Transaction Yönetimi – Saga

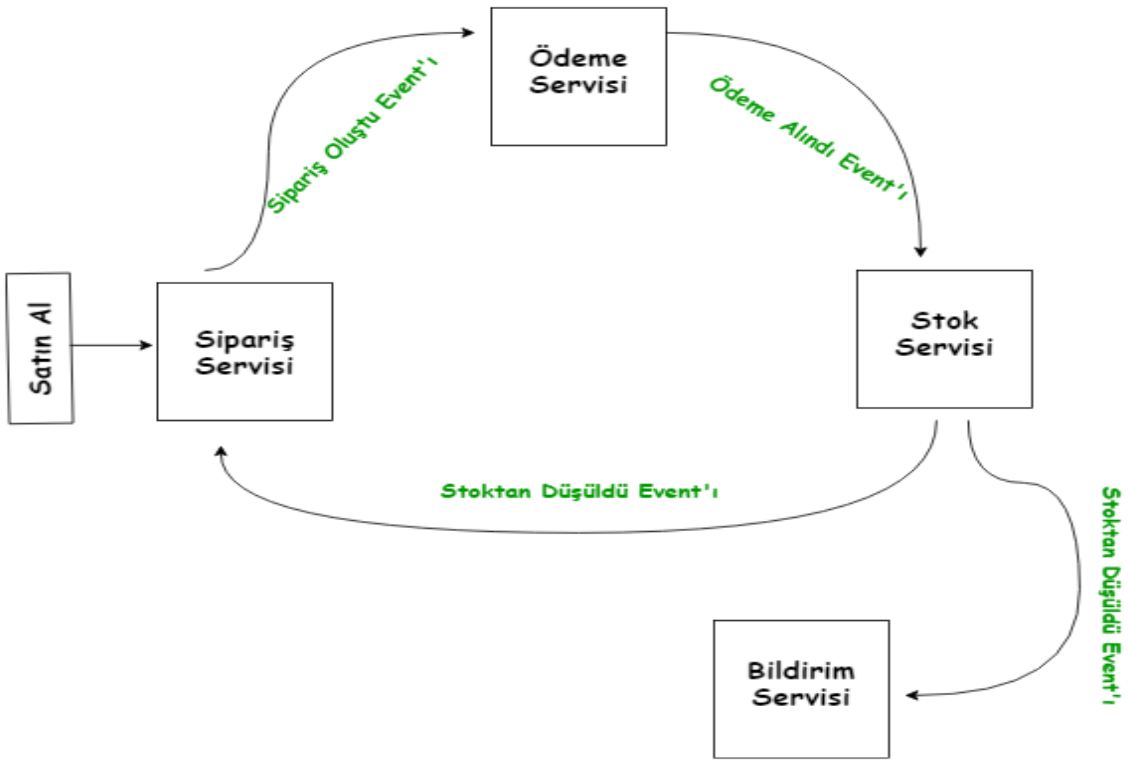
Eğer microservice mimari söz konusuysa, bu aslında birden fazla servisin ard arda çalışması anlamına gelir. Arda arda çalışan transaction'lar dizisinin yönetilmeye ihtiyacı duyulmatadır.

Saga Pattern
Dağıtık mimarilerde transaction yönetimi için en bilindik yöntemlerden birisi olan Saga Pattern, ilk olark 1987 yılında akademik bir makalede ortaya atıldı.

