

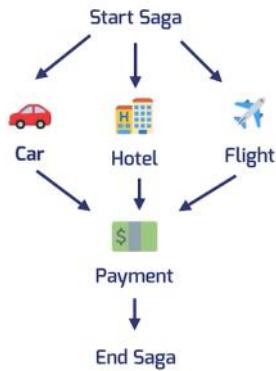
Distributed Saga - protocol for coordinating microservices

7 января 2021 г. 19:08

Distributed Saga - protocol for coordinating microservices

элементы саги могут выполняться паралельно

*Distributed Saga
Directed Acyclic Graph*



сага состоит из транзакций и их компенсаций

A Distributed Saga is a Collection of Requests

Book Hotel Book Car Book Flight Charge Money

and Compensating Requests

Cancel Hotel Cancel Car Cancel Flight Refund Money

*TE
Fringe*
that represent a single business level action

сага либо вся заканчивается успешно либо вся откатывается

Distributed Saga Guarantee

All requests were completed successfully



Or a subset of requests and the corresponding compensating requests were executed



Distributed Saga Guarantee

No Atomicity

Visible before
Saga Completes

No Isolation



компенсационные запросы нельзя отменить

Requests	Compensating Requests
Idempotent	Idempotent
Can Abort	Commutative Can Not Abort

сами компенсационные запросы должны быть идемпотентными

Compensating Requests

Compensating Requests

Must Be Idempotent

компенсационные запросы должны быть коммутативными с запросами
транзакция + компенсация == компенсация + транзакция

Compensating Requests

Must Be Commutative with Requests



is the same as



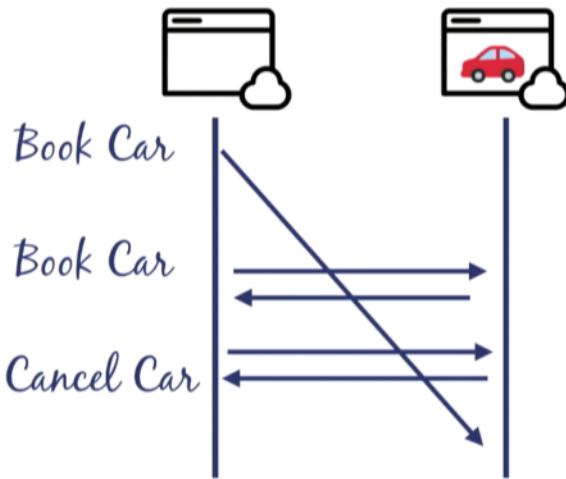
Identity Empowers Confusion

- Writes may arrive at the “wrong” place
 - Durable at some replica not in the original plan
 - Must eventually show them home to the “right” place
 - The “right” place is a fuzzy concept
- Writes may arrive in the wrong order
 - Issue log writes to buffers 1, 2, 3, 4, 5, 6
 - May arrive at a replica as 4, 6, 2, 3, 5, 1
 - May arrive in different orders at different replicas
- Intended order must be assigned by the DBMS
 - The identity of the buffer must be intrinsic to its identity
 - Must be *reorderable* by each separate replica to intended order

Assigning the Order at the DBMS or Client Allows Durable Writes at Any Replica While Preserving Order

Tolerance of *Where* You Write
Tightens the SLA for
When You’re Durable!

из-за идемпотентности мы имеем два запроса book car
если вдруг первый запрос потерялся (а затем мы уже успели еще раз арендовать и отменить) то
после того как запрос все-таки придет, он не должен иметь никакого влияния

