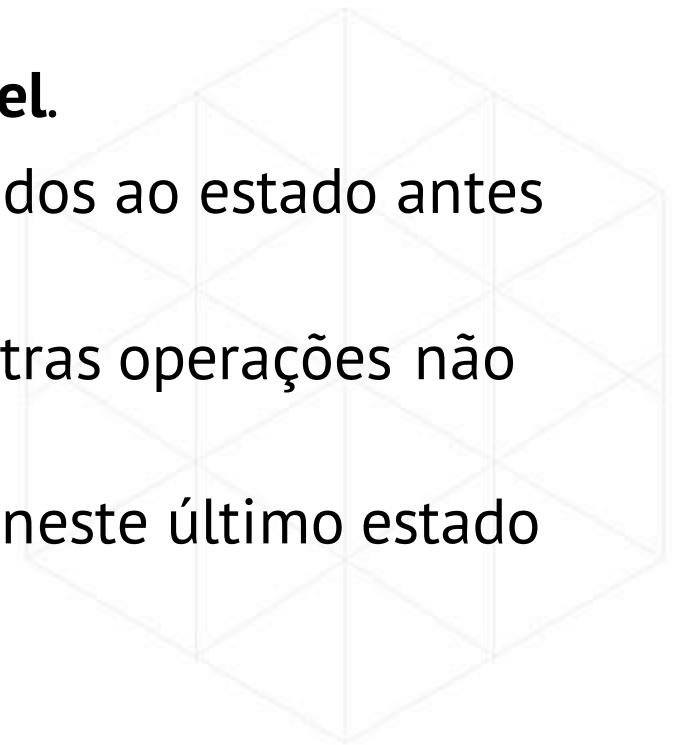




Transações em uma arquitetura de microsserviços

Transações tradicionais

- Sequência de operações tratada como um **bloco indivisível**.
- Criam um novo *estado válido* dos dados (ou retorna os dados ao estado antes da transação iniciada).
- Isolam acessos na mesma massa de dados. Quaisquer outras operações não devem interferir nesta até finalização.
- Validados de tal forma que os dados estarão disponíveis neste último estado correto.



