&Bcy;&rcy;&ocy;&jsercy; &scy;&lcy;&ocy;&bcy;&ocy;&dcy;&ncy;&icy;&khcy; &mcy;&iecy;&scy;&tcy;&acy; &zcy;&acy; &scy;&iecy;&pcy;&tcy;&iecy;&mcy;&bcy;&acy;&rcy;&scy;&kcy;&icy; &ucy;&pcy;&icy;&scy;&ncy;&icy; &rcy;&ocy;&kcy; 2012**Univerzitet u Nišu, Elektronski fakultet**

**Katedra za računarstvo**

**Miloš Panić**

**Mikroservisna aplikacija arhitekture vođene događajima za pregled online sadržaja**

**Diplomski rad**

**Niš, 2022**

Sadržaj

[1. Uvod 3](#_Toc95034853)

[1.1. Definisanje problema 3](#_Toc95034854)

[1.2. Organizacija teze 5](#_Toc95034856)

[2. Mikroservisi i mikroservisna arhitektura 6](#_Toc95034857)

[2.1. Mikroservisi 6](#_Toc95034858)

[2.2. Evolucija od monolitne arhitekture na mikroservisnu arhitekturu 7](#_Toc95034859)

[2.3. Mikroservisna arhitektura 8](#_Toc95034862)

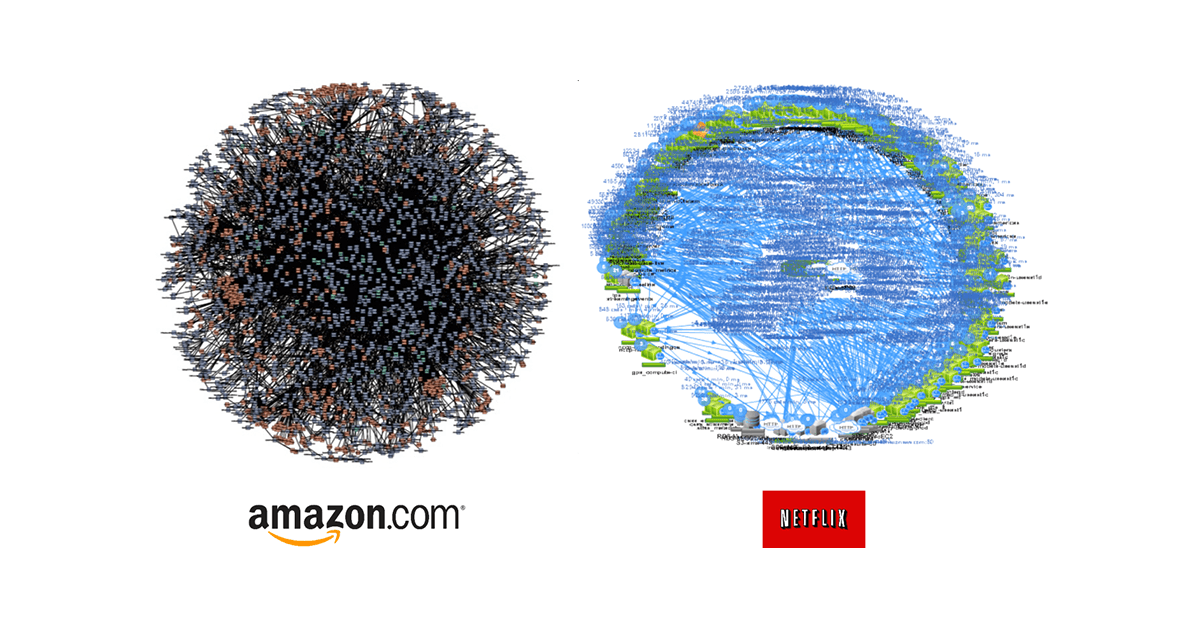
# Uvod

## Definisanje problema

Deljenje video sadržaja u realnom vremenu ili sama mogućnost da aplikacija poseduje opciju za deljenjem kao i pregledavanjem video sadržaja je oduvek bila aktuelna tema i verujem da se to neće menjati u daljem periodu. Počev od samih društvenih mreža, čiji je osnovni cilj upravo međusobno deljenje video sadržaja i slika između korisnika pa do nekih web aplikacija čija je edukaciona svrha, obezbeđivanje različitih video snimaka koje korisnici mogu pregledavati u svrhe učenja i izučavanja različitih oblasti. Sve te aplikacije možda pokrivaju različite domene, međutim sve one imaju jedan zajednički cilj, a to je obezbediti korisnicima mogućnost da pregledavaju različite video snimke bez ikakvog kašnjenja i čekanja da se video snimak učita. Možda to deluje kao naizgled očigledan zahtev i nešto oko čega ne bi trebalo biti problema u današnje vreme, u poređenju sa brzinama internet konekcija, međutim, ukoliko sama arhitektura web aplikacije nije isprojektovana da podnese veliki broj korisnika u realnom vremenu, koji preuzimaju video snimke, odmah se dolazi do problema.

U tom slučaju, na scenu stupa mikroservisna arhitektura koja bi sama po sebi, u skladu sa svojim prednostima i mana, trebala biti odličan izbor za rešavanje prethodno pomenutog problema. Mikroservisna arhitektura, čija popularnost je u naglom porastu, trebalo bi u potpunosti da podnese veliku količinu protoka podataka kroz samu arhitekturu između samih komponenti sistema ali i veliki broj korisnika koji zahtevaju podatke u realnom vremenu. Kako joj i samo ime kaže, mikroservisna arhitektura je sačinjena od velikog broja mikroservisa koji rade zajedno na obezbeđivanju različitih funkcionalnosti korisnicima, u zavisnosti od domena i potreba same aplikacije koja se razvija. U suštini, po meni, glavna prednost je u tome što se može napraviti veliki sistem, sačinjen od velikog broja komponenti između kojih se može vršiti balansiranje opterećenja, odnosno prosleđivanje zahteva ravnomerno mikroservisima kako se ne bi postiglo preopterećenje niti jedne komponente sistema. Tim pristupom odmah dobijamo odgovor zašto bi baš mikroservisna arhitektura bila pogodna za opsluživanje velikog broja korisnika koji zahtevaju veliku količinu podataka u realnom vremenu, kako preuzimanje tako i deljenje podataka.

Mikroservnisna arhitektura, tačnije takozvana mikroservisna arhitektura vođena dogajima(Event-driven architecture), je dosta korišćena danas i biva sve popularnija. Bez ikakve dalje priče, za primer bih najpre naveo Netflix kao kompaniju koja je u potpunosti implementirala mikroservisnu arhitekturu za manje više iste svrhe koje obezbeđuje i aplikacija razvijena za potrebe ovog rada. Njihova arhitektura je sačinjena od velikog broja mikroservisa(po nekim informacijama Netflix danas poseduje preko 1000 mikroservisa koji rade u pozadini i svaki je nadležan za odgovarajući deo aplikacije)[2].



