# Saga

## Servisno orijentisane arhitekture

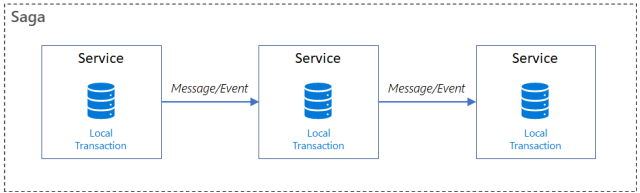
Univerzitet u Novom Sadu
Fakultet tehničkih nauka

# Problem

- Transakcije koje se izvršavaju u okviru jednog mikroservisa i dalje jednostavno mogu zadovoljiti ACID (atomicity, consistency, isolation, durability) svojstva
- Problem se javlja kada je potrebno izmeniti podatke u više od jednog mikroservisa, gde svaki od njih poseduje svoju bazu podataka
- Protokoli za upravljanje distribuiranim transakcijama (2PC, 3PC) obezbeđuju da se izmene izvrše ili kod svih učesnika transakcije ili ni kod jednog, ostavljajući sistem u konzistentnom stanju
- 2PC poseduje SIngle Point of Failure (otkazom koordinatora sistem ostaje u nekonzistentnom stanju)
- 2PC je blokirajući protokol, što znači da će sistem biti spor koliko i najsporija transakcija u čitavom procesu

# Saga

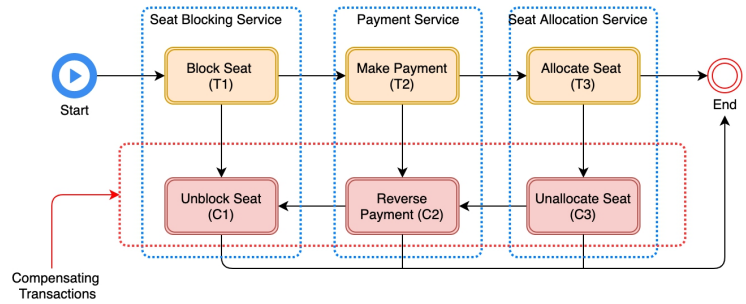
- Saga predstavlja sekvencu lokalnih transakcija, takvih da svaka od njih vrši atomičnu izmenu podataka u okviru jednog servisa
- Kada neki zahtev inicira sagu, počinje da se izvršava prva lokalna transakcija i po njenom završetku započinje naredna lokalna transakcija
- Koristimo publish/request - async response stil komunikacije među učesnicima, tako da će svaki korak biti izvršen čak i kada je neki od učesnika privremeno nedostupan
- Moramo obezbediti at least once garanciju isporuke



# Compensating transakcije

- Šta uraditi u situacijama kada neka lokalna transakcija ne može da se izvrši?
- Moramo obezbediti rollback mehanizam za prethodne uspešno izvršene lokalne transakcije
- Compensating transakcija neke transakcije vrši njen undo, obezbeđuje povratak u prvobitno stanje (kao da se transakcija nije ni desila)
- Ako je prvih n transakcija bilo uspešno, a n+1-a je neuspešna, treba da izvršimo n compensating transakcija, u obrnutom redosledu u odnosu na originalne transakcije
- Read-only koraci sage nemaju compensating transakciju

