Predlog skalabilne arhitekture aplikacije

Uvod

U današnje vreme, kao jedno od najvećih pitanja pri razvoju internet aplikacija jeste kako postaviti arhitekturu aplikacije tako da ona može opsluživati veliki broj korisnika istovremeno. Može se reći da je ova tema relativno nova I da još uvek ne postoje sveobuhvatna I univerzalna rešenja za sve probleme koji se nameću kada govorimo o ovoj temi. Jedno od rešenja čija je upotreba značajno porasla u proteklih nekoliko godina je mikroservisna arhitektura.

Uvođenje mikroservisne arhitekture uvodi veliku kompleksnost u celokupan sistem i zahteva visok stepen sinhronizacije, kako sa tehničkog tako i sa ljudskog aspekta. Bitno je napomenuti da statistički gledano, veliki broj projekata propada upravo zbog izbora mikroservisne arhitekture, gde se kao najveći uzrok toga izdvaja činjenica da razvojni timovi preusmeravaju razvoj svojih aplikacija na mikroservisnu arhitekturu bez prethodnog iskustva u ovoj oblasti ili realne potrebe za takvim potezom. Takođe, sa ljudskog aspekta, agilnost timova koji će razvijati mikroservise je jedan od kritičnih faktora koji utiču na krajnji uspeh.

Velike internet aplikacije u današnje vreme moraju da zadovolje sledeća 3 kriterijuma:

- Availability
- Reliability
- Scalability

U nastavku ćemo prezentovati neke od načina kojima bi naša aplikacija zadovoljila gorepomenute zahteve.

Predlog

Naš predlog za postizanje visokog nivoa skalabilnosti jeste uvođenje mikroservisne arhitekture, gde bi se za svaku od 3 namene naše aplikacije (*rezervacija letova, hotelskog smeštaja I iznajmljivanje vozila*) kreirali posebni mikroservisi za prikazivanje I pretragu podataka, rezervisanje, plaćanje, kao I generalni servisi za slanje notifikacija, prijateljstva I pozivnica.

Sam početak pri dizajnu softvera u mikroservisnoj arhitekturi je poprilično težak iz prostog razloga što nije moguće unapred proceniti tok razvoja celokupnog sistema, ne poseduje se dovoljno znanja iz domena problema i nepoznato je kako će se zahtevi menjati kroz vreme. Iz tog razloga se savetuje da se aplikacija razvija po monolitnoj arhitekturi iz koje će se kasnije po potrebi izdvajati mikroservisi u zavisnosti od toga koji delovi sistema prvi postanu preopterećeni I ne uspevaju da obrade sve zahteve koji dospevaju do samog sistema.

U trenutku kada naš monolitni sistem ne bi bio sposoban da uspešno opsluži ogroman broj zahteva, preduzeli bismo sledeće korake:

1. Izdvajanje dela sistema koji je najopterećeniji i usporava rad celokupnog sistema
2. Detaljno proučavanje poslovnog domena I granica konteksta (*bounded context*), a zatim donošenje odluke gde će se povući granica u poslovnom domenu oko koje ćemo “odseći” servis.
3. Kreiranja posebne baze podataka za mikroservis.

Mikroservisna arhitektura nam garantuje da će naša aplikacija moći da podrži veliki broj istovremenih korisnika iz razloga koji ćemo navesti u daljem tekstu.

Skalabilnost

S obzirom na to da se većina današnjih rešenja deploy-uje na cloud servise poput AWS, Azure, GCP itd. upravo to nam pruža mogućnost da automatizujemo skalabilnost našeg sistema.

Jednostavno rešenje koje bi primenili prilikom deploy-a je smeštanje mikroservisa u automatski skalabilne grupe ispred kojih bi stojao load balancer. Na taj način bi load balancer donosio odluke na koju instancu servisa će proslediti zahtev, a auto scaling grupe bi nam omogućile da se na osnovu određenih kriterijuma *(npr. CPU usage, RAM usage itd.)* grupa servisa skalira na gore (*poveća broj aktivnih instanci*) ili na dole (*smanji broj aktivnih instanci*). Na ovaj način, naša aplikacija bi bila skalabilna po X-osi gde se skalira broj trenutno pokrenutih instanci. Skaliranje po Y-osi bi obuhvatalo skaliranje svakog servera sa dodavanjem dodatne procesorske moći I količine RAM memorije koju taj server poseduje.