### Slika 1.1 – Mikroservisna arhitektura Netflix-a i Amazon-a

Na slici 1.1 možemo videti na interesantan način reprezentovane arhitekture Netflix-a i Amazon-a. Na prvi pogled, deluje da je to neki preveliki nered u kojem se teško stvari mogu razumeti i koji uz toliko konekcija između komponenata prosto ne može da obezbedi odične performanse korisnicima ali stvarnost je naravno potpuno drugačija. Rezultati su naravno izvanredni i po pitanju Netflix-a prelazak na mikroservisnu arhitekturu sa stare monolitne arhitekture je bio pun pogodak. Broj korisnika je u stalnom porastu, a skalabilnost same mikroservisne arhitektura bi u potpunosti trebalo da podnese taj rast. Netflix trenutno broji preko 93 miliona korisnika.  
  
Distribuirani sistemi, paralelni sistemu su oduvek bili aktuelni i oni su meni bila glavna motivacija za učenjem i razvijanjem aplikacija kod kojih je u osnovi mikroservisna arhitektura. Sama paralelizacija, podela posla između komponenti sistema, samo razmišljanje na taj način da više delova sistema mogu paralelno obavljati poslove u cilju zadovoljenja potreba korisnika je meni oduvek bila fascinantna i privlačna za učenjem. Mikroservisi ciljaju na probleme monolitne arhitekture podržavanjem principa rada po imenu razdvajanje briga što bi bio grubi prevod(Separation of concerns)[1]. Taj princip podržava modularno razbijanje velikih programa u manje softverske komponente pri čemu svaka od tih softverskih komponenti poseduje sopstvenu enkapsulaciju podataka.

## Organizacija teze

U okviru ove podsekcije potrudiću se da ukratko predočim šta će ovaj rad sadržati u nastavku po poglavljima.   
U narednom poglavlju, poglavlju broj 2 pod imenom ,,Mikroservisi i mikroservisna arhitektura” ću se najviše dotaći teorije i pokušati da teoretski opravdam upotrebu mikroservisne arhitekture za potrebe deljenja video sadržaja ali naravno u okviru tog poglavlja biće reči i uopšte o samoj mikroservisnoj arhitekturi.

Treće poglavlje, pod imenom ,,Metode i tehnologije za razvoj mikroservisnih aplikacija” treba da se dotakne nekih dobrih praksi koje su se istakle kroz rad softverskih inženjera kao i tehnologija nastalih u okviru tog procesa i pokušaću da istaknem par tehnologija koje bi bile odlične za razvoj mikroservisnih aplikacija kao i tehnologije koje su danas najviše korišćenije.

Četvro poglavlje će biti u potpunosti posvećeno aplikaciji koja je razvijena za potrebe ovog rada i koja prati ovaj rad. U okviru tog poglavlja pokušaću da kroz različite dijagrame objasnim samu arhitekturu aplikacije, njene funkcionalne i nefunkcionalne zahteve ali i različite implementacione detalje.

U poglavlju nakon četvrtog poglavlja, priča se nastavlja o samoj aplikaciji i u okviru tog poglavlja ću pokušati da još detaljnije opišem funkcionalne zahteve te ću se i dotaći samog testiranja i evaluacije aplikacije.

Poslednje, odnosno šesto poglavlje bi trebalo da bude zaključno poglavlje i u okviru tog poglavlja pokušaću da iznesem neke svoje zaključke o samoj mikroservisnoj arhitekturi te i da sumiram sve stvari koje sam ja naučio u toku razvoja aplikacije.

# Mikroservisi i mikroservisna arhitektura

## Mikroservisi

Kod mikroservisne arhitekture, kao što joj i samo ime govori, osnovna gradivna jedinica, osnovni element u strukturi te arhitekture, je u pravo mikroservis. Mikroservis može da se posmatra kao nezavisna komponenta koja u saradnji sa ostalim mikroservisima obezbeđuje korisnicima različite funkcionalnosti u zavisnosti od potreba i domena same aplikacije za koju se razvija mikroservisna arhitektura.  
  
Mikroservisi su softverske komponte, nezavnisne jedinice deployment-a i razvoja koje sačinjavaju mikroservisnu arhitekturu i oni su u suštini osnovna gradivna jedinica te arhitekure. Mikroservisna arhitektura, odnosno, mikroservisi imaju za cilj da u potpunosti reše neke probleme sa kojima se susretala inicijalna monolitna arhitektura.

Osnovne karakteristike koje bi trebalo da ispunjava svaki mikroservis:

* Pogodan izuzetno za održavanje i testiranje
* Mikroservisi koji sačinjavaju mikroservisnu arhitekturu bi trebali biti slabo spregnuti
* Nezavisna jedinica deployment-a
* Nezavisna jedinica razvoja
* Tehnologije za izradu mikroservisa ne moraju biti iste

U nastavku ću pokušati da malo preciznije pojasnim svaku od ovih pojedinačnih karakteristika.

Da bi se postigla karakteristika mikroservisa, koja kaže da svaki mikroservis treba da bude nezavnisna jedinica deployment-a i razvoja, to ne bi bilo nikako moguće ukoliko postoji čvrsta sprega između mikroservisa koji čine arhitekturu neke aplikacije koja u osnovi ima mikroservisnu arhitekturu. Pod slabom spregom između mikroservisa misli se na to, da ne postoji čvrsta ,,veza’’ između mikroservisa. Oni nisu u čvrstoj vezi po pitanju osetljivosti na promene. Dakle, ukoliko dođe do modifikacije dizajna, implementacije ili ponašanja nekog od mikroservisa, to ne bi trebalo nikako da izazove promene u ostalim mikroservisima[4]. Na taj način se postiže da je svaki mikroservis nezavisna celina, i samim tim se razvoj pojedinačnih mikroservisa može dodeliti pojedinačnom timu. Jedan od glavnih loših pristupa, koji može da izazove čvrstu spregu između mikroservisa je upravo deljenje skladišta podataka, neke baze podataka na primer[4].   
Svakako, ukoliko svaki mikroservis ima posebnu i nezavisnu bazu koda(codebase), opet može da dođe do čvrste sprege ukoliko mikroservisi međusobno zavise od istih biblioteka.

Pošto smo opravdali i objasnili karakteristiku mikroservisa da oni međusobno treba da imaju laku spregu, sada je dosta lako objasniti da svaki mikroservis treba da bude nezavisna jedinica deployment-a i razvoja. Samim tim, pošto ne postoji čvrsta sprega između mikroservisa, lako je nezavisno razvijati bilo koji mikroservis u poređenju sa ostalim mikroservisima. Dobra praksa je da svaki mikroservis ne treba da bude naročito prevelik te se razvoj i održavanje mikroservisa može dodeliti ne toliko prevelikom timu.