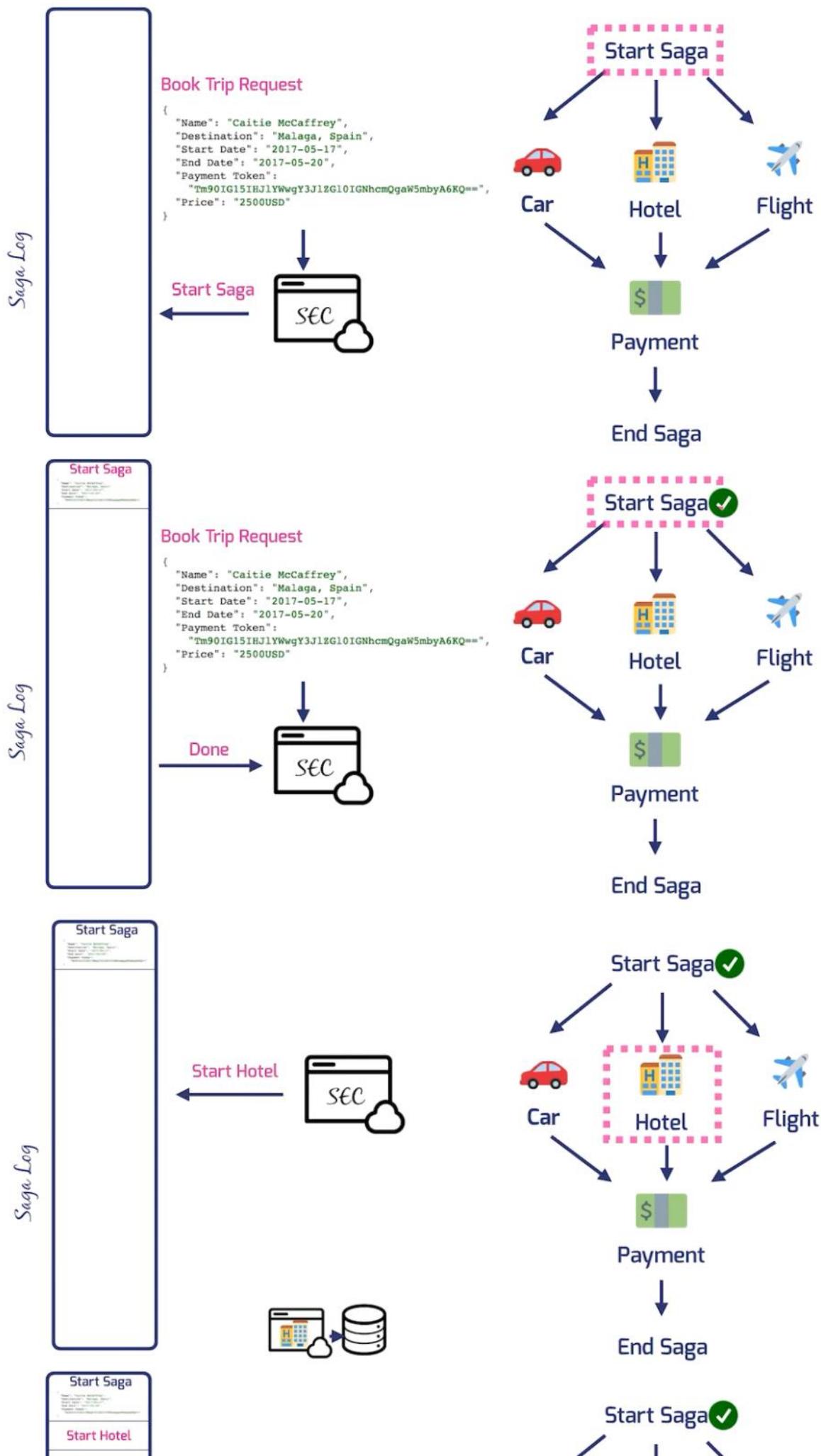


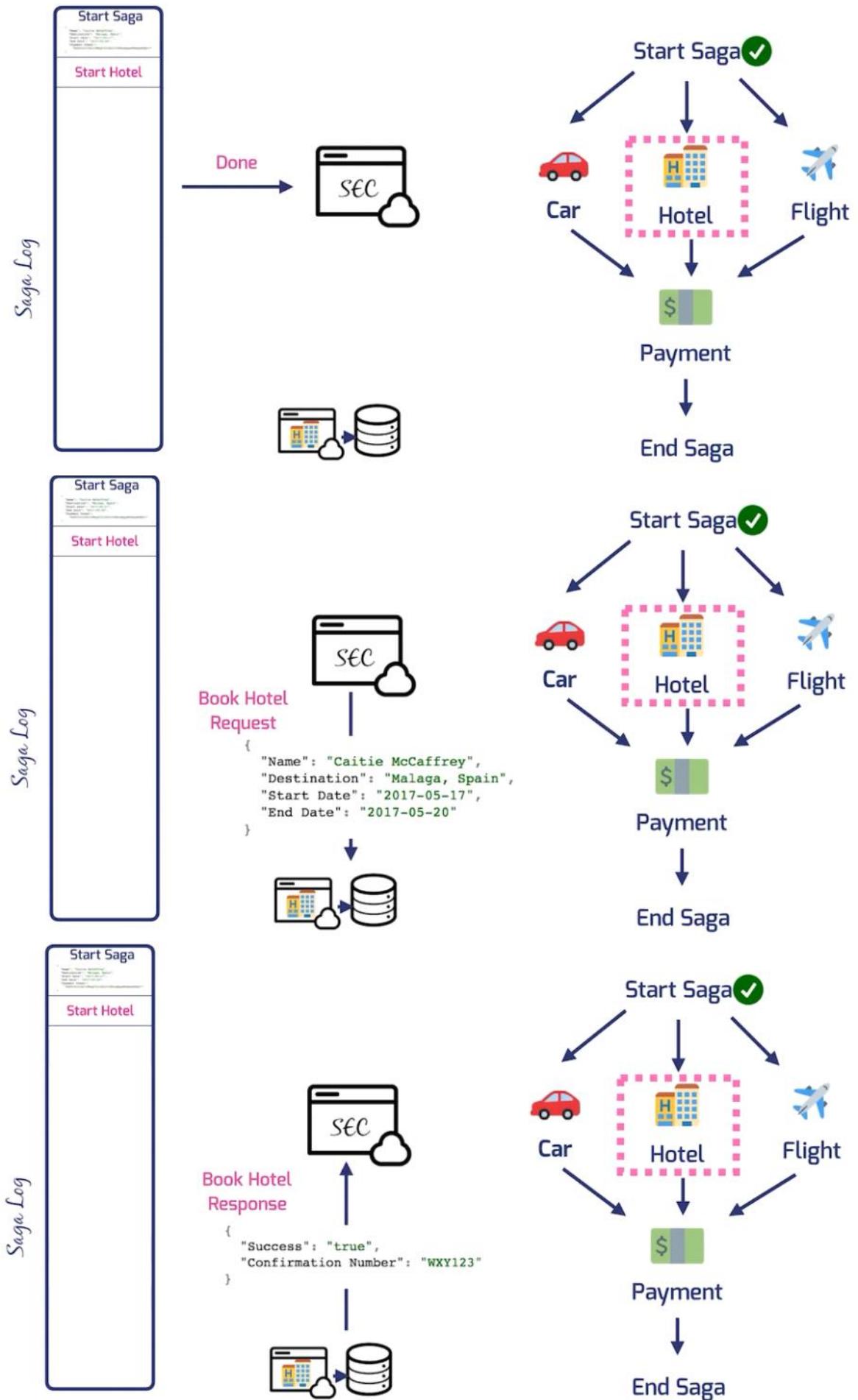
в кафке можно просто послать всем сообщение об отмене
благодаря принципу сохранения порядка сообщений в партиции: к каждому
микросервису сначала гарантировано EoS придет сообщение о бронировании, а затем
вслед за ним сообщение об отмене

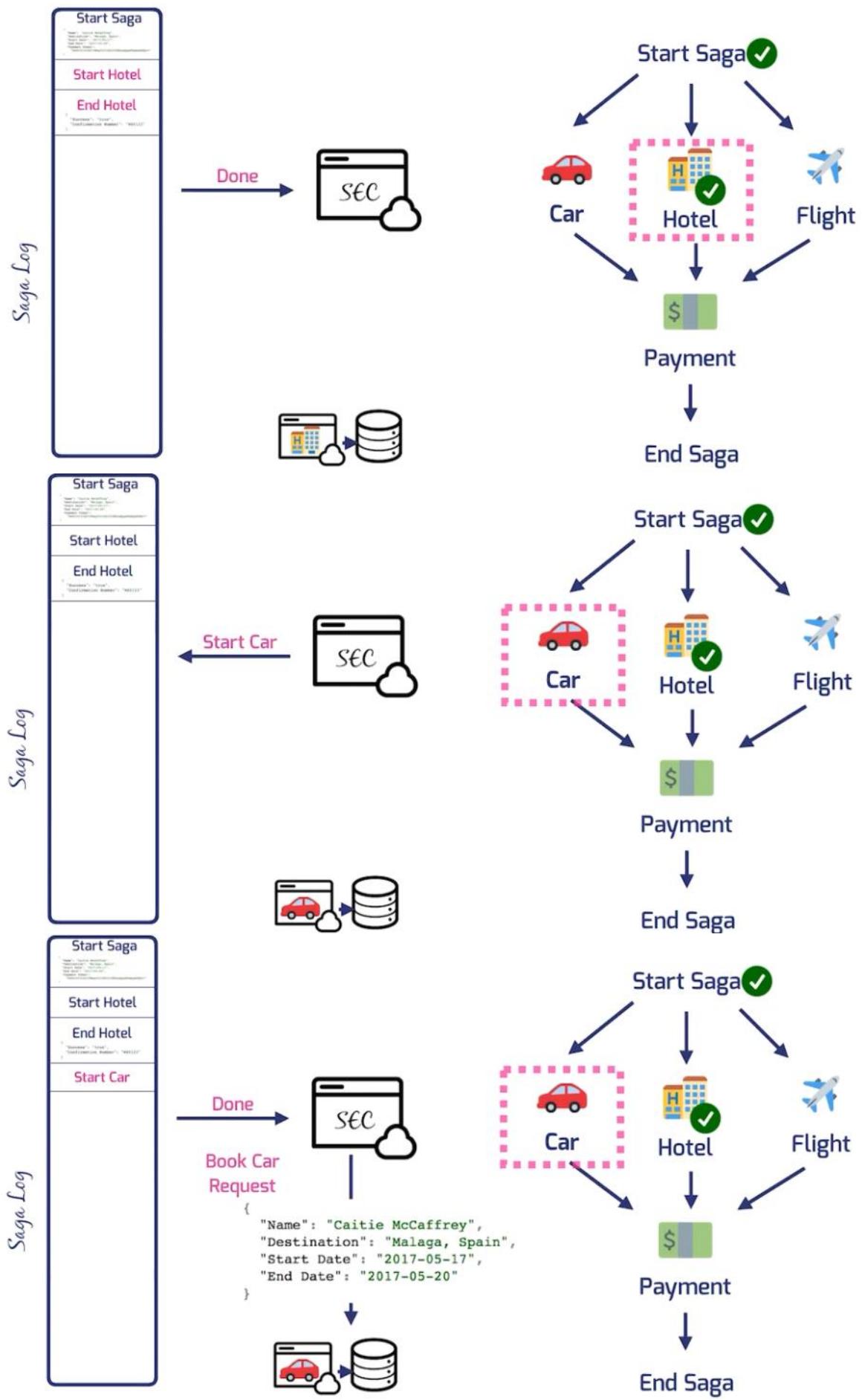
- тот микросервис, который упал и не смог провести транзакцию, по идее на сообщение об отмене ничего не должен делать (ведь транзакцию то он так и не провел)

