&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](#_Toc95418502)

[1.1. Definisanje problema 3](#_Toc95418503)

[1.2. Organizacija rada 4](#_Toc95418505)

[2. Mikroservisi i mikroservisna arhitektura 6](#_Toc95418506)

[2.1. Mikroservisi 6](#_Toc95418507)

[2.2. Evolucija od monolitne arhitekture u mikroservisnu arhitekturu 7](#_Toc95418508)

[2.3. Mikroservisna arhitektura 8](#_Toc95418511)

[2.4. Koristi mikroservisne arhitekture 10](#_Toc95418513)

[2.5. Najbolje prakse i šabloni dizajniranja 11](#_Toc95418514)

[3. Tehnologije za razvoj mikroservisnih aplikacija 13](#_Toc95418515)

[3.1. Tehnologije za realizaciju konkretnih mikroservisa i streaming funkcionalnosti 15](#_Toc95418518)

[4. OnlineEventsOrganizer – aplikacija i servis za organizovanje online događaja 17](#_Toc95418519)

[4.1. Generalna specifikacija zahteva 17](#_Toc95418520)

[4.2. Arhitektura i dizajn 18](#_Toc95418521)

[4.3. UserMicroservice – Implementacioni detalji 21](#_Toc95418526)

[4.4. EventMicroservice – Implementacioni detalji 24](#_Toc95418530)

[4.5. StreamingMicroservice – Implementacioni detalji 25](#_Toc95418532)

[4.6. Kafka broker poruka – Implementacioni detalji 27](#_Toc95418533)

[5. Opis funkcionalnosti, testiranje i evaluacija 30](#_Toc95418538)

[5.1. Testiranje i evaluacija 38](#_Toc95418550)

[5.2. Testiranje mikroservisne arhitekture 40](#_Toc95418554)

[6. Zaključak 41](#_Toc95418555)

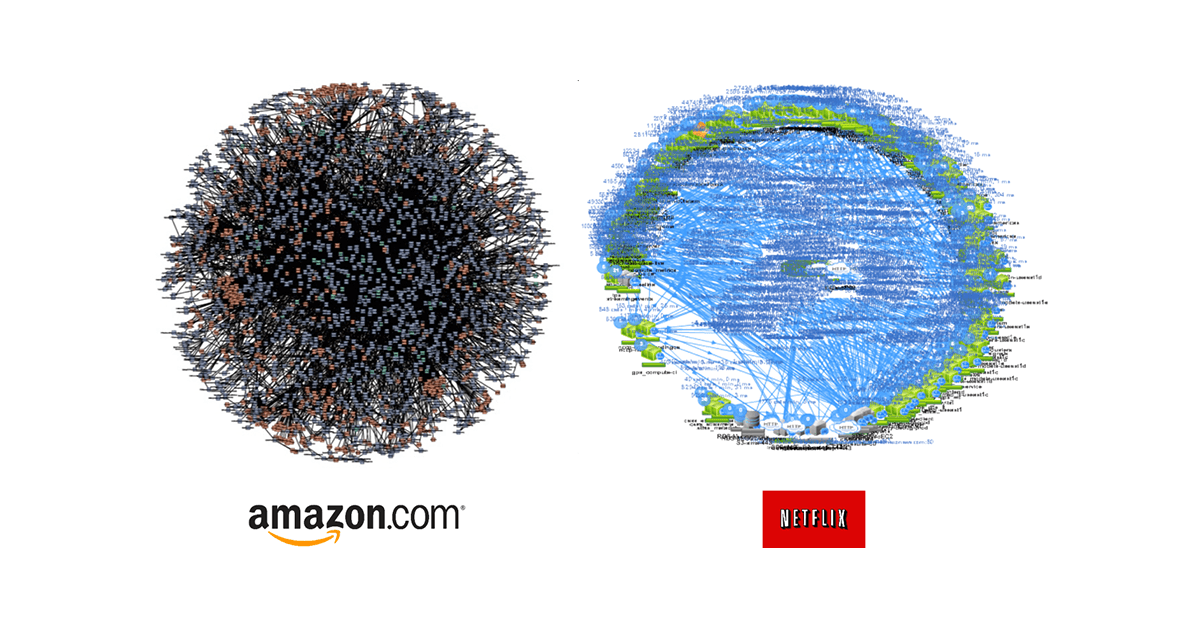
# Uvod

## 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 projektovana 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.

Mikroservisna arhitektura, tačnije takozvana mikroservisna arhitektura vođena događajima (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 rada

U okviru ove podsekcije predstavljen je sadržaj rada po poglavljima. U narednom poglavlju, poglavlju broj 2 pod imenom ,,Mikroservisi i mikroservisna arhitektura” predstavljena je teorijska osnova i opravadana je upotreba 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” opisuje dobre prakse koje su se istakle kroz rad softverskih inženjera kao i tehnologije nastale u okviru tog procesa i predstavlja par tehnologija koje bi bile odlične za razvoj mikroservisnih aplikacija kao i tehnologije koje su danas najviše korišćenije.

Četvrto poglavlje će biti u potpunosti posvećeno aplikaciji koja je razvijena za potrebe ovog rada i koja prati ovaj rad. U okviru tog poglavlja predstavljena je kroz različite dijagrame sama arhitektura aplikacije, njeni funkcionalni i nefunkcionalni zahtevi ali i različiti implementacioni detalji.

U petom poglavlju, priča se nastavlja o samoj aplikaciji i u okviru tog poglavlja predstavljeni su još detaljnije funkcionalni zahtevi te je opisano i samo testiranje i evaluacija aplikacije.

Poslednje, odnosno šesto poglavlje bi trebalo da bude zaključno poglavlje i u okviru tog poglavlja predstavljeni su neki moji zaključci o samoj mikroservisnoj arhitekturi te su i opisane 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, nezavisne 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 u 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 [5]

## Mikroservisna arhitektura



### Slika 2.2 Mikroservisna arhitektura [6]

Na slici 2.2 možemo videti sistem sačinjen od 4 mikroservisa koji obezbeđuju funkcionalnosti korisnicima 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. Uglavnom 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, uglavnom 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.

## Koristi 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 iskoristiti 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 obrazac (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 iskoristiti 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.

**Osigurač (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].

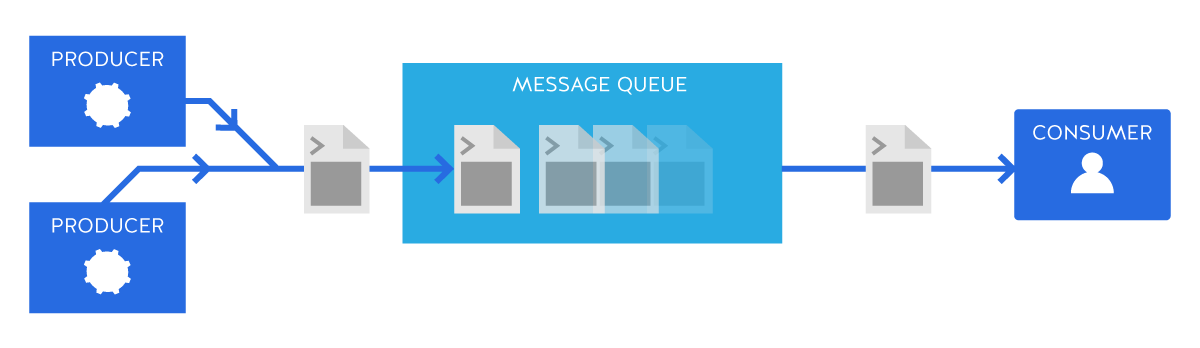
Ovo su samo neki od šablona i dobrih praksi koje sam pomenuo. Pored njih, postoje i drugi pristupi koji se mogu iskoristiti, ja sam se lično dotakao ovih iz razloga što prate samu aplikaciju koja je razvijena ali i obezbeđuju dobre rezultate, odnosno dosta su korišćeni u modernim aplikacijama.

# Tehnologije za razvoj mikroservisnih aplikacija

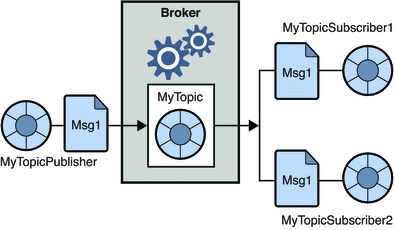
U okviru prethodnog poglavlja, bilo je dosta teorijske osnove o samom tipu arhitekture i pojedinačnim delovima. Verovatno ste primetili da se konkretnih tehnologija koje se mogu iskoristiti nisam previše dotakao iz razloga što smatram da je za to potrebno posvetiti celokupno poglavlje jer može mnogo 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 [14]



### Slika 3.2 Vizuelizacija rada brokera poruka zasnovanog na publish and subscribe šablonu komunikacije [15]

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 protokola. 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.

# OnlineEventsOrganizer – aplikacija i servis za organizovanje online događaja

Kroz prethodna 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.