Svaki mikroservis koji sačinjava neku mikroservisnu arhitekturu takođe treba da bude i nezavisna jedinica deployment-a. Odnosno, u okviru sistema, svaki mikroservis se može nezavisno ažurirati i deploy-ovati bez uticaja na ostale module koji sačinjavaju aplikaciju odnosno sistem.

Još jedna od karakteristika mikroservisa koja je direktna posledica toga da mikroservisi međusobno poseduju slabu spregu je upravo to da za razvoj svakog mikroservisa u sistemu se može koristiti različita tehnologija. Dakle, potpuno je validno razviti jedan mikroservis korišćenjem ASP .NET Core tehnologije i C# programskog jezika dok ostali mogu biti razvijeni upotrebom programskog jezika Java i Spring Boot framework-a. To je potpuna nezavisnost i ovo u potpunosti podržava različite timove sačinjene od inženjera sa znanjem iz različitih sfera i tehnologija. U pozadini bi sve, naravno, funkcionisalo kako treba bez ikakvih problema, jer se komunikacija i sinhronizacija između samih mikroservisa može realizovati na različite načine. To se može postići i sinhronim putem i asinhronim putem ali o ovome malo više reči u poglavlju 2.3 koje će se više pozabaviti samom mikroservisnom arhitekturom u potpunosti.

## Evolucija od monolitne arhitekture na mikroservisnu arhitekturu

Monolitna arhitektura je bila trend od samog početka i u potpunosti ispunjava sve zahteve koje je potrebno da ispunjava. Danas, sa sve većim brojem korisnika i zahteva u realnom vremenu, monolitna arhitektura se suočava sa problemima po pitanju skalabilnosti, agilnosti ali i ukoliko želimo da uvedemo odnosno primenimo upotrebu neke nove tehnologije u naš sistem, opet ćemo doći do problema[5]. Skalabilnost je u potpunosti rešena uvođenjem mikroservisne arhitekture, jer svaki mikroservis može biti u okviru sistema repliciran više puta i uz upotrebu odgovaraćujeg raspoređivača opterećenja(Load balancer), skalabilnost ne bi trebala da predstavlja nikakav problem.

Do ovih problema dolazi tako što sve, barem većina aplikacija u samom početku krenu kao manje aplikacije, pri čemu upotreba monolitne arhitekture nad takvim potrebama je u potpunosti opravdana naravno i performanse koje isporučuje su u potpunosti u skladu sa svim zahtevima. Kako vreme odmiče i kako aplikacija raste u širinu, kako po pitanju podataka i broja korisnika, te time i broja zahteva u jedinici vremena, aplikacija u jednom trenutku postane ogromna. U tom trenutku, održavanje i unapređenje monolitne arhitekture postaje pravi izazov, poput dodavanja novih funkcionalnosti aplikaciji kako bi se održao rast i unapredilo iskustvo samih klijenata [5]. U jednom trenutku, dolazi se do pozicije da se cela upotreba aplikacije dovodi u pitanje, jer preveliki rast nad monolitnom arhitekturom u jednom trenutku može početi da se održava kroz performanse u aplikaciji.  
Iz očiglednih razloga, rast i skaliranje aplikacije ni u kojem trenutnu ne bi trebali stati, te je potrebno pronaći izlaz iz monolitne arhitekture i jedno od rešenja je upravo razbijanje monolita na nekoliko mikroservisa radi lakšeg rada[5].

### 

### Slika 2.1 Uporedni prikaz monolitne i mikroservisne arhitekture

## Mikroservisna arhitektura



### Slika 2.2 Mikroservisna arhitektura

Na slici 2.2 možemo videti sistem sačinjen od 4 mikroservisa koji obezbeđuju funkcionalnosti korisncima i svi zajedno čine jedan veliki sistem. Celokupna arhitektura, kao što je već više puta pomenuto, je sačinjena od mikroservisa koji zajedno funkcionišu, saradjuju i komuniciraju a sve to u cilju obavljanja različitih funkcionalnosti. Potrebno je da je svaki mikroservis bude nezavisan, da postoji slaba sprega između mikroservisa. Svaki mikroservis može posedovati posebno skladište podataka, bilo da je to neka relaciona baza podataka ili neka NoSQL baza podataka. Mikroservisi međusobno komuniciraju preko API-ja koji su dobro definisani pri čemu su interni implementacioni detalji sakriveni od ostalih učesnika u komunikaciji [6].

U nastavku bih se potrudio da posvetim pažnju svakoj pojedinačnoj komponenti o kojoj do sada nije bilo reči kako bih još detaljnije objasnio funkcionisanje celokupne arhitekture.

Menadžment ili orkestracija je komponenta odgovorna za postavljanje servisa na čvorovima, otkrivanje grešaka i problema, rebalansiranje servisa kroz čvorove i tako dalje. U glavnom se za potrebe obavljanja ovih funkcija koriste neke gotove komponente, poput Kubernetes-a koji je izuzetno dobar alat za orkestraciju kontejnera[6]. Dakle, mikroservisi, s obzirom da ih može biti veliki broj u okviru jednog sistema, potrebno je imati neku centralnu kontrolu nad njima po pitanju upravljanja, balansiranja. Tu na scenu stupa Docker, kao poznat i dosta korišćen alat za kontejnerizaciju. On upravo obezbeđuje kreiranje takozvanih slika(Docker images) čijim korišćenjem se može instancirati veći broj kontejnera(više instanci jednog istog mikroservisa) i uz upotrebu odgovarajućeg raspoređivača težine(Load balancer) moguće je obezbediti da svi mikroservisi u okviru jednog sistema budu ravnomerno opterećeni po pitanju zahteva koji stižu od strane klijenata. Docker, kao alat za kontejnerizaciju je upravo iskorišćen i za upravljanje mikroservisnom aplikacijom koja je razvijena za potrebe ovog rada a o kojoj će biti više reči u narednim poglavljima.