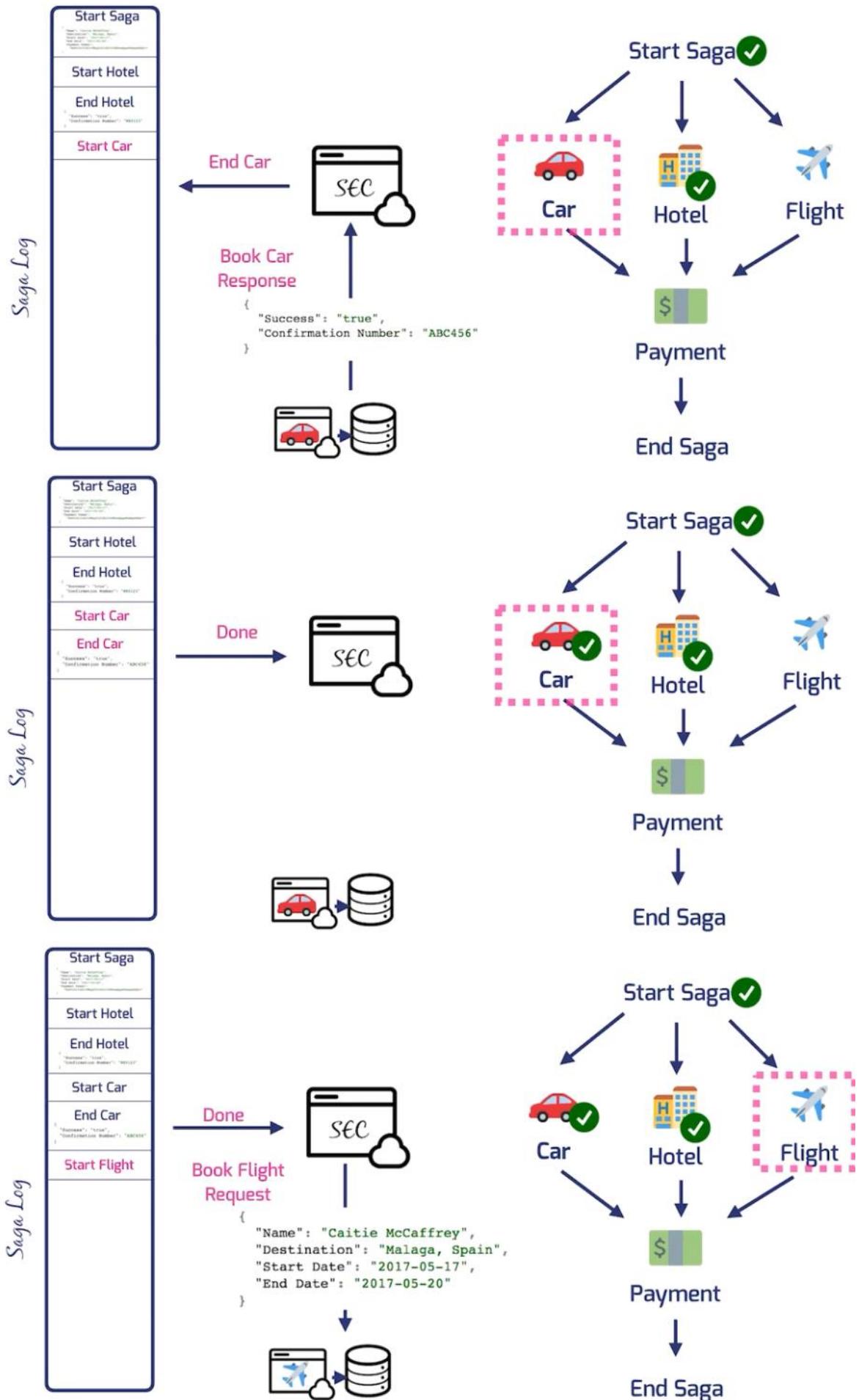
запрос в саге должен быть идемпотентным

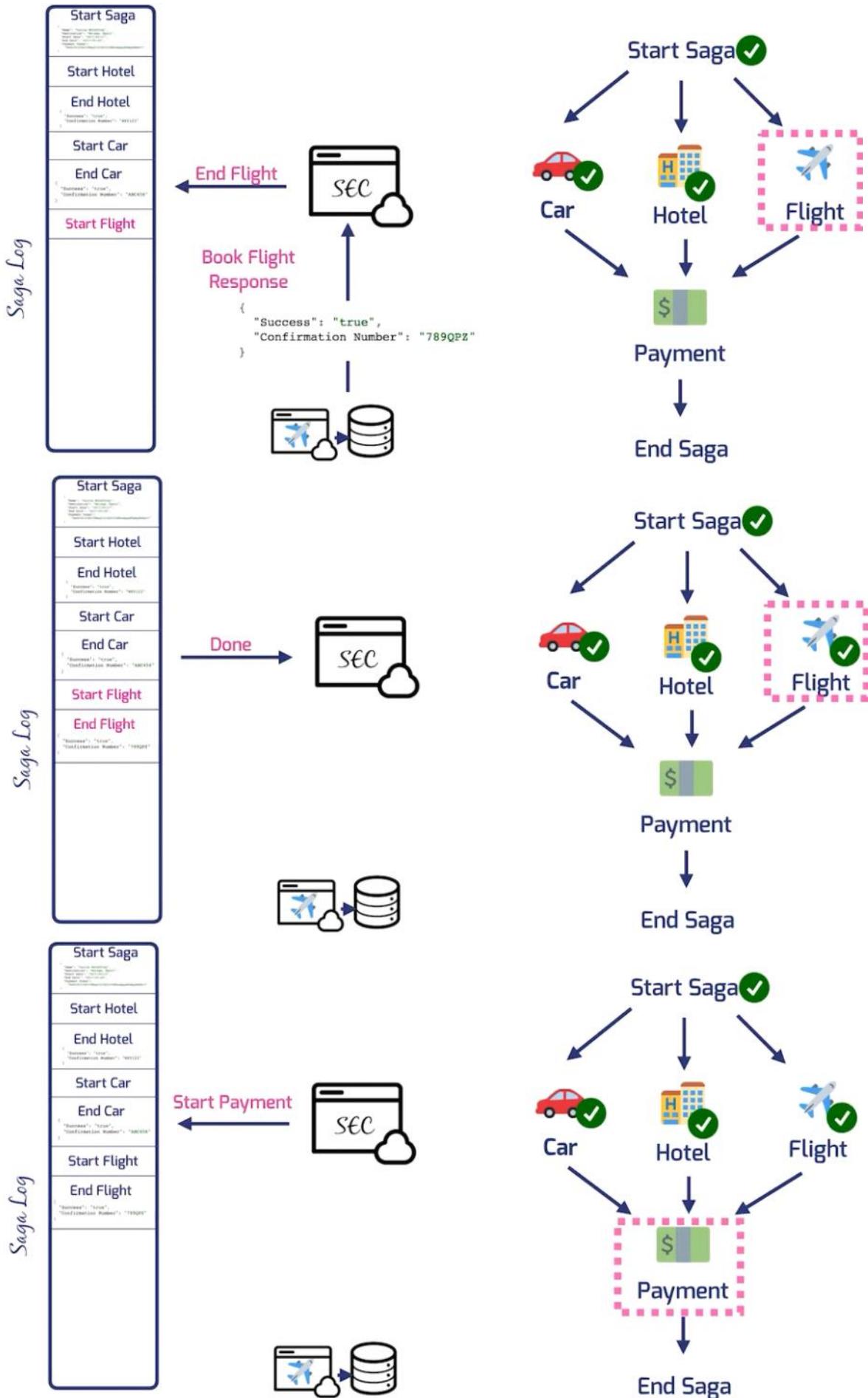


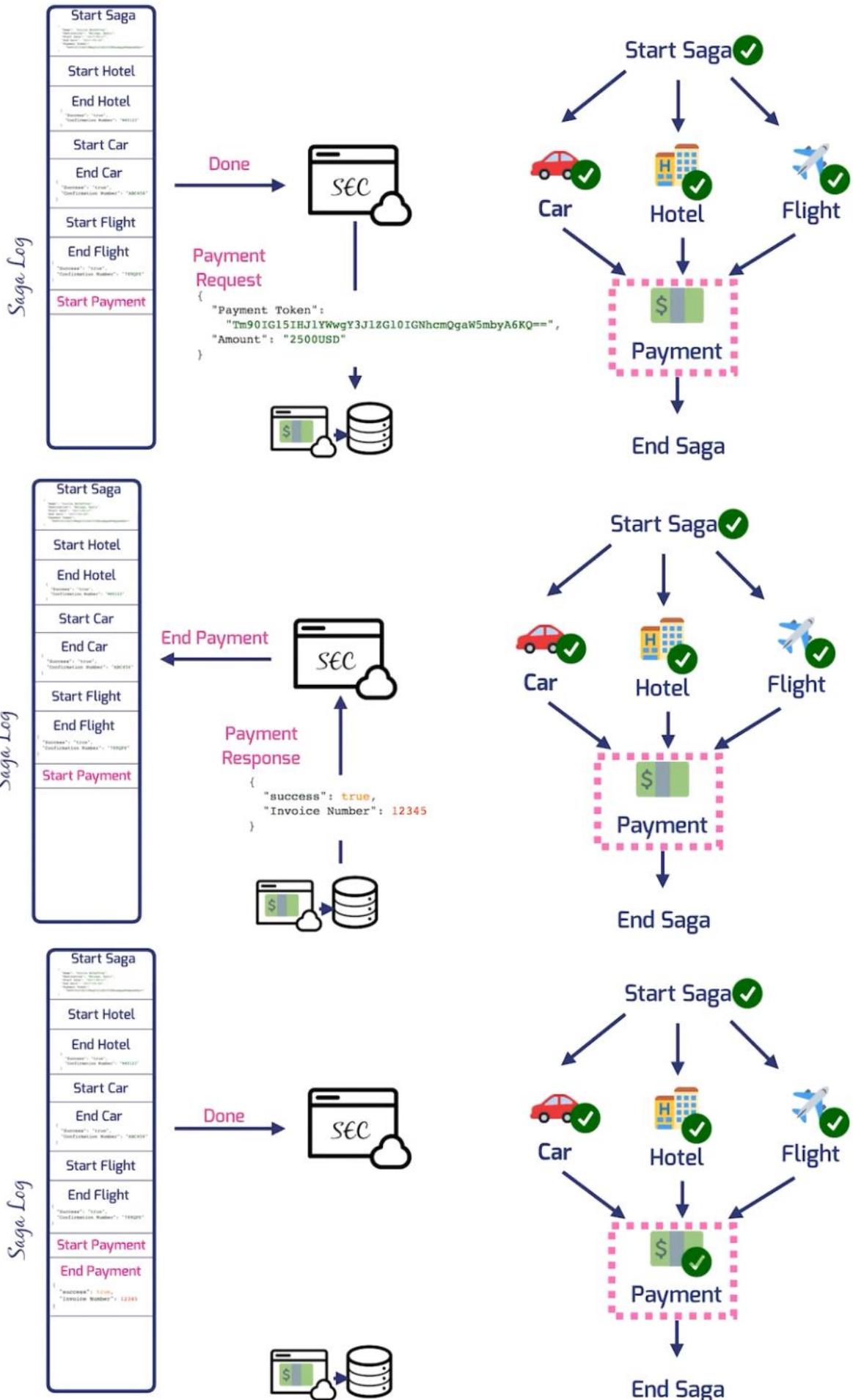


