Servisno orijentisane arhitekture

Predavanje 8:Mikroservisi i paterni, Obrasci za upravljanje podacima, SAGA, Kompozicija API-ja

Univerzitet u Novom Sadu
Fakultet Tehničkih Nauka

Obrasci za upravljanje podacima

- Jedna baza po servisu
- Deljene baze
- Saga
- Kompozicija API-ja
- CQRS (Command Query Responsibility Segregation)
- Domain Event (domenski dogadjaji)
- Event Sourcing (izvori dogadjaja)

Uvod

- Većina servisa ima potrebu da trajno sačuva odredjene podatke.
- Koja arhitektura baze podataka je pogodna za mirkoservisnu arhitekturu?

Deljena baza

- Servisi treba da su medjusobno slabo zavisni kako bi se mogli razvijati, puštati u rad i skalirati nezavisno jedan od drugog.
- Odredjene poslovne transakcije moraju sačuvati odredjene nepromenljive podatke koje koristi više servisa.
- Poslovne transakcije treba da pretražuju podatke koji pripadaju nekim drugim servisima.
- Pretraživanja podataka treba da spoje podatke koji postoje u bazama koje pripadaju drugim servisima.

- Baze podataka se ponekad moraju replicirati ili izdeliti ("sharding") kako bi se skalirale.
- Različiti servisi mogu imati različite zahteve u pogledu čuvanja podataka, za neke su relacione baze dobro rešenje, za neke to nisu, za neke su najbolje rešenje graf-orijentisane baze podataka ili repozitorijumi nestrukturiranih podataka.

Moguće rešenje

- Koristiti jednu bazu podataka, koju dele svi mikroservisni moduli.
- Svaki servis može slobodno da pristupa i podacima koji su u vlasništvu drugih mikroservisa, koristeći lokalne transakcije (ACID - Atomicity, Consistency, Isolation, Durability compliant).

Rezultat primene

- Dobre osobine
  - Programer se koristi već dobro poznatih tehnikama za pristup podacima (transakcije) kako bi garantovao konzistenciju podataka.
  - Jednu zajedničku bazu je lakše održavati.
- Medjuzavisnost mikroservisnih modula tokom razvoja – programer koji radi na razvoju jednog servisa mora da se koordiniše pri promenama ili čeka rezultat razvoja dela baze podataka koji primarno koristi neki drugi servis. Ovo usporava razvoj.
- Medjuzavisnost servisa tokom izvršavanja programa – pošto svi servisi pristupaju istoj bazi i potencijalno istim tabelama može doći do preklapanja u pristupu i do eventualnog blokiranja jednog servisa od strane drugog koji drži zaključanim deo baze.
- Jedna jedinstvena baza se može pokazati kao loše rešenje sa stanovišta raspoloživog prostora za čuvanje podataka.

SAGA obrazac

- Implementiran je obrazac jedne baze po servisu.
- Odredjene poslovne transakcije ipak zahtevaju da više servisa odradi odredjeni deo posla, pa je neophodno obezbediti mehanizam kojim se postiže konzistencija podataka izmedju servisa.
- Pri tome se ne može raditi lokalnom ACID transakcijom, jer se radi o potpuno odvojenim bazama.

