Система координації транзакцій між мікросервісами на основі патерну Saga

Студент: Шандрик А.В. ІПЗ–4
Науковий керівник: Бабич Т. А. ст. викл.

Актуальність

Кілька сервісів, кожен зі своєю БД, мають виконати пов'язані операції. У моноліті це одна транзакція. У мікросервісах — ні.

Класичні рішення (2PC, XA, DTC) погано поєднуються з мікросервісами: блокують ресурси, знижують доступність, вимагають єдиного координатора.

Подвійний запис

Зміна стану + публікація події мають бути атомарними

Ідемпотентність

Брокери дають at-least-once: повторна доставка неминуча

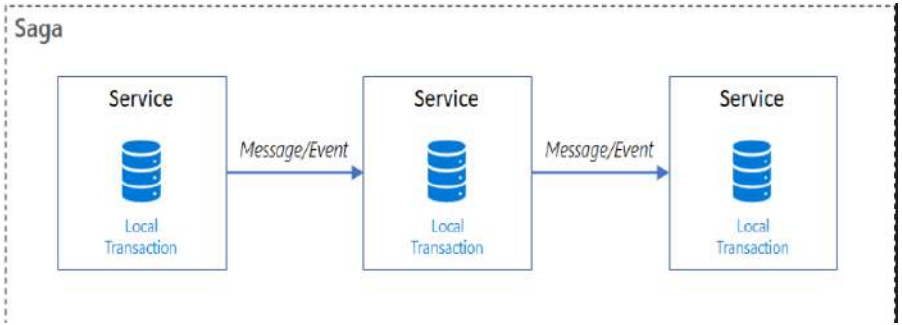
Компенсація

Відкат — лише через зворотні бізнес-дії, не SQL ROLLBACK

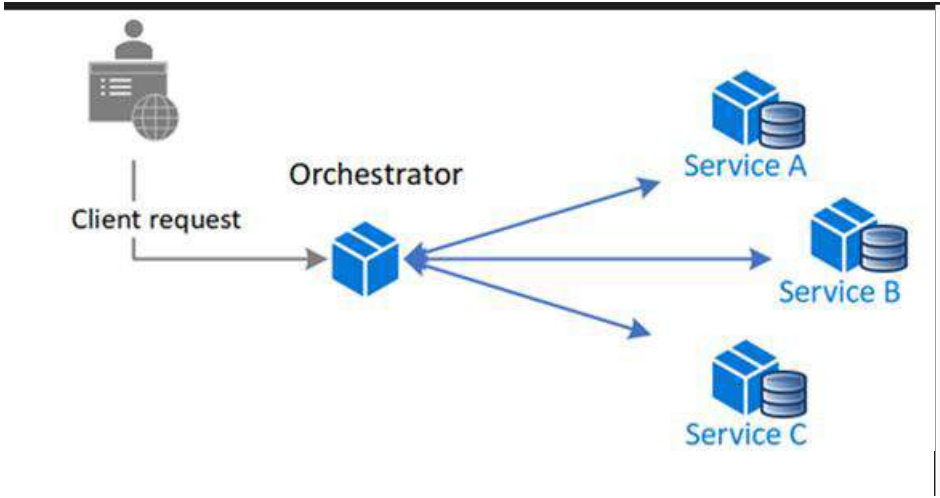
Мета

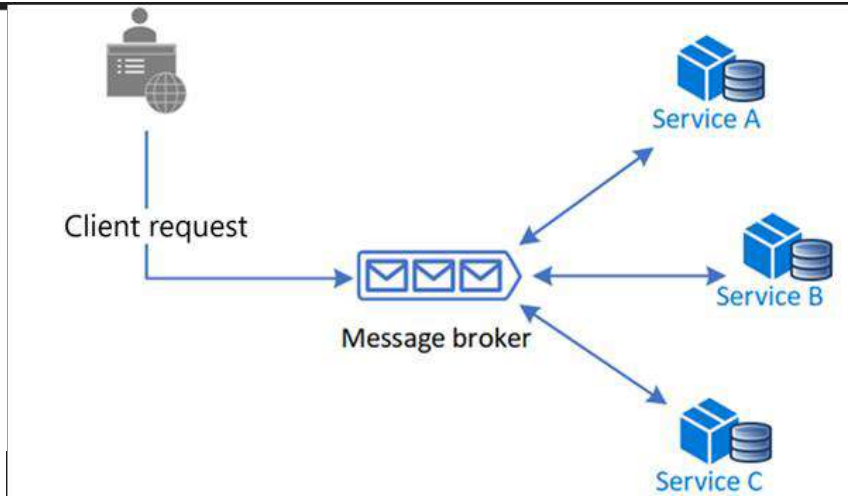
Спроєктувати та реалізувати систему координації розподілених транзакцій між мікросервісами на основі оркестрації Saga

із підтримкою компенсувальних дій, ідемпотентності та керованих політик повторів



Оркестрація vs хореографія





SagaFlow: порти й адаптери

Ядро не знає про конкретні брокери чи СКБД. Зміна транспорту — без перекомпіляції бізнес-коду.

5 пакетів NuGet · новий брокер чи СКБД — це новий адаптер, а не зміна ядра

РЕАЛІЗАЦІЯ

01 Декларативний DSL

02 Transactional Outbox

03 Дворівнева ідемпотентність

04 Конкурентний доступ

```
Initially(
    When<PlaceOrderCommand>()
        .Then(ctx => ApplyInitiator(ctx.Instance, ctx.MessageContext.Message))
        .Publish(ctx => new ReserveInventoryCommand(
            ctx.Instance.CorrelationId,
            ctx.Instance.OrderId,
            ctx.Instance.ProductId,
            ctx.Instance.Quantity))