API prolaz (API Gateway), kako bi bio bukvalan prevod sa engleskog jezika je upravo prolaz koji vodi klijentske zahteve u svet mikroservisne arhitekture. Prolaz kao prolaz, u glavnom vodi od tačke A do tačke B, međutim, API Gateway obezbeđuje dinamično rutiranje. Pod dinamičnim rutiranjem, želim da istaknem da svaki zahtev, upućen od strane korisnika bez obzira kom mikroservisu, bi najpre trebao da prođe kroz API Gateway komponentu mikroservisne aplikacije te da bi ta komponenta bila potom zadužena da taj zahtev prosledi, po odgovarajućem algoritmu, na pravo odredište. Verovatno se pitate, kako ovo može biti brže i imati bolje performanse od monolitne arhitekture gde klijent direktno šalje svoj zahtev odredišnoj komponenti bez ikakvog posrednika. Zamislite, recimo sledeću situaciju gde u jednom trenutku postoji milion zahteva od strane korisnika ka jednoj komponenti, bilo da je to jedna glavna komponenta (kod monolitne arhitekure) ili samo jedan deo nekog većeg sistema (više instanci jednog mikroservisa u mikroservisnoj arhitekturi). Monolitna komponentna bi se u tom trenutku morala sama izboriti sa svim novopristiglim zahtevima dok bi API Gateway u mikroservisnoj arhitekturi obavio ključnu ulogu. API Gateway u glavnom poseduje i obavlja funkciju takozvanog, prethodno pomenutog, raspoređivača težine (Load balancer). Njegova uloga je da ravnomerno rasporedi zahteve od strane korisnika ka nekom konkretnom mikroservisu između svih instanci tog mikroservisa kojem su zahtevi upućeni, te se na taj način postiže da svaka instanca tog mikroservisa bude ravnomerno opterećena.

Pored raspoređivanja težine, API Gateway može obavljati i druge različite funkcije za poboljšanje performansi:

* Razdvaja u potpunosti klijenta od servisa, te na taj način servisi se mogu menjati bez ikakve potrebe da dolazi do promena na klijentskoj strani[6]
* Autentikacija
* Različite transformacije, validacije, keširanja[6]

U okviru mikroservisne arhitekture koja je dizajnira za potrebe ovog rada, takođe je uključen API Gateway kao ulazna tačka u ceo sistem.

Komunikacija u mikroservisnom svetu može biti i asinhrona i sinhrona, kako je već prethodno rečeno. Mikroservisi mogu međusobno komunicirati sinhronim putem, korišćenjem HTTP zahteva i REST-a dok se komunikacija između mikroservisa može obezbediti i asinhronim putem gde se, drugačije rečeno, mikroservisi sinhronizuju događajima odnosno pravi se mikroservisna arhitektura vođena događajima. U tom slučaju, koristi se centralna komponenta, takozvani broker poruka (Message broker). Uloga te centralne komponente je održavanje različitih redova poruka, ili topic-a preko kojih se vrši sinhronizacija između mikroservisa. Mikroservisi međusobno proizvode poruke u topic-ima i redovima poruka kada se odigra neki događaj od interesa, a za koji su oni odgovorni, te drugi mikroservisi koji su pretplaćeni na prethodno pomenutim redovima poruka ili topic-a preko istih bivaju obavešteni o tom događaju, primaju podatke vezane o tom događaju preko samih poruka i obavljaju naredne akcije kako bi se celokupna akcija odigrala do kraja. Ovaj pristup je izuzetno brz i dosta korišćen u modernim mikroservisnim arhitekturama.

## Benifiti mikroservisne arhitekture

* **Agilnost** – Samim tim pošto su mikroservisi nezavisne jedinice deployment-a mnogo je lakše organizovati ispravljanje postojećih grešaka i organizovati naredna ažuriranje aplikacije[6]. Prostije rečeno, moguće je obaviti ažuriranje jednog mikroservisa bez potrebe da se cela aplikacije menja, ponovo postavlja.
* **Manji, fokusiraniji timovi** – Ne treba preterivati sa veličinom mikroservisa, svaki mikroservis treba da ima dobro zaokruženu funkcionalnost i da je obavlja u potpunosti u okviru sistema te je na taj način dovoljno obezbediti i manje, fokusiranije timove za razvoj i održavanje svakog mikroservisa u okviru sistema
* **Različite tehnologije** – kao što je pre bilo reči, za razvoj svakog mikroservisa u sistemu moguće je iskrostiti različite tehnologije
* **Izolacija grešaka (Fault isolation)** – Ako neki, individualni mikroservis postane nedostupan to ne bi trebalo da onemogući rad cele aplikacije, sve dok mikroservisi iznad su dizajnirani tako da mogu da podnesu otkazivanje nekog od mikroservisa ispod u hijararhiji [6]. Za obezbeđivanje ove funkcionalnosti postoje različiti šabloni dizajniranja o kojima će biti više reči u narednom potpoglavlju.
* **Skalabilnost** – O skalabilnosti je bilo više reči do sada. Ključ u proširljivosti je u tome što svaki pojedinačni deo ceo sistema se može skalirati tako što će se povećati broj mikroservisa koji čine taj pojedinačni deo sistema
* **Izolacija podataka** – Mnogo je lakše izvršavati različita ažuriranja šema podataka nad različitim bazama podataka jer bi u tom slučaju samo jedan mikroservis bio pogođen tom promenom dok bi kod monolitne arhitekture bilo koja promena nad šemom baze podataka izazvala promene u različitim delovima aplikacije [6].

Pored različitih prednosti koje donosi mikroservisna arhitektura, kao i većina stvari i mikroservisna arhitektura ima svoje nedostatke. Ukoliko se iskoristi na pravi način za prave potrebe, ti nedostaci se svode na minimum. Neki od nedostataka su: kompleksnost, nedostatak upravljanja pošto se koristi adekvatan decentralizovani pristup, zagušenje mreže i kašnjenje ukoliko se naprave lanci komunikacije ,,bez krajeva’’, pošto svaki pojedinačni mikroservis može imati sopstvenu bazu podataka, dolazimo do problema održavanja konzistentnosti nad podacima, međutim to je jedan od standardnih problema u modernim distribuiranim sistemima koji se uz upotrebu adekvatnih, dokazanih pristupa rešava.

## Najbolje prakse i šabloni dizajniranja

Višestruko korišćenje kroz istoriju akumuliralo je različite pristupe za dizajniranje i implementaciju mikroservisne arhitekture. Softverski inženjeri su razvijali različite mikroservisne arhitekture te su kroz iskustvo istakli odgovarajuće pristupe koji su se pokazali kao najbolji. U okviru ovog potpoglavlja potrudiću se da prezentujem par šablona koje smatram da bi bili korisni za implementaciju i dizajniranje različitih mikroservisnih arhitektura.

**Baza podataka po mikroservisu (Database per service)** – Većini mikroservisa je potrebna adekvatna perzistencija podataka. Da bi se izbegla čvrsta sprega između mikroservisa uglavnom se teži ka tome da svaki mikroservis poseduje posebnu bazu podataka. Glavni problem koji stvara ovaj pristup je obezbeđivanje konzistentnosti između baza podataka različitih mikroservisa. Kod monolitne arhitekture je to dosta bilo jednostavno, pošto, na primer ako se koriste relacione baze podataka, bilo je lako organizovati pristup kroz transakcije dok ovde, kada se koriste različite baze podataka to stvara problem. Jedno od rešenja za ovaj problem je takozvani Saga pattern [7].

