu koristiti za identifikaciju objekata. Ključ se sa druge strane ne menja, pa je pogodan kao identifikator. Da bi se došlo do adekvatnog ključa, neophodno je dobro poznavanje domena.

Entitet je moguće modelovati u više različitih bounded context-a (pa samim tim i mikroservisa). To ne znači da će isti entitet (sa istim atributima i logikom) biti implementiran na identičan način u više različitih bounded context-a. Na primer, ukoliko posmatramo neki web shop sa mikroservisnom arhitekturom i izdvojenim bounded context-ima, entitet kupca u nekom zamišljenom *Identity* delu treba da poseduje sve atribute jedne osobe. Sa druge strane, u nekom *ordering* delu, taj isti kupac će imati manje atributa zato što nisu baš svi atributi jedne osobe neophodni kako bi se obavila jedna online kupovina.

Da ne bi model bio anemičan, već bogat, neophodno je da entiteti implementiraju i svoje ponašanje, a ne samo takozvane “data atribute“ (gett-ere i sett-ere).

Sve ove informacije je moguće grupisati u jedan “pattern“ DDD-a: **Domain Entity pattern**, a to je moguće i videti na sledećoj slici: [6]



Value object nema svoj identifikator. To je skup atributa bez identiteta, a sam value object je nepromenljiv. Ukoliko je potrebno promeniti neki od atributa value object-a, praksa je da se kreira novi value object, a ne menja konkretno samo 1 atribut value object-a (**nepromenljivost value object-a**). [7]

Value object-i nose informaciju o tome šta su oni, a ne ko su oni. Zato im i ne treba identifikator i shodno tome nalaze i primenu u DDD-u. Najčešče su neke brojne vrednosti i stringovi value object-i, ali to nije pravilo, mogu i dosta složenije grupe atributa da budu value object-i.

Interesantno je primetiti da ponekada entitet iz jednog bounded context-a nije entitet i u drugom, već value object. Razlog je to što objekti imaju različito značenje u različitim bounded context-ima. Na primer, adresa u nekoj web shop aplikaciji ne mora da ima identitet. To je grupa atributa korisnika sistema koja nosi informaciju o tome gde on živi ili gde želi da mu se pošiljka dostavi. Ovde nije neophodno postojanje identiteta, pa je samim tim ovo value object, a ne entitet. Sa druge strane, adresa u nekom sistemu za praćenje potrošnje električne energije, adresa predstavlja dosta kompleksnu instancu. Ovde je adresa korisnika skup atributa bitnih za biznis logiku. Neophodan je i identifikator da bi sistem za naplatu računa mogao da poveže koristnika sa njegovom adresom, tako da je ovde adresa definitivno entitet. [7]

## Aggregate pattern

Aggregate predstavlja skup od više domenskih objekata (entiteta i value object-a) koji se mogu tretirati kao jedinica i koji imaju određenu funkcionalnost. Na primer, jedna narudžbina (order) i njeni elementi narudžbine (order item) predstavljaju jednu celinu-**aggregate**. [8]

Aggregate je osnovna jedinica transfera izvora podataka-zahteva se učitavanje ili čuvanje čitavog aggregate-a. Transakcija ne bi trebalo da izlazi iz okvira jednog aggregate-a, pa se shodno tome i dizajniraju aggregate-i.

Identifikacija aggregate-a može biti teška. Aggregate je grupa objekata koji moraju biti konzistentni zajedno. [6]

## Aggregate root pattern

Svaki aggregate ima makar jednu komponentu koja predstavlja **aggregate root**. To je ujedno i glavna komponenta. Aggregate može imati pored aggregate root-a i druge entitete i value object-e.

Svaka referenca ka aggregate-u treba da ide preko aggregate root-a. Aggregate root garantuje integritet aggregate-a. Takođe garantuje i konzistentnost, pa svako ažuriranje aggregate-a treba da ide preko aggregate root-a. Dakle, da bi se promenio bilo koji entitet u aggregate-u, mora se ići preko aggregate root-a. Ukoliko se promeni neki entitet ili value object van aggregate root-a, on više ne može da garantuje da je aggregate u validnom stanju. [6]

## Repository pattern

Svaki bounded context raspolaže (domenskim) informacijama koje je u nekom trenutku potrebno perzistirati. Za to su zadužene komponente za perzistiranje podataka.

Repository pattern je DDD pattern zadužen za **perzistiranje izvan domenskog modela**. Apstrakcije za perzistiranje podataka (interface) su definisane ili u domenskom delu, ili u aplikativnom delu (ako se radi sa clean architecture-om poznatim design pattern-om koji se jako često kombinuje sa DDD [9]). Ove apstrakcije imaju svoje implementacije u drugim delovima aplikacije zaduženim za infrastrukturni deo. Implementacije repozitorijuma su zapravo klase koje enkapsuliraju logiku neophodnu za pristup podacima. Centralizuju funkcionalnosti pristupa podacima i tako se dobija razdvajanje infrastrukture i tehnologije korišćene za pristup podacima iz izvora podataka (baza podataka najčešće, ali ne i uvek).

Repository pattern je jedan poznati pattern za rad sa izvorom podataka i nije nešto što je osmišljeno konkretno za DDD (ali je i ovde našlo primenu). Repository je enterprise pattern koji funkcioniše kao posrednik između domenskog modela i mapiranja podataka. Klijentski objekti doslovno kreiraju upite i šalju ih repozitorijumima kako bi dobili odgovor. Repozitorijumi enkapsuliraju set objekata koji se nalaze u izvoru podataka (bazi) i operacije koje je moguće izvršiti nad njima. [10]

Repository pattern u DDD-u je usko povezan sa aggregate patternom. Za svaki aggregate (pa samim tim i aggregate root) je neophodno kreirati jedan repository. U mikroservisnoj arhitekturi baziranoj na Domain Driven Design-om, repository je jedino mesto koje se koristi za ažuriranje baze podataka. To je zbog veze 1:1 repository:aggregate root, a aggregate root kontroliše konzistentnost aggregate-a.