Transações tradicionais

A

Atomic

All changes to the data must be performed successfully or not at all

C

Consistent

Data must be in a consistent state before and after the transaction

I

Isolated

No other process can change the data while the transaction is running

D

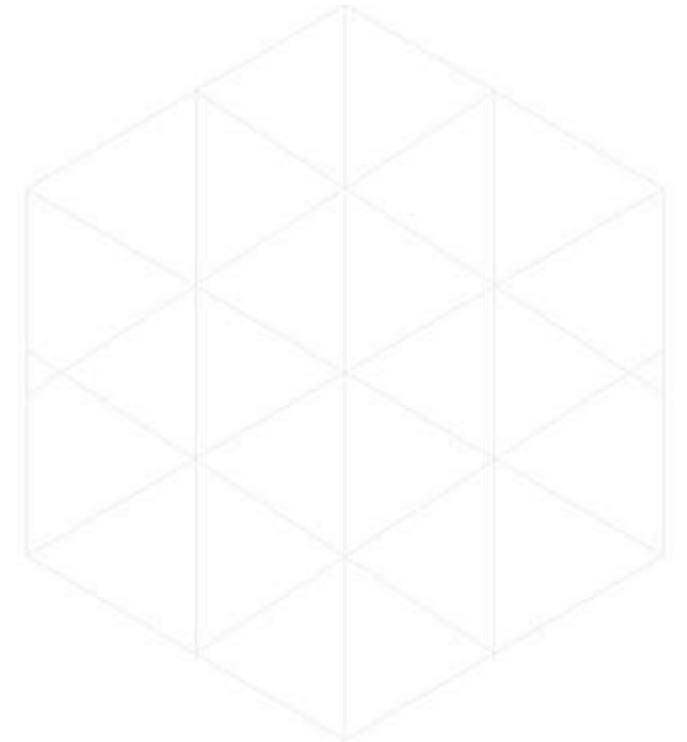
Durable

The changes made by a transaction must persist



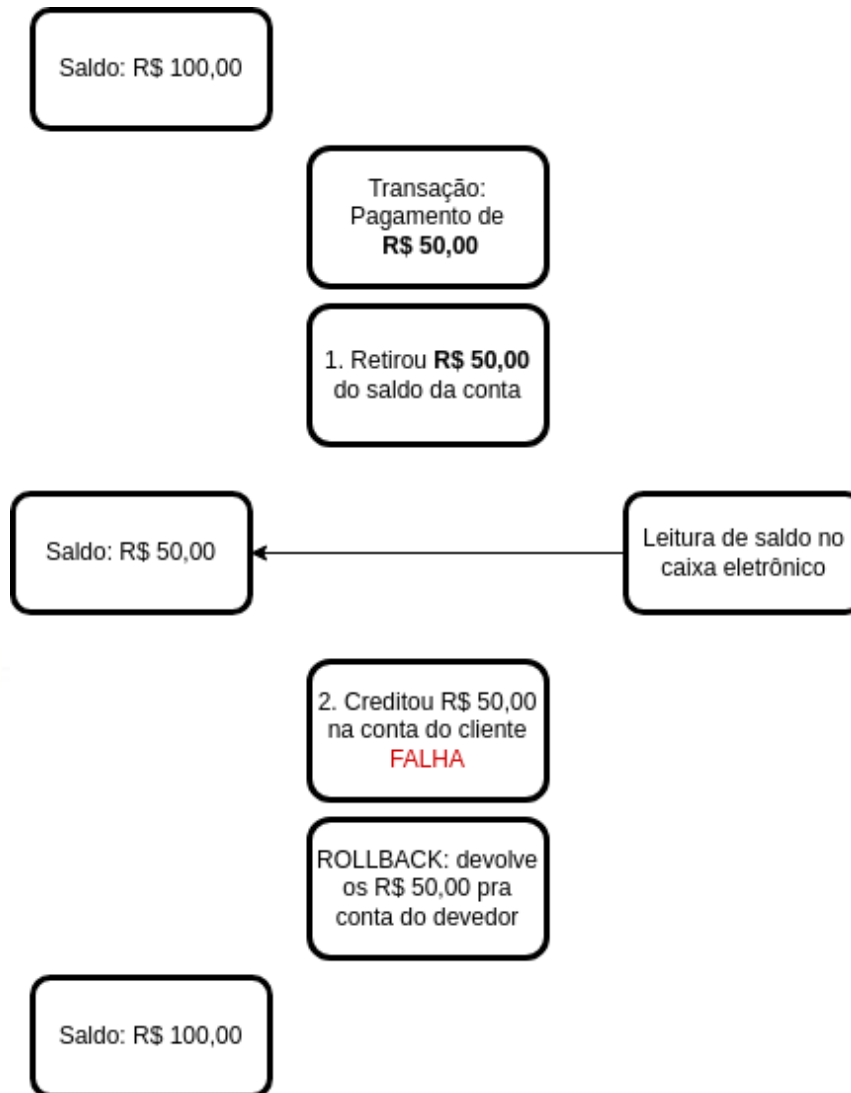
Fenômenos indesejados em transações

- Dirty Read
- Nonrepeatable read



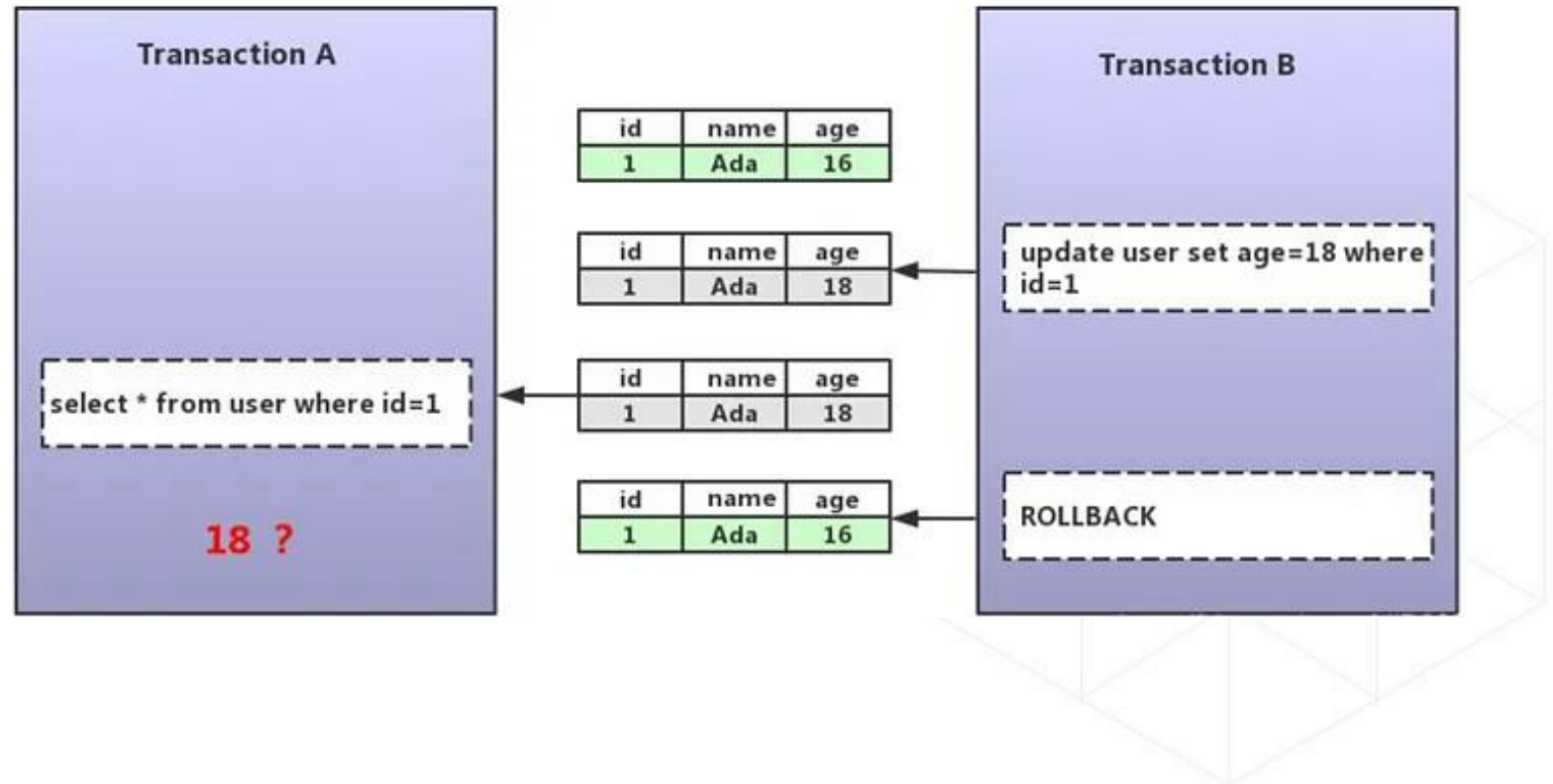
Dirty Read

- Leitura de dados não comitados



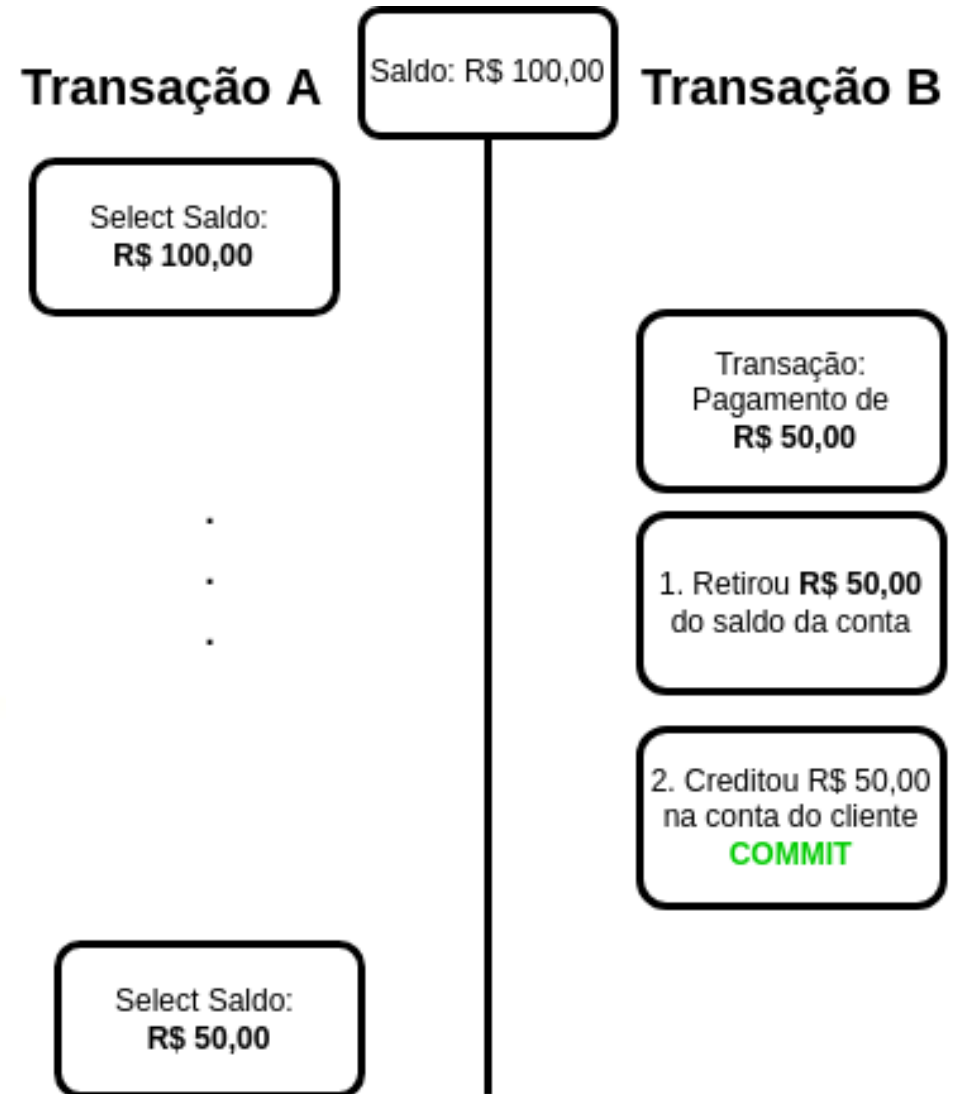
Dirty Read

- Leitura de dados não comitados



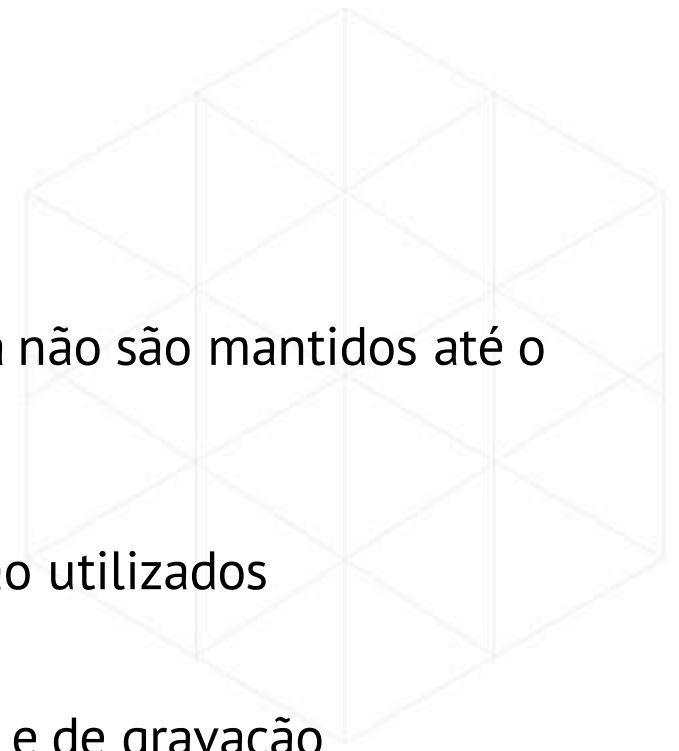
Nonrepeateble Read

- Transação lê dados uma segunda vez, que foram modificados por uma outra transação



Níveis de isolamento de uma transação

- Read uncommitted
 - Menor nível de isolamento. Permite os dirty reads
- Read committed
 - Utiliza locks na escrita de um registro, porém os locks de leitura não são mantidos até o final da transação
- Repeatable read
 - Utiliza Locks de leitura e gravação, porém os *Range Locks* não são utilizados
- Serializable
 - Maior nível de isolamento de transação. Requer Locks de leitura e de gravação
 - Também utiliza *range locks*

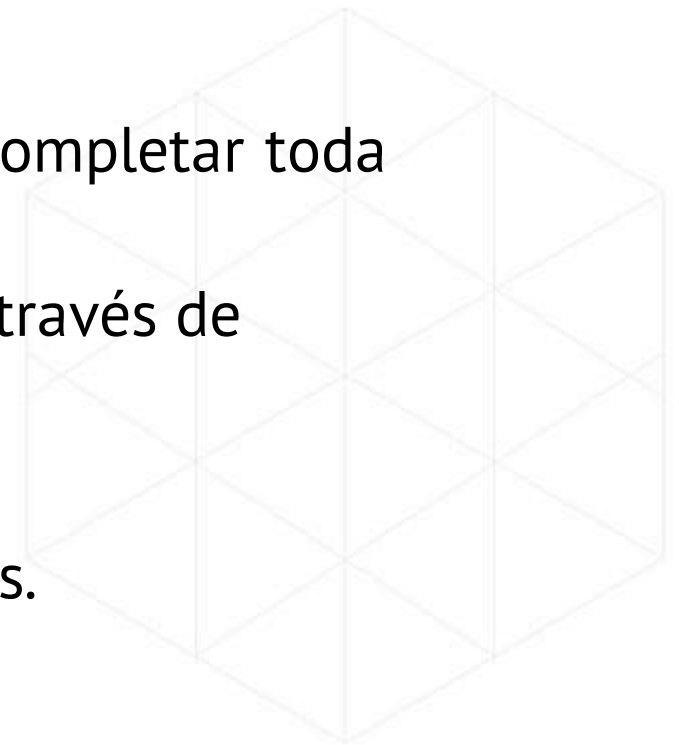


Níveis de isolamento VS Fenômenos

Nível de isolamento	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possível	Possível	Possível
Read committed	Impossível	Possível	Possível
Repeatable read	Impossível	Impossível	Possível
Serializable	Impossível	Impossível	Impossível

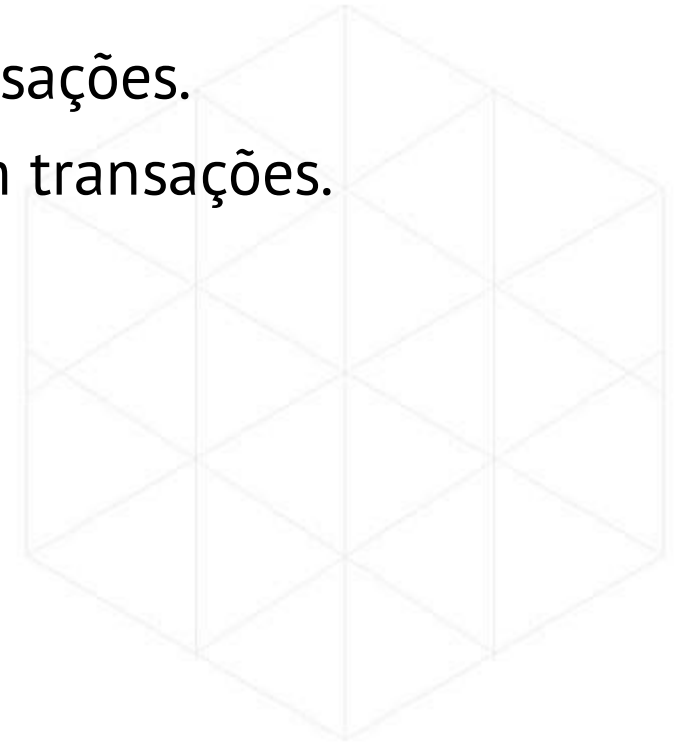
Transações em microserviços

- Uma transação é distribuída para vários serviços que são chamados sequencialmente ou paralelamente para completar toda a transação.
- Esse é o grande desafio, conseguir executar transações através de múltiplos serviços.
- Cada serviço tem seu próprio banco de dados.
- Deve haver uma coordenação *externa* entre esses serviços.



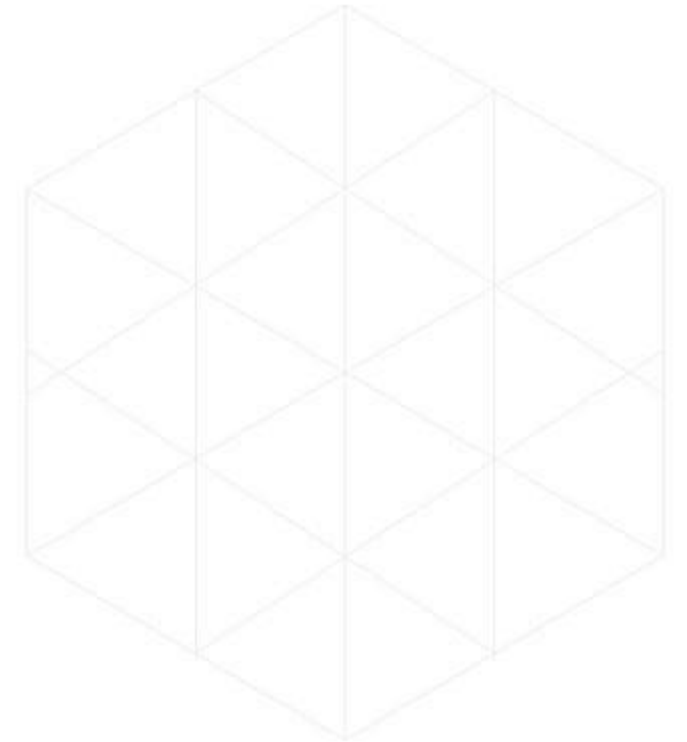
Transações em microserviços

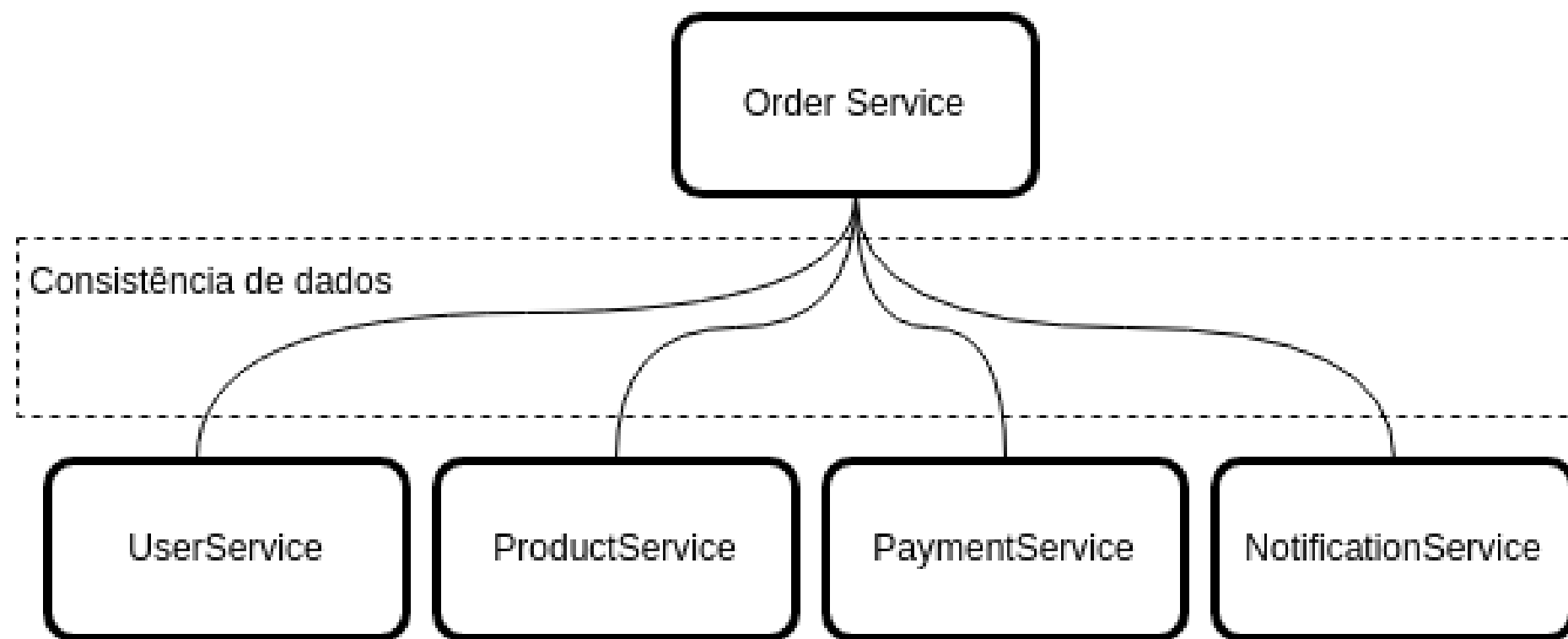
- Bancos como MongoDB ou Cassandra não suportam transações.
- Message brokers como Kafka ou RabbitMQ não suportam transações.
- Transações são síncronas.



