1. SLAJD

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.

Obezbediti korisnicima mogućnost da pregledavaju različite video snimke bez ikakvog kašnjenja i čekanja da se video snimak učita. Ukoliko sama arhitektura web aplikacije nije projektovana da podnese veliki broj korisnika u realnom vremenu, koji preuzimaju video snimke, odmah se dolazi do 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. 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.

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).

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. SLAJD

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.

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. Ispunajvanje tog zahteva automatski obezbeđuje nezavisan deployment i razvoj.

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.

1. SLAJD

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

6. SLAJD

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.

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. Kako celokupha arhitektura može biti sačinjena od velikog broja mikroservisa, sa povećanjem upravo broja mikroservisa tako ova komponenta dobija sve više na značaju.

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.

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.

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).

Napomenuo bih prilikom prezentovanja ove teoretske osnove o mikroservisnoj arhitekturi i mikroservisima da su sve pomenute komponente i pristupi upravo realizovani u okviru aplikacije koja prati ovaj rad. Dakle, aplikacija implementirana za potrebe ovog rada obezbeđuje kako sinhronu tako i asinhronu komunikaciju između mikroservisa, API Gateway komponentu, broker poruka i druge prateće stvari.

7. SLAJD

* **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.

U okviru samog rada može se pronaći i sekcija u okviru koje su opisani pristupi, odnosno šabloni dizajniranja kod mikroservisne arhitekture koji su se pokazali adekvatnim, odnosno koji su se pokazali kao dobra rešenja kroz praksu softver inženjera. Na primer jedan od njih je takozvani šablon zvani osigurač (circuit breaker) koji obezbeđuje izolaciju grešaka, odnosno obezbeđuje komunikaciju između mikroervisa u slučaju ako dođe do otkaza nekog od mikroservisa. 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.

8. SLAJD

**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. 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.

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**

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.

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

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.

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) .

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.

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.

Pored podrške za rad sa različitim bazama podataka,za ASP .NET Core 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).

9. SLAJD

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.

10. SLAJD

**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

11. SLAJD

* 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

12. SLAJD

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.

15. SLAJD

*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.

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.

*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.

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.

17. SLAJD

*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 u sistemu 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. 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, kao što je prikazano na prethodnom sekvencijalnim dijagramima.

18. SLAJD

*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

19. SLAJD

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.

S obzirom da je prilikom pokretanja sistema potrebno obezbediti postojanje 2 topic-a, realizovan je mali kontejner koji se pokreće uporedo sa celokupnim sistemom i vrši upravo kreiranje potrebnih topica ukoliko oni već ne postoje pre nego što celokupan sistem bude spreman za korišćenje.

20. SLAJD

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.

21. SLAJD

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:

ZAKLJUČAK:

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.