Logika je da se preko repozitorijuma popune podaci u memoriji iz baze podataka u obliku domenskih entiteta. Kada su entiteti u memoriji, oni se mogu promeniti i posle opet preko repozitorijuma perzistirati u bazu podataka.

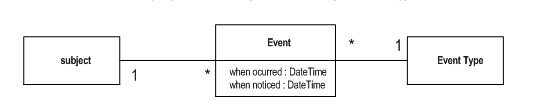
## Domain event (domenski događaj)

Martin Fowler na interesantan način objašnjava domenske događaje [11]:

“Odlazim u Babur (indijski restoran) da večeram u četvrtak i plaćam kreditnom karticom. Ovo može biti modelovano kao događaj, gde je tip događaja *kupovina*¸ gde je subjekat kreditna kartica i trenutak dešavanja je četvrtak. Ako Babur koristi stari ručni sistem za plaćanje, novčana transakcija se neće obaviti do petka, pa će trenutak obaveštavanja biti petak“.

Stvari se dešavaju. Nisu sve interesantne. Neke su vredne beleženja, ali ne izazivaju nikakvu reakciju. Posebno su interesantne one koje izazivaju reakciju. Mnogi sistemi moraju da reaguju na njima zanimljive događaje.

Pattern domenskog događaja je moguće videti na sledećoj slici:



Subject (subjekat) je onaj koji pokreće čitav događaj. On može pokrenuti više događaja, ali jedan konkretan događaj pokreće samo jedan subjekat. Događaj ima trenutak dešavanja (when ocurred) i trenutak obaveštavanja (when noticed). Svaki događaj ima svoj tip (event type), dok više različitih događaja mogu biti istog tipa.

Korišćenjem domain event-a se implementiraju bočni efekti usled promena koje su se desile u domenu. Ovi bočni efekti se odnose na različite aggregate-e. Kada se koriste domain event-i postiže se eventualna konzistentnost sistema.

**Event (događaj)** je nešto što se dogodilo u prošlosti. **Domain event (domenski događaj)** je nešto što se desilo u domenu i neophodno je da i ostali delovi istog domena budu svesni toga, odnosno budu obavešteni o tome. Kada budu obavešteni, oni reaguju na ranije definisan način.

Preko domenskih događaja se prenose domenska pravila bazirana na ubiquitous language-u koja se dobijaju od domenskih eksperata. Domenski događaji pružaju razdvajanje zaduženja (**separation of concerns**) između klasa istog domena.

Domenski događaji su slični događajima baziranim na poruke (message-style events), ali postoji jedna razlika. Kod pravih sistema zasnovanih na porukama, message queue-ima, message brokerima, poruka je namenjena delovima koji integrišu više bounded context-a, mikroservisa ili događaja iz više aplikacija. Sa druge strane, domenski događaji su korisni kada je potrebno obavestiti o događaju, ali su bočni efekti potrebni u okviru istog domena. [12]

## Integration event

Pored domain event-a (domenskih događaja), postoje i **integration event-i (integracioni događaji)**. U osnovi su to 2 iste stvari: “**notifikacija** o nečemu što se desilo“. Međutim, njihova **implementacija mora biti različita**. Domenski događaji su samo poruke objavljene preko nekog dispečera domenskih događaja (in-memory mediator). Sa druge strane, svrha integracionih događaja je propagirati transakciju i ažurirati/obavestiti druge podsisteme/mikroservise/bounded context-e/druge aplikacije...Treba napomenuti da do integracionih događaja treba doći isključivo ako se entitet uspešno perzistirao u sistemu gde se desio događaj.

Domenski događaji mogu da budu sinhroni i asinhroni, dok integracioni događaji moraju biti asinhroni. [12]

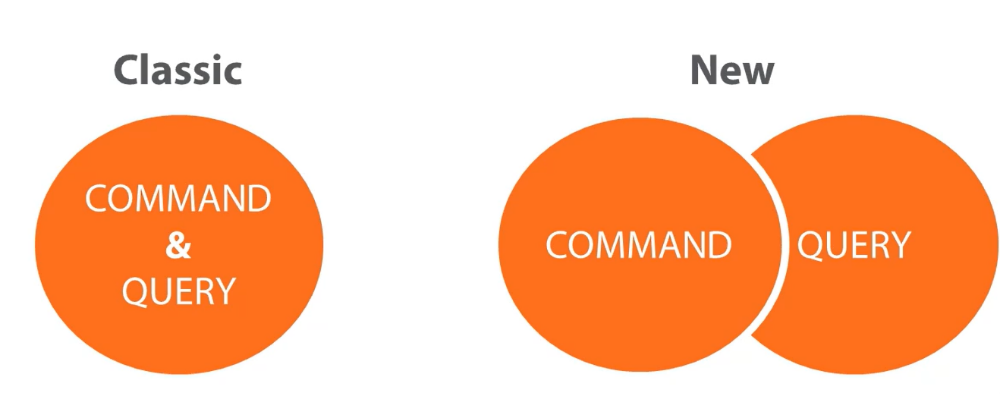
DOGAĐAJI NAJČEŠĆE KORISTE MESSAGE BROKER-E ZA OBAVEŠTAVANJE IZMEĐU RAZLIČITIH MIKROSERVISA.