Exemplo de transação

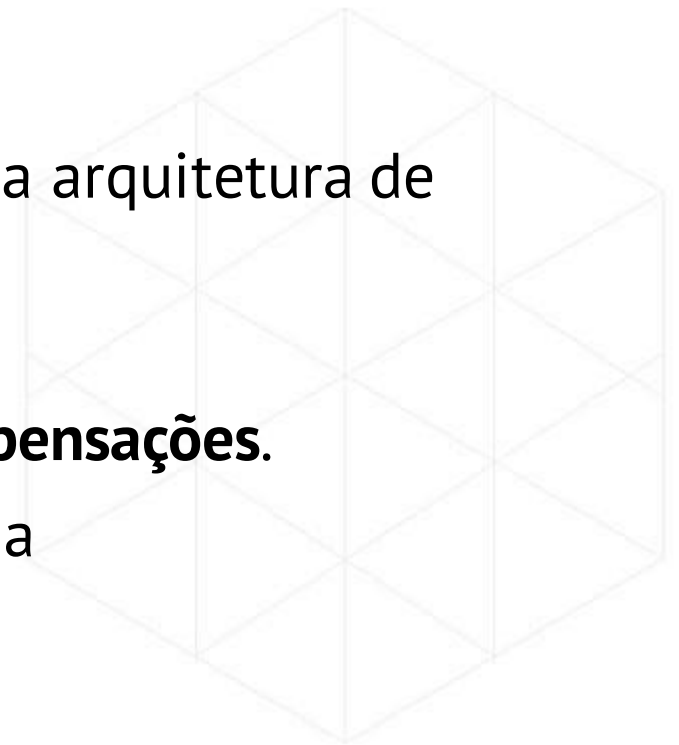
- Usuário compra um produto de R\$100,00
 1. Cria o registro Order
 2. Validar os dados do usuário
 3. Verificar disponibilidade do produto
 4. Bloquear quantidade do produto
 5. Processar pagamento
 6. Processar produto
 7. Enviar e-mail de pedido Disponível





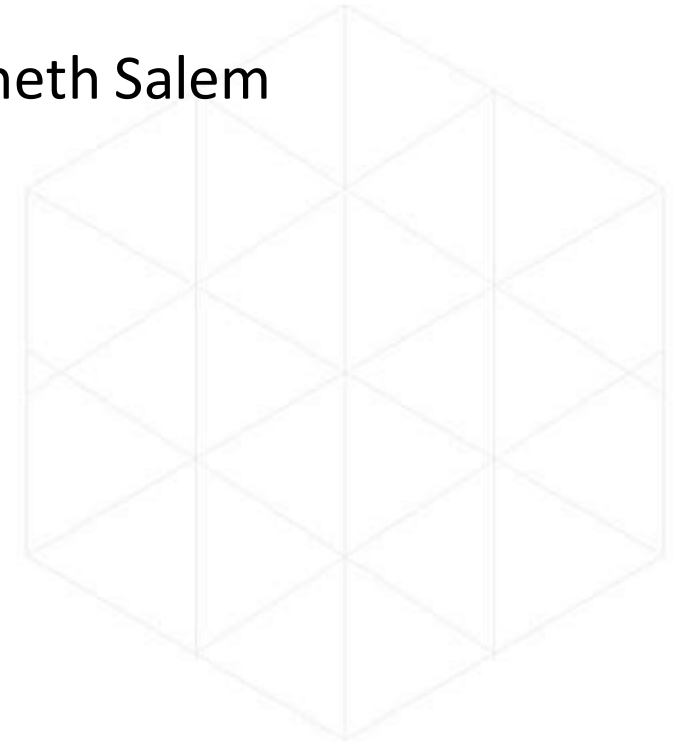
Saga

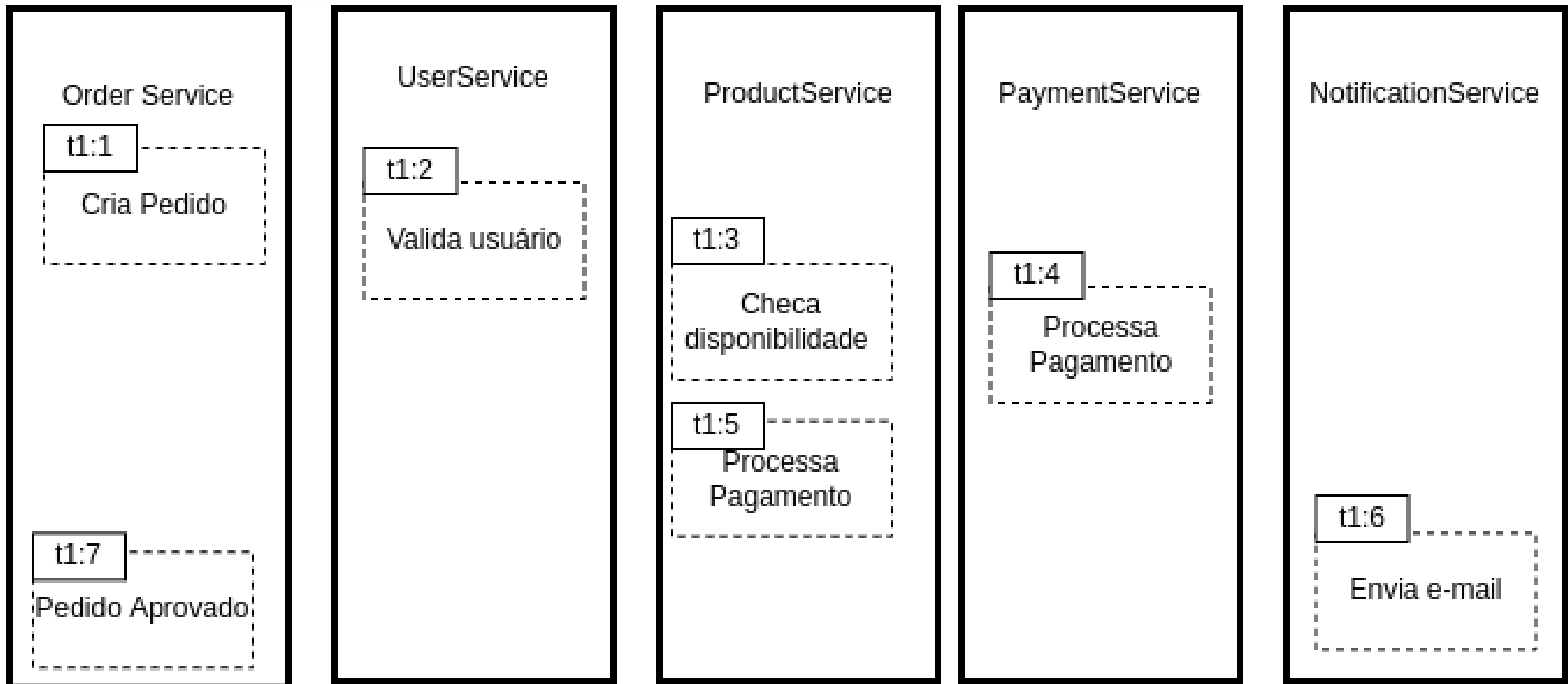
- <http://microservices.io/patterns/data/saga.html>
- Mecanismos para manter a consistência de dados em uma arquitetura de microsserviços
- Sequência de transações locais
- É definida por meio de dois conjuntos: **transações** e **compensações**.
- É mais uma metodologia de desenvolvimento do que uma tecnologia propriamente dita



Saga

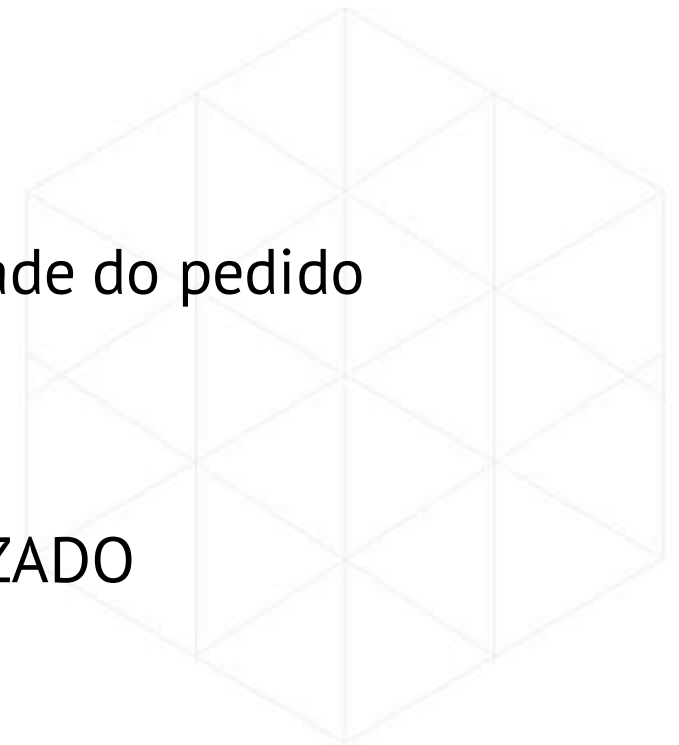
- Modelo proposto em **1987** por Hector Garcia-Molina e Kenneth Salem para transações de longa duração.
- <https://dl.acm.org/doi/10.1145/38714.38742>





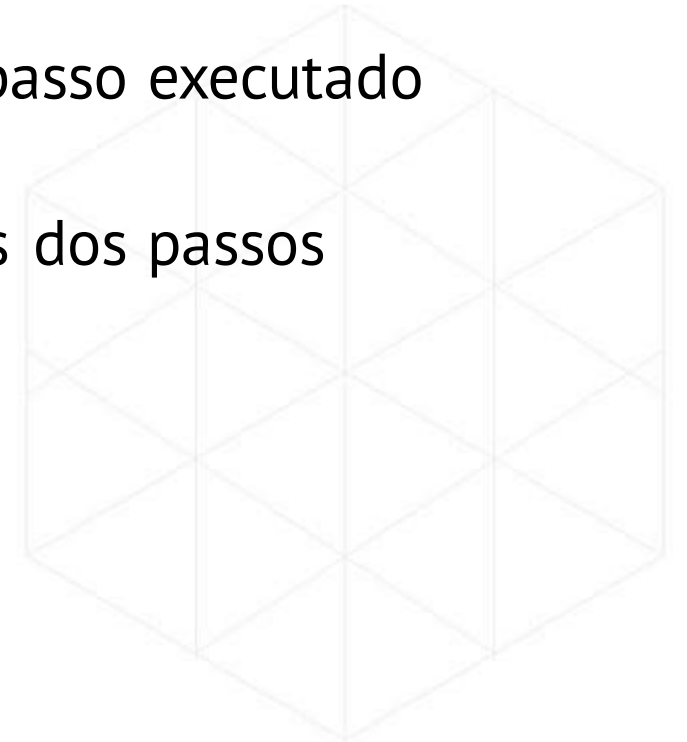
Saga

1. Criar Pedido com status PENDENTE
2. Validar usuário
3. Verificar se produto está disponível e *lockar* a quantidade do pedido
4. Criar Pagamento com status PENDENTE
5. Atualizar o Status do pagamento para AUTORIZADO
6. Atualizar o status do pedido para PAGAMENTO_REALIZADO
7. Dar baixa no estoque do produto e liberar *lock*
8. Enviar e-mail de notificação
9. Alterar Status do pedido para Aprovado



Rollback?

- Como dito antes, em uma implementação de saga, cada passo executado explicitamente deve ter uma **compensação**.
- Se uma transação falhar no passo **N**, todas compensações dos passos anteriores devem ser executadas.

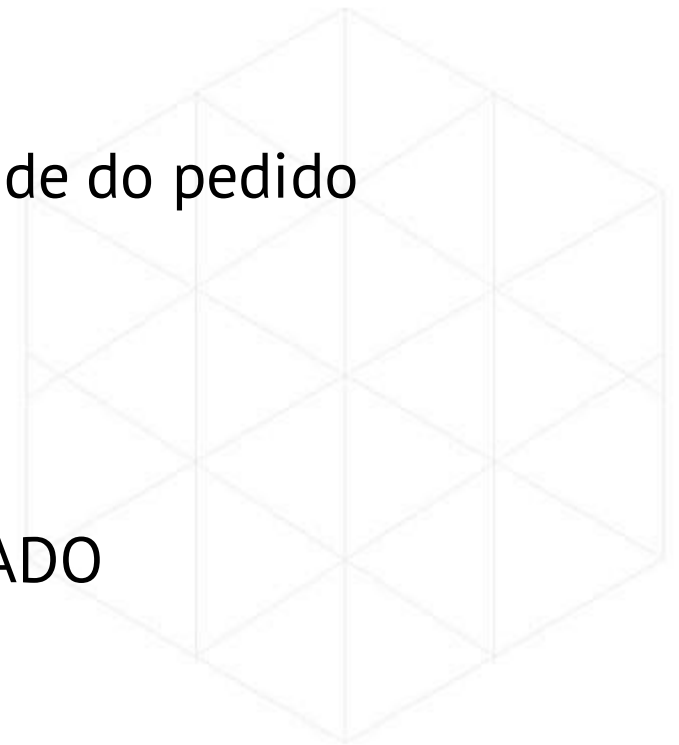


Rollback

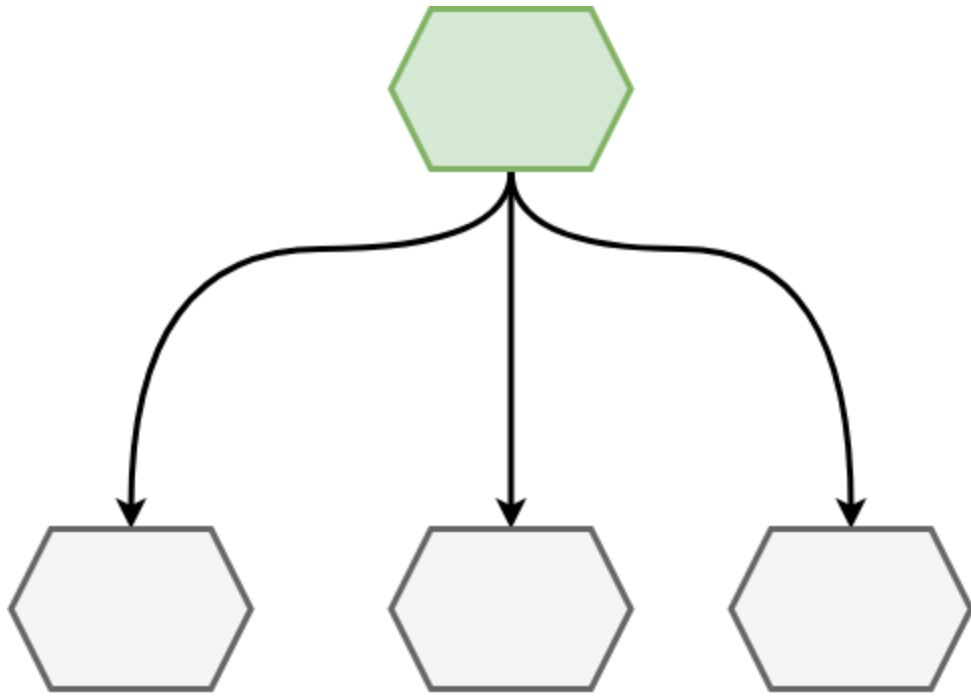
Passo	Serviço	Transação	Compensação
1	OrderService	Criar pedido vinculado ao usuário	-
2	User Service	Validar dados do usuário	CancelarPedido
3	ProductService	Verificar disponibilidade do produto	CancelarPedido
4	PaymentService	Processar pagamento	CancelarPedido; LiberarProdutos;
5	ProductService	Processar estoque	-
6	NotificationService	Enviar e-mail de pedido criado	-
7	OrderService	Atualiza status para APROVADO	-