# Compensating transakcije



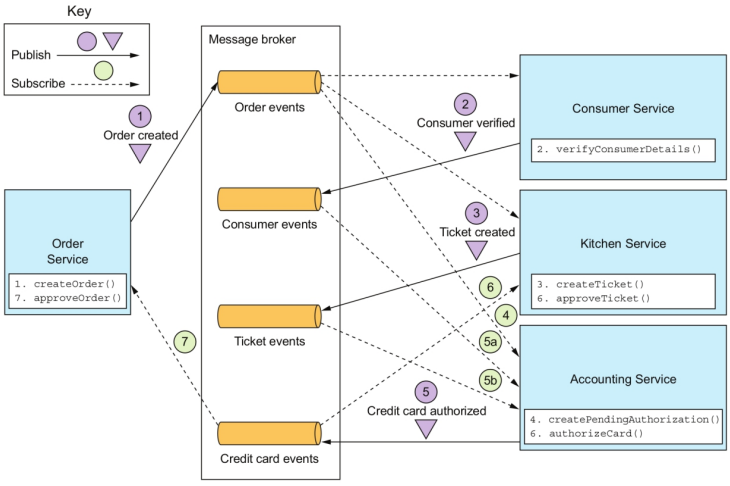
# Napomene

- Saga ne pruža svojstvo izolovanosti, ono se mora obezbediti na aplikativnom nivou (semantic lock)
- Moramo da odlučimo da li ćemo klijentu vratiti odgovor odmah nakon iniciranja sage, pa će on kasnije proveriti ishod, ili ćemo čekati dok se saga ne završi u potpunosti i tek onda vratiti odgovor
- Događaji koje učesnici sage objavljuju moraju se izvršiti atomično sa izmenom u bazi podataka (moraju se izvršiti ili obe akcija ili nijedna)
- Svaki događaj mora u sebi sadržati informacije o tome na koje podatke u servisima se odnosi, odnosno za koju sagu je vezan

# Koordinacija sage

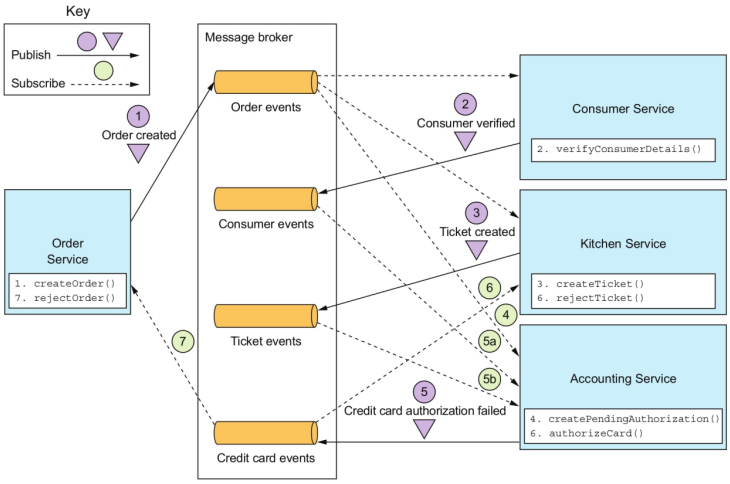
- Treba odlučiti ko je zadužen da signalizira učesnicima sage da je red na njih da izvrše svoju lokalnu transakciju
- Razlikujemo dva pristupa:
  - **Koreografija** - Nemamo komponentu koja igra ulogu koordinatora, već servisi osluškuju događaje koje je poslao neki drugi učesnik, a koji predstavljaju signal za započinjanje njihove transakcije (pogodno za jednostavnije sage)
  - **Orkestracija** - Orkestrator je komponenta koja šalje komandu servisu (učesniku sage) i na osnovu njegovog odgovora izdaje narednu komandu (pogodno za kompleksnije sage jer centralizujemo upravljačku logiku)

# Koreografija - Uspešan scenario



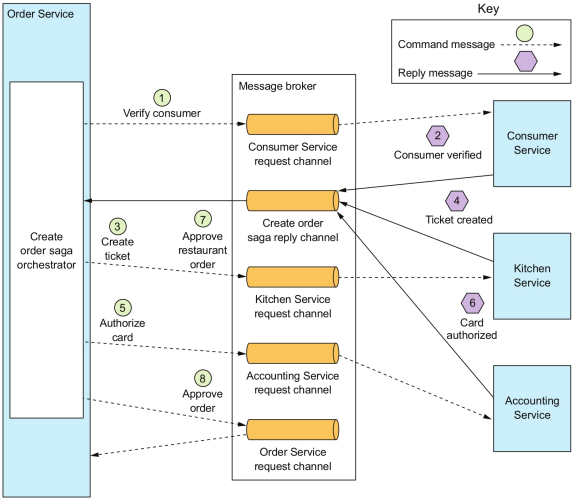
1. Order Service kreira Order koji je u *ApprovalPending* stanju i objavljuje *OrderCreated* događaj
2. Consumer Service registruje *OrderCreated* događaj, verifikuje da li ta mušterija može da izvrši narudžbinu i objavljuje *ConsumerVerified* događaj
3. Kitchen Service registruje *OrderCreated* događaj, validira Order i kreira Ticket u *CreatePending* stanju, a zatim šalje *TicketCreated* događaj
4. Accounting Service reaguje na *OrderCreated* događaj i kreira CreditCardAuthorization u *Pending* stanju
5. Accounting Service registruje *TicketCreated* i *ConsumerVerified* događaje, menja stanje na korisnikovoj kreditnoj kartici i objavljuje *CreditCardAuthorized* događaj
6. Kitchen Service registruje *CreditCardAuthorized* događaj i menja stanje Ticket-a u *AwaitingAcceptance*
7. Order Service prima *CreditCardAuthorized* poruku i menja stanje Order-a u *Approved*