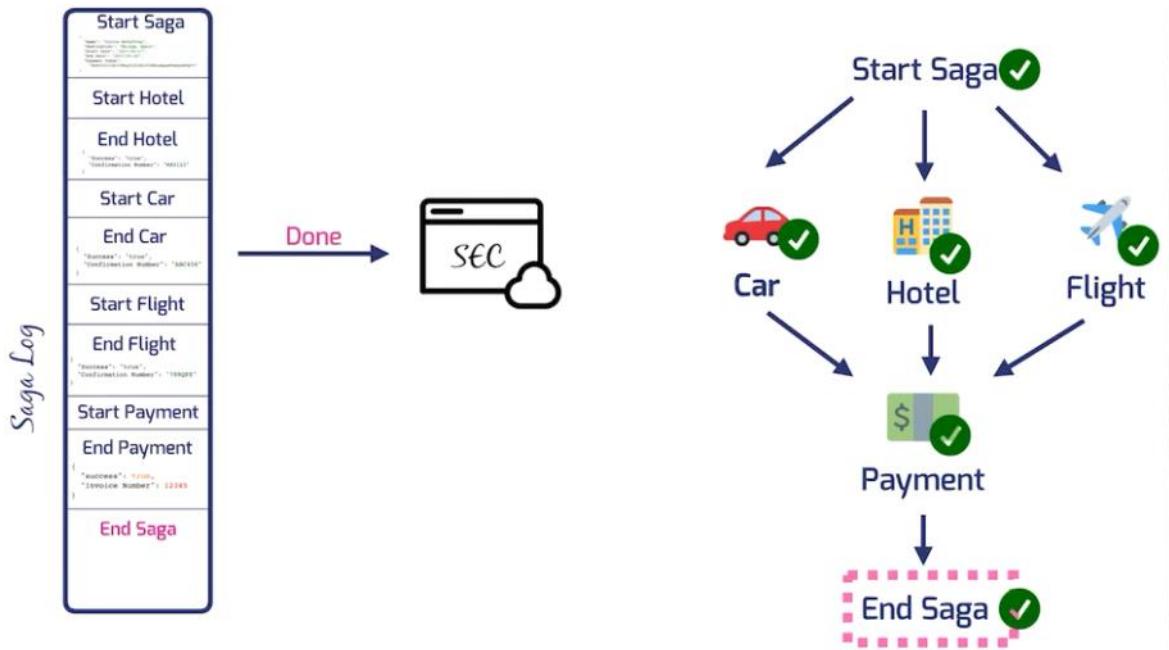




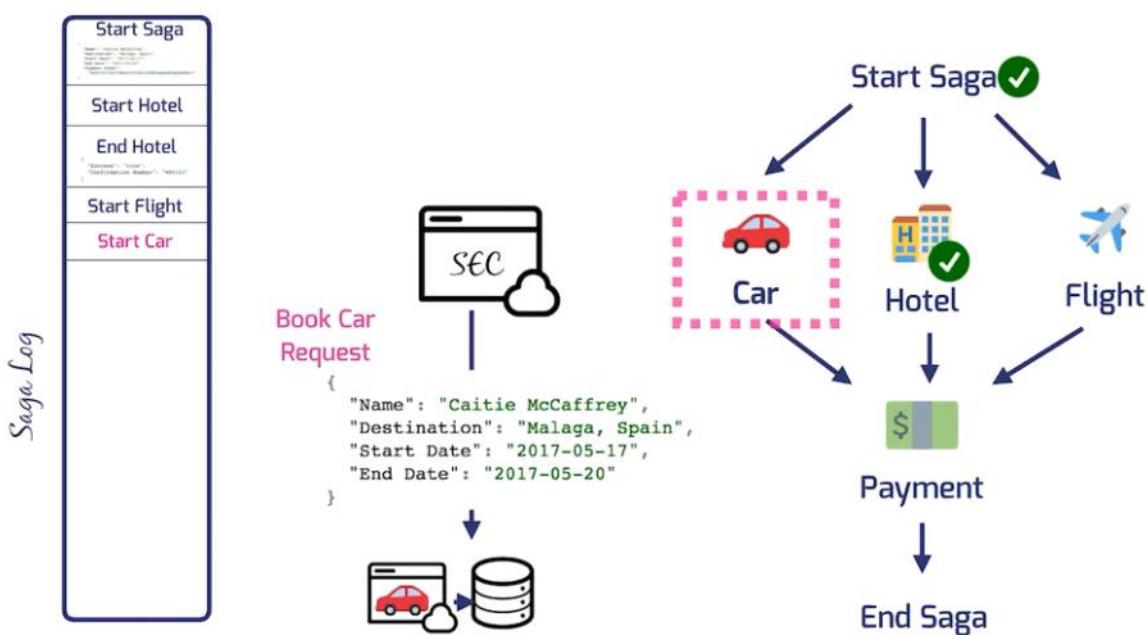


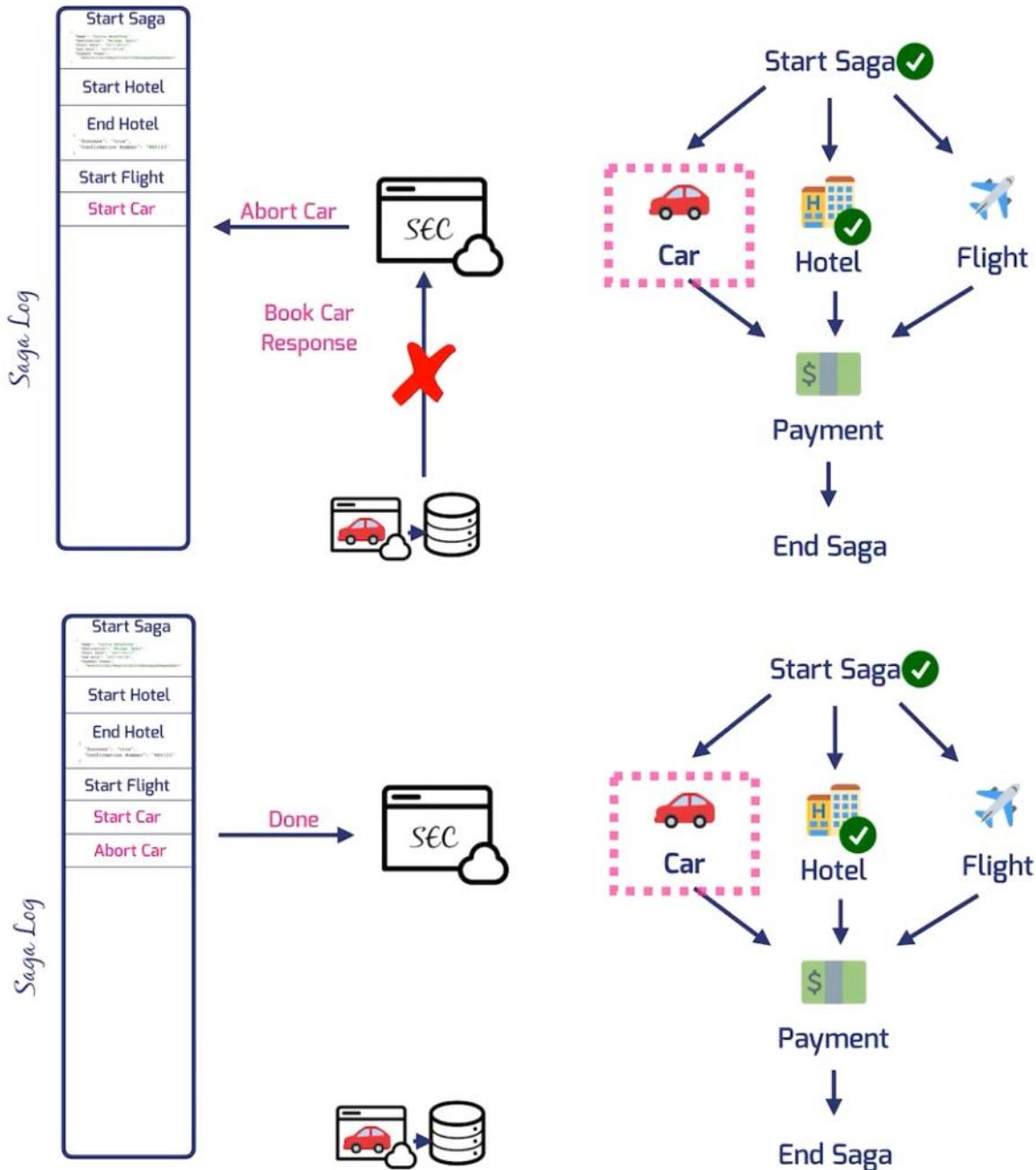




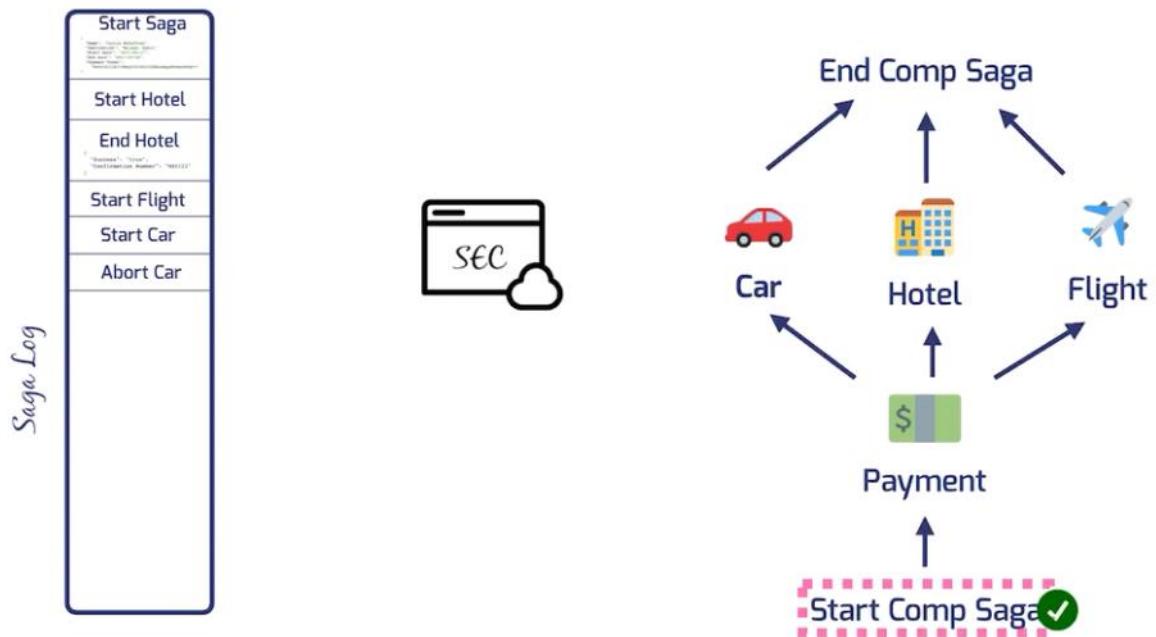