Rollback

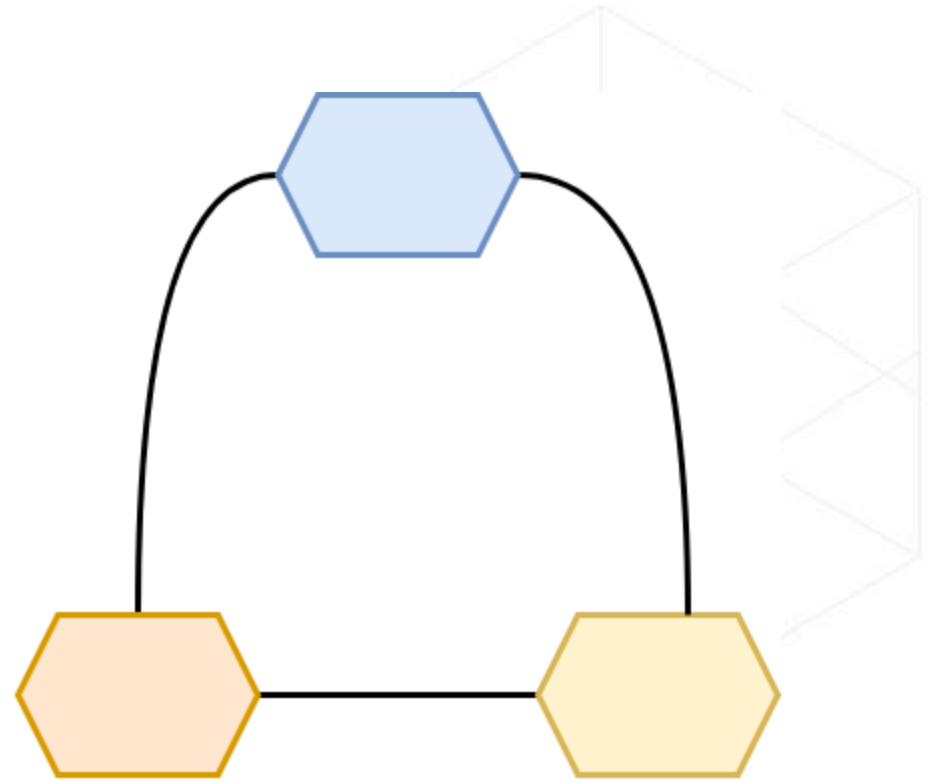
1. Verificar se usuário está autenticado e autorizado
2. Verificar se produto está disponível e *lockar* a quantidade do pedido
3. Criar Pedido com status PENDENTE
4. Criar Pagamento com status PENDENTE
5. Atualizar o Status do pagamento para NEGADO
6. Atualizar o status do pedido para PAGAMENTO_REJEITADO
7. Liberar *lock* de produto
8. Enviar e-mail de notificação de pagamento rejeitado



Coordenando Sagas



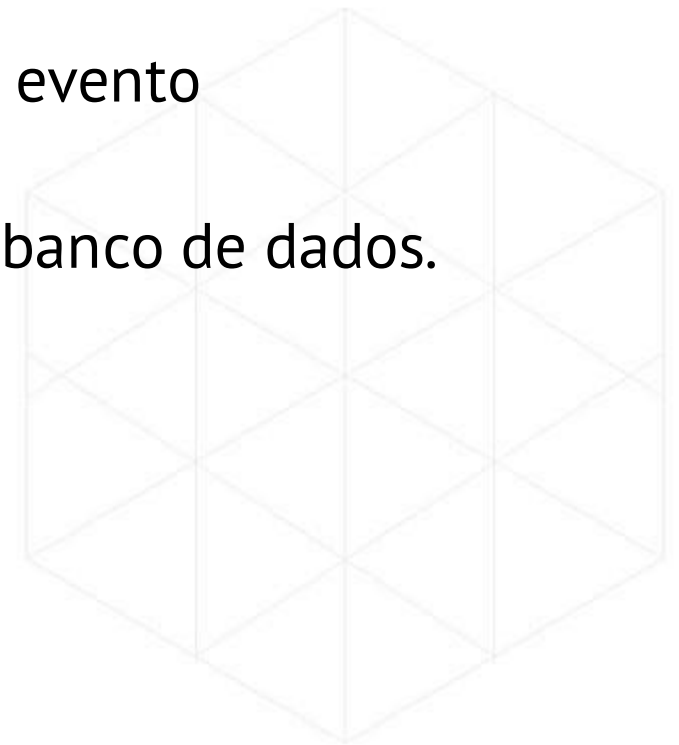
Orquestração



Coreografia

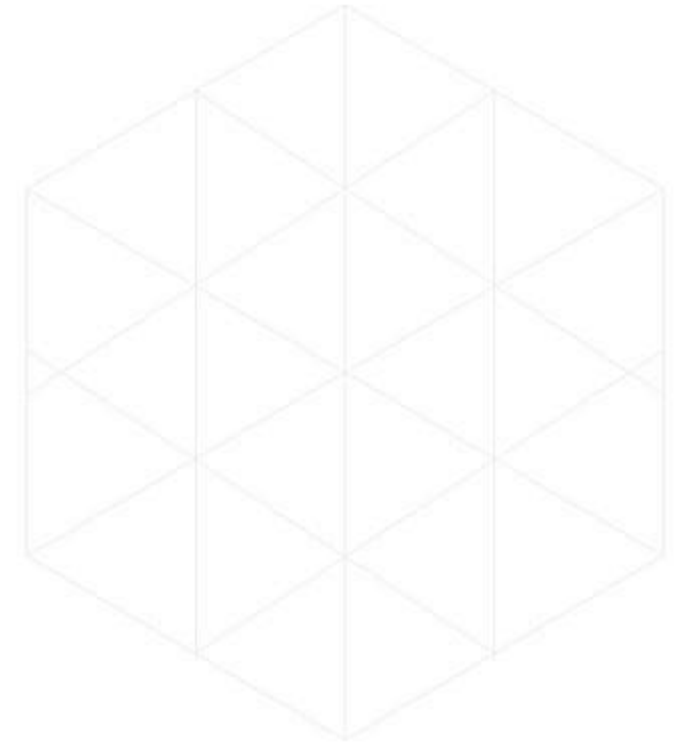
Comunicação baseada em eventos

- Cada passo deve atualizar seu próprio banco e emitir um evento correspondente.
- Cada participante deve mapear o evento em seu próprio banco de dados.



Eventos - Vantagens

- Simplicidade de implementação.
- Baixo acoplamento entre os participantes.
- Excelentes para sagas pequenas.



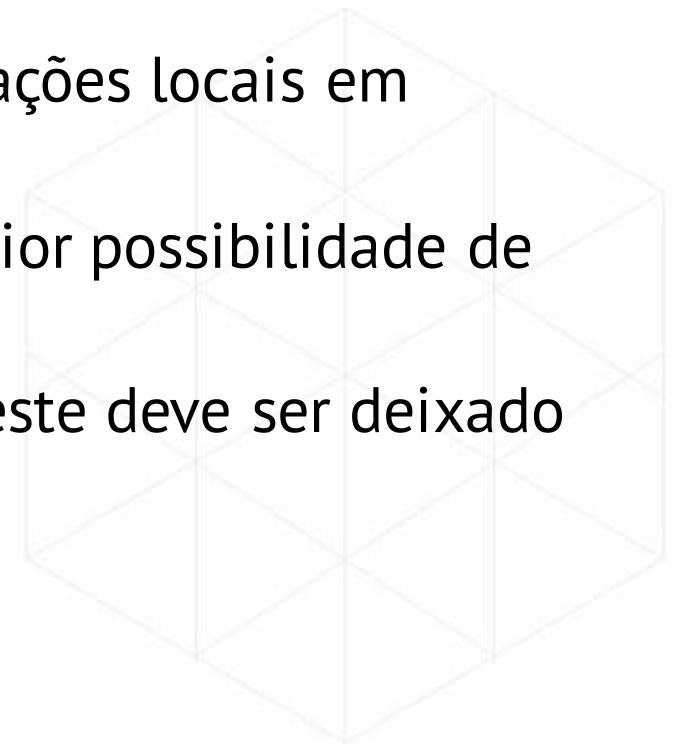
Eventos - Desvantagens

- Curva de aprendizado longa.
- Dependências cíclicas.
- Risco de acoplamento, devido a dependência na ordem de execução.



Saga Coreografada

- Cada transação local publica eventos que acionam transações locais em outros serviços
- É importante, neste estágio, que todos os passos com maior possibilidade de falha sejam executados PRIMEIRO
- Caso haja algum passo sem a possibilidade de rollback, este deve ser deixado por último



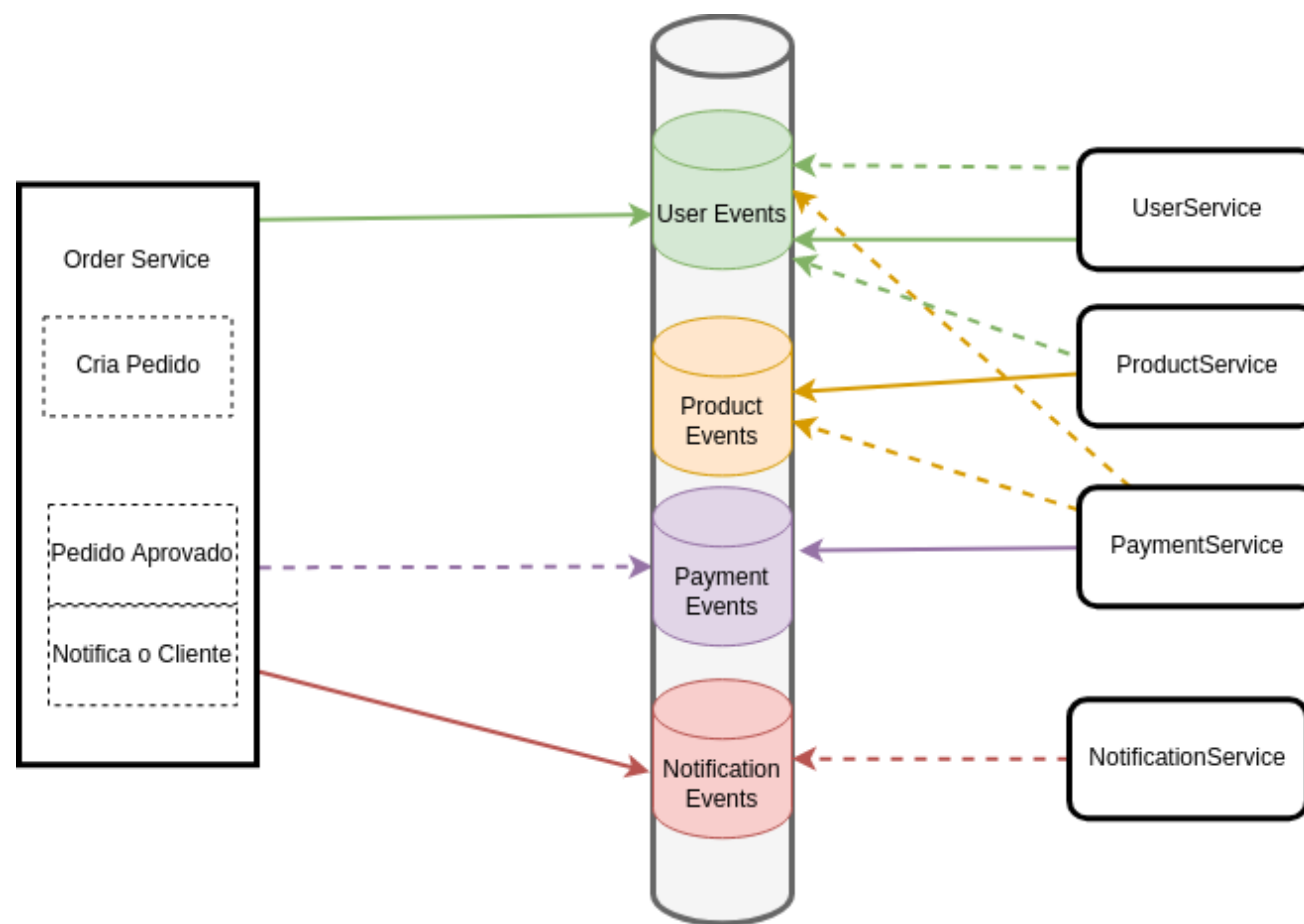
Saga Coreografada

1. Order Service cria Order como PENDENTE e publica o evento ORDER_CREATED
2. User Service lê o evento ORDER_CREATED e valida se o usuário tem as devidas permissões para criar o Order e publica USER_APPROVED
3. Product Service lê o evento ORDER_CREATED e USER_APPROVED, processa os respectivos produtos e publica o evento PRODUCT_APPROVED