Baza podataka

Ukoliko bi naš sistem dostigao nivo gde baza podataka predstavlja usko grlo sistema, onda bi bilo potrebno primeniti I skaliranje po Z-osi odnosno skaliranje baze podataka. Prvi korak ka skaliranju baze podataka bilo bi particionisanje tabela (*table partitioning*). Kada zahtevi prerastu I ovo skaliranje, dodatno bi bilo potrebno kreirati *database shard* koji bi bio distribuiran na više servera. Ovo bi uvelo dodatnu kompleksnost u celokupan sistem, iz prostog razloga što je pri svakoj izmeni podataka, podatke potrebno replicirati na više servera uz očuvanje konzistentnosti podataka. U današnje vreme je sve više popularna priča o tzv. *Eventual consistency* koja se javlja upravo zbog potrebe da podatke držimo konzistentim.

Distribuirane transakcije

Još jedan od ključnih problema u mikroservisnoj arhitekturi jesu distribuirane transakcije. Ukoliko imamo više mikroservisa, gde jedan mikroservis zavisi od niza događaja koji treba da se izvrše na nekom drugom (*ili više drugih*) mikroservisa, postavlja se pitanje kako osigurati transakcije. Osiguravanje transakcija na mikroservisima je veoma skupa operacija I implementira se jedino kada je zaista neophodno – npr. kod bankarskih sistema gde se moraju zadovoljiti protokoli plaćanja I obezbediti visoka sigurnost podataka. Jedno od rešenja koje bi otklonijlo ovaj problem je *two-phase commit* protokol (*2PC*).

Otkazivanje servisa

Naredni problem o kome treba voditi računa je – kako reagovati na otkazivanje jednog od servisa? Za rešenje ovog problema upotrebili bi *circuit breaker pattern* koji za svaki servis definiše šta će biti fallback u slučaju da on otkaže. Kako bi zadovoljini dostupnost, bitno nam je da korisniku prikažemo *neke* podatke umesto poruke o grešci. Poruku o grešci bi prikazivali jedino ukoliko nemamo ni jedan aktivan fallback na koji možemo da se oslonimo u datom trenutku.

Kao primer na našoj aplikaciji, recimo da karta za let A ima redovnu cenu i cenu sa popustom baziranu na individualnom korisniku koja se kreira na osnovu mnogobrojnih parametara poput njegove aktivnosti na aplikaciji, prethodnih rezervacija, broja prijatelja, bonus poena itd. U slučaju da servis koji određuje cenu karte za svakog individualnog korisnika otkaže, naš fallback bi bio da korisniku prikažemo pravu cenu leta bez ikakvih pogodnosti. Na ovaj način korisniku dajemo utisak da naša aplikacija radi, i ne samo to, već mu pružamo mogućnost rezervacije karte po ceni bez popusta.

Kada se govori o fallback-ovima, često se spominje i redundantnost podataka. Kako bismo mogli da omogućimo da imamo fallback za neki servis, moramo da pronađemo servis koji ima najsličnije podatke kako bismo mogli da ih pružimo korisniku. Iz ovog razloga, mikroservisnoj arhitekturi redundantnost podataka ne predstavlja preveliki problem.

Dodatne prednosti

Pored svih navedenih problema i predloga rešenja, bitno je napomenuti da mikroservisi pružaju veliki broj prednosti koje dolaze same po sebi bez dodatnog ulaganja napora. Kao neke od njih možemo navesti:

1. Mogućnost upotrebe različitih programskih jezika za različite delove sistema
2. Odvojene odgovornosti po servisima (*single responsibility*)
3. Agilan razvoj u više različitih timova
4. Nezavisan deployment servisa

CDN

Dodatno bi korišćenjem CDN (Content Delivery Network) za smeštanje statičkog sadržaja, skratili vreme pristupa našoj aplikaciji, I dodatno smanjili broj zahteva koji bi naša aplikacija morala da opsluži.

Continuous Delivery

Na samom kraju, pri deploymentu bi koristili tzv. Blue/Green deployment kako bi smanjili nedostupnost I rizik od otkazivanja, gde bi na novu verziju servisa na određeno vreme preusmeravali mali deo saobraćaja. Na taj način, krajnji korisnici bi neko vreme testirali novu verziju, a zatim bi se vratila stara verzija. Za to vreme, naš tim bi mogao da analizira log-ove i na osnovu njih zaključi da li je deploy bio uspešan.