пример компенсационной саги

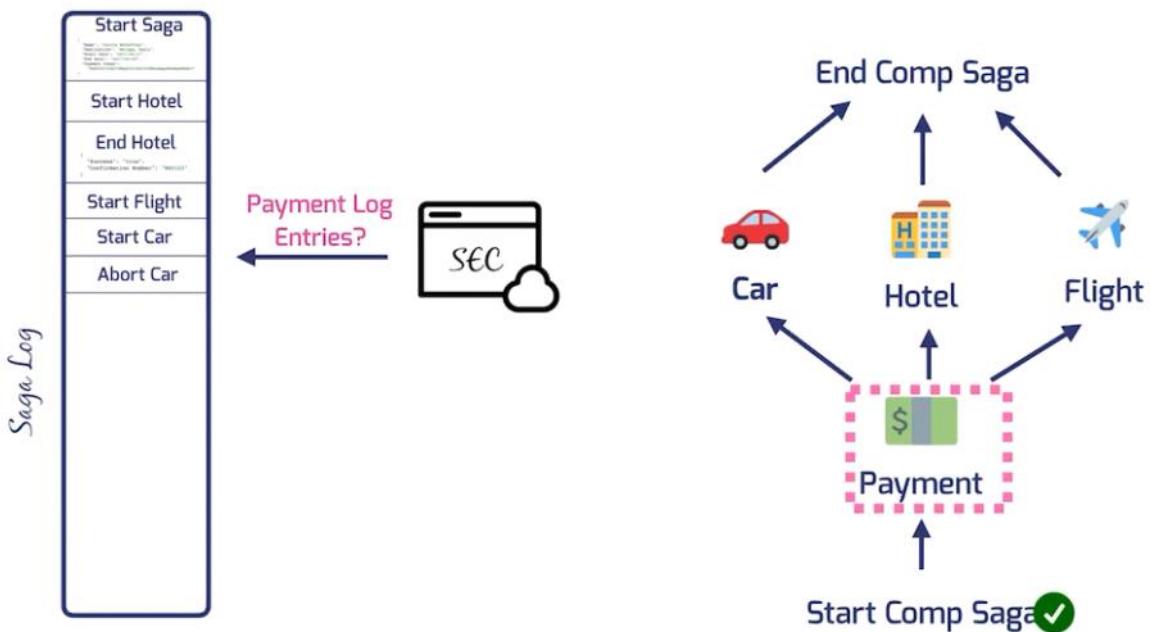


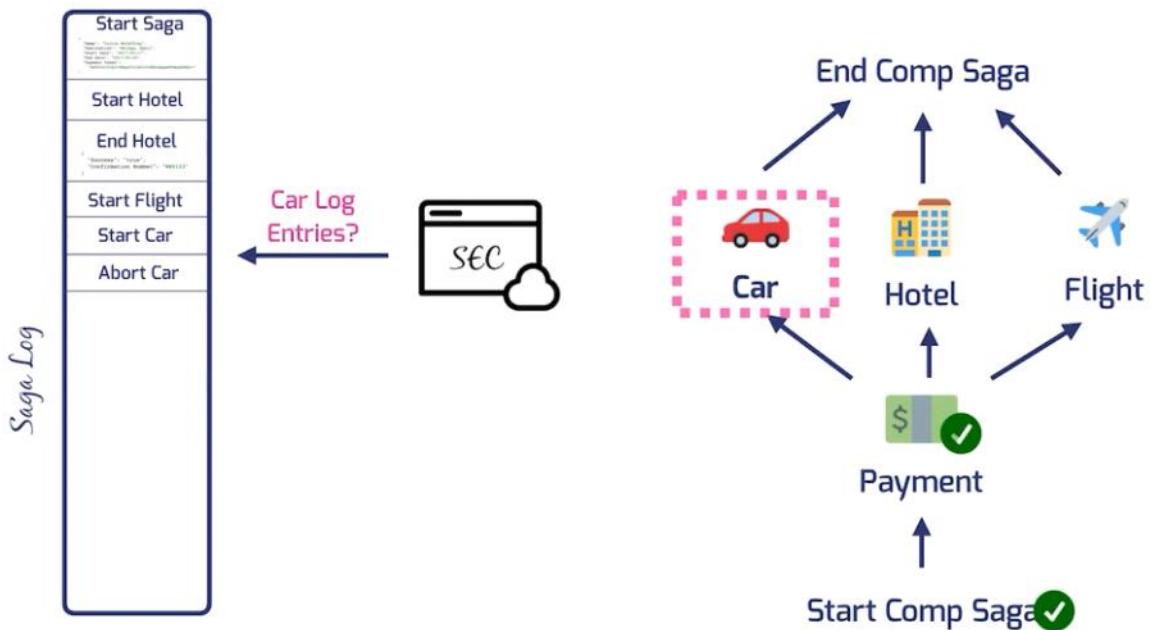


если ошибка то запускается отдельная компенсационная сага