Još jedna od prednosti koje uvodi ovaj pristup je upravo mogućnost da se za različite mikroservise iskoristi različit tip skladišta podataka. Na taj način se dobija na raznovrsnosti ali i performansama, s obzirom da nisu modeli podataka mikroservisa isti i nemaju iste zahteve. Svaki mikroservis može iskoritit poseban tip skladišta podataka, bilo da je to relaciona baza podataka ili neka NoSQL baza podataka u skladu sa odgovarajućim modelom podataka koji treba da implementira [7].

**Šablon pod imenom izvor događaja(Event sourcing) –** Svaki mikroservis u okviru sistema treba da bude sposoban da ažurira svoje skladište podataka i da proizvodi poruke, odnosno događaje. Na taj način se postiže sinhronizacija između mikroservisa i izbegava se nekonzistentnost odnost problemi.

**Razmenjivanje poruka (Messaging) –** Za sarađivanje između mikroservisa korišećnjem sinhronizacije preko poruka i message broker-a koji mogu implementirati različite protokole komunikacije. Jedan od najboljih pristupa je preko Publish/Subscribe šablona koji obezbeđuju različiti message broker-i. Ovaj pristup je upravo i iskorišćen u aplikaciji koja je razvijena za potrebe ovog rada. Publish/Subscribe šablon obezbeđuje komunikaciju između mikroservisa preko takozvanih tema(Topic-a), gde kada jedan publikant objavi svoju poruku nad temom, svi pretplaćeni mikroservisi bivaju obavešteni o tome i primaju poruku. Na taj način se može obezbediti mikroservisna arhitektura vođena događajima, odnosno kada se odigra događaj u jednom mikroservisu on preko odgovarajućeg topic-a može obavestiti ostale mikroservise o tome da preduzmu odgovarajuće akcije i obave transakcije u potpunosti.

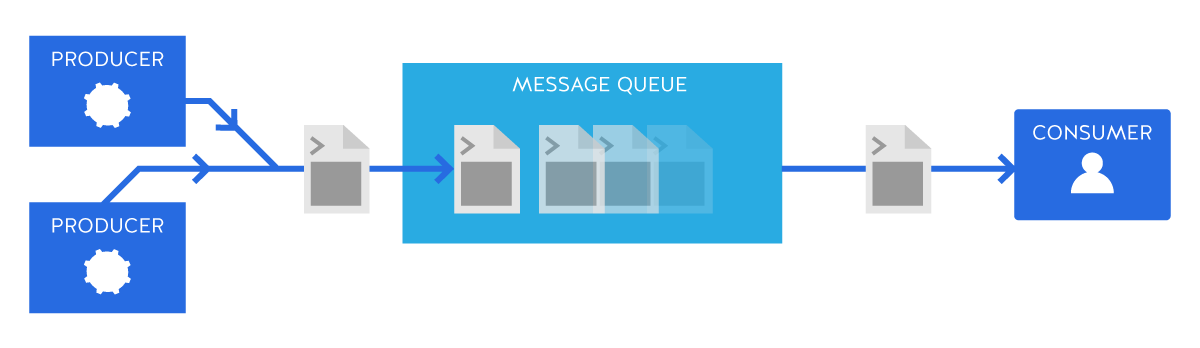
**Razbijanje kola (Circuit breaker) šablon** Često u mikroservisnoj arhitekturi se može doći u situaciju, da prilikom komunikacije između 2 mikroservisa se dođe u situaciju da pozivani mikroservis nije dostupan, usled nekog problema, ili se jednostavno predugo čeka odgovor od pomenutog mikroservisa. Rešenje za ove pomenute probleme je da mikroservis koji poziva drugi mikroservis treba da obavi to preko nekog proksija (posrednika) koji funkcioniše na sličan način kao i električni prekidač. Kada se dođe do detekcije problema, nakon prelaska preko odgovarajućeg praga (proteklo vreme bez odgovora ili adekvatan broj pokušaja) sam prekidač se isključuje i više ne dozvoljava zahteve ka mikroservisu koji očigledno ima neki problem. Postoji odgovarajući period tokom kojeg će prekidač biti isključen te potom ponovo se aktivira po odgovarajućem algoritmu [10].

Ovo su samo neki od šablona i dobrih praksi koje sam pomenuo. Pored njih, postoje i drugi pristupi koji se mogu iskoristiti, ja sam se lično dotakao ovih iz razloga što prate samu aplikaciju koja je razvijena ali i obezbeđuju dobre rezultate, odnosno dosta su korišćeni u modernim aplikacijama.

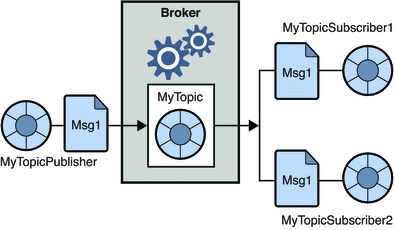
# Tehnologije za razvoj mikroservisnih aplikacija

U okviru prethodnog poglavlja, bilo je dosta teorijske osnove o samom tipu arhitekture i pojedinačnim delovima. Verovatno ste primetili da se konkretnih tehnologija koje se mogu iskoristiti nisam previše dotakao iz razloga što smatram da je za to potrebno posvetiti celokupno poglavlje jer može mnogo što šta da se kaže. Ova tema dobija na svom značaju upravo zbog toga što se za razvoj svakog pojedinačnog mikroservisa u sistemu u potpunosti mogu iskoristiti međusobno različite tehnologije.

**Docker** - Samim tim pošto je celokupna arhitektura sačinjena od većeg broja manjih gradivnih jedinica, odnosno mikroservisa, potrebno je obezbediti centralizovan alat za upravljanje celim sistemom, raspoređivanje opterećenja. Za takve potrebe upravo stupa na scenu Docker. Docker je alat koji se koristi za kontejnerizaciju komponenti, odnosno od svakog mikroservisa se prave takozvane Docker slike (Docker images) koje su poput klasa u kontekstu objektno orjentisane paradigme. Kako se klase mogu instancirati i od njih praviti objekti, koji postoje sve vreme dok se aplikacija izvršava, tako se i Docker slike mogu instancirati i od njih praviti kontejneri te svaki kontejner predstavlja jednu instancu odgovarajućeg mikroservisa. Upravo na taj način je i moguće instancirati više puta jedan mikroservis i dobiti više instanci istog mikroservisa u okviru sistema. Docker obezbeđuje centralizovani pogled na ceo sistem, centralizovani logging sistem te se lako može pratiti rad celog sistema i status svakog pojedinačnog mikroservisa. Da bi sve to funkcionisalo lepo, i da bi bilo lako upravljati sistemom, uz Docker uglavnom ide i poseban alat za orkestraciju, odnosno upravljanje kontejneri. Jedan primer takvog alata je upravo **Kubernetes** o kojem je prethodno bilo reči.