        .TransitionTo(ReservingInventory));

During(
    ReservingInventory,
    When<InventoryReservedEvent>()
        .Then(ctx => ApplyInventoryReserved(ctx.Instance, ctx.MessageContext.Message))
        .WithCompensation(saga => new ReleaseInventoryCommand(saga.CorrelationId))
        .Publish(ctx => new ProcessPaymentCommand(
            ctx.Instance.CorrelationId,
            ctx.Instance.OrderId,
            ctx.Instance.Amount))
        .TransitionTo(ProcessingPayment));

During(
    ProcessingPayment,
    When<PaymentProcessedEvent>()
        .Then(ctx => ApplyPaymentProcessed(ctx.Instance, ctx.MessageContext.Message))
        .WithCompensation(saga => new RefundPaymentCommand(saga.CorrelationId))
        .Publish(ctx => new ShipOrderCommand(
            ctx.Instance.CorrelationId,
            ctx.Instance.OrderId,
            ctx.Instance.ShippingAddress))
        .TransitionTo(ShippingOrder),
    When<PaymentFailedEvent>()
        .Then(ctx => ApplyFailure(ctx.Instance, ctx.MessageContext.Message.Reason, nameof(PaymentFailedEvent)))
        .Compensate()
        .TransitionTo(Failed));
```

```
[SagaConcurrency(SagaConcurrencyMode.Optimistic)]
[SagaConcurrency(SagaConcurrencyMode.Pessimistic)]
```

```
[HandlerRetryPolicy(
    MaxRetries = 3,
    RetryDelayMilliseconds = 500,
    Strategy = BackoffStrategy.Exponential,
    UseJitter = false)]
3 references
public sealed class PaymentConsumer(
```

```
1 reference
public sealed record SendNotificationCommand(
    Guid CorrelationId,
    [property: IdempotencyKeyPart(0)] string UserId,
    [property: IdempotencyKeyPart(1)] string NotificationId,
    string Content) : IMessage;
```

Дякую за увагу!