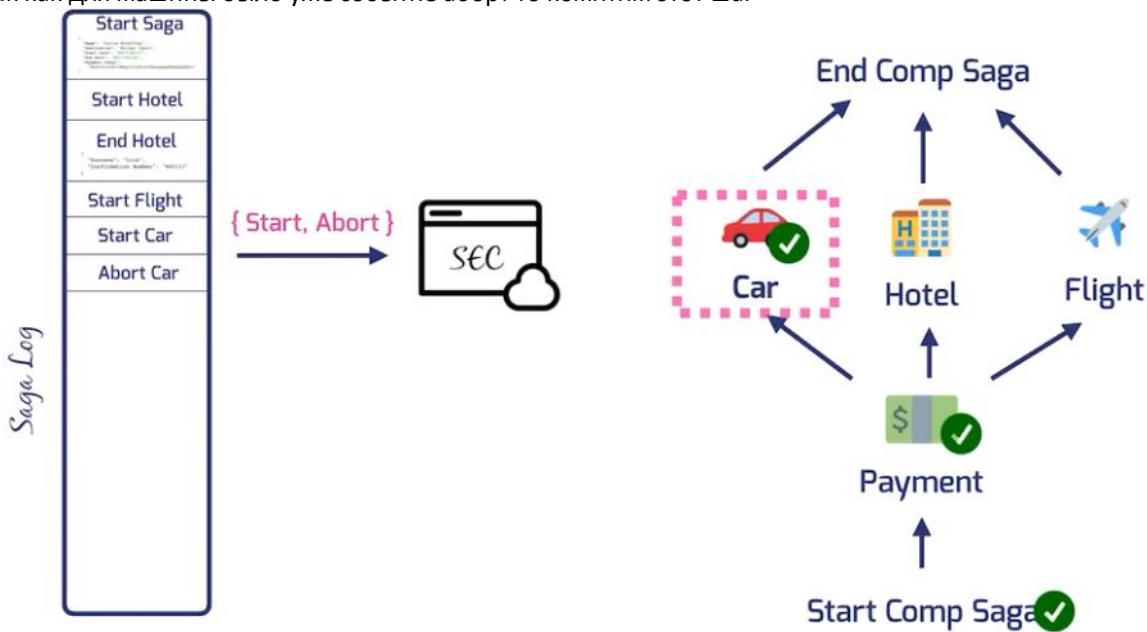


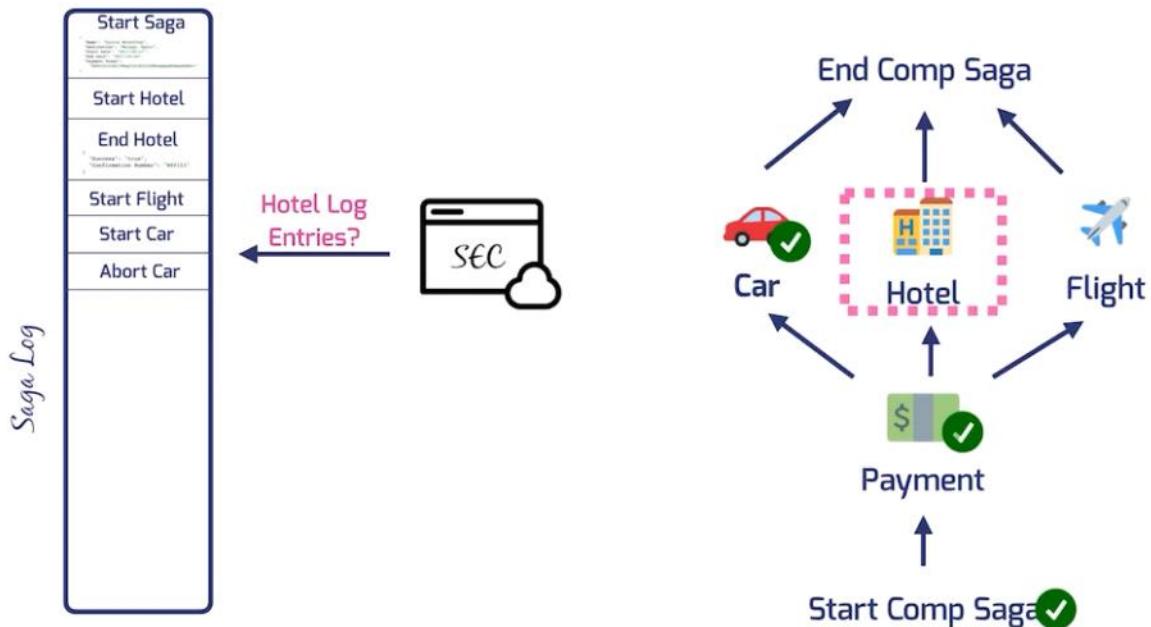
так как payment не был проведен то и нечего компенсировать (то комитим пеймент шаг)