**Message brokeri** – U prethodnom poglavlju je dosta bilo reči o samoj komunikaciji i sinhronizaciji između mikroservisa. Jedan od glavnih alata koji se koristi danas za sinhronizaciju između mikroservirsa su pravo message broker-i ili u prevodu na naš jezik, takozvani brokeri poruka. Njihova glavna uloga je da obezbede komponentu u sistemu, koja će obezbeđivati različite strukture podataka preko kojih se može obezbediti komunikacija, odnosno sinhronizacija mikroservisa. Postoje različiti protokoli i šabloni koje mikroservisi implementiraju te ću ja pokušati ukratko da objasnim njihov rad. Cilj brokera poruka je da obavljaju rutiranje poruka između zainteresovanih strana, od pošiljaoca do odgovarajuće destinacije ili destinacija. Postoje u glavnom dva šablona po kojima se odvija komunikacija, a to je komunikacija od tačke do tačke preko različitih redova poruka ili komunikacija preko Publish and subscribe modela koji obezbeđuje komunikaciju između jednog pošiljaoca i većeg broja primaoca [8].  


### Slika 3.1 Vizuelizacija rada brokera poruka sa redom poruka



### Slika 3.2 Vizuelizacija rada brokera poruka zasnovanog na publish and subscribe šablonu komunikacije

Message broker koji je iskorišćen u aplikaciji koja je razvijena za potrebe ovog rada je **Kafka**. Kafka je message broker koji je razvijen od strane Apache fondacije i obezbeđuje takozvani tok događaja(Event streaming). Uz pomoć Kafke dosta je lako razviti sistem koji je otporan na greške i obezbediti sistem zasnovan na događajima, sistem kod kojeg se komponente koje ga sačinjavaju sinhronizuju na osnovu događaja koji se odigravaju u sistemu. Sama Kafka je u suštini nezavisan distribuirani sistem koji se sastoji od servera i klijenata koji komuniciraju preko visoko kvalitetnog TCP mrežnog protokla. Kafka klaster je visoko skalabilan i otporan na greške, odnosno ako jedan od servera padne ostali će preuzeti njegovu odgovornost bez ikakvog gubitka u podacima, što je za moderne sisteme koji opslužuju veliki broj korisnika jedan od glavnih zahteva [9]. Pored Kafke, pomenuo bih i message broker koji obezbeđuje Amazon Web Services(AWS) – Amazon MQ, Eclipse Mosquitto MQTT Broker razvijen od strane Eclipse fondacije, ZeroMQ i drugi.

**Skladišta podataka** – Ovoj sekciji ne bih posvetio previše pažnje iz razloga što je za izbor odgovarajućeg skladišta podataka, za odgovarajući mikroservis, potrebno uzeti u obzir model podataka sa kojim bi mikroservis trebao da radi. To je specifično od mikroservisa do mikroservisa, od domena aplikacije do domena aplikacije ali bih iskoristi priliku da ponovim kako je to veliki benefit po pitanju performansi, jer modeli podataka od mikroservisa do mikroservisa mogu biti različiti te se za implementaciju skladišta podataka mogu iskoristiti adekvatne tehnologije, od relacionih baza podataka do odgovarajućih NoSQL baza podataka te se i na taj način obezđuje slaba sprega između mikroservisa. Konkretno u okviru same aplikacije koja je razvijena za potrebe ovog rada iskorišćene su i relaciona baza podataka (MS SQL Server) i više vrsti NoSQL baza podataka (Redis za keširanje različitih stvari ali i MongoDB baza podataka za čuvanje odgovarajućih događaja koji će biti dostupni za pregledavanja/streaming u odgovarajućem terminu.

**API Gateway** – Konkretno, u okviru aplikacije koja je razvijena za potrebe rada, za implementaciju API Gateway funkcionalnosti (API kapija/prolaz), iskorišćena je Ocelot biblioteka razvijena u .NET Core tehnologiji. Pored standardne funkcionalnosti API Gateway, ona obezbeđuje i naredne funkcionalnosti: raspoređivanje težine (Load balancing), autentikacija, autorizacija, razna keširanja i druge stvari [11]. Pored navede biblioteke, iskoristio bih priliku da se ponovo osvrnem na kompaniju Netflix i njihovu upotrebu mikroservisne arhitekture. S obzirom da je njihova kompletna arhitektura mikroservisna, oni poseduju tim u okviru kompanije zadužen za razvijanje novih tehnologija i alata koje pružaju na raspolaganju svetu. Jedan od alata koji je razvijen je upravo i Zuul API Gateway. Netflix potpuno ponosno, a verovatno i opravdana dosta promoviše taj svoj proizvod i po nekim njihovim informacijama iz 2018. godine, poseduju 80 klastera sa Zuul-om koji prosleđuju saobraćaj ka 100 klastera servisa a to je oko 1 milion zahteva svake sekunde [12].

## Tehnologije za realizaciju konkretnih mikroservisa i streaming funkcionalnosti

Pre svega, najpre bih se osvrnuo na tehnologiju koja je iskorišćena za realizaciju mikroservisa aplikacije koja je razvijena za potrebe ovog rada te bih kasnije pokušao da pomenem i par drugih tehnologija, ravnopravnih, pogodnih za razvoj mikroservisa.

**ASP .NET Core** – Svi mikroservisi koji su implementirani za potrebe aplikacije koja prati ovaj rad, su implementirani upravo korišćenjem ASP .NET Core tehnologije. Svi mikroservisi su u suštini Web API-ji i svaki od njih ima tačno zaokruženu funkcionalnost, što je u suštini i jedan od zahteva koji treba da se uzme u obzir prilikom projektovanja mikroservisne arhitekture. Ja sam se odlučio za ovu tehnologiju iz više razloga. Jedan od njih je naravno, upravo zato što sam do početka rada na ovom projektu imao najviše iskustva sa tom tehnologijom kako za razvijanje aplikacija sa monolitnom arhitekturom ali tako i za razvijanje aplikacija sa mikroservisnom arhitekturom. ASP .NET Core tehnologija pruža izuzetno dobar način za razvijanje servisa koji će raditi na serveru i opsluživati zahteve od strane korisnika i uz pomoć dodatnih biblioteka i Object Relation Mapper-a (ORM), za koje poseduje podršku, obezbeđuje inženjerima mogućnost da razviju servis koji će komunicirati sa bilo kojom bazom i obavljati perzistenciju podataka. S obzirom da je u planu, u inicijalnom dizajnu arhitekture, bilo da aplikacija treba da bude u mogućnosti da komunicira sa 3 različite baze pokrenute u nezavisnom kontejneru, ASP .NET Core je bilo logično rešenje za realizaciju. Pored podrške za rad sa različitim bazama podataka, postoji i biblioteka koja je omogućila i rad sa Kafkom, odnosno mogućnost da se kreiraju proizvođači i potrošači (Consumer i Producer) poruka nad odgovarajućim topic-ima brokera poruka. Što se tiče potrošača poruka, on je u okviru aplikacije realizovan kao nezavisni pozadinski servis, koji je u blokiranom stanju sve dok ne primi poruku.