Saga Coreografada

1. Payment Service lê o evento `PRODUCT_VALIDATED` e cria uma cobrança no cartão de crédito com o status `WAITING_AUTHORIZATION`
2. Quando o pagamento é autorizado, é emitido o evento `PAYMENT_AUTHORIZED`
3. O notification Service lê o evento `PAYMENT_AUTHORIZED` e envia o respectivo e-mail ao usuário

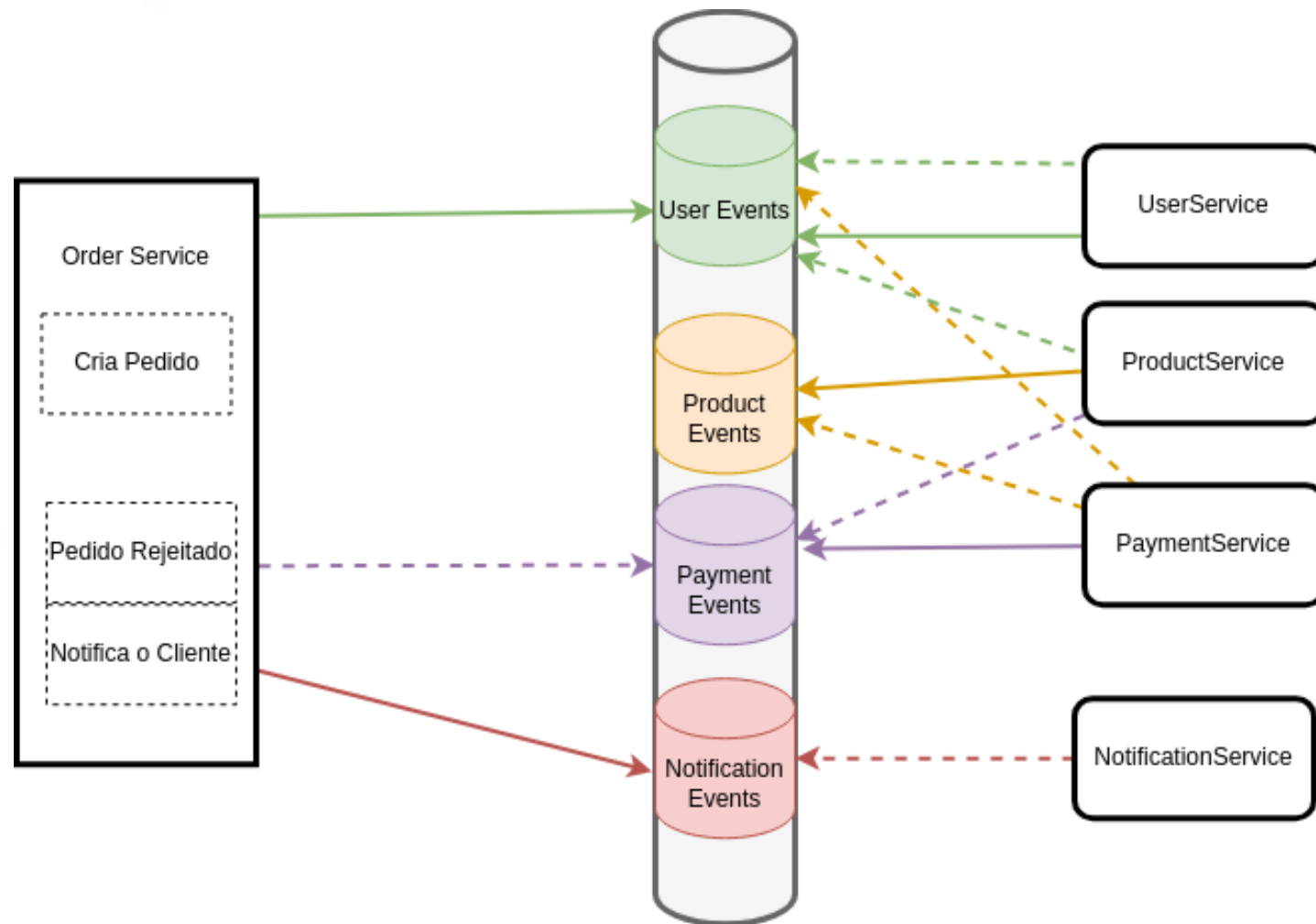
Saga Coreografada



Saga Coreografada - Rejeitada

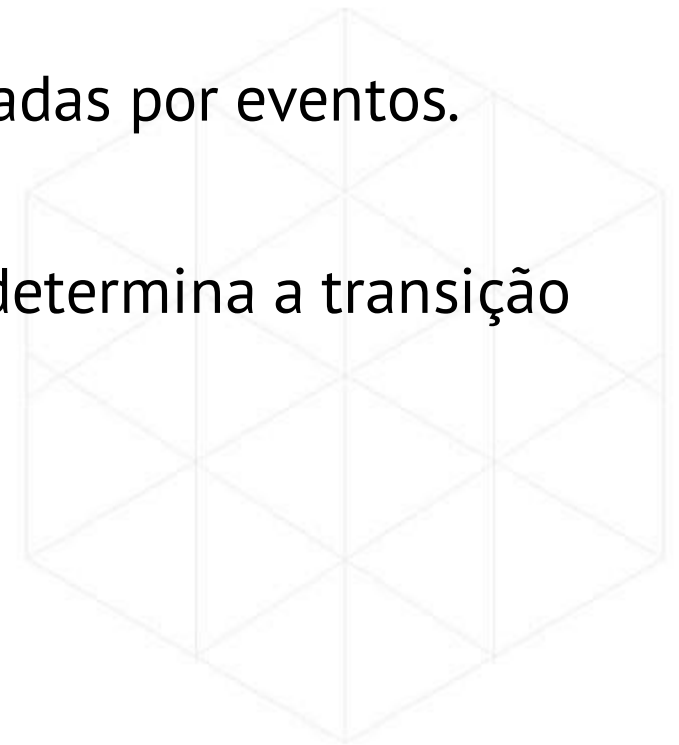
1. Payment Service lê o evento `PRODUCT_VALIDATED` e cria uma cobrança no cartão de crédito com o status `WAITING_AUTHORIZATION`
2. Quando o pagamento não é autorizado, é emitido o evento `PAYMENT_UNAUTHORIZED`
3. O notification Service lê o evento `PAYMENT_UNAUTHORIZED` e envia o respectivo e-mail ao usuário
4. Order Service lê o evento `PAYMENT_UNAUTHORIZED` e muda o status do pedido para cancelado
5. O Product Service lê o evento `PAYMENT_UNAUTHORIZED` e volta os produtos ao estoque
6. O notification Service lê o evento `PAYMENT_UNAUTHORIZED` e notifica o usuário

Saga Coreografada - Rejeitada

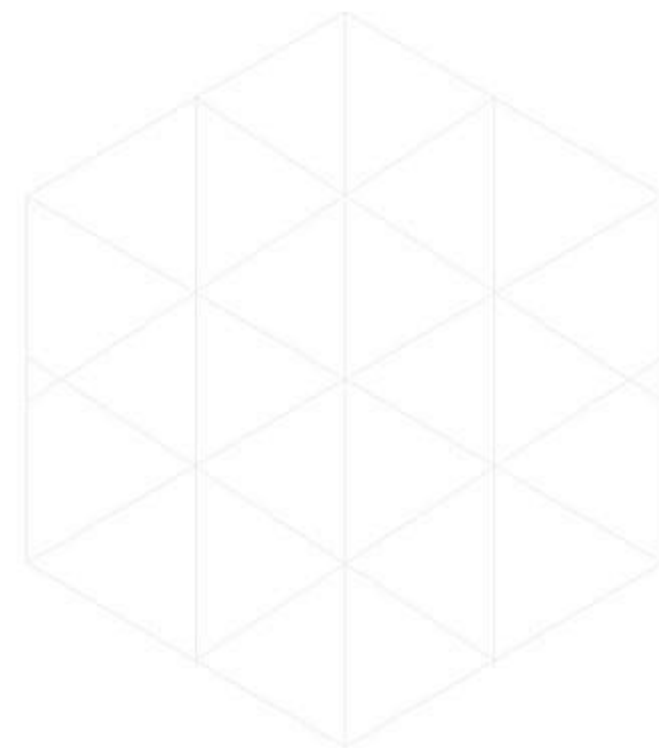
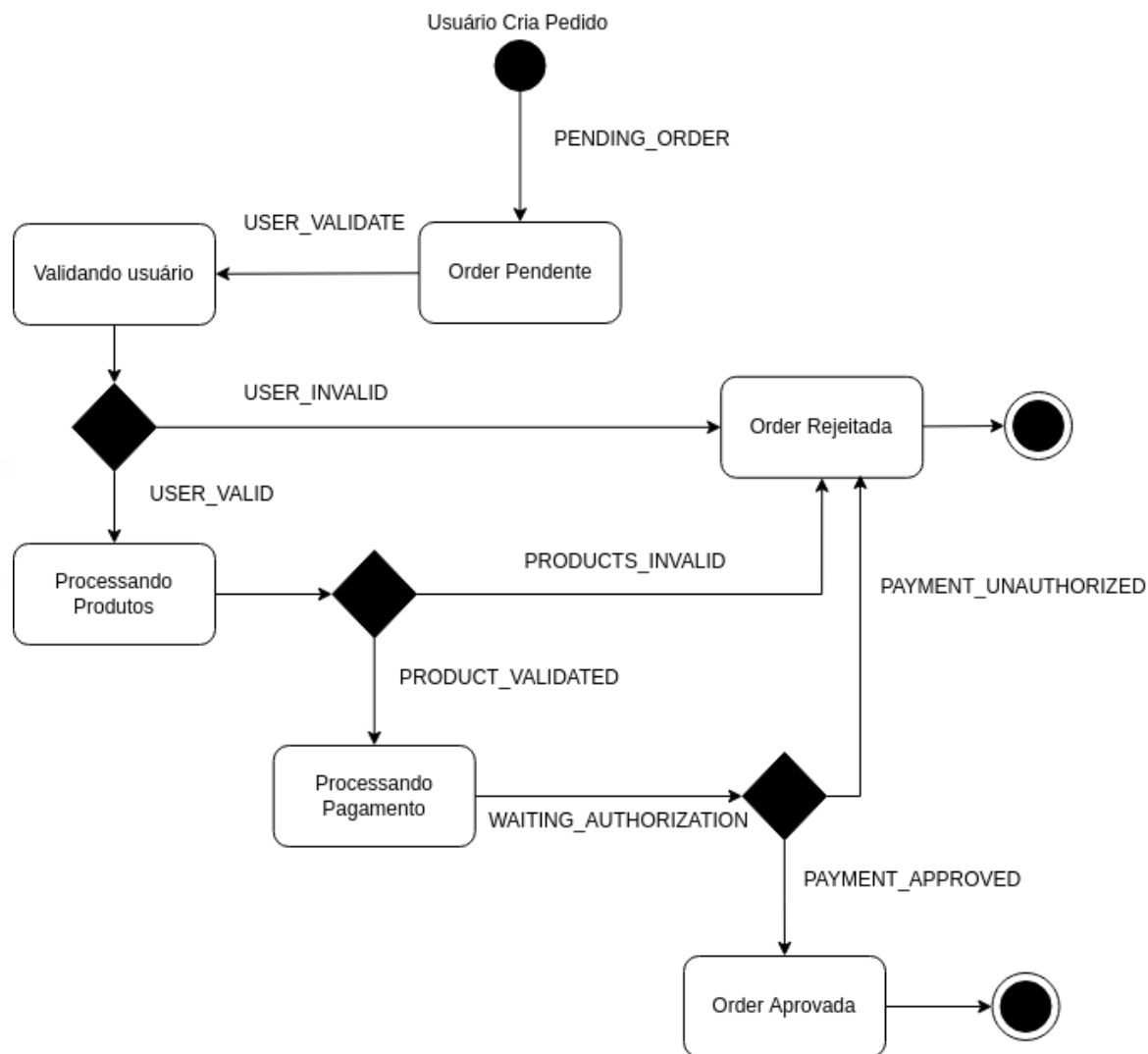


Sagas como Máquinas de Estados

- Uma série de estados e um conjunto de transições acionadas por eventos.
- Cada transição invoca um participante da saga.
- O estado atual da saga e o resultado da transação local determina a transição de estado e que ação performar.