OnlineEventsOrganizer 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 OnlineEventsOrganizer

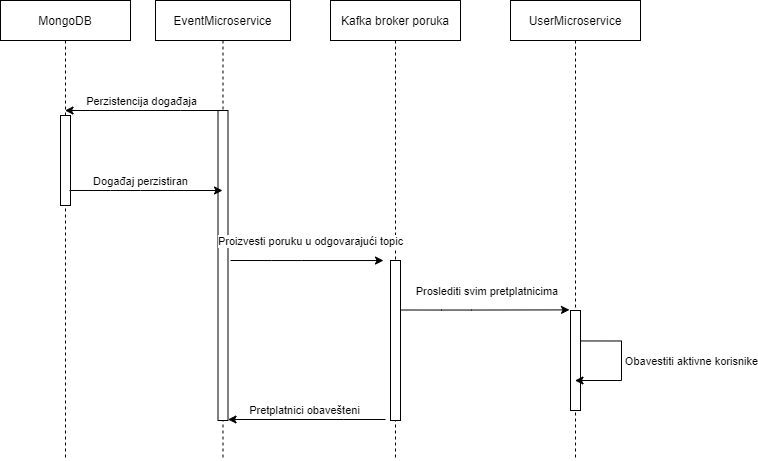
Sa dijagrama se može videti upravo sve prethodno navedeno. OnlineEventsOrganizer je izrađena korišćenjem mikroservisne arhitekture, sačinjene od 3 različita mikroservisa pri čemu svaki dizajnirani mikroservis obavlja jednu posebnu funkcionalnost.

**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 broker poruka (message broker) startovan kao Docker kontejner, 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 OnlineEventsOrganizer) , 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 OnlineEventsOrganizer.



### 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 OnlineEventsOrganizer) 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:

### 

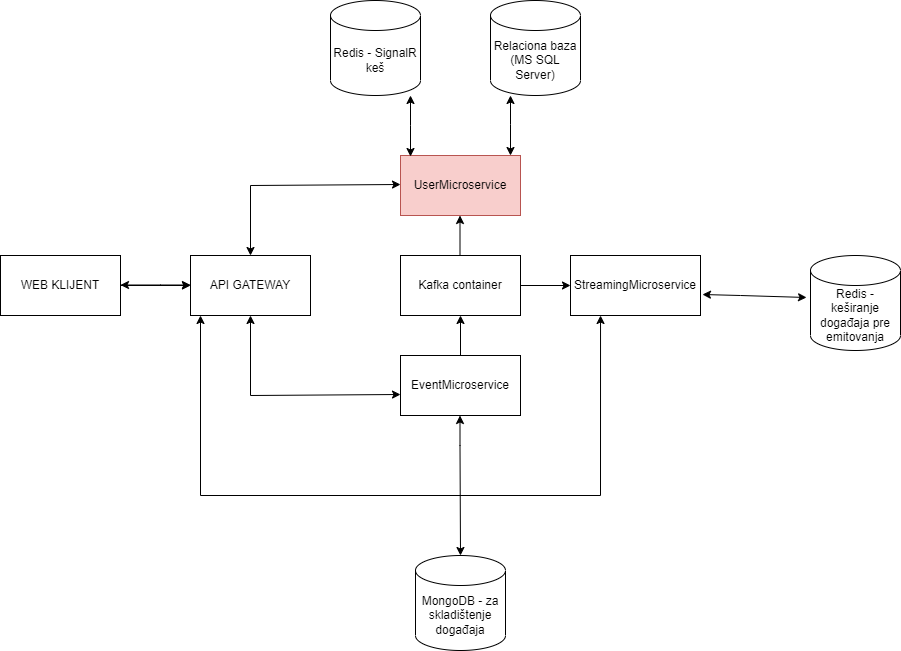
### Slika 4.3 Sekvencijalni dijagram početka događaja

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.

U okviru ovog poglavlja bilo je reči o svakom pojedinačnom mikroservisu, kratak opis bez nekih većih implementacionih detalja. Naredna par poglavlja bih posvetio svakom pojedinačnom mikroservisu i implementaiconim detaljima vezanim za njih, kako bi pokušao da što je detaljnije moguće predstavim svaki od njih.

## UserMicroservice – Implementacioni detalji

*UserMicroservice* je možda i najveći mikroservis u okviru arhitekture aplikacije iz razloga što upravo on obavlja dosta funkcionalnosti. Uloge pomenutog mikroservisa su da obavlja slanje notifikacija korisnicima, kako o tome kada se desi kreiranje nekog događaja od interesa tako i kada je potrebno obavestiti aktivne korisnike ukoliko neki događaj za koji su se oni pretplatili počinje. Pored obavljanja funkcionalnosti slanja notifikacija, *UserMicroservice* je zadužen i za perzistenciju korisnika, logovanje samih korisnika u sistem. Za potrebe pomenute funkcionalnosti perzistencije korisnika, *UserMicroservice* koristi relacionu MS SQL Server bazu podataka.

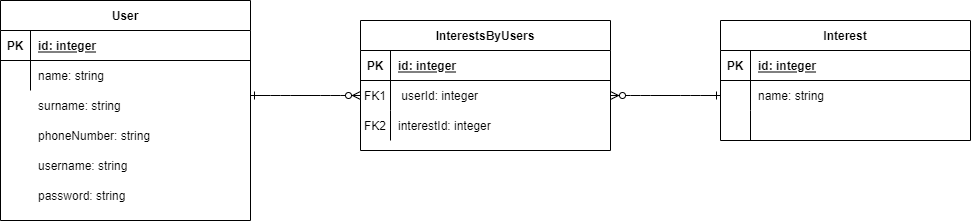


### Slika 4.4 Lokacija UserMicroservice u sistemu

Na slici 4.4 možemo jasno videti koja je lokacija UserMicroservica u okviru celokupnog sistema i na koji način učestvuje u komunikaciji između internih komponenti koje čine realizovani sistem. Može se primetiti da je upravo strelica od brokera poruka okrenuta ka UserMicroservicu. Razlog za to je što je *UserMicroservice* samo pretplatnik na događaje u celom sistemu, dakle on ne generiše nijedan događaj koji je od interesa ostalim mikroservisima, već jednostavno reaguje na događaje koje generišu ostali mikroservisi u sistemu (konkretno *EventMicroservice*) i obavlja odgovarajuće akcije kada dođe do toga.

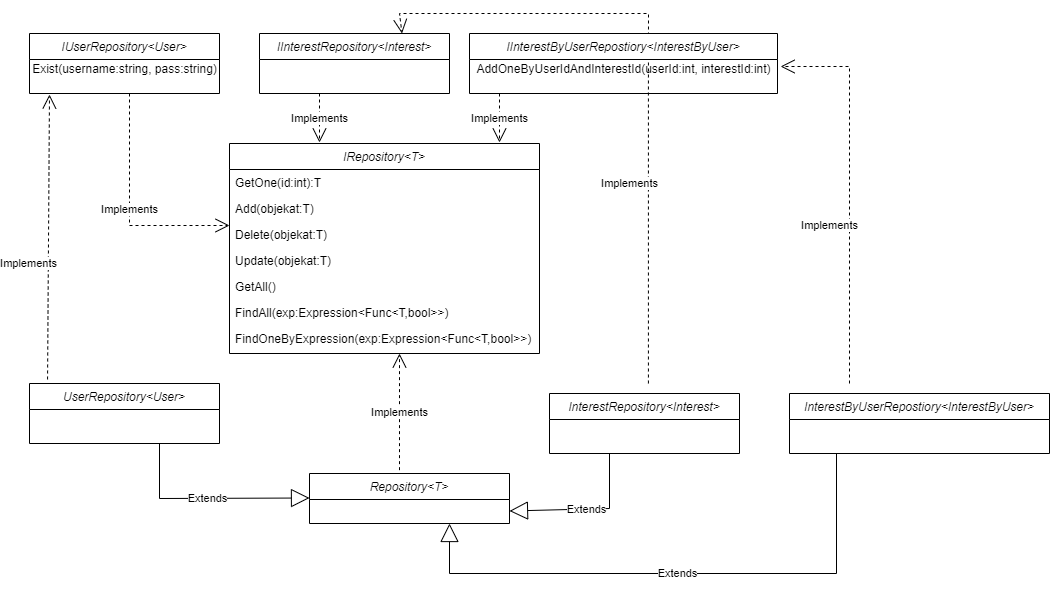
Kao što sam već pomenuo, *UserMicroservice* kao i ostala dva mikroservisa su realizovana uz pomoć ASP .NET Core tehnologije i uz upotrebu dodatnih biblioteka i alata kako bi se olakšao rad sa bazama podataka. Konkretno celokupan pristup relacionoj bazi podataka je realizovan uz pomoć Microsoftovog Entity Framwork Core-a koji obavlja funkcionalnosti ORM-a (Object relational mapper) i olakšava rad sa samom bazom podataka.