Jedan od funkcionalnih zahteva bio je i obezbediti mogućnost slanja notifikacija korisnicima kada se neki događaj koji je njima od interesa dogodi. Za realizaciju te funkcionalnosti, iskorišćen je WebSocket protokol i SignalR tehnologija za koju ASP .NET Core u potpunosti poseduje podršku.  
  
Za realizaciju toka (streaming-a) video snimaka koji treba da budu dostupni korisnicima za pregledavanje, ja sam se odlučio da iskoristim standardni HTTP protokol i isporučivanje datoteka kao stream(tok) podataka koji se sa klijentske strane preuzima u manjim delovima (chunk-ovima). Pre svega, ovo je dobar vid i brzi vid prebacivanja podataka sa serverske strane do klijenta. Pored ovog pristupa, u obzir su dolazili i različiti pristupi korišćenjem WebSocket protokola (SignalR) ili preko gRPC-a, međutim kako sam naišao prilikom istraživanja na dosta realizacija korišćenjem jednostavnih HTTP zahteva i HTTP protokola, odlučio sam se da i ja realziujem taj pristup u okviru svoje aplikacije.

Kako sam već naveo na početku ovog potpoglavlja, na kraju bih iskoristio priliku da se osvrnem i na druge tehnologije koje su dostupne za realizaciju mikroservisa. Možda i najpopularnija tehnologija, koja bi u potpunosti zamenila ASP .NET Core bi bio upravo Java programski jezik i Spring Boot framework. Za realizaciju REST Web API-ja, uz podršku za rad sa različitim bazama podataka i ORM-ovima, Spring Boot framework bi bio izvanredan izbor. S obzirom da su Netflix-ovi alati koji su razvijeni od strane tima iz navedene kompanije uglavnom orjentisani ka Javi, ovo bi bio još dodatan plus na stranu Spring Boot frameworka jer bi bilo moguće iskoristiti neke alate koje Netflix uveliko koristi u svojem sistemu i koji su se pokazali kao pouzdano rešenje za dosta problema sa kojima se susreće mikroservisna arhitektura.

Pored prethodne dve navedene tehnologije, za realizaciju mikroservisa bi bilo moguće iskoristiti i Node.JS, Python, Golang i druge programske jezike. Svaki od njih naravno donosi svoje prednosti i mane, razne mogućnosti ali i razna ograničenja.

# MyStreamingApp – aplikacija i servis za organizovanje online događaja

Kroz prethodna par poglavlja, potrudio sam se da postavim teoretsku osnovu i neke tvrdnje koje bi aplikacija koja je razvijena trebalo da opravda. S toga, u ovom poglavlju bih se okrenuo ka samoj aplikaciji i pokušao da napravim paralelu sa teorijom i prezentujem kako je sve, prethodno predstavljeno, realizovano kroz samu aplikaciju.

MyStreamingApp je aplikacija koja je dizajnira tako da poseduje mikroservisnu arhitekturu vođenu događajima. Odnosno, u pozadini same aplikacije postoje 3 različita mikroservisa koja obezbeđuju sve potrebne funkcionalnosti korisnicima i koji uz pomoć Kafke, kao brokera poruka, međusobno komuniciraju i uspevaju da se sinhronizuju. Realizovana aplikacija obezbeđuje korisnicima da kreiraju različite online događaje i obezbede video snimke koji bi trebalo da prezentuju kreirane događaje, primaju notifikacije o događajima ali pregledavaju događaje za koje su se pretplatili u vreme kada bi događaj trebalo da se odigra.

Prilikom prezentovanja aplikacije, najpre bih se osvrnuo na funkcionalne i nefunkcionalne zahteve koje aplikacija treba da obezbedi.

## Generalna specifikacija zahteva

**Funkcionalni zahtevi**:

* Obezbediti korisnicima mogućnost registrovanja u sistem
* Obezbediti korisnicima različite kategorije interesovanja o kojima korisnici mogu biti zainteresovani a koje mogu selektovati prilikom registrovanja
* Obezbedi korisnicima mogućnost kreiranja događaja uz obavezno postavljanje video snimka koji treba da bude emitovan u izabrano vreme emitovanja događaja
* Obezbediti korisnicima da ručno izaberu koje kategorije od interesa događaj pokriva
* Obezbediti korisnicima mogućnost pretplate na događaje
* Obezbediti korisnicima mogućnost brisanja pretplate na događaj, ukoliko izgube interesovanje za pregledavanjem odgovarajućeg događaja
* Obezbediti korisnicima mogućnost ponovnog pregledavanja događaja
* Obezbediti korisnicima mogućnost pregledavanja događaja sa zakašnjenjem, ukoliko propuste početak događaja
* Obezbediti korisnicima obaveštenja kada neki događaj na koji su se oni pretplatili počinje
* Obezbediti korisnicima obaveštenja ukoliko se neki događaj od interesa kreira – pod događajem od interesa se misli na to da je kreiran neki događaj, koji pokriva kategoriju koja je interesantna samom korisniku

**Nefunkcionalni zahtevi**:

* Obezbediti visoku skalabilnost – mogućnost da se, ukoliko dođe do povećanja broja korasnika, proširi lako sistem kako bi mogao da podnese veći broj korisnika
* Omogućiti korisnicima pregledavanje video snimaka bez ikakvog kašnjenja
* Omogućiti pravovremeno pristizanje obaveštenja korisnicima, bez ikakvog kašnjenja
* Obezbediti mogućnost korišćenja funkcionalnosti raspoređivača režine (Load balancing) prema mikroservisima
* Obezbediti da sistem može da funkcioniše i opslužuje veliki broj zahteva od strane korisnika
* Obezbediti mogućnost brzog preuzimanja video datoteka prilikom emitovanja događaja
* Obezbediti centralni sistem preko kojeg će moći da se nadgleda rad mikroservisa koji čine sistem (Centralni logging)
* Visoke performanse po pitanju izvršenja jednostavnih operacija poput kreiranja korisnika i isporučivanja listi događaja – operacije koje će se stalno odvijati
* Bezbednost, autentikacija korisnika
* Obezbediti lako održavanje mikroservisa – zaokružena i ne prevelika funkcionalnost svakog mikroservisa

U okviru ovog potpoglavlja predstavljeni su funkcionalni i nefunkcionalni zahtevi o kojima se vodilo računa prilikom realizacije i implementacije aplikacije. Svi funkcionalni zahtevi su realizovani i svaki nefunkcionalni zahtev je uzet u obzir prilikom dizajniranja arhitekture mikroservisne aplikacije.