## CQRS

CQRS je arhitekturni pattern koji razdvaja model za čitanje i upis podataka. Sličnu stvar je pružao i **CQS-Command Query Separation**. Osnovna ideja je da se podele operacije sistema u 2 kategorije:

1. query (upit) – operacije vraćaju rezultat/podatke i ne menjanju stanje sistema (bez bočnih efekata)
2. command (komanda) – operacije koje menjaju stanje sistema (i “idealno“ ne vraćaju rezultat/podatke) [13]

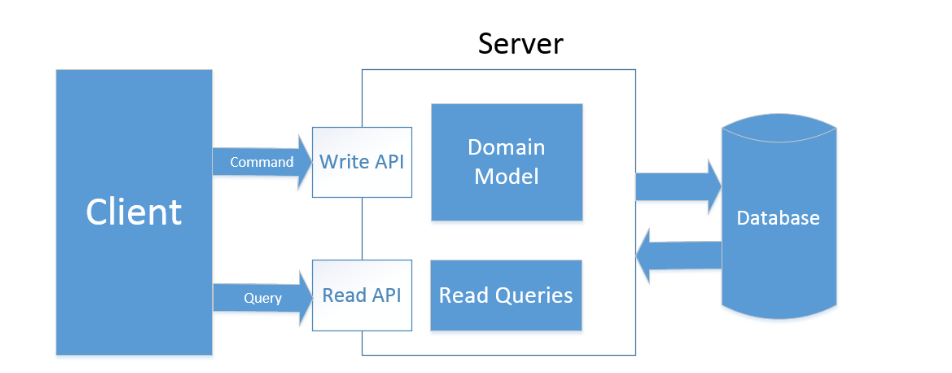
Na narednoj slici je moguće videti simboličnu demonstraciju razdvajanja upita i komandi:



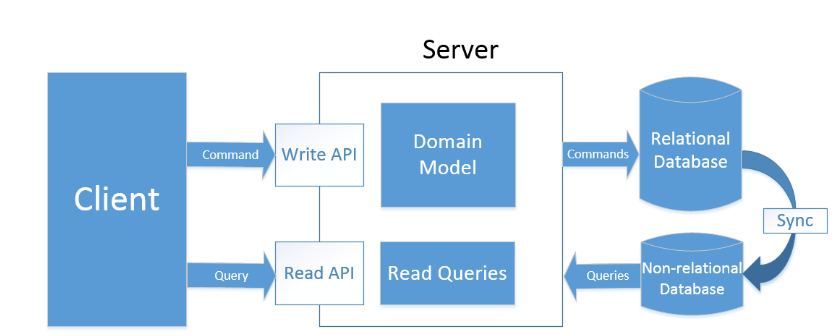
Dakle, CQS je jednostavniji koncep. Reč je o metodama posvećenim jednom objektu. Svaka metoda ili vraća stanje ili menja stanje, ali nikako oba! Čak i ako se radi samo sa repozitorijum pattern-om, moguće je implementirati CQS i odvojiti operacije čitanja i upisa.

**CQRS – Command Query Responsibility Segregation** predstavlja sve što predstavlja i CQS-u, ali je detaljniji. To je pattern zasnovan na query-ima, command-ama i event-ima. Postoje više različitih tipova CQRS-a [14]:

1. **razdvajanje strukture klasa** – klase za čitanje i upis su različite (poseban DTO)
2. **razdvajanje modela** – posebni su modeli i API pozivi za čitanje i upis. Ovde su u okviru domenskog modela operacije kreiranja, ažuriranja i ostale operacije koje će kasnije biti korišćene od strane komandi. Za upit se koristi takozvani *read query* – odnosno dto za operacije čitanja. Ovo objašnjenje je moguće videti na sledećoj slici:

****

1. **razdvajanje baza** – po mnogima pravi CQRS. Kreiraju se posebna baze podataka za upite druga posebna baze podataka za komande. Ovde je potencijalni problem konzistentnost podataka između 2 baze. Zbog toga se obavlja sinhronizacija. Ona se obavlja kao proces u pozadini i može da zahteva neko vreme kako bi se kompletno sinhronizovalo. Zato se ovde dobija **eventualna** **konzistentnost**. Na sledećoj slici je moguće videti dijagram ovog tipa CQRS-a:



Prednosti korišćenja CQRS-a:

* bolja prilagođenost modela – razdvajanjem komandi i upita se postiže zasebne optimizacije čitanja i upisa podataka.
* brže izvršenje operacija čitanja i upisa – neke NoSQL baze podataka su optimizovane za upise, a neke za čitanje. Ukoliko se koristi CQRS sa razvojenim bazama, moguće je iskoristiti bazu optimizovanu za čitanja kao bazu za query deo, a bazu optimizovanu za upis iskoristiti za command deo.
* Bolja skalabilnost – može se skalirati samo query ili command infrastruktura ako je neophodno – nezavisne su.
* Lakše održavanje koda – razdvojenost upita i komandi pruža jednu čistu arhitekturu aplikacije.

Mane CQRS-a:

* Povećava vreme razvoja – kroišćenje CQRS-a može povećati vreme inicijalnog razvoja aplikacije usled izazova oko različitog modela, validacije podataka i održavanja konzistentnosti.
* Moguća nekonzistentnost – ukoliko se koristi CQRS sa 2 baze za upite i komande, definitivno će u nekom trenutku biti nekonzistentnost sistema, samo je pitanje da li je za sistem to prihvatljivo.
* Bespotrebna složenost za jednostavne sisteme.
* Teže privikavanje novih developera na projektu koji implementira CQRS ukoliko nisu upoznati sa CQRS-om.

## CQRS i DDD

CQRS i DDD su 2 različite stvari. CQRS je arhitekturni pattern, dok je DDD pristup u razvoju softvera koji se fokusira na razvoj modela domena nekog problema. Međutim, ova 2 koncepta se jako često koriste zajedno, a njihova kombinacije može doprineti boljem modelovanju i arhitekturi sistema.