U nastavku sledi sama šema relacione baze podataka:



### Slika 4.5 Šema relacione baze podataka

Relaciona baza podataka se sastoji iz 3 tabele, i jedne veze više prema više. Za svakog korisnika se beleži koja interesovanja on iskazuje i na osnovu tih podataka, korisnik prima obaveštenja o tome kada se kreira neki događaj koji obuhvata interesovanja za koje je korisnik zainteresovan.  
  
*UserMicroservice* je organizovan tako da sam mikroservis predstavlja višeslojnu arhitekturu, odnosno u strukturi samog mikroservisa postoji nivo servisa, u kojem je smeštena celokupna poslovna logika kao i nivo pristupa podacima koji implementira takozvani Unit of work šablon dizajniranja u kombinaciji sa Repository dizajn šablonom. U nastavku sledi klasni dijagram koji opisuje realizaciju prethodno pomenutog:



### Slika 4.6 Repository pattern

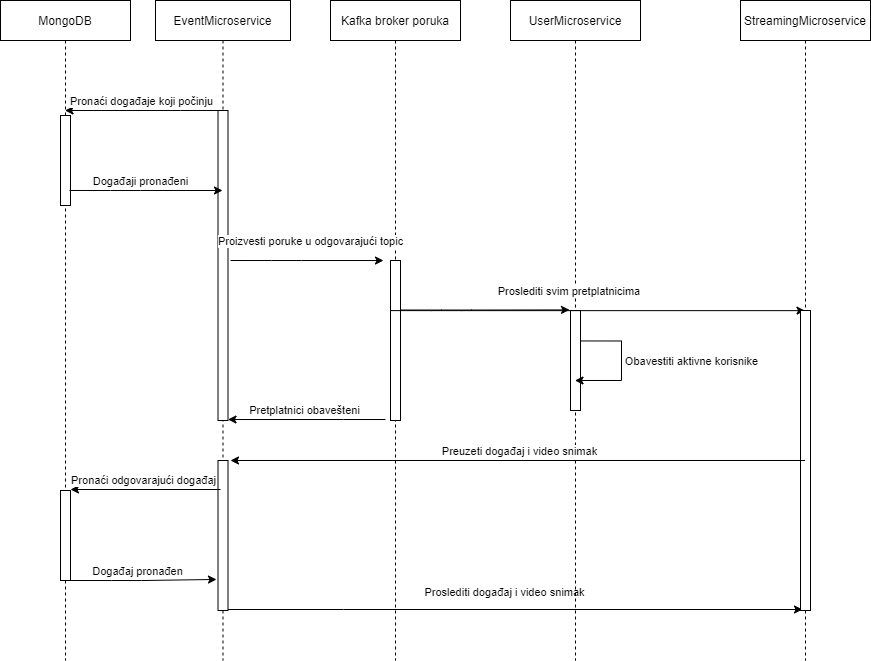
Na slici 4.5 se može videti klasni dijagram kako je realizovan repository dizajn šablon. Pored njega, realizova je i Unit of work dizajn šablon sloja pristupa podacima koji obezbeđuje lakši način organizovanja transakcija prema relacionoj bazi podataka.  
  
Za slanje obaveštenja korisnicima iskorišćen je SignalR alat i Redis baza podataka startovana kao Docker kontejner kao podrška za keširanje svih potrebnih stvari za rad sa SignalR-om. Pod potrebnim stvarima pre svega mislim na keširanje ID-jeva konekcija aktivnih korisnika. Sam model podataka Redis baze podataka je dosta pogodan za takve potrebe i intuitivan te je za potrebe same OnlineEventsOrganizer bio dosta očigledno rešenje. Konkretno u OnlineEventsOrganizer u Redis bazi podataka u svakom trenutku postoji po jedna lista za svaku kategoriju interesovanja i kako se korisnici povezuju u sistem/isključuju sa sistema, tako se knjihove konekcije dodaju i brišu iz lista i uz pomoć ovakvog pristupa u svakom trenutku imamo listu aktivnih korisnika kojima je moguće poslati notifikacije. Za lakši rad sa Redis-om i pristup podacima, iskorišćena je open source biblioteka pod imenom: ServiceStack.Redis

## EventMicroservice – Implementacioni detalji

*EventMicroservice* je mikroservis čija je uloga u sistemu da obavlja sve aktivnosti vezane za perzistenciju i aktiviranje događaja koji treba da budu emitovani. Dakle, pomenuti mikroservis obavlja perzistenciju svih događaja i konkretnih bajtova video snimaka koji prate događaje. Događaji se kreiraju na zahtev korisnika i bivaju dostupni za pregledanje svim pretplaćenim korisnicima u vreme u koje je postavljeno prilikom kreiranja samog događaja.  
*EventMicroservice* je glavni proizvođač događaja pomoću kojih je realizovana mikroservisna arhitektura vođena događajima (Event-driven) . Dakle, on je zadužen da obavesti ostale mikroservise kada se kreira neki događaj i kada neki događaj treba da počne sa emitovanjem. To se obavlja preko brokera poruka i upravo zbog toga je strelica u dijagramu arhitekture okrenuta od *EventMicroservice*-a ka brokeru poruka. On proizvodi događaje, ostali mikroservisi bivaju obavešteni o tome te preduzimaju dalje akcije prilikom toga.

Celokupna perzistencija događaja se obavlja u bazi podataka orjentisanoj dokumentima (Document oriented NoSQL baza potadaka) . Konkretno je MongoDB iskorišćen kao skladište za perzistenciju svih događaja kreiranih od strane korisnika. Postoji jedna kolekcija dokumenata i u okviru te kolekcije dokumenta se smeštaju svi događaji koji su kreirani od strane korisnika.

U toku rada *EventMicroservice*-a, u pozadini je stalno pokrenut pozadinski servis koji svakog minuta proverava da li je potrebno početi sa emitovanjem nekog od događaja i ukoliko je taj uslov ispunjen, putem brokera poruka se obaveštavaju ostali mikroservisi da treba početi sa emitovanjem odgovarajućeg događaja. Svaki od preostala dva mikroservisa ima svoje aktivnosti koje obavlja u tom slučaju, a to je najlakše prezentovati adekvatnim UML sekvencijalnim dijagramom sa kojim smo se već upoznali:



### Slika 4.7 Aktivnosti prilikom početka emitovanja nekog od događaja

Kada pozadinski servis *EventMicroservice* mikroservisa detektuje da je potrebno početi sa emitovanjem nekog od događa, on obaveštava ostale mikroservise o tome. Konkretno, *UserMicroservice* preduzima nadležnost o tome da obavesti aktivne pretplaćene korisnike da događaj za koji su oni zainteresovani i pretplaćeni počinje te da bi trebalo da se uključe i pregledaju događaj dok *StreamingMicroservice* preuzima celokupan događaj i prateću video datoteku od EventMicroservica, kešira tu datoteku i potom čeka zahteve od korisnika za emitovanjem događaja.

Za pristup i rad sa MongoDB-jem je iskorišćena pomoćna biblioteka pod imenom MongoDB.Driver koja obezbeđuje olakšani pristup samoj bazi podataka, kako za čitanje tako i za upis i modifikaciju postojećih podataka u samoj bazi.

## StreamingMicroservice – Implementacioni detalji

*StreamingMicroservice* je mikroservis koji ima najmanje funkcionalnosti ali je upravo mikroservis koji treba da bude najbrži i da obezbeđuje korisnicima preuzimanje podataka bez ikakvih kašnjenja. Ovaj mikroservis ima ulogu u sistemu da obezbedi korisnicima da preko jednostavnih HTTP zahteva preuzimaju video datoteke koje prate događaje na koje su se oni pretplatili te da nakon toga kroz svoje web pregledače pregledavaju događaje.