# Koreografija - Neuspešan scenario



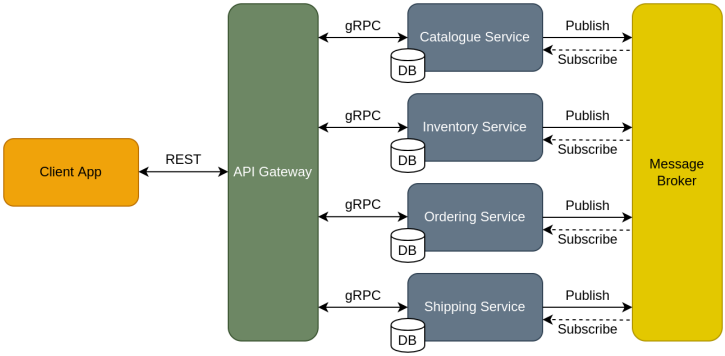
1. Order Service kreira Order koji je u *ApprovalPending* stanju i objavljuje *OrderCreated* događaj
2. Consumer Service registruje *OrderCreated* događaj, verifikuje da li ta mušterija može da izvrši narudžbinu i objavljuje *ConsumerVerified* događaj
3. Kitchen Service registruje *OrderCreated* događaj, validira Order i kreira Ticket u *CreatePending* stanju, a zatim šalje *TicketCreated* događaj
4. Accounting Service reaguje na *OrderCreated* događaj i kreira CreditCardAuthorization u *Pending* stanju
5. Accounting Service registruje *TicketCreated* i *ConsumerVerified* događaje, okušava da izmeni stanje na korisnikovoj kartici i objavljuje *CreditCardAuthorizationFailed* događaj
6. Kitchen Service registruje *CreditCardAuthorizationFailed* događaj i menja stanje Ticket-a u *Rejected*
7. Order Service prima *CreditCardAuthorizationFailed* poruku i menja stanje Order-a u *Rejected*

# Orkestracija - Uspešan scenario



1. Orkestrator šalje *VerifyConsumer* komandu Consumer Service-u
2. Consumer Service odgovara *ConsumerVerified* porukom
3. Orkestrator šalje *CreateTicket* komandu Kitchen Service-u
4. Kitchen Service odgovara *TicketCreated* porukom
5. Orkestrator šalje *AuthorizeCard* komandu Accounting Service-u
6. Accounting Service odgovara *CardAuthorized* porukom
7. Orkestrator šalje *ApproveTicket* komandu Kitchen Service-u
8. Orkestrator šalje *ApproveOrder* komandu Order Service-u

# Arhitektura aplikacije



# Implementacije sage

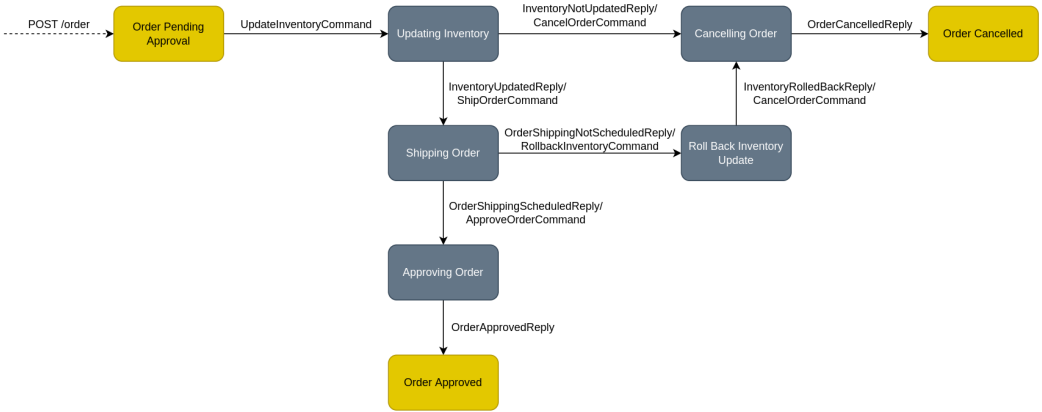
- Operacija kreiranja porudžbine zahteva izmenu podataka u Ordering, Inventory i Shipping servisu
- Koristićemo koordinaciju orkestracijom, a orkestrator će biti deo Ordering servisa
- Odgovor ćemo odmah vratiti klijentu, a krajnji ishod sage moći će da proveri dobavljanjem porudžbine po identifikatoru
- Komande koje orkestrator izdaje slaće se na NATS subject *order.create.command*, a učesnici će odgovor slati na subject *order.create.reply*
- Primer ne pokriva atomično objavljivanje događaja i at-least-once isporuku