DDD promoviše concept agregata kao središnje jedinice za održavanje konzistentnosti podataka. Da bi se odžala konzistentnost podataka, operacije idu preko aggregate root-a, a kod CQRS se komande i upiti mogu napraviti baš na nivou aggregate root-a i tako podržati DDD.

DDD i CQRS zajedno podržavaju asinhronu komunikaciju preko događaja, što može biti korisno za modelovanje sistema koji reaguju na promene.

DDD promoviše modeliranje granica (bounded) context-a kako bi se jasno odredile granice konzistentnosti u domenskom modelu. Sa druge strane, CQRS omogućava odvajanje upita i komandi što može bolje izraziti granice i konzistentnost.

CQRS podržava odvajanje modela za upite od modela za komande. Ova odvojenost modela omogućava različite modele za čitanje i pisanje podataka, što je u skladu sa principima DDD-a gde se modelovanje vrši prema kontekstu.

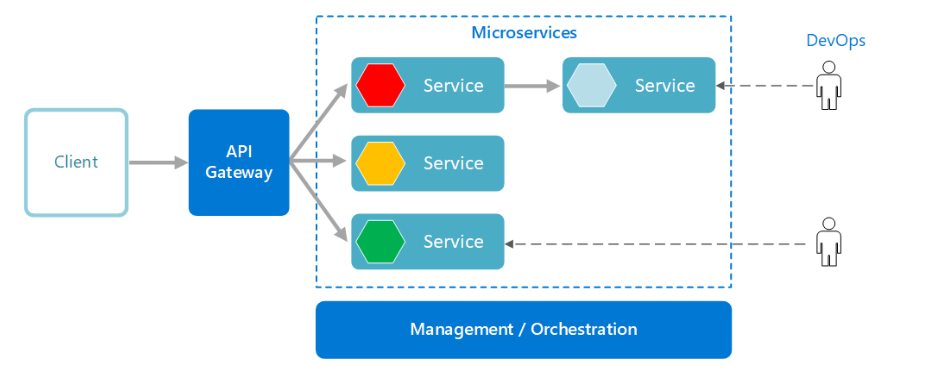
# Mikroservisi

**Mikroservisi** su arhitekturni pristup razvoja softvera gde se softver sastoji iz više malih nezavisnih servisa koji komuniciraju međusobno.

Mikroservisi su **samo-održivi** i zasebna celina za sebe-ne treba da zavise direktno od ostalih mikroservisa. Svaki mikroservis se **nezavisno deploy-**uje. To je jako velika prednost zato što nije neophodno rebuild-ovanje i redeploy-ovanje i ostalih delova sistema. Promena u okviru jednog mikroservisa ne treba da poremeti neki servis sa kojim taj mikroservis radi, sve dok ostaje kompatibilan sa eksternim API pozivima i interface-ima za poruke. Dakle, mikroservis treba da ima **granicu (boundary)** oko sebe. U okviru te granice mikroservis se treba fokusirati na svoja specifična ponašanja, a podatke može modelovati kako njemu odgovara (kako odgovara poslovnoj logici). Svaki mikroservis ima svoj neki **context**,terminologiju i jezik koji se koristi za definisanje pojmova. Zbog svega toga se **jako često mikroservisi tretiraju kao bounded context-i.** Naravno, ne mora uvek da postoji preslikavanje 1:1.

**Mikroservisna** arhitektura omogućava bolje i lakše skaliranje, brž razvoj i smanjuje vreme potrebno za izbacivanje novog feature-a.

Na narednoj slici je moguće videti strukturu mikroservisne arhitekture [15]:



Centralna tačka su definitivno **(mikro)servisi**. Servisi treba da budu **loosely coupled** – “labavo povezan“. Različiti delovi sistema su projektovani tako da zavise jedni od drugih što manje. Nepisano pravilo je da jedan mali tim (developera) radi na razvoju i održavanju jednog servisa. Takođe je poželjno da svaki servis bude u posebnom projektu kako bi se razvoj lakše obavljao između timova. Svaki servis je zadužen za perzistiranje svojih podataka. To povlači činjenicu da je za svaki od servisa neophodna posebna baza podataka. To pruža mogućnost izbora baze podataka za svaki servis koja odgovara fukcionalnim zahtevima servisa i tako se može povećati kvalitet i efikasnost svakog servisa. Svaki od servisa se može razvijati posebnom tehnologijom, programskim jezikom i framework-om. To je posledica posebnog deploy-a svakog servisa.

Servisi međusobno komuniciraju. Ta komunikacija može da bude **asinhrona** (preko nekog **message queue**-a), ili **sinhrona** (preko klasičnih **API poziva**).

Na prethodnoj slici je moguće videti i deo **management/orchestration**. Ova komponenta je zadužena za deploy-ovanje aplikacije na konkretne čvorove, identifikaciju otkaza čvorova, rebalansiranje čvorova,…Uglavnom je ovo već gotova komponenta koja se koristi, na primer: **Kubernetes, Docker Swarm**.

**Devops** predstavlja osobu sa određenim programersko-infrastrukturnim zaduženjima na projektu (deploy, podešavanje pipeline-a,…). Međutim, to je mnogo veći pojam koji obuhvata teoriju, praksu i alate zadužene za ubrzanje razvoja softvera spajanjem programersko-infrastrukturnih zaduženja. Kada se pominje devops i generalno mikroservisi, nemoguće je nepomenuti i **docker**. Docker predstavlja sistem za deliver/deploy servisa (jako pogodno za mikroservise). Koristi u pozadini virtualizaciju na nivou operativnog sistema. Za isporuku sistema se koriste “paketi” koji se zovu **kontejneri (containers)**.