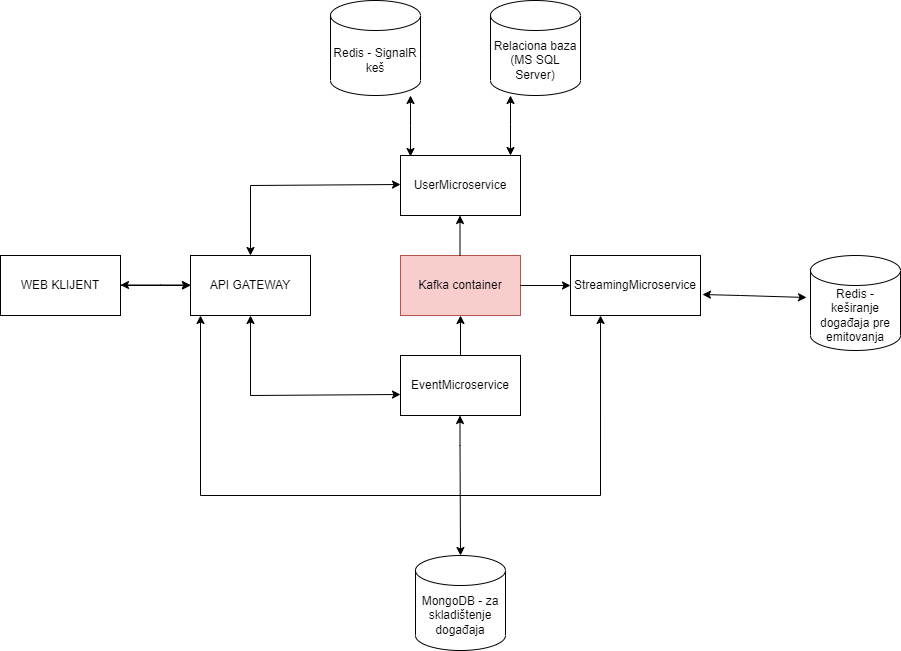
S obzirom da je jedan od zahteva koje mikroservisna arhitektura treba da ispuni, upravo zahtev da između mikroservisa treba da postoji slaba sprega, odnosno da mikroservisi same arhitekture međusobno ne zavise jedan od drugog. *StreamingMicroservice* je u potpunosti nezavisan mikroservis od ostalih i u skladu sa tim on može da bude repliciran više puta, skaliran, te da ukoliko se naglo poveća broj korisnika koji pregledaju video snimke, podnese to bez ikakvih problema. Mikroservis bi se instancirao više puta, zahvaljući API Gateway-u kao jedinstvenom ulazu u naš sistem obavljala bi se funkcionalnost rutiranja i raspoređivanja težine (Load balancing) , odnosno svi zahtevi ka *StreamingMicroservice*-u bi se ravnomerno prosleđivali svim instancama tog mikroservisa, svaki od njih bi obavljao podjednak obim posla i svo kašnjenje i obim posla koja obavlja jedna instanca mikroservisa bi se značajno smanjila te bi se to naravno odrazilo na poboljšanje samih performansi te i zadovoljstvu krajnjeg korisnika.

Mikroservis zadužen za emitovanje video sadržaja korisnicima funkcioniše u saradnji sa zasebnom instancom Redis baze podatake u okviru koje on održava informacije o svim aktivnim događajima koji bi trebalo da budu dostupni korisnicima ali i vrši keširanje bajtova samih video snimaka. Celokupan rad pomenutog mikroservisa se zasniva na tome da korisnici video snimke preuzimaju korišćenjem više HTTP zahteva tako što se video snimci, video datoteke, preuzimaju deo po deo (chunk by chunk) svakim pojedinačnim HTTP zahtevom.

Za rad sa samom Redis bazom podataka i ovaj mikroservis koristi open source biblioteku sa istim imenom koja je iskorišćena i u *UserMicroservice* mikroservisu. Pomenuta biblioteka omogućava developerima mogućnost da jednostavnim pozivima funkcija se pristupa listama, heš tabelama, ključevima i uz pomoć pomenutih funkcija je takođe moguće lako vršiti perzistenciju kao i modifikaciju već perzistiranih podataka u okviru baze podataka.

*StreamingMicroservice* kao i ostali mikroservisi u potpunosti zavisi od Kafke (brokera poruka) jer putem nje on biva obavešten o tome da je potrebno početi sa emitovanjem odgovarajućeg događaja te nakon toga on preuzima sve prateće stvari oko događaja od mikroservisa zaduženog za perzistenciju događaja, kešira te informacije u svom kešu i nakon toga on je spreman da prima zahteve od korisnika i isporučuje korisnicima video sadržaj.

## Kafka broker poruka – Implementacioni detalji

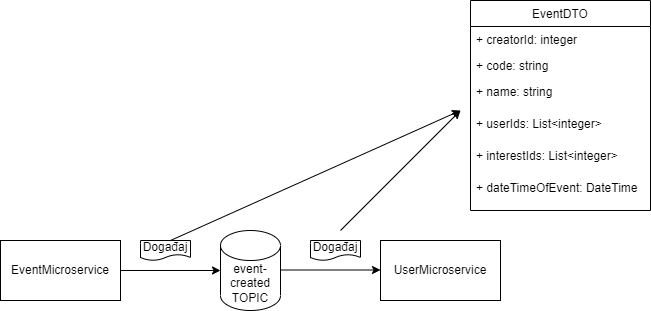


### Slika 4.8 Broker poruka kao srce celog sistema

Kafka broker poruka je centar celokupnog sistema, preko njega se obavljaju sve aktivnosti u samom sistemu i preko njega se obavlja glavna i najbitnija komunikacija i sinhronizacija između mikroservisa. Pošto je implementirana arhitektura mikroservisna arhitekura vođena događajima, celokupan sistem funkcioniše tako što se u sistemu dešavaju odgovarajući događaji, postoje mikroservisi koji su zaduženi za generisanje događaja ali i ostali mikroservisi koji preduzimaju odgovarajuće akcije kada dođe do nekog od događaja koji su od interesa za te mikroservise.

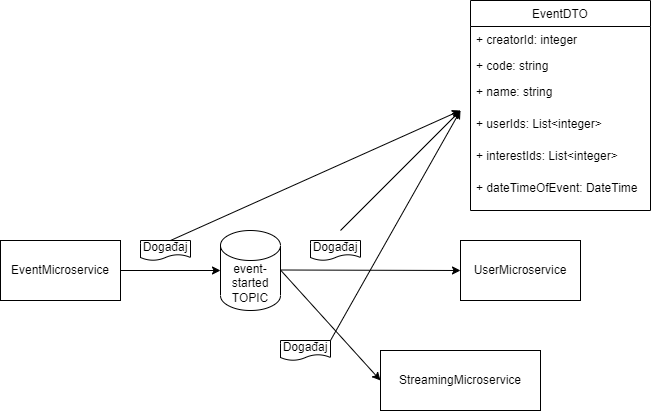
Kafka broker poruka implementira takozvani ,,Publish and subscribe’’ model komunikacije koji zahteva održavanje u okviru samog kontejnera brokera poruka odgovarajuće strukture podataka takozvane ,,topic’’ i preko tih struktura podataka se obavlja komunikacija između mikroservisa. Navedi model komunikacije obezbeđuje tip komunikacije jedan na više, odnosno kada jedan od mikroservisa generiše poruku, ta poruka može biti prosleđena na više odredišta u zavisnosti od broja pretplaćenih mikroservisa na taj događaj/topic.

Za realizaciju OnlineEventsOrganizer sistema, u okviru Kafke postoje dva topic-a preko kojih se obavlja komunikacija između mikroservisa. Jedan od njih je nazvan event-started topic kojim se obavlja komunikacija između *EventMicroservice* mikroservisa i *UserMicroservice* mikroservisa. Inicijator je *EventMicroservice* koji putem navedenog topic-a obaveštava *UserMicroservice* da je došlo do kreiranja novog događaja u sistem, i kao sadržaj poruke šalje upravo podatke o tom novokreiranom dođaju pri čemu ti podaci uključuju i kategorije koje taj događaj obuhvata. Na slici 4.9 možemo videti vizuelizovano kako izgleda komunikacija putem event-created topic-a.



### Slika 4.9 Vizuelizacija komunikacije preko event-created topic-a

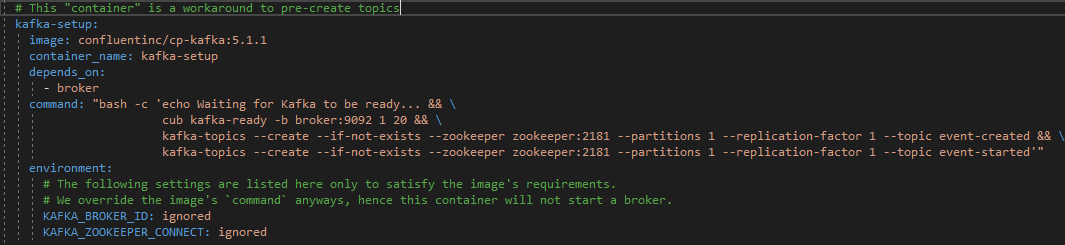
Pored event-created topic-a, koristi se i event-started topic koji ima ulogu da se preko njega signalizira od strane *EventMicroservice* mikroservisa ostalim mikroservisima da je potrebno obezbediti početak emitovanja odgovarajućeg događaja. Pomenuti mikroservis šalje putem navedenog topic-a informaciju o događaju čije emitovanje treba da počne, a ostali mikroservisi obavljaju dalje akcije koje potrebno da obave. Na slici 4.10 možemo videti vizuelizovano kako teče tok komunikacije putem event-started topic-a:



### Slika 4.10 Vizuelizacija komunikacije preko event-started topic-a

S obzirom da je Kafka broker poruka srce same arhitekture, za rad celokupnog sistema, pre pokretanja svih mikroservisa, potrebno je bilo obezbediti da je broker poruka pokrenut i da poseduje u sebi instancirane odnosno kreirane sve pomenute topic-e. Ovaj zahtev je realizovan uz pomoć alata koji je iskorišćen za orkestraciju Docker kontejnera (docker-compose) . Uz pomoć pomenutog alata, postignut je potreban redosled pokretanja mikroservisa i pratećih komponenti tako da se komponente i mikroservisi pale po redosledu zavisnosti, odnosno prvo se pokreću delovi sistema koji ne zavise od drugih delova te potom oni delovi koji vuku zavisnost od nekih drugi delova.