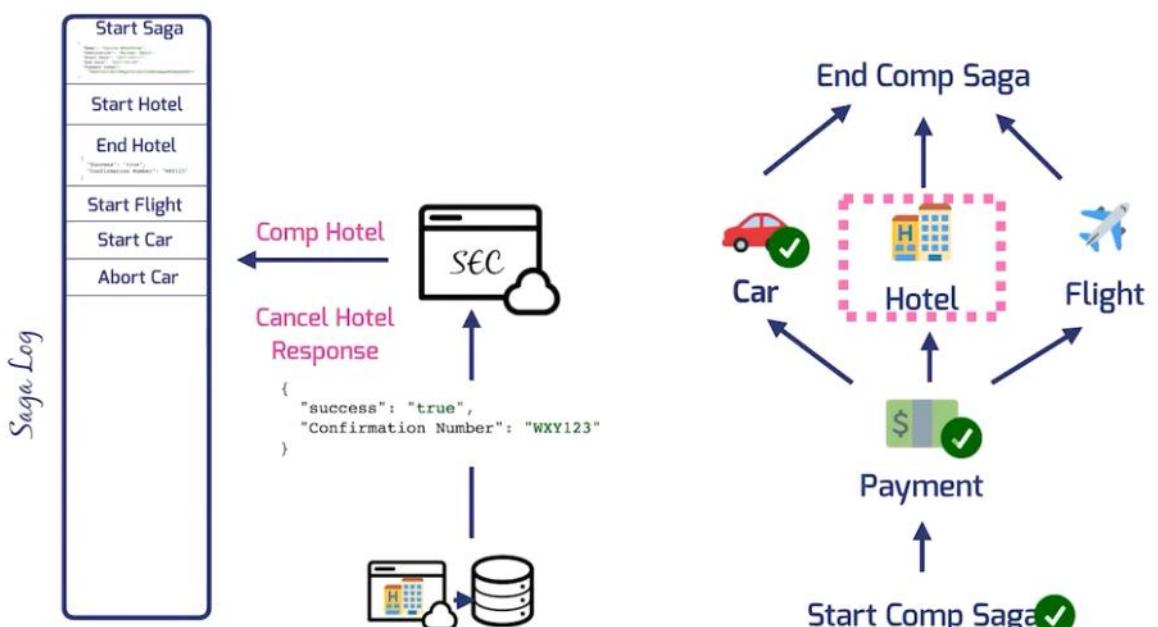
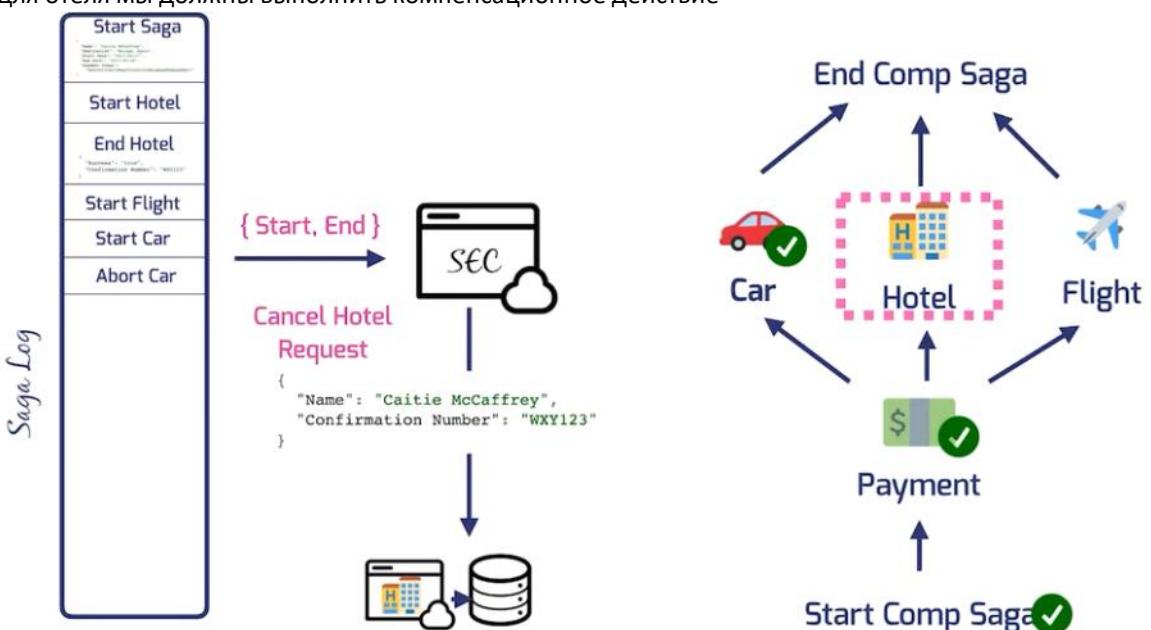


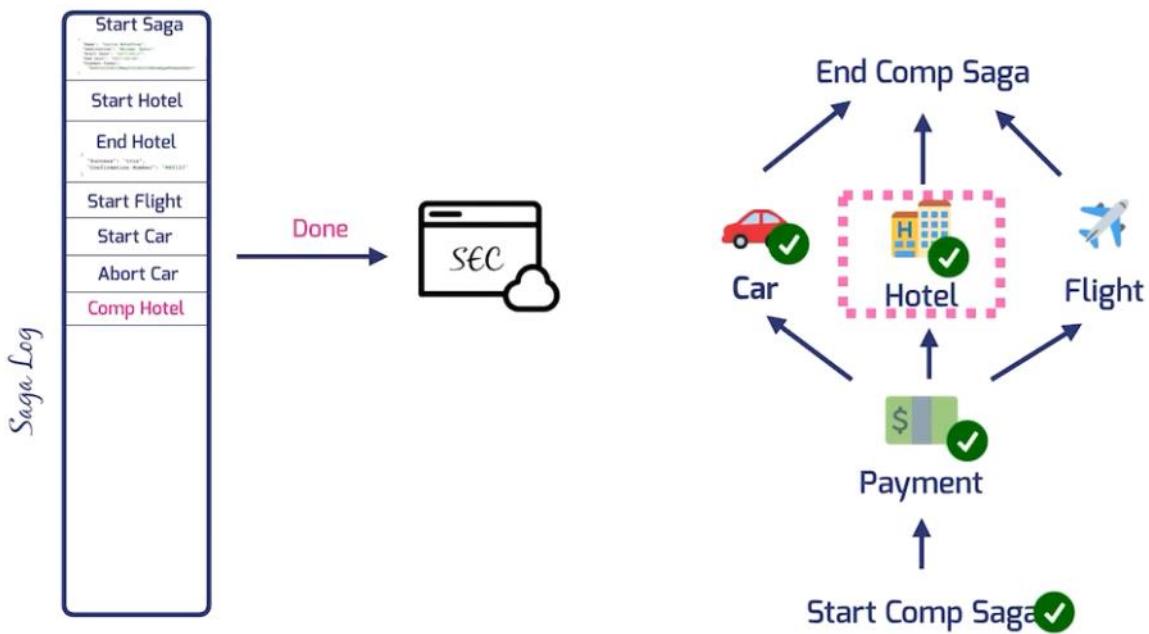
так как для машины было уже событие.abort то комитим этот шаг