**API gateway** predstavlja jako bitan deo svakog mikroservisnog sistema. Nije obavezan, ali ga je poženo imati. On predstavlja ulaznu tačku klijenata u servis. Dakle, umesto direktne komunikacije klijenata sa servisima, klijenti pozivaju API gateway koji prosleđuje pozive konkretnim servisima.

Prednosti korišćenja API gateway-a su:

* Razdvaja klijente od servisa, tako da servisi mogu da budu verzionisanji i refaktorisani (čak i njihovi potpisi u API-u) bez potrebe za ažuriranjem kod klijenata
* Servisi mogu da koriste protokole za razmenu poruka koji nisu podržani od strane WEB-a (na primer AMQP-Advanced Message Queuing Protocol).
* API gateway takođe može da posluži i kao centralizovano mesto za obavljanje funkcionalnosti koje to zahtevaju/dozvoljavaju. Na primer, API gateway je parvo mesto za autentifikaciju, logovanje, load balancing, validaciju, keširanje,…

Neki od **benefita** korišćenja mikroservisa su:

* **agilnost**. Pošto se svaki mikroservis deploy-uje nezavisno, lakše je ispravljati bug-ove i kreirati release-e. Moguće je ažurirati servis bez ponovnog deploy-a čitave aplikacije. Takođe, ako pođe nešto po zlu, potrebno je samo rollaback-ovati 1 servis pošto su mikroservisi međusobno nezavisni. Kada se pojavi bug u jednom delu tradicionalne monolitne arhitekture on može blokirati čitav release proces. Nova funkcionalnost može biti zadržana i nedeploy-ovana zbog tog bug-a sve dok se on ne reši, integriše, testira i publishuje.
* **mali fokusirani timovi.** Mikroservis treba da bude dovoljno mali da bi jedan tim mogao da ga razvija, održava, testira i deploy-uje. Mali timovi su agilniji i efikasniji. Kod većih timova je uglavnom teža i usporena komunikacija, koja na kraju krajeva i usporava development.
* **Miks tehnologija**. Svaki tim može za svoj mikroservis da izabere tehnologiju koja najviše odgovara servisu, ili timu zbog njihovog znanja i veština.
* **Izolacija**. Ako jedan od mikroservisa postane nedostupan, to neće napraviti problem u čitavoj aplikaciji. Ako servisi komuniciraju međusobno sinhrono preko api poziva (nepreporučljivo), moguće je implementirati **retrial policy**, pa probati sa zahtevom nakon nekog vremena. Ako servisi komuniciraju asinhrono preko nekog message queue-a (preporučljivo) samo drugi servis neće pročitati poruku koja je njemu namenjena. Tako će doći do nekonzistentnosti u sistemu. To je ponekad bitno, a ponekad nije. Ukoliko je bitno, postoje brojne tehnike za sprečavanje nekonzumiranja poruka kada su sistemi nedostupni, na primer **transactional outbox pattern** o kome će kasnije biti više reči u radu.
* **Skalabilnost**. Servisi se mogu skalirati nezavisno. To dozvoljava sa se skaliraju delovi aplikacije koji zahtevaju više resursa bez skaliranja čitave aplikacije (što nije slučaj sa monolitnim aplikacijama). Korišćenjem orkestracionih alata kao što je Kubernetes na primer, moguće je to skaliranje obavljati i automatski.
* **Izolacija podataka**. Mnogo je lakše obaviti ažuriranje šeme podataka pošto se ta šema odnosi samo na 1 mikroservis. Kod monolitnih aplikacija, ažuriranje šeme postaje jako izazovno zato što je mogu koristiti i različiti drugi delovi. Kod klasičnih, dugoodržavanim monolitnih arhitektura, posle određenog vremena monoliti postaju preveliki i svaka promena je rizik od uvođenja velikog bug-a.

Benefiti mikroservisa su neosporni, međutim, postoje i mnogi **izazovi** sa kojima se susreće prilikom njihovog korišćenja:

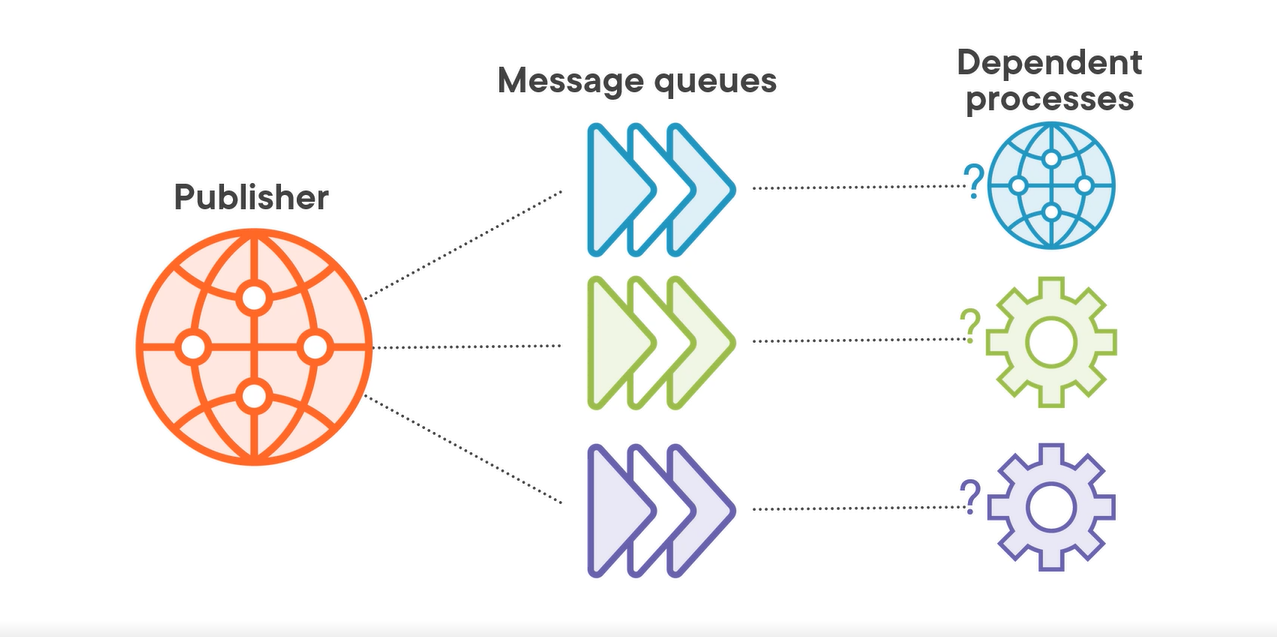
* **Kompleksnost**. Mikroservisna aplikacija ima puno aktivnih/promenljivih delova. Svaki deo (mikroservis) je jednostavniji, ali čitav sistem postaje vremenom dosta kompleksan i neophodno je jako dobro domensko znanje kako bi se svi servisi povezali i gradili smislenu celinu.
* **Nedostatak upravljanja**. Decentralizovana arhitektura definitivno ima svoje prednosti, ali može da vodi do problema. Mikroservisna aplikacija onda može da preraste u aplikaciju sa puno različitih programskih jezika, tehnologija i framework-a tako da celokupna aplikacija može da postane teška za održavanje. Poželjno je da postoje definisani barem neki standardi i ograničenja kako aplikacija ne bi prerasla u nešto jako teško za održavanje.
* **Zagušenje mreže i kašnjenje**. Korišćenje više malih, granuralnih servisa može dovesti do toga da postoji jako puno međuservisne komunikacije. Lanac zavisnosti može postati jako dug. Na primer servis 1 poziva servis 2 koji poziva servis 3,… i sve to da bi se došlo do jedne funkcionalnosti.

Osim mikroservisne, postoji i **monolitna** arhitektura. Kod nje su svi procesi usko povezani i izvršavaju se kao jedan servis. To znači da ukoliko jedan process aplikacije ima trenutno veću potrebu za korišćenjem od strane korisnika, neophodno je skalirati čitavu aplikaciju. Takođe, dodavanje i ispravljanje postojećih funkcionalnosti u monolitnoj arhitekti vremenom postaje jako zahtevno.

## Message queue

U mikroservisnoj arhitekturi su različite funkcionalnosti podeljene između više različitih servisa. Ovi servisi su međusobno nezavisni, ali zajedno čine celinu, pa je potreban neki vid povezanosti. Uglavnom servisi ne mogu da kompletno izvrše jednu funkcionalnost bez pomoći ostalih servisa. Baš zbog toga je neophodno da sistem ima mehanizam koji omogućava sesrvisima da ostanu u kontaktu bez tradicionalnog request-response mehanizma. **Asinhrona komunikacija** koja se postiže preko message queue može da reši ovaj problem kod mikroservisa.

Message queue je koncept asinhrone komunikacije koji omogućava komunikaciju između servisa preko slanja poruka u **queue (red)**. On omogućava neko privremeno skladište podataka između sender-a/pošiljaoca/publisher-a i receiver-a/primaoca. Na ovaj način pošiljalac može da nastavi sa nesmetanim radom čak i kada primalac nije dostupan.



Queue je red/linija stvari koje čekaju na obradu u sekvencijalnom redu počevši od početka reda. Message queue (red poruka) je queue sa porukama poslatim između aplikacija.

Message (poruka) je podatak transportovan između pošiljaoca i primaoca. Kod mikroservisne arhitekture su najčešće poruke zapravo integracioni događaji koji obaveštavaju neki od servisa da krenu sa obradom nekog konkretnog zadatka.

Najkorišćeniji message queue-ovi su: RabbitMQ, Kafka, Heron, Amazon SQS,... [16]

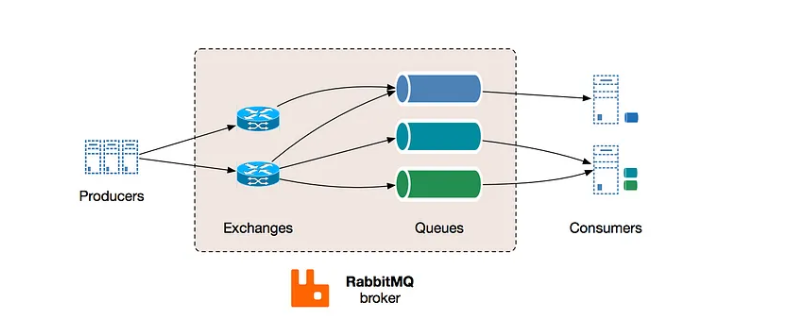
### RabbitMQ

RabbitMQ je jedan od najpoznatijih message queue-ova koji nalazi primenu kao takozvani “mailman” u mikroservisnoj arhitekturi.

RabbitMQ se sastoji od:

1. Producer-a – klijent koji kreira poruku
2. Consumer-a – prima poruku
3. Queue-a – čuva poruke
4. Exchange-a – rutira poruke i šalje ih na queue

Na sledećoj slici [16] je moguće videte arhitekturu RabbitMQ-a:



RabbitMQ sistem funkcioniše na sledeći način:

1. Producer kreira poruku i šalje je na exchange
2. Exchange prima poruku i rutira je na queue koji je pretplaćen na exchange
3. Consumer prima poruku preko queue-a na koji je on pretplaćen

### Outbox pattern

Kada se u nekom mikroservisu desi neka promena koju drugi mikroservisi treba da zna, mikroservis kreira događaj koji opisuje tu promenu. Događaj se putem poruke upisuje u message queue i drugi servis će pročitati i obaviti očekivanu aktivnost. Problem nastaje ukoliko je trenutno drugi mikroservis nedostupan i on neće moći da pročita poruku sa message queue-a. U tom slučaju je poruka trajno izgubljena i sistem će zauvek biti u nekom vidu **nekonzistentnosti**.