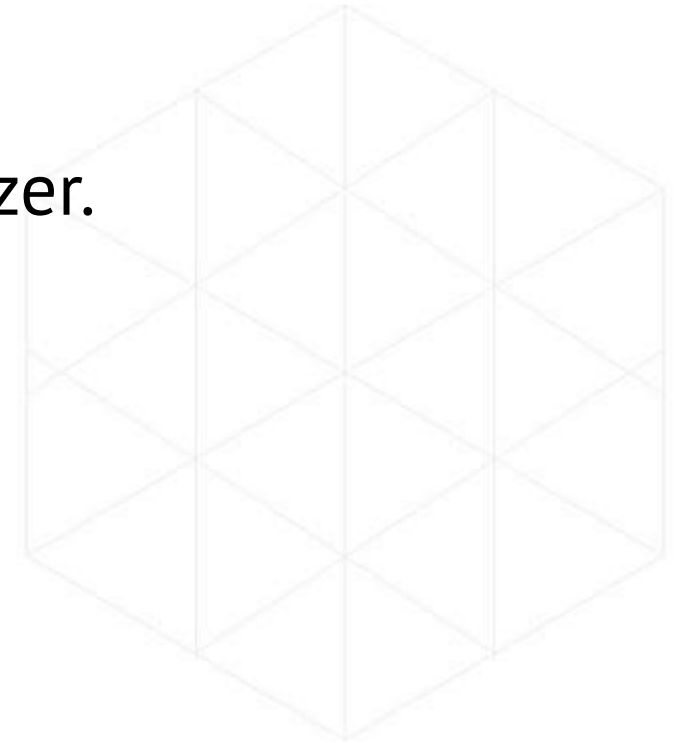
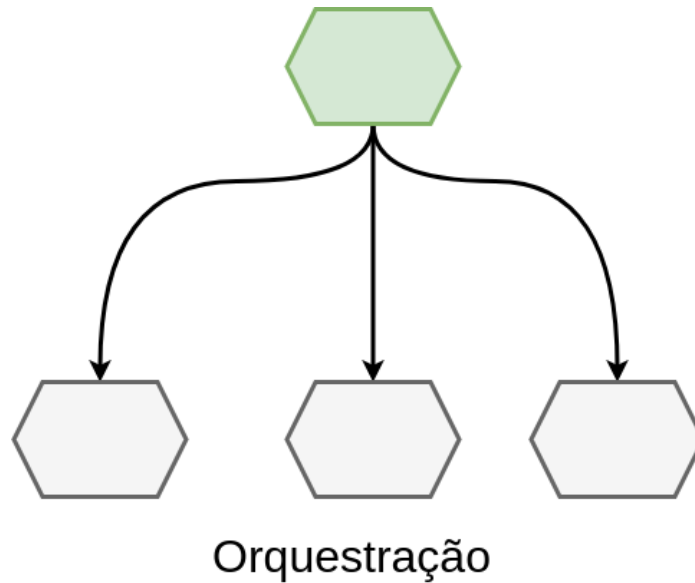


Sagas como Máquinas de Estados



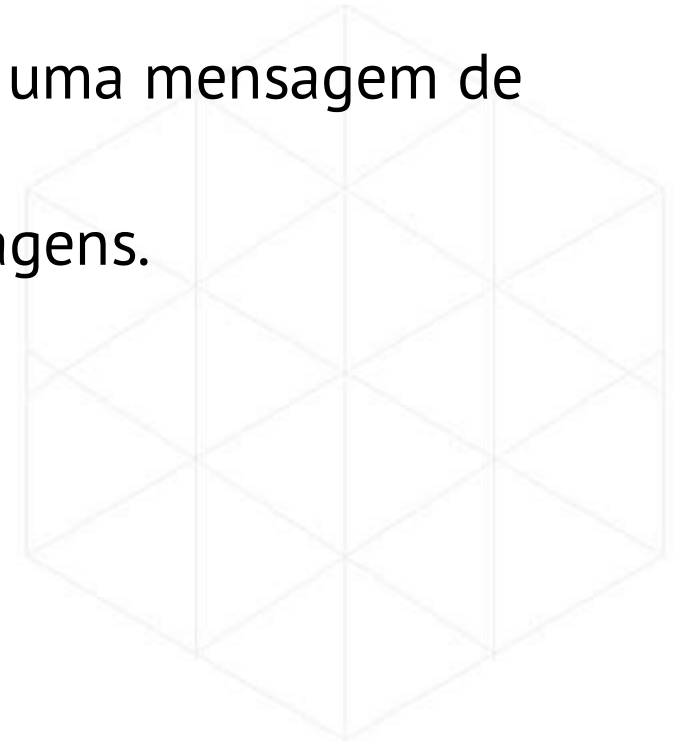
Orquestração

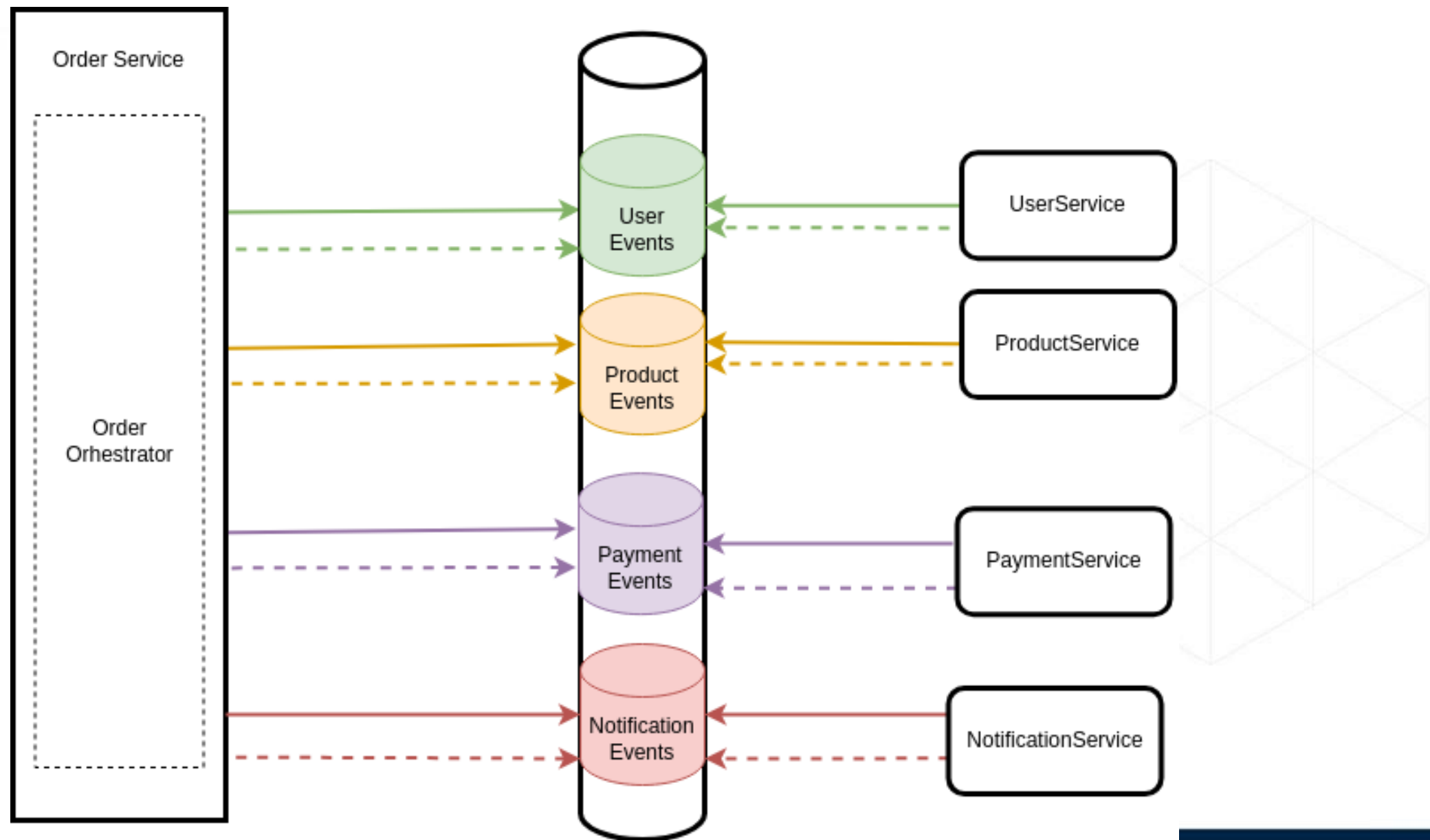
- Componente centralizado.
- Responsável por dizer aos participantes o que fazer.



Sagas orquestradas

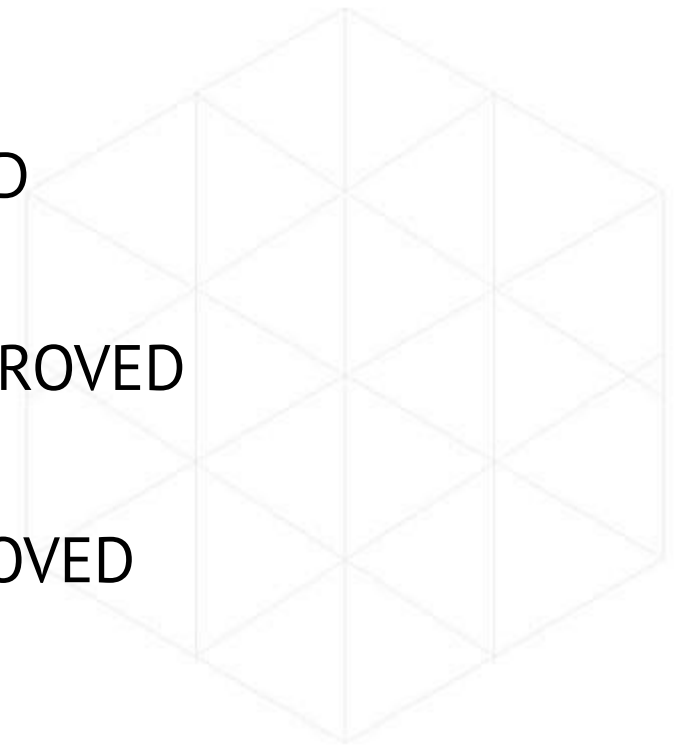
- Cada passo deve atualizar suas próprias bases e publicar uma mensagem de evento.
- Devem ser utilizados os conceitos já estudados de mensagens.





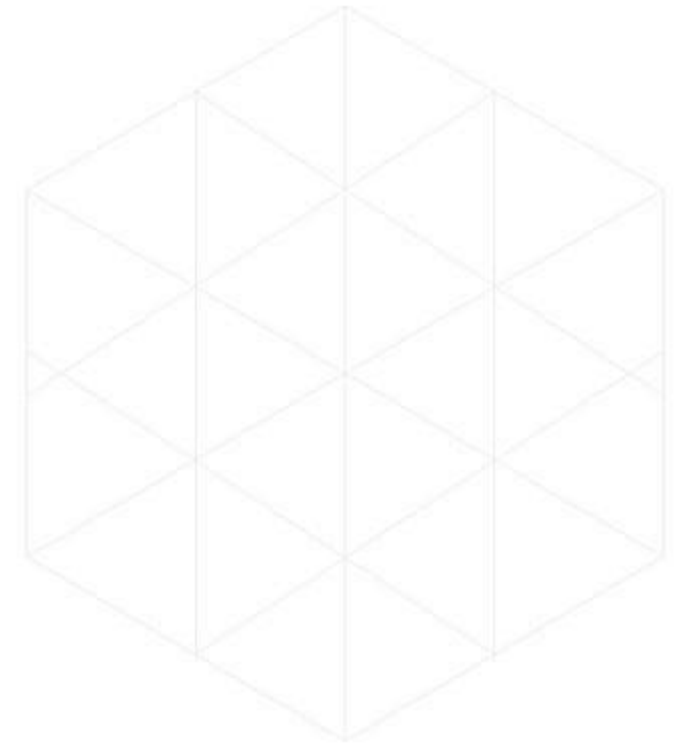
Saga - Orquestração

1. Orquestrador envia o evento USER_VALIDATE
2. UserService responde com um evento USER_APPROVED
3. Orquestrador envia o evento PROCESS_PRODUCTS
4. ProductService responde com o evento PRODUCT_APPROVED
5. O orquestrador envia o evento PAYMENT_APPROVE
6. O Payment Service responde com um PAYMENT_APPROVED
7. O orquestrador envia um comando de notificação
8. O Notification Service envia as notificações ao usuário
9. O Order Service altera o ORDER para APPROVED



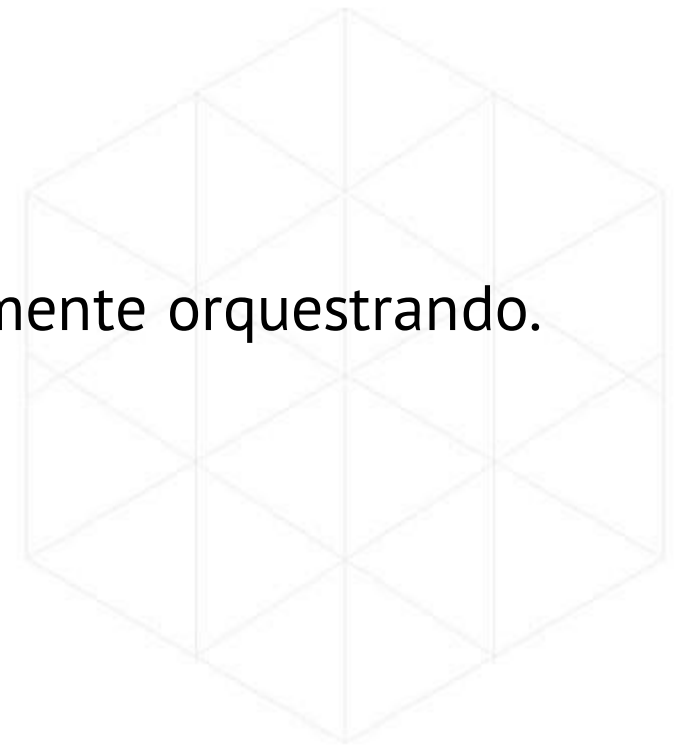
Orquestração - Vantagens

- Simplifica a dependência entre os serviços.
- Não causa dependência cíclica.
- Baixo acoplamento entre os serviços.
- Simplifica o modelo e regras de negócios.
- Funciona melhor para sagas mais complexas.