# Poruke

```
type CreateOrderCommandType int8

const (
   UpdateInventory CreateOrderCommandType = iota
   RollbackInventory
   ApproveOrder
   CancelOrder
   ShipOrder
   UnknownCommand
)

type CreateOrderCommand struct {
   Order OrderDetails
   Type  CreateOrderCommandType
}
```

```
type CreateOrderReplyType int8

const (
   InventoryUpdated CreateOrderReplyType = iota
   InventoryNotUpdated
   InventoryRolledBack
   OrderShippingScheduled
   OrderShippingNotScheduled
   OrderApproved
   OrderCancelled
   UnknownReply
)

type CreateOrderReply struct {
   Order OrderDetails
   Type  CreateOrderReplyType
}
```

# Tok izvršavanja sage



# Orkestrator

- Kreiranje porudžbine započinje komandom *UpdateInventory* koju treba da izvrši Inventory servis

```
func (o *CreateOrderOrchestrator) Start(order *domain.Order, address string) error {
        event := &events.CreateOrderCommand{
                Type: events.UpdateInventory,
                Order: events.OrderDetails{
                        Id:      order.Id.Hex(),
                        Items:   make([]events.OrderItem, 0),
                        Address: address,
                },
        }
        // convert domain order to saga order model
        return o.commandPublisher.Publish(event)
}
```

- Kada kreiramo orkestrarora, on odmah počne da sluša na reply subject-u i *handle* funkcijom obrađuje pristigle odgovore

```
func NewCreateOrderOrchestrator(publisher saga.Publisher, subscriber saga.Subscriber)
↪ (*CreateOrderOrchestrator, error) {
        o := &CreateOrderOrchestrator{
                commandPublisher: publisher,
                replySubscriber:  subscriber,
        }
        err := o.replySubscriber.Subscribe(o.handle)
        ...
        return o, nil
}

func (o *CreateOrderOrchestrator) handle(reply *events.CreateOrderReply) {
        command := events.CreateOrderCommand{Order: reply.Order}
        command.Type = o.nextCommandType(reply.Type)
        if command.Type != events.UnknownCommand {
                _ = o.commandPublisher.Publish(command)
        }
}
```

- Na osnovu tipa trenutno pristiglog odgovora odlučujemo koju sledeću komandu treba da izdamo

```
    func (o *CreateOrderOrchestrator) nextCommandType(reply events.CreateOrderReplyType)
↪ events.CreateOrderCommandType {
            switch reply {
            case events.InventoryUpdated:
                    return events.ShipOrder
            case events.InventoryNotUpdated:
                    return events.CancelOrder
            case events.InventoryRolledBack:
                    return events.CancelOrder
            case events.OrderShippingScheduled:
                    return events.ApproveOrder
            case events.OrderShippingNotScheduled:
                    return events.RollbackInventory
            default:
                    return events.UnknownCommand
            }
    }
```

# Učesnici sage

- Učesnici sage treba da slušaju poruke na command subject-u i obrade ih ukoliko je poruka namenjena njima (primer iz Inventory servisa)

```
func (handler *CreateOrderCommandHandler) handle(command *events.CreateOrderCommand) {
        reply := events.CreateOrderReply{Order: command.Order}

        switch command.Type {
        case events.UpdateInventory:
                products := mapUpdateProducts(command)
                err := handler.productService.UpdateQuantityForAll(products)
                if err != nil {
                        reply.Type = events.InventoryNotUpdated
                        break
                }
                reply.Type = events.InventoryUpdated
        case events.RollbackInventory:
                products := mapRollbackProducts(command)
                err := handler.productService.UpdateQuantityForAll(products)
                if err != nil {
                        return
                }
                reply.Type = events.InventoryRolledBack
        default:
                reply.Type = events.UnknownReply
        }
        if reply.Type != events.UnknownReply {
                _ = handler.replyPublisher.Publish(reply)
        }
}
```

# Zadaci

- Na osnovu specifikacije projekta identifikovati operacije koje je pogodno implementirati upotrebom sage
- Za svaku sagu odrediti učesnike u komunikaciji i odabrati način koordinacije
- Definišite model poruka koje će učesnici razmenjivati i tok saga