Međutim, postoji mogućnost da se izbegne gubitak poruka preko takozvanog **outbox pattern-a**. Umesto da se odmah upiše poruka o događaju u message queue, upisuje se u **lokalnu outbox tabelu** baze podataka servisa koji je generisao događaj. Postoji periodičan **process** koji s vremena na vreme proverava outbox tabelu i šalje događaj drugim mikroservisima preko message queue-a. Nakon uspešnog slanja događaja, on se može obeležiti kao uspešno završen u outbox tabeli. Mikroservis koji je primio događaj preko message queue-a ima kod sebe u bazi podataka **inbox** tabelu u kojoj pamti informaciju o asinhronom primanju događaja.

Ovaj pristup omogućava mikroservisima da komuniciraju asinhrono i smanjuje direktnu zavisnost između mikroservisa. Takođe pomaže u rešavanju problema sa konzistentnošću u situacijama gde slanje događaja i ažuriranje lokalne baze podataka može dovesti do nesaglasnosti i ovako se dobija jedna **distrubuirana transakcija**.

## API gateway

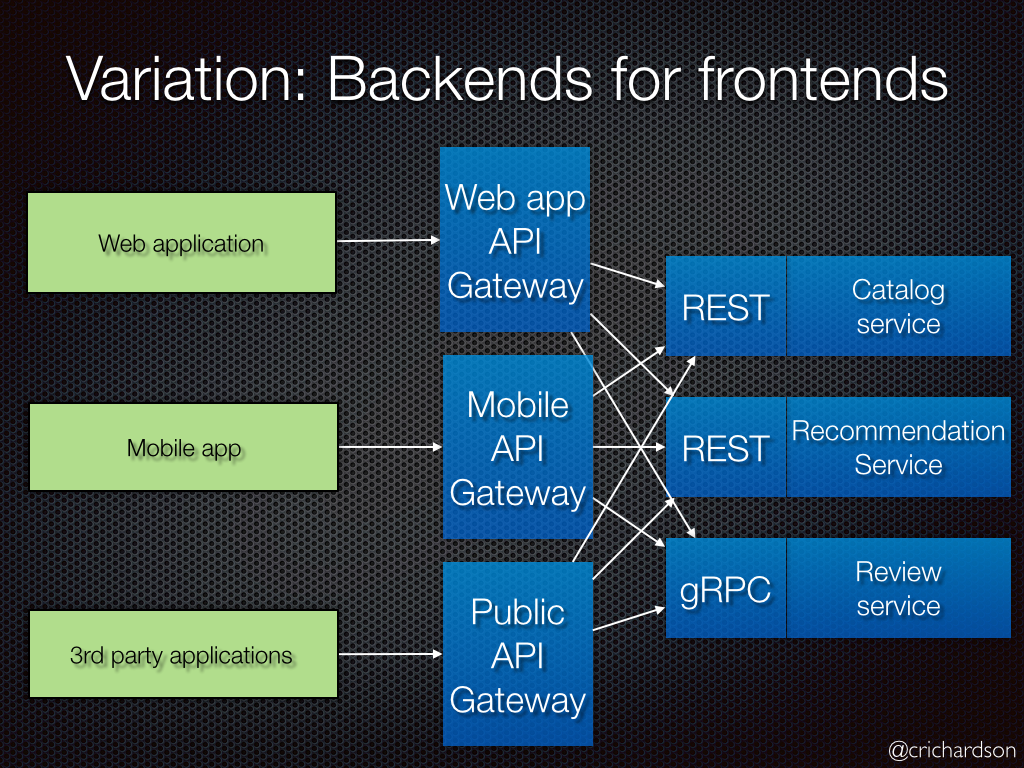
API gateway je **centralizovani servis** u arhitekturi mikroservisa koji pruža jednostavan i konzistentan pristup različitim mikroservisima i njihovim funkcionalnostima putem jedinstvenog **interfejsa**. Dakle, klijenti najčešće ne pristupaju direktno konkretnom mikroservisu, već preko API gateway-a. Ovo olakšava rad klijentima zato što nisu direktno zavisni od mikroservisa i ne moraju sa svakim zasebno komunicirati.

U današnje vreme dinamičnog razvoja softvera postojanje verzija endpoint-a je jako često. Kako klijent ne bi morao da menja svoj kod za pristup servisu, API gateway upravlja verzijama API-a.

API gateway kao centralizovano mesto u mikroservisnoj arhitekturi je pogodan za upravljanje bezbednošću, autentifikacijom, autorizacijom,…

Broj API gateway-a u sistemu može da zavisi od arhitekture i potreba sistema. Neretko se dešava da čak jedan mikroservisni sistem ima više API gateway-a. Podela se najčešće radi po tipu klijenta. Na primer, poseban API gateway za web aplikaciju, poseban za mobilnu aplikaciju,… Ovo može olakšati prilagođavanje funkcionalnosti API-ja prema potrebama svake vrste klijenta u sistemu.