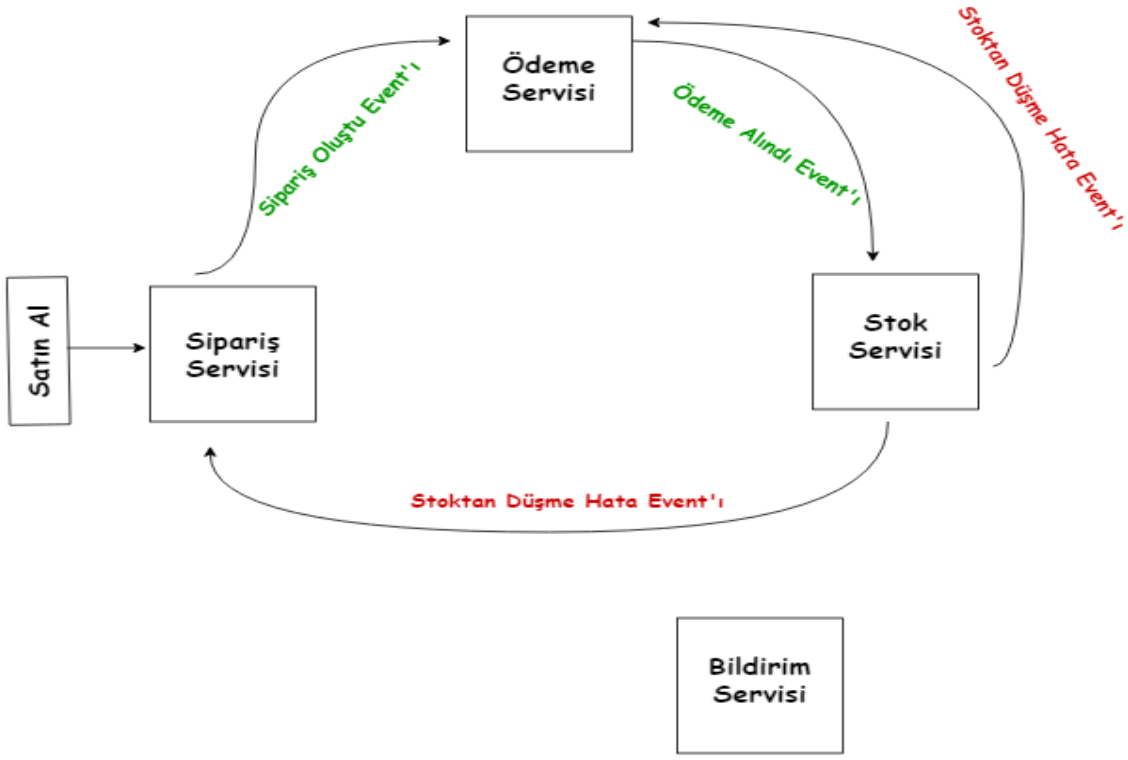
Saga, her transaction'ın farklı ve bağımsız bir servis üzerinde lokal olarak çalıştığı ve yine o servis içerisinde verisini güncellediği transaction'lar dizisidir. Bu tasarım kalıbına göre, ilk transaction, dış bir etki ile (kullanıcının kaydet butonuna tıklaması gibi) tetiklenir ve artık sonraki tüm transaction'lar bir önceki transaction'ın başarılı olması durumunda tetiklenecektir. Transaction'lardan herhangi birisinde meydana gelecek bir hata durumunda ise tüm süreç iptal edilerek Atomicty presibine bağlılık sağlanmış olur.

Events/Choreography Yöntemiyle Saga Uygulaması
Bu yöntemde ilk servis işini icra ettikten sonra bir event fırlatır ve bu event'ı dinleyen servis veya servisler tetiklenerek kendi local transaction'larını çalıştırır. Yani her servis aslında bir önceki servisin "ben işimi hallettim sıra sende" demesini bekler. Son servis çalıştıktan sonra artık bir event fırlatmaz ve süreç sonlanır.

Örnek Başarılı İşlem Senaryosu (Commit)



Örnek Başarısız İşlem Senaryosu (Rollback)



Spring ile uyuglama:
Örnek uygulama: https://eventuate.io/docs/manual/eventuate-tram/latest/getting-started-eventuate-tram-sagas.html

```
private SagaDefinition<CreateOrderSagaData> sagaDefinition =
        step()
          .withCompensation(this::reject)
        .step()
          .invokeParticipant(this::reserveCredit)
        .step()
          .invokeParticipant(this::approve)
        .build();


@Override
public SagaDefinition<CreateOrderSagaData> getSagaDefinition() {
  return this.sagaDefinition;
}
```

```
@Configuration
...
@Import({SagaOrchestratorConfiguration.class,
...
TramMessageProducerJdbcConfiguration.class,
EventuateTramKafkaMessageConsumerConfiguration.class
})
public class OrderConfiguration {
...
```

```
public class OrderService {

  @Autowired
  private SagaInstanceFactory sagaInstanceFactory;

  @Autowired
  private OrderRepository orderRepository;

  @Transactional
  public Order createOrder(OrderDetails orderDetails) {
    ResultWithEvents<Order> oe = Order.createOrder(orderDetails);
    Order order = oe.result;
    orderRepository.save(order);
    CreateOrderSagaData data = new CreateOrderSagaData(order.getId(), orderDetails);

    sagaInstanceFactory.create(createOrderSaga, data);

    return order;
  }

}
```