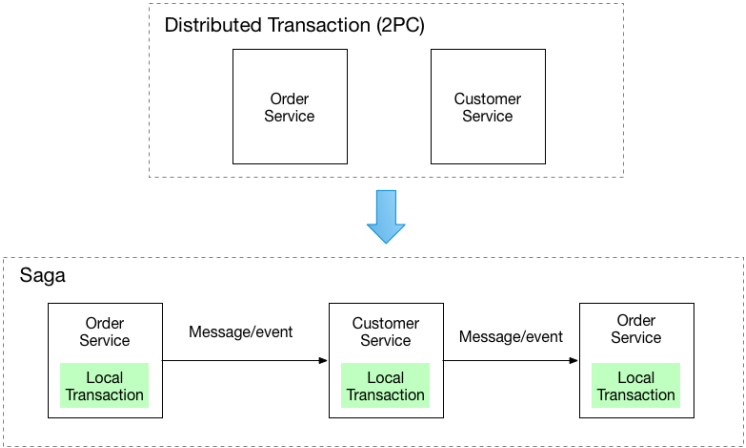
Problem

- Kako održati konzistenciju podataka izmedju različitih servisa?
- Primena Two phase commit (2PC) protokola nije odgovarajuća opcija.
  - 2PC je protokol za distribuirane transakcije koji je transparentan ka krajnjem korisniku, podrška za upravljanje transakcijama (koordinator) često se ugradjuje u samu platformu distribuiranog sistema.
  - Transakcije se obavljaju u dva koraka
    1. commit-request/voting korak u kome koordinator priprema sve procese koji učestvuju u transakciji i da glasaju Da/Ne (izvršiti komit?)
    2. commit korak - koordinator na osnovu glasanja svih učestuvjućih procesa donosi odluku da li da izvrši komitovanje (samo ako su svi glasali Da).

Rešenje

- Implementirati poslovnu transakciju koja zahteva učešće više servisa kao **sagu**.
- Saga predstavlja sekvencu lokalnih transakcija.
- Svaka lokalna transakcija (unutar servisa) ažurira stanje svoje baze i generiše dogadjaj koji se koristi kao okidač za sledeću transakciju u sekvenci.
- Ukoliko lokalna transakcija ne uspe, jer narušava neko od postavljenih pravila, u sagi se izvršava niz kompenzacionih transakcija kojima se poništavaju prethodne promene izazvane lokalnim transakcijama povezanim u sagu.

SAGA vs 2PC



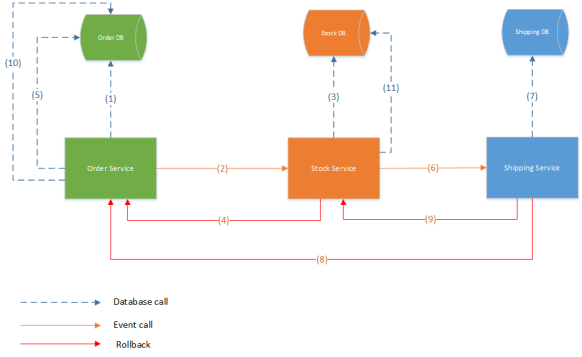
(https://microservices.io/patterns/data/saga.html)

Vrste

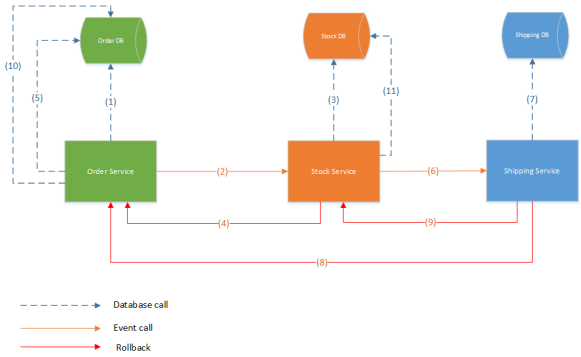
- Postoje dva načina koordinacije transakcija u sagi:
  1. Koreografija - svaka lokalna transakcija generiše dogadjaj i tako trigeruje lokalne transakcije u drugim servisima
  2. Orkestracija - poseban objekat (orkestrator) upravlja učesnicima tako što utvrdjuje koju lokalnu transakciju treba da izvrše

Koreografija

- Scenario 1 – greška sa Stock servisom
  1. Order servis pravi zapis u Order DB sa statusom *Verifying*
  2. Order servis kreira event i Stock servis osluškuje na te event-ove
  3. Stock sevis pokušava da ažurira stanje i dolazi do kreške
  4. Stock servis kreira *rollback* event
  5. Order servis osluškuje na ove event-ove i ažurira stanje u *Failed*