Da bi se obezbedio takođe i zahtev da pre pokretanja svih mikroservisa postoje već napravljeni topic-i za komunikaciju, realizovan je mali kontejner koji se pokreće uporedo sa kontejnerom brokera poruka i koji u okviru petlje sa pauzama, čeka sve dok broker poruka nije aktivan i kreira potrebne topic-e ukoliko oni ne postoje. Na slici 4.11 može se videti isečak koda iz .yml fajla koji se koristi za konfigurisanje docker-compose-a za orkestraciju kontejnera. Na isečku se upravo nalazi deo koda kojim se instancira opisani kontejner kao i prikaz implementacije samog kontejnera.



### Slika 4.11 kafka-setup kontejner

Pored pomenutih zavisnosti oko brokera poruka, obezbeđeno je orkestracijom da se prvo aktiviraju sve baze podataka a tek potom mikroservisi koji koriste te baze podataka, što bi naravno i bio neki logički redosled akcija.

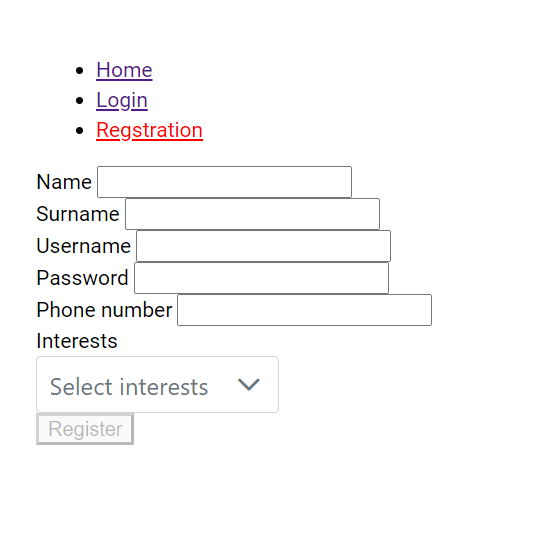
U okviru ovog poglavlja, upoznali smo se sa implementacionim detaljima svih komponenti i mikroservisa koji čine OnlineEventsOrganizer sistem. U narednom poglavlju ću se potruditi da detaljnije obradim sve funkcionalnosti OnlineEventsOrganizer sistema te da prikažem kako sve to izgleda i kako ide sama upotreba u aplikaciji. Osvrnuću se još jednom na sve funkcionalne zahteve koji su postavljeni na početku i prikazati kako su oni realizovani u samoj aplikaciji. Pre svega toga, napomenuo bih da je sama aplikacija razvijena u demonstracione svrhe, odnosno da je samo razvijena za potrebe testiranja realizovane arhitekture, tako da u skladu sa tim zahtevom, dizajn i korisnički interfejs same aplikacije je na niskom nivou.

# Opis funkcionalnosti, testiranje i evaluacija

Kroz prethodna poglavlja bilo je dosta reči o teoriji koja prati samu mikroservisnu arhitekturu zasnovanu događajima, o savetima i stvarima koje trebe uzeti u obzir prilikom projektovanja same arhitekture, ali i o samoj realizovanoj arhitekturi kroz aplikaciju (OnlineEventsOrganizer) koja je razvijena za potrebe ovog rada. Potrudio sam se da prvo postavim teoretsku osnovu te da bih potom sve to prikazao kroz primer, odnosno kako je sve to, barem većina stvari koje sam pomenuo, realizovana praktično kroz aplikaciju OnlineEventsOrganizer. Počev od samih mikroservisa do njihove integracije uz pomoć brokera poruka i uz pomoć API Gateway koji služi kao jedinstven ulaz u sistem.

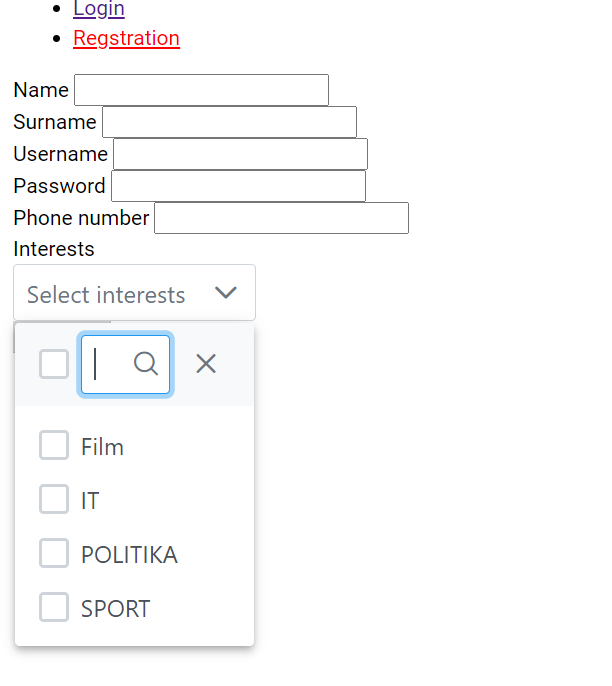
U okviru ovog poglavlja, posvetio bih pažnju samim funkcionalnim zahtevima koji su pomenuti u okviru poglavlja 4.1, potrudiću se da ih objasnim još detaljnije te i da pokažem kako su oni realizovani u samoj aplikaciji i kako sve to izgleda. Ovo poglavlje će pratiti dosta slika koje prikazuju kako je sve realizovano u samoj aplikaciji kako bi lakše mogao da prezentujem sve.

Prvi i najosnovniji zahtev glasi da je potrebno obezbediti korisnicima mogućnost registrovanja u sistem. OnlineEventsOrganizer je sistem koji omogućuje korisnicima kreiranje različitih događaja, koje drugi korisnici mogu pregledavati ali i pretplaćivanje i pregledavanje događaja kreiranih od strane drugih korisnika. Da bi sve to bilo moguće, potrebno je naravno obezbediti mogućnost korisnicima da kreiraju naloge te i da se potom prijave u sistem uz pomoć kreiranog naloga.   
Prilikom registracije u sistem, korisnici moraju da izaberu odgovarajuće kategorije za koje su oni zainteresovani. Upravo ta tačka prilikom pravljenja naloga je možda i najbitnija, jer celokupna poslovna logika će se potom vrteti oko toga. Dakle, korisnik prilikom registracije bira više kategorija ka kojima on gaji interesovanje i upravo će sistem biti naklonjen tome da obaveštava korisnika kada bilo koji dođaj bude kreiran u sistem ukoliko, naravno, taj događaj pokriva neku od kategorija o kojima je korisnik zainteresovan.



### Slika 5.1 Izgled registracione forme

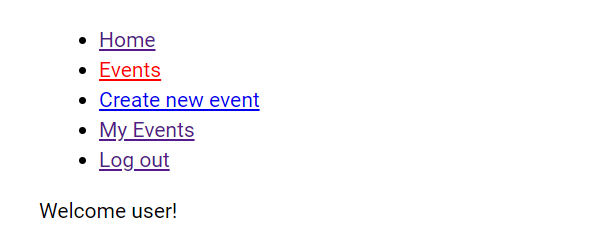
Na slici 5.1 možemo videti kako izgleda registraciona forma OnlineEventsOrganizer sistema. Pored interesovanja, korisnik je dužan i da unese neke podatke o sebi.



### Slika 5.2 Padajuća lista sa interesovanjima

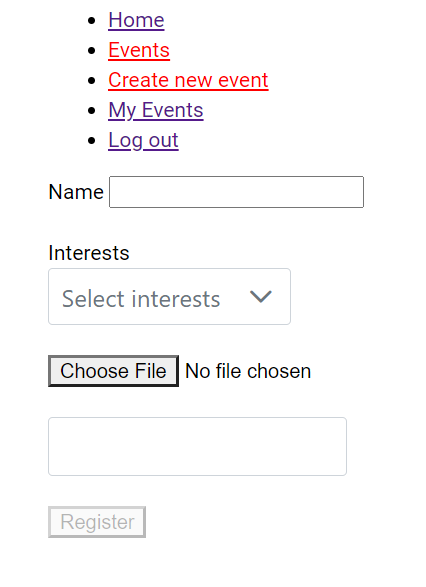
Korisniku je ostavljena mogućnost, kao što je već rečeno da prilikom registracije izabere veći broj kategorija događaja (interesovanja) za koje je on zainteresovan. S obzirom da je ovo demo aplikacija, razvijena samo za potrebe testiranja projektovane mikroservisne arhitekture, trenutno postoji samo par kategorija pri čemu bi naravno, prilikom komercijalizacije aplikacije sve to bilo opširnije.