## Arhitektura i dizajn

U okviru ovog poglavlja prezetovao bih ukratko svaku od komponenti arhitekture aplikacije pojedinačno ali i zajedno kroz jedan ,,box and line’’ dijagram te bih se u narednom poglavlju detaljnije osvrnuo na implementacione detalje svake od komponenti.

### Slika 4.1 Box-line dijagram arhitekture MyStreamingApp

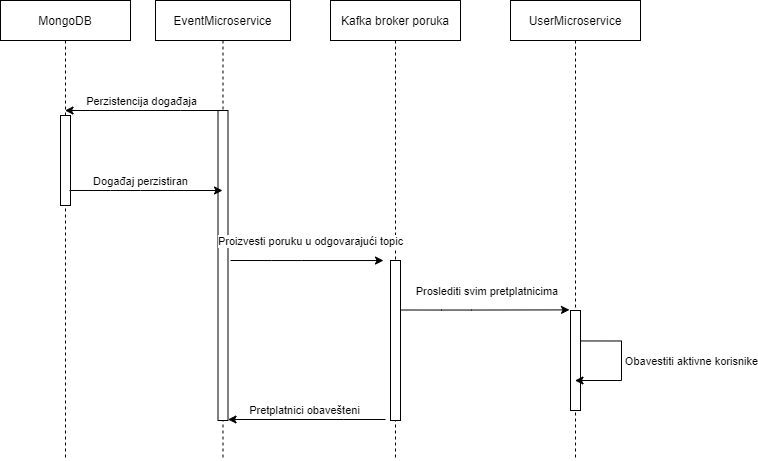
Sa dijagrama se može videti upravo sve prethodno navedeno. MyStreamingApp je izrađena korišćenjem mikroservisne arhitekture, sačinjene od 3 različita mikroservisa pri čemu svaki dizajnirani mikroservis obavlja jednu posebnu funkcionalnost.

**UserMicroservice** je mikroservis čije je zaduženje registrovanje korisnika i obavljanje slanja notifikacija korisnicima. Perzistencija korisnika i njihovih interesovanja se obavlja u relacionoj bazi podataka (MS SQL Server) dok se za potrebe keširanja SignalR konekcija, povezivanje konekcija sa odgovarajućim parametrima (posebne liste konekcija za svaku kategoriju interesovanja) obavlja u Redis kontejneru.

**EventMicroservice** je mikroservis čije je zaduženje da obavlja kreiranje novih događaja i obaveštava ostale mikroservise putem Kafke i toka događaja o kreiranju novog događaja i o tome da emitovanje nekog događaja treba da počne. EventMicroservice obavlja perzistenciju događaja u kolekciji dokumenata MongoDB baze podataka. Svi podaci o događaju i bajtovi samog događaja se smeštaju upravo u navedenoj bazi podataka.

**StreamingMicroservice** je mikroservis čija je namena u potpunosti da obezbeđuje datoteke korisnicima i obavlja takozvani streaming, odnosno emitovanje samog događaja. Njegova uloga je da kada bude obavešten o tome da odgovarajući događaj počinje, pribavi događaj, kešira ga i isporuči svim korisnicima koji se budu odlučili za pregledavanje događaja na koji su se pretplatili. Sam StreamingMicroservise funkcioniše uz pomoć zasebnog Redis kontejnera u okviru kojeg obavlja keširanje bajtova samog video snimka te prilikom svakog emitovanja korisnicima, emituje bajtove iz keša.

U srcu same arhitekture se nalazi Kafka kontejner, odnosno message broker koji nam upravo obezbeđuje mikroservisnu arhitekturu vođenu događajima. Pod terminom ,,vođena događajima’’ se upravo misli na to, da se mikroservisi koji čine samu arhitekturu, sinhronizuju upravo događajima koji se odigravaju u susednim mikroservisima. Odnosno, kada se u nekom mikroservisu odigra neki događaj (poput kreiranje događaja u MyStreamingApp), putem brokera poruka se obaveštavaju svi susedni mikroservisi zainteresovani za taj događaj i svaki od tih mikroservisa obavlja adekvatne akcije koje bi trebalo da uslede i na taj način mikroservisi međusobno sarađuju i obezbeđuju kompletne funkcionalnosti korisnicima. U nastavku ću pokušati putem UML sekvencijalnog dijagrama da bolje objasnim i predstavim događaje koji se odigravaju u MyStreamingApp.



### Slika 4.2 Sekvencijalni dijagram kreiranja događaja

Sa dijagrama se jasno može videti koja je uloga u ovom slučaju brokera poruka. Suština je da kada se dođe do kreiranja novog događaja, mikroservis koji je zadužen za obavljanje te funkcionalnosti vrši perzistenciju samog događaja u bazu podataka, generiše poruku kojom obaveštava zainteresovane mikroservise da je došlo do odgovarajućeg događaja u sistemu(kreiranje novog događaja u MyStreamingApp) te oni potom preuzimaju odgovarajuće akcije kako bi obavili celokupnu funkcionalnost do kraja, u ovom slučaju UserMicroservice obaveštava sve aktivne korisnike da je odgovarajući događaj od interesa kreiran u sistem.

Drugi bitan događaj preko kojeg se obavlja sinhronizacija mikroservisa je početak nekog od perzistiranih događaja u sistemu:

### 

### Slika 4.3 Sekvencijalni dijagram početka događaja

Na slici 4.3 možemo videti UML sekvencijalni dijagram koji reprezentuje aktivnosti između mikroservisa prilikom početka nekog od perzistiranih događaja u sistemu. U ovom slučaju imamo situaciju u kojoj sva tri mikroservisa imaju svoje akcije prilikom početka nekog događaja. EventMicroservice u sebi ima pozadinski servis koji svakog minuta proverava da li ima nekih događaja koji bi trebalo da počnu upravo tog minuta, te nakon toga obaveštava ostale mikroservise putem odgovarajućeg topic-a da je došlo do događaja ,,početak nekog od perzistiranih događaja u sistemu’’. UserMicroservice nakon toga šalje obaveštenje o tome svim aktivnim pretplaćenim korisnicima dok StreamingMicroservice obavlja akcije pripreme, odnosno pribavlja celu datoteku i informacije o događaju, kešira ih i čeka zahteve od korisnika za emitovanjem.