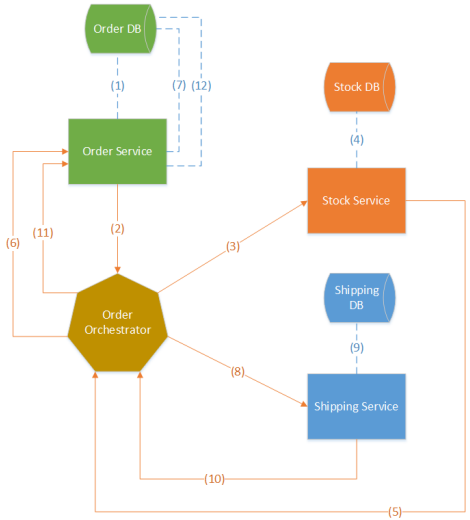


- Scenario 2 – greška sa Shipping servisom
  6. Stock servis generiše uspešan dogadjaj, na koji Shipping osluškuje
  7. Shipping servis kopuša da ažurira stanje, ali dolazi do greške
  8. Shipping servis generiše *rollback* event na koji osluškuje Stok servis
  9. Shipping servis generiše *rollback* event na koji osluškuje Order servis
  10. Order servis osluškuje na ove event-ove i ažurira stanje u *Failed*
  11. Stok servis vraća stanje na ono pre koraka (3)



Orkestracija

- Situacija vrlo slična kao u prethodnom slučaju
- Razlika je sada to što nije svaki servis za dužen za propagaciju poruka, uspešnih ili *rollbacck*
- Za to postoji posebna komponenta čija je to namena
- Servisi direktno komuniciraju samo sa svojom bazom



Osobine

- Dobre osogbine:
  - Omogućava aplikaciji da obezbedi konzistentnost podataka medju servisima, a da pri tome ne koristi distribuirane transakcije.
- Loše osobine:
  - Komplikuje se programsko rešenje.
  - Moraju se dobro osmisliti kompenzacione transakcije koje poništavaju promene postignute u prethodnim koracima sage.

Dodatan problem

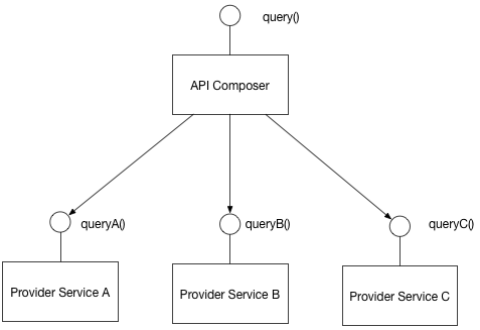
- Kako bi se postigla pouzdanost, servis mora atomično ažurirati svoju bazu, a potom publikovati poruku/dogadjaj.
- Ne može se osloniti na tradicionalne mehanizme distribuiranih transakcija i message brokere.
- Kako bi se to postoglo mora se osloniti na dodatne šablone (Event Sourcing, Transactional Outbox, Aggreagates, Domain Events).

Uvod

- Implementirana je mirkoservisna arhitektura, a za čuvanje podataka obrazac jedne baze po servisu.
- Kao rezultat tog pristupa nije više jednostavno prikupiti podatke koji pripadaju različitim servisima.
- Problem – Kako implementirati upite koji povlače podatke sa različitih mikroservisa?

Moguće rešenje

- Upiti se realizuju tako što se definiše komponenta koja radi kompoziciju API-ja (API Composer).
- Ova komponenta poziva svaki mikroservis ponaosob, a zatim obavlja spajanje pojedinačnih rezultata u memorijskim strukturama (in-memory) pre vraćanja rezultata krajnjem korisniku.
- U praksi često komponenta koja je API Gateway obavlja i ovu ulogu.



(https://microservices.io/patterns/data/api-composition.html)

Osobine

- Dobre osobine:
  - Obezbedjuje jednostavan način da se u mikroservisnoj arhitekturi obave složeniji upiti.
- Loše osobine:
  - Neki upiti mogu rezultovati u velikim skupovima medjurezultata i neefikasnom obradom.

Dodatni materijali

- Building Microservices, Sam Newman
- Microservices • Martin Fowler • GOTO 2014
- What are microservices?
- Microservices patterns
- SAGAS originalan rad

Kraj predavanja

Pitanja? :)