Pored registrovanja u sistem, drugi funkcionalni zahtev koji je dosta bitan je upravo i samo kreiranje događaja. Kreiranje događaja takođe pratim forma slična formi za registrovanje korisnika i da bi korisnik uopšte mogao da kreira događaj, da bi mu se otvorila ta opcija, potrebno je da korisnik poseduje nalog u okviru sistema na koji će se prijaviti i tek potom prijave u sistem korisniku će se prikazati drugačiji meni koji će posedovati upravo i opciju za kreiranje događaja.



### Slika 5.3 Izgled menija nakon prijave korisnika

Kao što je već rečeno, nakon prijavljivanja u sistem korisniku se prikazuje potpuno drugačiji meni koji upravo sadrži i opciju za kreiranjem događaja. Klikom na tu opciju korisniku se otvara pomenuta forma i korisnik je dužan da unese osnovne potrebne stvari o samom događaju. Pod osnovnim stvarima se naravno i podrazumeva lista kategorija koje događaj koji se kreira pokriva i upravo na osnovu te liste se šalje notifikacija svim aktivnim korisnicima da je kreiran događaj od interesa. Taj dođaj će biti od interesa naravno samo korisnicima koji su prilikom registracije postavili da su zainteresovani za neku od kategorija koje novokreirani događaj pokriva.



### Slika 5.4 Izgled menija za kreiranje događaja

Na slici 5.4 možemo videti kako pomenuta forma za kreiranje događaja i izgleda. Sve stvari koje se nalaze na formi su obavezne i to su neke najosnovnije informacije koje je potrebno obezbediti. Pored same video datoteke koja će biti poslata ka serveru i kasnije biti emitovana, potrebno je izabrati i vreme emitovanja događaja kao i kategorije koje taj događaj pokriva. Klikom na polje iznad samog dugmeta za potvrdu kreiranja se otvara kalendar preko kojeg je lako izabrati vreme i datum emitovanja događaja.

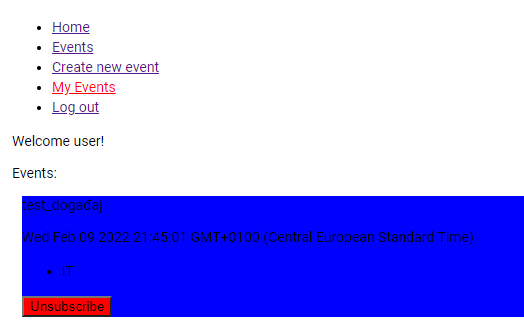
Pored opcije za kreiranje događaja, meni sadrži još opcije za prikaz svih događaja koji još uvek nisu povezani sa samim korisniku (pod povezanim se misli da ih korisnik nije kreirao i da to nisu događaji na koje se korisnik pretplatio) ali i za prikaz povezanih događaja.   
Nakon registrovanja, odnosno kreiranja događaja korisnik biva premešten na stranicu koja prikazuje listu svih događaja sa kojima je on povezan.   
  
U okviru liste koja sadrži sve događaje sa kojima korisnik nije povezan još uvek, za svaki prikazani događaj je ostavljena opcija pretplate, odnosno ostavlja se mogućnost korisniku da se poveže sa željenim događajem odnosno da se pretplati na taj događaj. Opcija pretplate omogućava korisniku mnoge stvari, pre svega najbitniju, omogućava korisniku pregledavanje događaja, bez pretplate korisniku nije ostavljena mogućnost pregledavanja događaja. Korisnik se može pretlatiti na događaj sve dok se ne počne sa emitovanjem događaja odnosno dok se ne dođe do termina koji je postavljen kao datum i vreme emitovanja događaja prilikom kreiranja događaja. Korisniku će stizati notifikacije o tome kada događaji na koje se on pretplatio počinju.



### Slika 5.5 Lista događaja sa kojima korisnik nije povezan

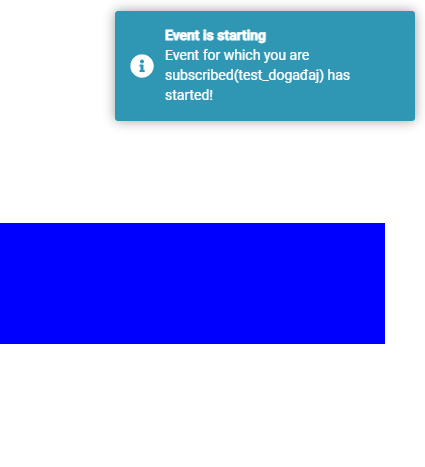
Kao što je rečeno, korisniku se ostavlja mogućnost pretlate na događaj sve do samog početka emitovanja događaja. Jednostavnim klikom na dugme za pretplatu, korisnik ostvaruje pretplatu na događaj i biva preusmeren na stranicu koja prikazuje listu svih događaja sa kojima je on povezan. Tamo će biti prikazani svi događaji na koje se korisnik pretplatio ali i događaji koje je sam korisnik kreirao. Podrazumeva se da je korisnik automatski pretplaćen na sve događaje koje je on kreirao.

Lista sa povezanim događajima sadrži dve opcije, jedna će biti aktivna uvek, sve dok postoji mogućnost za pregledavanjem događaja, a to je opcija kojom se ukida pretplata na događaj dok je druga opcija upravo opcija kojom se započinje pregledavanje događaja. Opcija za pregledavanjem događaja će biti aktivna tek po početku emitovanja događaja i ostaje prisutna sve dok korisnik ima pravo za pregledavanjem događaja. Poslovna logika OnlineEventsOrganizer sistema omogućava korisnicima pregledavanje događaja 2 sata nakon početka emitovanja događaja i ne postoji ograničenje koliko puta korisnik može da pregleda događaj u tom vremenskom periodu. Time su ispunjena dva funkcionalna zahteva koja zahtevaju da korisniku bude omogućeno ponovno pregledavanje događaja ali i funkcionalni zahtev koji kaže da je potrebno omogućiti korisniku pregledavanje događaja sa zakašnjenjem, odnosno ukoliko korisnik ne bude u mogućnosti da odmah u trenutku emitovanja događaja pogleda događaj, ostavljeno je vreme od 2 sata za koje korisnik može da pregleda video snimak koji prati sami događaj.



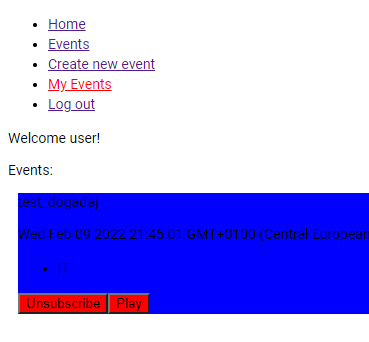
### Slika 5.6 Izgled liste događaja koji su povezani sa korisnikom

Na slici 5.6 možemo videti kako sama lista događaja, sa kojima je korisnik povezan, izgleda. Možemo videti da trenutni događaj još uvek nije počeo te da korisniku još uvek nije ostavljena mogućnost za pregledavanjem događaja. Kada sat otkuca odgovarajuće vreme, korisniku će biti prikazana i opcija u vidu dugmeta kojom korisnik može započeti pregledavanje događaja. Naravno, pre nego što korisnik započne pregledavanje samog događaja, ispunjava se još jedan funkcionalni zahtev odnosno korisniku stiže obaveštenje u gornjem desnom uglu o tome da događaj na koji je on pretplaćen počinje sa emitovanjem.



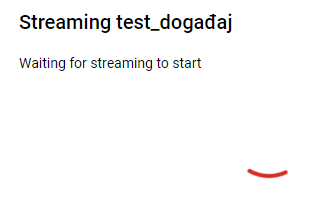
### Slika 5.7 Obaveštenje o početku događaja

Nakon pristizanja obaveštenja, kada korisnik otvori stranicu sa povezanim događajima, korisnik može videti da sada i ima opciju za pregledavanjem događaja u vidu dodatnog dugmeta.