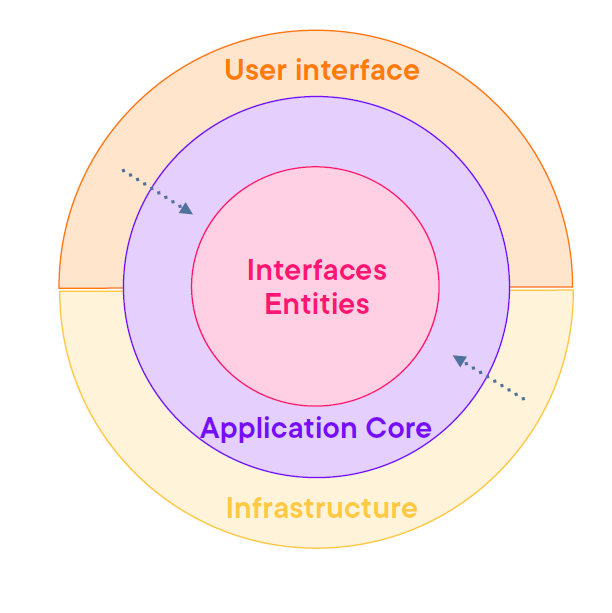
Na narednoj slici je moguće videti primer jednog mikroservisnog sistema gde postoji API gateway i za mobilnu i za web i za 3rd party aplikacije. One preko API gateway-a pristupaju različitim servisima koji mogu da budu implementirani preko različitih tehnologija i mehanizama (REST, gRPC), a API gateway-i sve to apstrakuju [17]:



# Ticket4U

Ticket4U predstavlja jedan web sistem za rezervaciju ulaznica. Frontend je urađen preko React-a. Backend je zasnovan na mikroservisnoj arhitekturi, gde je svaki sistem implementiran po Domain Driven Design pattern-ima. Takođe, svaki mikroservis prati clean architecture pattern. Clean Architecture je arhitekturni obrazac koji je razvijen kako bi omogućio razvoj softvera sa čistom organizacijom koda i lakoćom održavanja. Ovaj obrazac je definisao Robert C. Martin, poznat i kao "Uncle Bob". Clean Architecture promoviše jasnu podelu odgovornosti između različitih slojeva aplikacije i tako omogućava nezavisno testiranje. Promena jednog sloja ne utiče na druge delove sistema.

Clean architecture pattern se sastoji od više slojeva, koncentričnih kružnica. Unutrašnji krug predstavlja najviši nivo apstrakcije i ne zavisi od detalja implementacije unutrašnjih slojeva. Slojeve, kao i strelice koje ukazuju na smer zavisnosti (međusobno korišćenje slojeva) je moguće videte na sledećoj slici:



**Unutrašnji krug** se sastoji od **entiteta** sa poslovnom logikom koji su nezavisni od bilo kakvih spoljnih uticaja. Pored entiteta, ovde su i **interfejsi (interfaces)** koji se koriste za komunikaciju između entiteta i upotrebljavaju se u spoljašnjem sloju. Unutrašnji krug sadrži i takozvane **use case**-eve/poslovnu logiku. Kao takav, unutrašnji krug je pogodan za **implementaciju DDD pattern-a**.

**Središnji krug** predstavlja **Application Core** – centralni deo aplikacije sa aplikativnim servisima. Ovi servisi se koriste za povezivanje unutrašnjeg kruga sa spoljašnjim slojem, a servisi koriste entitete i ostale stvari iz unutrašnjeg kruga.

**Spoljašnji krug** je moguće podeliti na 2 dela:

1. User interface
2. Infrastructure

I**nfrastructure (infrastrukturni) deo** je zadužen za pristup eksternim sistemima i bazama podataka. **User interface** sadrži sve što je potrebno za interakciju sa korisnicima. To su uglavnom **kontroleri** kao tačka pristupa klijenata čitavoj aplikaciji.

U Ticket4U postoje 3 servisa:

1. Shows
2. Reservations
3. Users

Shows je katalog šouova. Šou predstavlja neki umetničko-muzički zabavni događaj. (Namerno se koristi šou-show-izbegnut je naziv event-događaj kako se ne bi mešalo sa pojmom događaja u DDD-u).

# Literatura

* <https://app.pluralsight.com/library/courses/fundamentals-domain-driven-design/table-of-contents> [1]
* [https://martinfowler.com/bliki/BoundedContext.html [2](https://martinfowler.com/bliki/BoundedContext.html%20%5b2)]
* Domain Driven Design Eric Evans [3]
* [https://en.wikipedia.org/wiki/Event\_storming [4](https://en.wikipedia.org/wiki/Event_storming%20%5b4)]
* [https://blog.jannikwempe.com/domain-driven-design-entities-value-objects [5](https://blog.jannikwempe.com/domain-driven-design-entities-value-objects%20%5b5)]
* <https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/microservice-domain-model> [6]
* <https://beetechnical.com/tech-tutorial/value-object-vs-entity-class/> [7]
* [https://martinfowler.com/bliki/DDD\_Aggregate.html [8](https://martinfowler.com/bliki/DDD_Aggregate.html%20%5b8)]
* [https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html [9](https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html%20%5b9)]
* Patterns of Enterprise Application Architecture Martin Fowler [10]
* <https://www.martinfowler.com/eaaDev/DomainEvent.html> [11]
* [https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/domain-events-design-implementation [12](https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/domain-events-design-implementation%20%5b12)]
* [https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/apply-simplified-microservice-cqrs-ddd-patterns [13](https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/apply-simplified-microservice-cqrs-ddd-patterns%20%5b13)]
* <https://enterprisecraftsmanship.com/posts/types-of-cqrs/> [14]
* [https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices [15](https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices%20%5b15)]
* [https://medium.com/must-know-computer-science/system-design-message-queues-245612428a22 [16](https://medium.com/must-know-computer-science/system-design-message-queues-245612428a22%20%5b16)]
* <https://microservices.io/patterns/apigateway.html> [17]