Orquestração - Desvantagens

- Risco de centralização de muita lógica no orquestrador.
- O ideal é ***não manter lógica de negócio*** no orquestrador.
- Criar um serviço como uma espécie de roteador, simplesmente orquestrando.



A Falta de isolamento

A

Atomic

All changes to the data must be performed successfully or not at all

C

Consistent

Data must be in a consistent state before and after the transaction

I

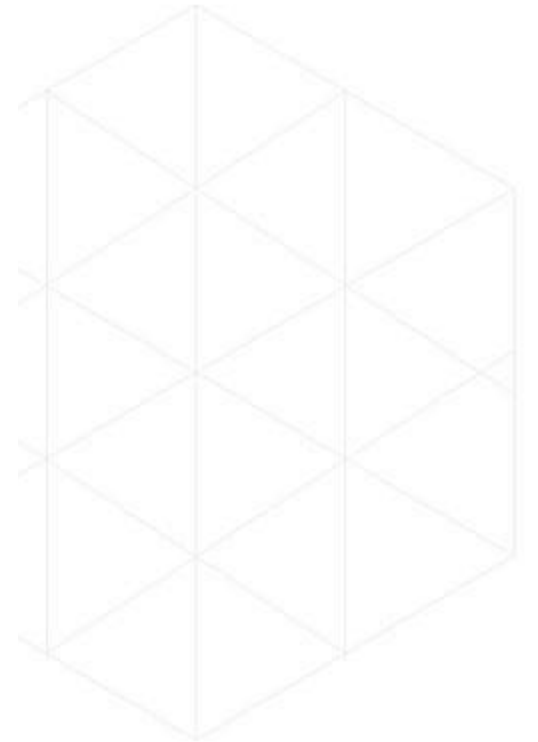
Isolated

No other process can change the data while the transaction is running

D

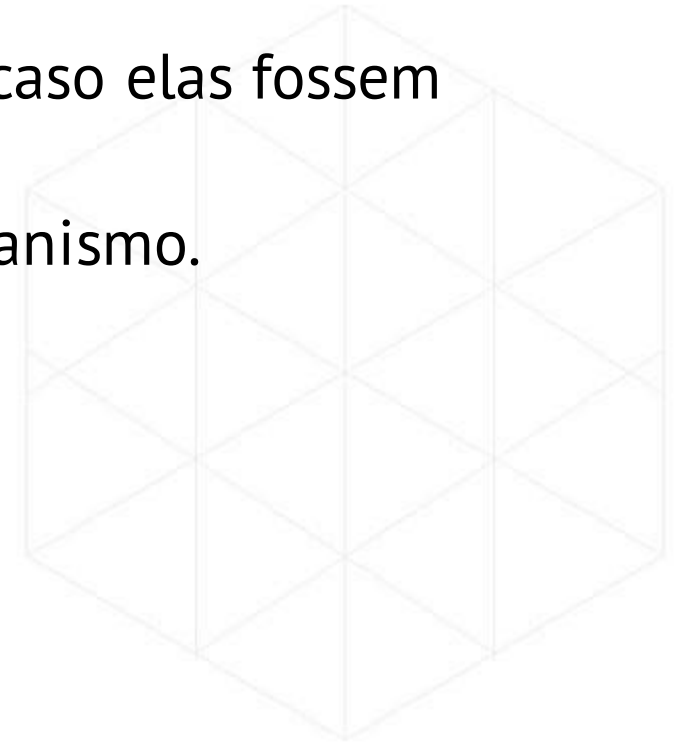
Durable

The changes made by a transaction must persist



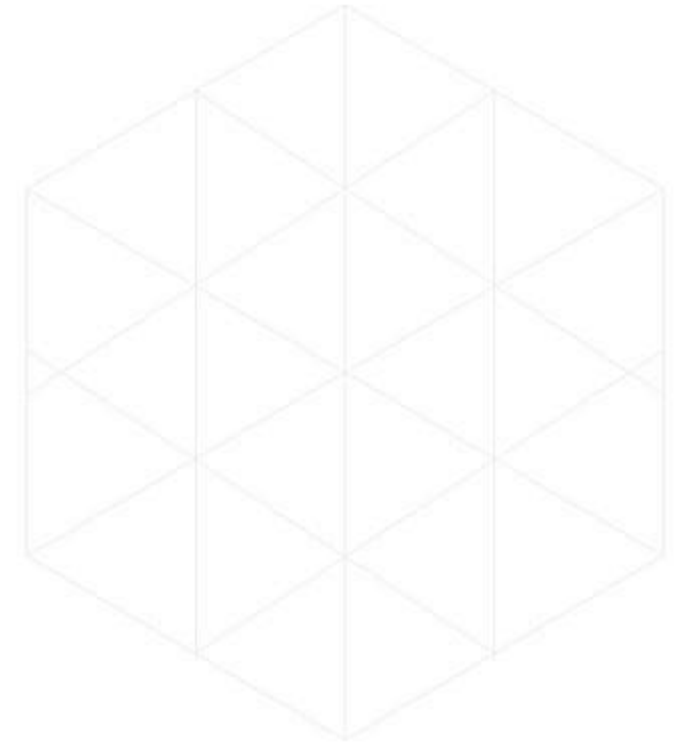
A Falta de isolamento

- O resultado das múltiplas transações deve ser o mesmo caso elas fossem executadas de maneira serial.
- Em microsserviços não é possível a utilização desse mecanismo.
- Cada update imediatamente é visível a outras sagas.
- Podem acabar tratando com dados inconsistentes.



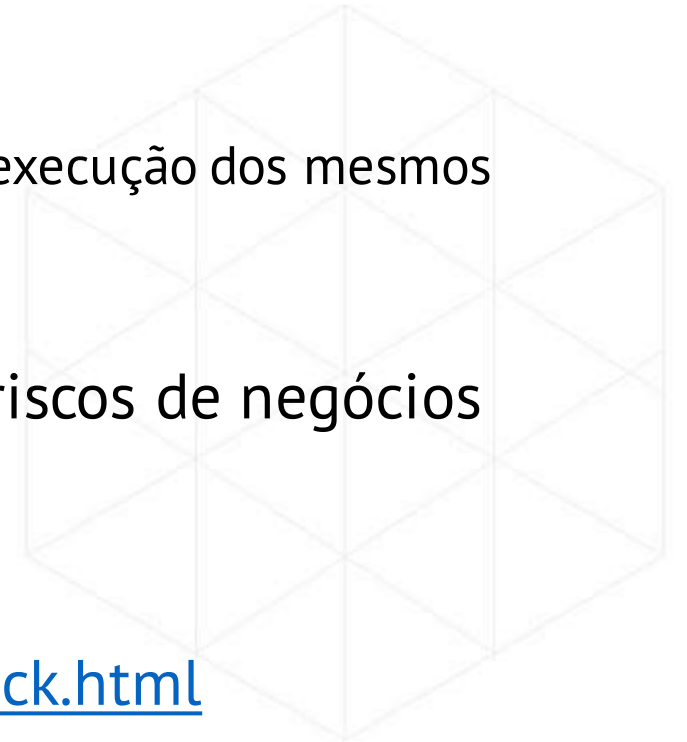
Anomalias da falta de isolamento

- Nonrepeatable Read
- Dirty Read
- Ghost Read



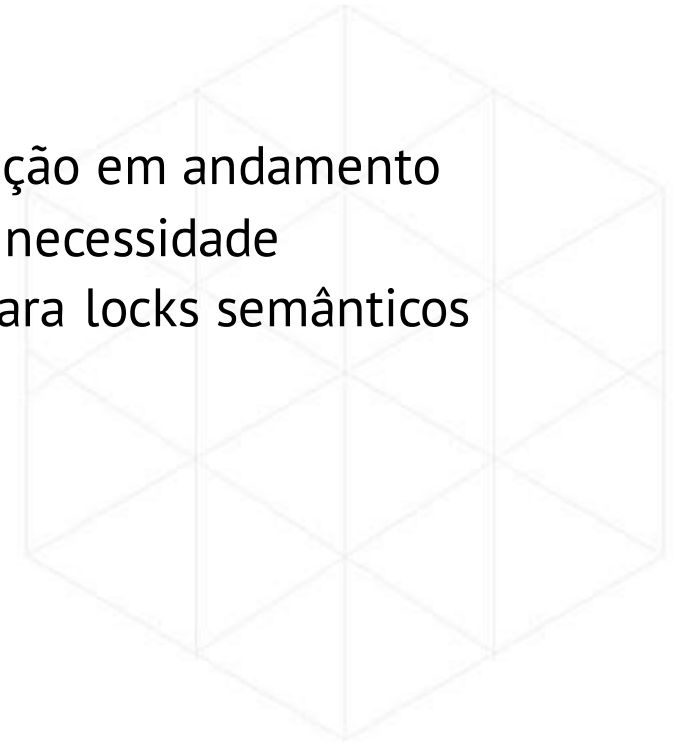
Contramedidas

- Updates comutativos
 - Updates devem ser desenhados de maneira que o resultado da execução dos mesmos seja a mesma em qualquer ordem
- Visão pessimista
 - Reordenar os passos de uma saga para minimizar os riscos de negócios
- Releitura de valores.
- Versionamento de updates.
- <https://martinfowler.com/eaCatalog/optimisticOfflineLock.html>



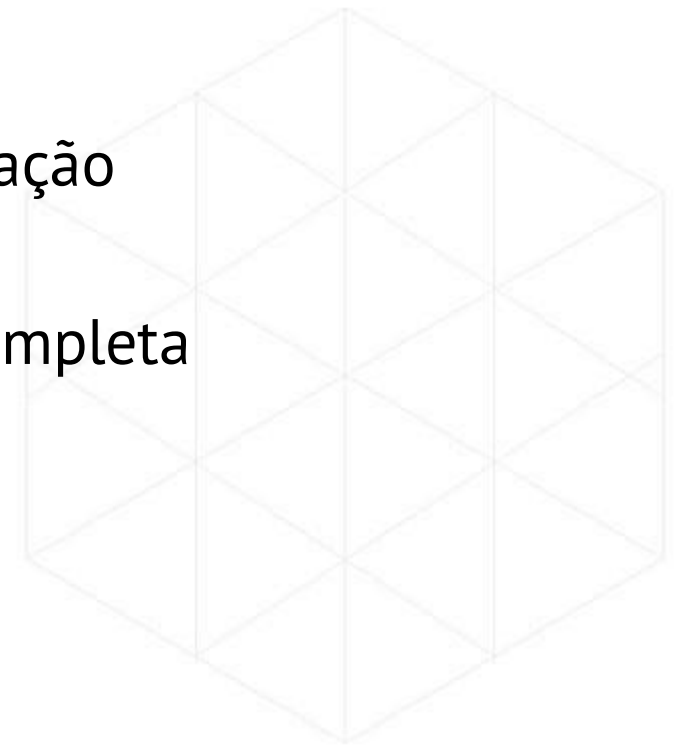
Contramedidas

- Lock semântico
 - Criação de *flags* dentro do registro indicando que há uma transação em andamento
 - Pode evitar que outras transações acessem o dado, se essa for a necessidade
 - Alguns campos, como STATUS podem ser perfeitos candidatos para locks semânticos






Tipos de componentes de uma saga

- Compensáveis
 - Podem, potencialmente ser canceladas com compensação
- Pivô
 - Dizem se a transação pode ou não ser considerada completa
 - Não pode ser cancelada nem com compensação
- Recuperáveis
 - Garantia de sucesso