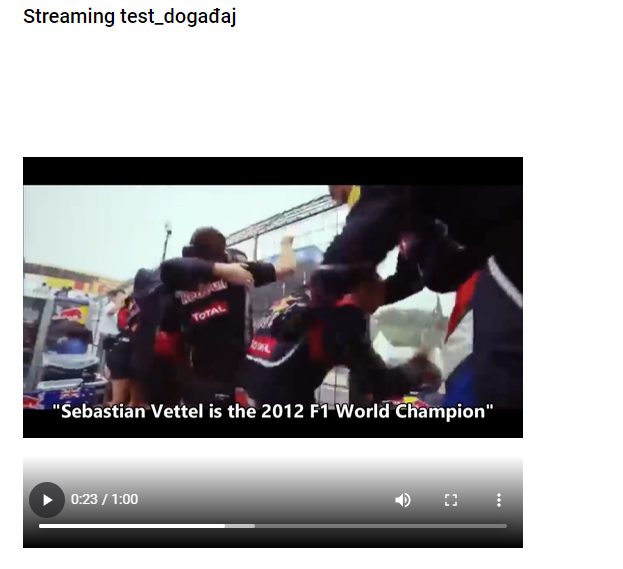
### Slika 5.8 Opcija za pregledavanjem događaja

Klikom na dugme za pregledavanje događaja, korisnik biva preusmeren na stranicu sa video plejerom preko kojeg se ostvaruje pregledavanje događaja. Klikom na play dugme video plejera će i sam streaming početi. Korisniku je ostavljena mogućnost da vrši različita premotavanja i pauziranja događaja te da sam kontroliše kako će streaming ići radi lakšeg hvatanja beleški ali i ponovnog preslušavanje nekih stvari.   
Ukoliko sam streaming video snimka iz nekog razloga kasni, korisniku će biti prikazan takozvani spinner koji ostaje aktivan sve dok emitovanje samog video snimka ne počne. U nastavku možete videti kako sve to izgleda.



### Slika 5.9 Čekanje na početak emitovanja događaja

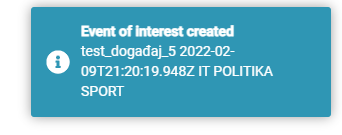
S obzirom da je cela arhitektura realizovana tako da su performanse na prvom mestu, ekran koji obaveštava korisnika da video snimak još uvek nije dostupan traje dosta kratko i velike su šanse da sam korisnik neće ni videti to. Razlog za postojanjem ove poruke je da se pokrije situacija da je korisnik toliko brzo pristupio video snimku da je bio brži od same razmene video snimka između *EventMicroservice* mikroservisa i *StreamingMicroservice* mikroservisa, što je u realnosti naravno malo teže postići.



### Slika 5.10 Pregledavanje video snimka koji prati događaj

Na slici 5.10 može se videti kako samo pregledavanje video snimka koji prati kreirani događaj izgleda.

U okviru ovog poglavlja, bilo je reči i o notifikacijama koje korisnik prima kada se kreira neki događaj koji je od interesa za tog korisnika i to je upravo poslednji funkcionalni zahtev koji je trebalo da OnlineEventsOrganizer realizuje. Pod događajem od interesa se misli pre svega da taj događaj pokriva neku od kategorija, može i sve, o kojima korisnik gaji interesovanje. Na slici 5.11 možemo videti izgled notifikacija koje korisnik prima u tom slučaju. Kao i notifikacija o početku događaja, notifikacija odnosno obaveštenje o kreiranom događaju od interesa se pojavljuje u gornjem desnom uglu web pregledača. U okviru obaveštenja, korisniku će biti prikazane i kategorije novokreiranog događaja.



### Slika 5.11 Izgled obaveštenja o novokreiranom događaju

U okviru ovog poglavlja pokušao sam da još detaljnije predstavim sve funkcionalne zahteve OnlineEventsOrganizer sistema i prikažem pojedinačno kako je svaki od njih realizovan u okviru aplikacije. Naravnom, da bi svaka aplikacija opravdala očekivanja i došla u poziciju da bude na korak od upotrebe od strane samih klijenata, svaka aplikacija mora da prođe kroz odgovarajuća testiranja i provere pre dolaska do samih klijenata. U okviru narednog poglavlja ću pokušati da pojasnim koje faze testiranja je OnlineEventsOrganizer aplikacija prošla i kako je moguće garantovati da je sam sistem i aplikacija spreman za upotrebu od strane korisnika.

## Testiranje i evaluacija

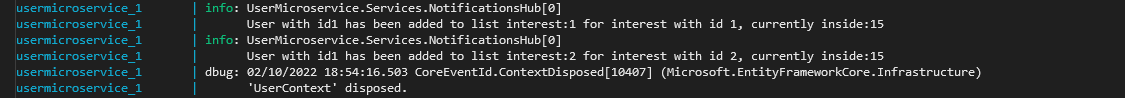
Faza testiranja je faza u razvoju softvera koja je podjednako bitna kao i samo razvijanje i programiranje. Testiranjem softvera se postize garancija kvaliteta, odnosno samim testiranjem softvera, dobrim testiranjem, izbegavaju se situacije da greške dolaze do klijenata. U današnje vreme, postoje mnogo različitih metodologija testiranja, od različitih jediničnih testova (unit testova) do integracionih testova. Zajedničko za sve ove testove je upravo to što kako god da teče razvijanje softvera, u kojem god smeru da ide, pomenuti testovi se uvek mogu pokrenuti i pomoću njih se uvek mogu proveriti funkcionalnosti koje navodno nisu bile u dodiru sa najnovijim modifikacijama na softveru. Kažem navodno iz razloga zato što verujem da nam se svima nama dešavalo da vršimo izmene na jednom delu aplikacije a to se manifestuje greškama na potpunoj drugoj strani. Svrha ovih testova je upravo da se spreče te stvari, odnosno da se ne izazovu neke greške o kojima nismo svesni u toku razvijanja softvera.

U okviru ovog potpoglavlja ću opisati kako sam ja istestirao aplikaciju razvijenu za potrebe ovog rada (OnlineEventsOrganizer) te bih se u poglavlju nakon ovog posvetio teoriji i objasnio koji sve vidovi testiranja postoje kod mikroservisne arhitekture.

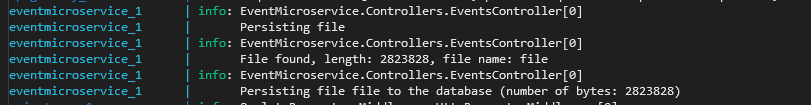
OnlineEventsOrganizer aplikacija je istestirana ručno i sama evaluacija aplikacije je obavljena na osnovu subjektivnog osećaja. Aplikacija je najpre istestirana putem različitih unosa, kako pravilnih tako i nepravilnih da se pre svega istestira korisnički interfejs i različite validacije koje se obavljaju u samim mikroservisima. Nakon toga, kada je donešen zaključak da funkcionalno sve funkcioniše i da su svi funkcionalni zahtevi ispunjeni i funkcionišu u skladu sa zahtevima, na aplikaciju je urađen blagi test opterećenja. Pored testa opterećenja, istestirane su i same notifikacije.   
Testiranje samih notifikacija, odnosno obaveštenja je obavljeno tako što je realizovana prijava više korisnika u aplikaciju putem različitih web pregledača na jednom računaru i obavljane su različite akcije kako bi se simulirale adekvatne situacije za slanjem notifikacija. Pod adevkatnim situacijama pre svega mislim na:

* Jedan prijavljen korisnik koji kreira događaj koji obuhvata kategorije o kojima je zainteresovan drugi prijavljeni korisnik. Ovaj splet okolnosti omogućava testiranje notifikacija koje stižu korisnicima o novokreiranim događajima
* Druga situacija je testiranje notifikacija o početku događaja. Ovaj test je jednostavniji i jedini zahtev je bio postojanje prijavljenog korisnika i pretplaćivanje na događaj te nakon toga čekanje na početak emitovanja samog događaja

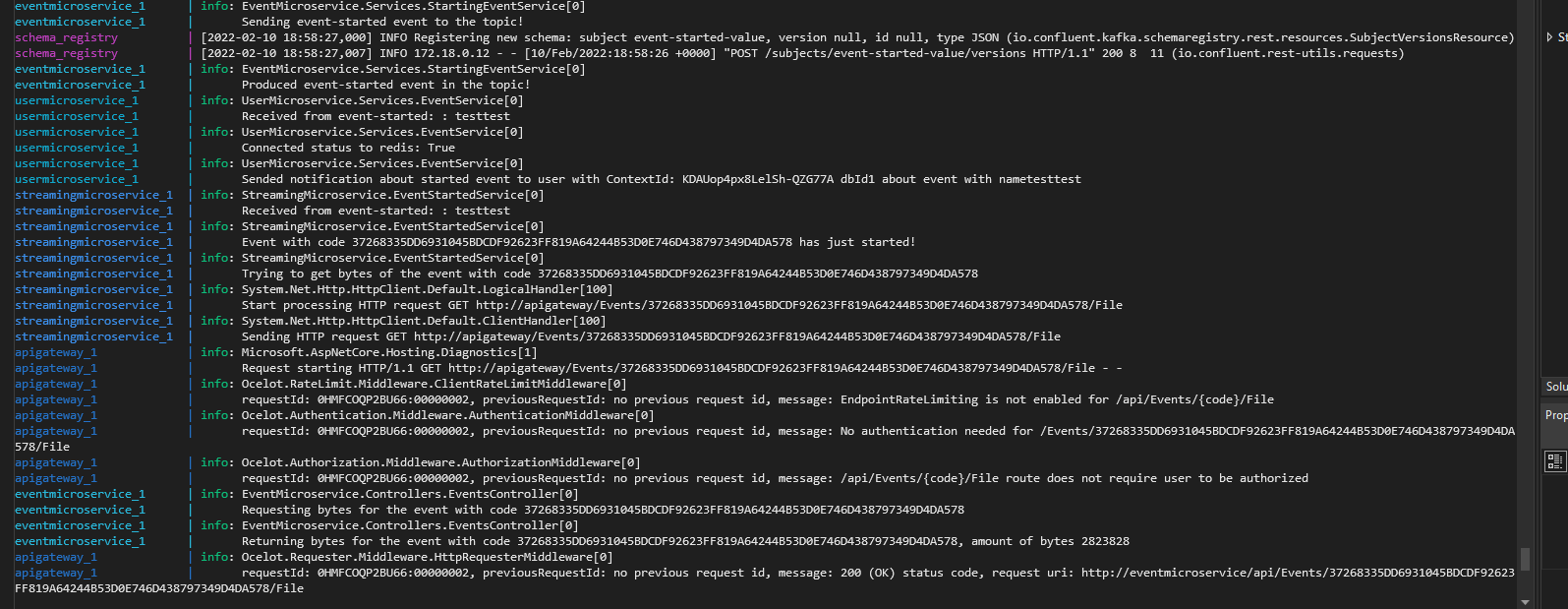
Pored ručnih testiranja, paralelno je rađen monitoring, odnosno nadgledanje centralnog logging sistema u Dockeru kako se slučajno ne bi uhvatila neka nepravilnost ali i proverilo kako funkcioniše sam sistem logovanja. Zakljućak donešen po tom pitanju je da je loging sistem kompletan, odnosno loguju se informacije o svim aktivnostima koje obavljaju mikroservisi te se lako iz samog pregledavanja može zaključiti kako je sve funkcionisalo u tom trenutku. Za *UserMicroservice* mikroservis koji uz pomoć ORM-a (Object relational mapper) Entity Framework Core-a pristupa bazi podataka je takođe podešeno takozvano osetljivo logovanje, odnosno sama tehnologija vrši ispisivanje SQL (Structured query language) upita u centralnom logging sistemu. Na sledećim slikama se vidi rad sistema odnosno pre svega kako izgleda sam logging sistem.



### Slika 5.12 Prijavljivanje korisnika u sistem i beleženje ID-ja njegove SignalR konekcije u odgovarajuće redis liste



### Slika 5.13 Perzistencija događaja



### Slika 5.14 Komunikacija između mikroservisa prilikom otkrivanja početka nekog od događaja

Na prehodnim slikama moglo se jasno videti kako celokupan logging sistem u okviru OnlineEventsOrganizer sistema funkcioniše.

Pored prethodno pomenutih testiranja, obavljeno je i poslednje testiranje, poslednji vid evaluacije. Takozvani test opterećenja, odnosno putem višestrukih konekcija u sistem istovremeno je pokrenuto preuzimanje odnosno pregledanje događaja kako bi se proverilo kako sistem funkcioniše kada veći broj korisnika istovremeno zahteva pregledavanje događaja. Rezultati su bili dobri, naravno to je bio mali broj konekcija u skladu sa mogućnostima. Svakako, *StreamingMicroservice* mikroservis zadužen za strimovanje same video datoteke koja prati događaj, je u potpunosti sklabilan, odnosno, u svakom trenutku je moguće izvršiti replikaciju pomenutog mikroservisa i uz upotrebu raspoređivača težine (Load balancer-a) , čiju funkcionalnost obezbeđuje Ocelot API Gateway, poboljšati performanse.

## Testiranje mikroservisne arhitekture

Pored pomenutih takozvanih jediničnih testova (Unit testova) koje je lako napisati na nivou svakog mikroservisa i koji proveravaju osnovne funkcionanosti jednog mikroservisa odnosno njegove poslovne logike, moguće je iskoristiti i integracione testove koji proveravaju kako je urađena integracija, odnosno saradnja između mikroservisa i da li se ceo proces odvija baš onako kako bi trebalo da se odvija

Preko integracionih testova se upravo proverava da li mikroservisi mogu međusobno komunicirati. Upravo ti testovi bi trebalo pokriti sve puteve između mikroservisa, kako uspešne puteve tako i neuspešne [13].

Još jedna vrsta integracionih testova su takozvani testovi ugovora (contract testing) . Kada god se neka komponenta poveže na interfejs druge komponente da bi iskoristila njeno ponašanje, ugovor je oformljen između tih komponenti. Taj ugovor zavisi od samih ulaza i od očekivanja same potrošačke komponente, odnosno komponente koja koristi funkcionalnosti druge komponente. Ova vrsta integracionog testa upravo verifikuje da li je odredišna komponenta ispunila ugovor i otuda naziv samom testu.

# Zaključak

Kao i većina stvari u softver inženjeringu, svaka odluka zahteva dosta analiza i procena isplativosti pre preuzimanja bilo kakvih koraka u implementaciji. Takva je i situacija sa mikroservisnom arhitekturom. Mikroservisna arhitektura, kao i mikroservisna arhitektura vođena događajima, ima puno koristi i može da ponudi puno dobrih stvari klijentima o kojima oni ne moraju ni biti svesni jer će se sve to odvijati u pozadini. Najbitnija stvar u svemu tome je što oni upravo neće biti svesni jer će sve funkcionisati sa aspekta performansi u najboljem redu te sami korisnici neće osetiti nikakve probleme i zamerke. Naravno, sve to u slučaju da je mikroservisna arhitektura realizovana kako treba i za pravi slučaj upotrebe.

Mikroservisna arhitektura je pre svega arhitektura koja obezbeđuje visoku skalabilnost. Svaka aplikacija koja u osnovi ima mikroservisnu arhitekturu bi trebala da bude visoko skalabilna, odnosno visoko proširiva te na taj način bilo koja veća ili manja varijacija u broju klijenata same aplikacije bi trebala da bude podnošljiva. Pored skalabilnosti, mikroservisnu arhitekturu prate još mnoge druge prednosti, poput same organizacije razvoja kroz timove i druge prateće stvari. Naravno, kao i sve u softver inženjeringu tako i mikroservisna arhitektura ima svoje mane koje bi naravno trebalo uzeti u obzir prilikom razmatranja koja arhitektura najviše godi aplikaciji koja treba da bude razvijena.

*,,Najveće poboljšanje u produktivnoj moći rada, i najveći deo veštine, spretnosti i rasuđivanja sa kojima se on bilo gde usmerava ili primenjuje, čini se da su efekti podele rada’’* – **Adam Smith**

Škotski moralni filizof Adam Smith je izuzetno dobro izneo kalibar mikroservisa kroz izjavu o tome kolika je važnost pametne podele rada i vodeći se tom izjavom u nekim situacijama je zaista dobro, nakon dobre procene, preći sa monolitne arhitekture na mikroservisnu arhitekturu ili u samom startu početi sa razvijanjem mikroservisne arhitekture umesto monolitne arhitekture. Ukoliko vaša aplikacija zahteva česte takozvane ,,releases’’, ukoliko vaša aplikacija zahteva visoku skalabilnost, visoku pouzdanost i fleksibilnost po pitanju upotrebe različitih tehnologija i njihovo povezivanje zajedno, onda je u tom slučaju mikroservisna arhitektura pravi izbor za vas.

**Literatura**

[1] Medium - https://medium.com/swlh/a-design-analysis-of-cloud-based-microservices-architecture-at-netflix-98836b2da45f – 13.02.2022.

[2] Geeks for geeks - https://www.geeksforgeeks.org/the-story-of-netflix-and-microservices/ - 13.02.2022.

[3] Microservices - https://microservices.io/ -13.02.2022.

[4] Capital one - https://www.capitalone.com/tech/software-engineering/how-to-avoid-loose-coupled-microservices/ - 13.02.2022.

[5] Cue logic - https://www.cuelogic.com/blog/microservices-in-practice-from-architecture-to-deployment -13.02.2022.

[6] Microsoft - https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices - 13.02.2022.

[7] Microservices - https://microservices.io/patterns/data/database-per-service.html - 13.02.2022.

[8] IBM - https://www.ibm.com/cloud/learn/message-brokers - 13.02.2022.

[9] Kafka - https://kafka.apache.org/intro - 13.02.2022.

[10] Microservices - https://microservices.io/patterns/reliability/circuit-breaker.html - 13.02.2022.

[11] Github (repozitorijum Ocelot API Gateway-a) - https://github.com/ThreeMammals/Ocelot - 13.02.2022.

[12] Netflix tech blog - https://netflixtechblog.com/open-sourcing-zuul-2-82ea476cb2b3 - 13.02.2022.

[13] Martin Fowler - https://martinfowler.com/articles/microservice-testing/#testing-integration-diagram - 13.02.2022.

[14] Nordic APIS - <https://nordicapis.com/whats-the-difference-between-event-brokers-and-message-queues/> - 13.02.2022.

[15] Oracle - https://docs.oracle.com/cd/E19879-01/821-0028/aerbk/index.html - 13.02.2022.