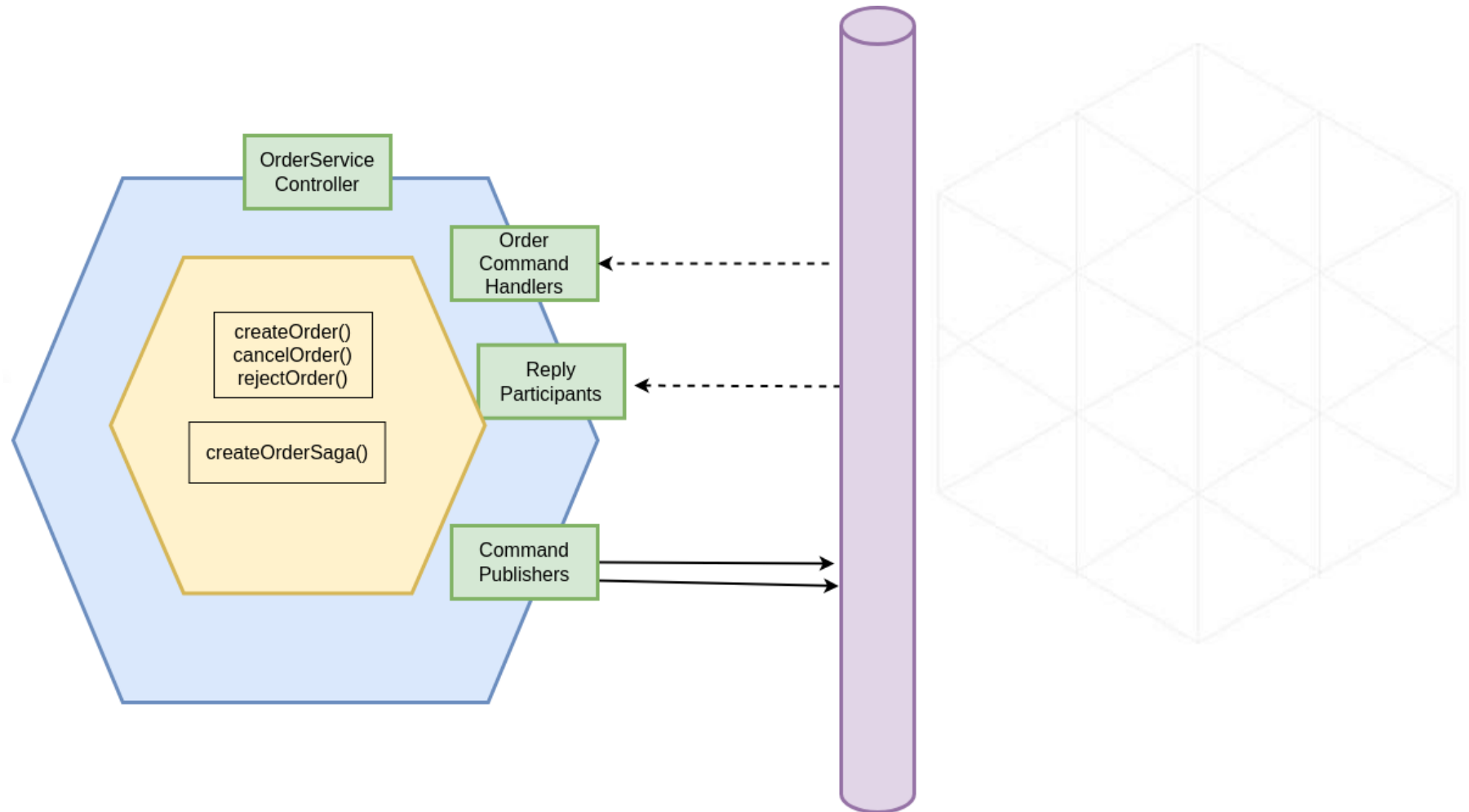


Tipos de sagas

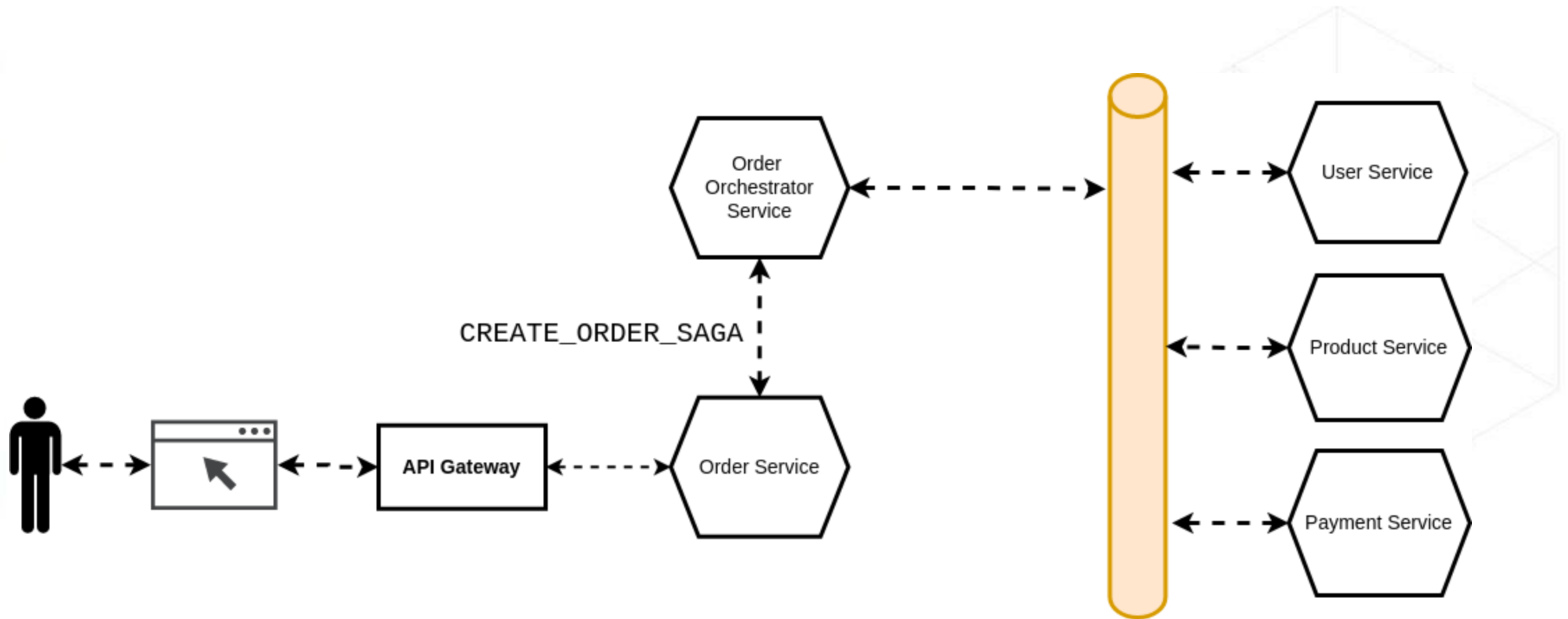
Passo	Serviço	Transação	Compensação
1	OrderService	Criar pedido vinculado ao usuário	-
2	User Service	Validar dados do usuário	CancelarPedido
3	ProductService	Verificar disponibilidade do produto	CancelarPedido
4	PaymentService	Processar pagamento	CancelarPedido; LiberarProdutos;
5	ProductService	Processar estoque	-
6	NotificationService	Enviar e-mail de pedido criado	-
7	OrderService	Atualiza status para APROVADO	-

 Compensáveis
 Pivô
 Recuperáveis

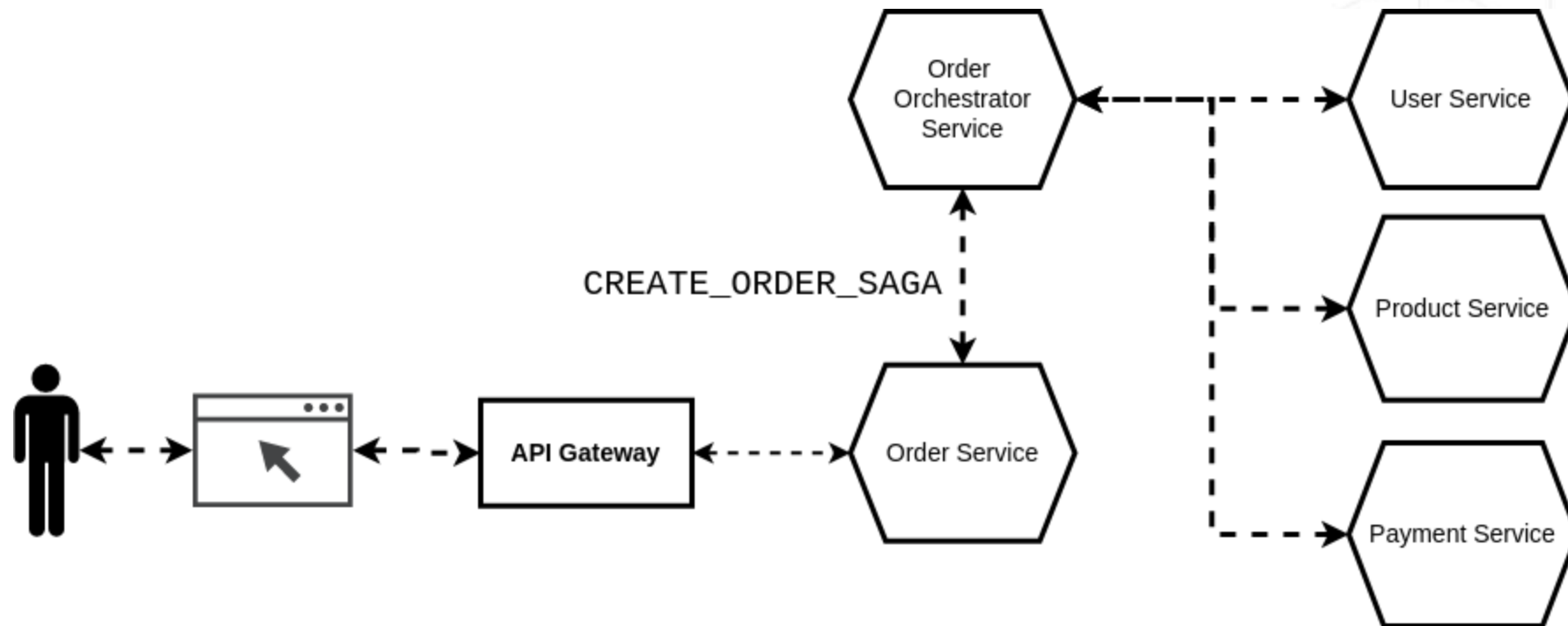
Um novo Serviço: Order Service Saga



Um novo Serviço: Order Service Saga



Um novo Serviço: Order Service Saga



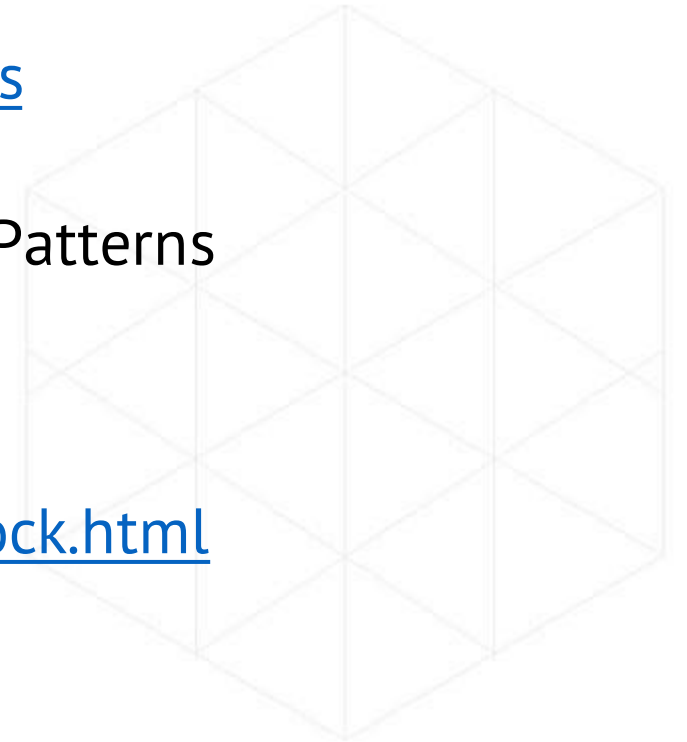
Order Service Saga

- Recebe Requests do Order Service.
- Responsável por criar e orquestrar a saga.
- Não detém nenhuma lógica sobre Order.
- Contém elementos para se relacionar com os outros membros do ecossistema.



Para saber mais

- <https://github.com/eventuate-tram/eventuate-tram-sagas>
 - Framework para orquestração de sagas
 - Descrito mais profundamente no livro Microservice Patterns
- <http://microservices.io/patterns/data/saga.html>
- <https://dl.acm.org/doi/10.1145/38714.38742>
- <https://martinfowler.com/eaCatalog/optimisticOfflineLock.html>
- <https://netflix.github.io/conductor/>
- <https://github.com/Netflix/conductor>



OBRIGADO!

Centro

Rua Formosa, 367 - 29º andar Centro, São Paulo - SP, 01049-000

Alphaville

Avenida Ipanema, 165 - Conj. 113/114 Alphaville, São Paulo - SP, 06472-002

+55 (11) 3358-7700

contact@7comm.com.br

7comm
Serviços e Soluções em TI