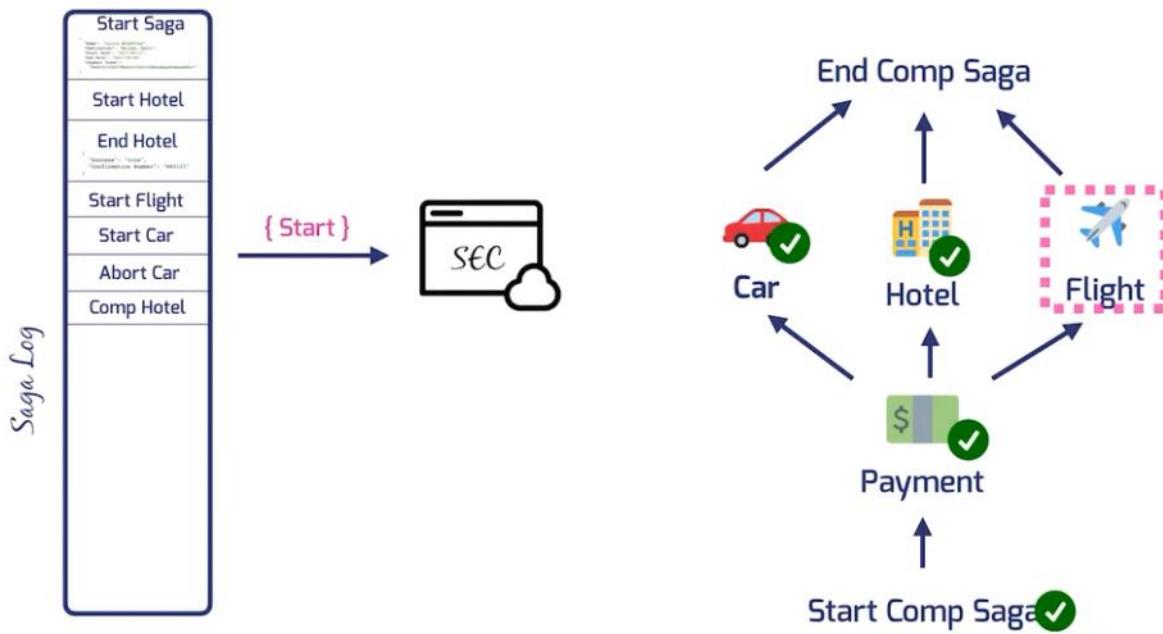


для отеля мы должны выполнить компенсационное действие

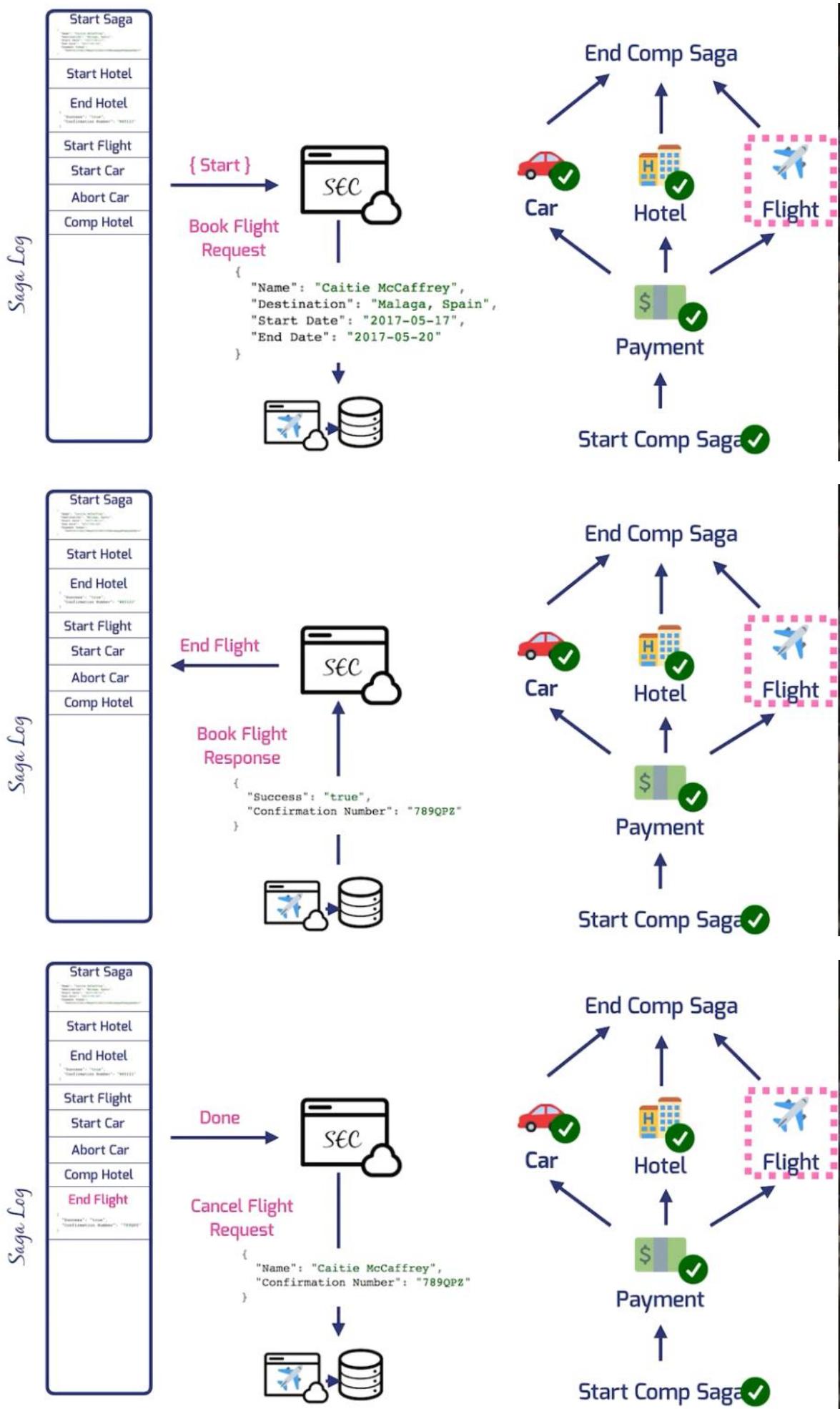


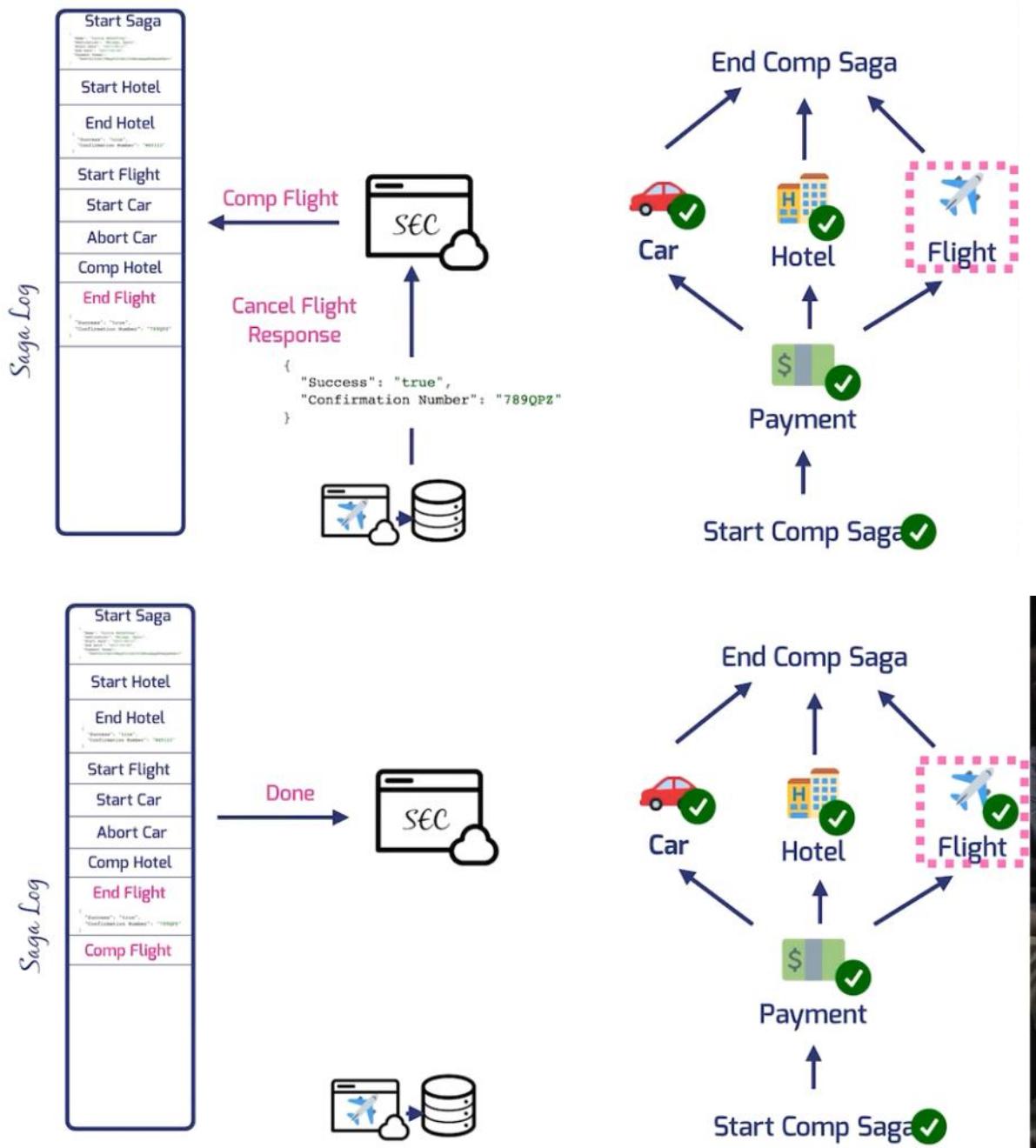


видим что билет на самолет так и не был куплен

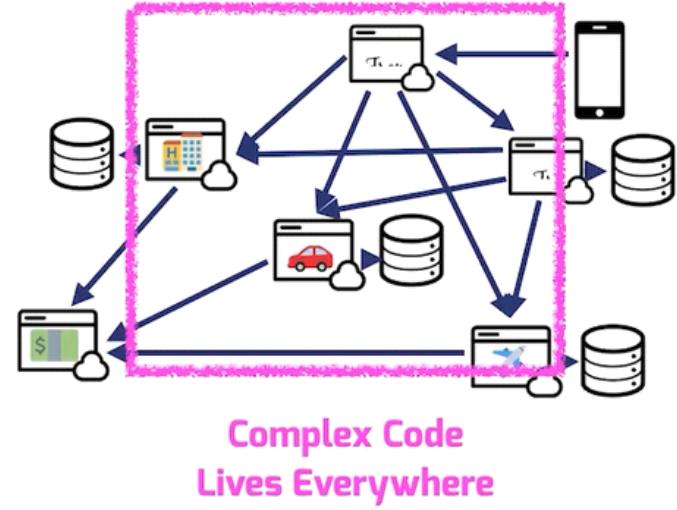
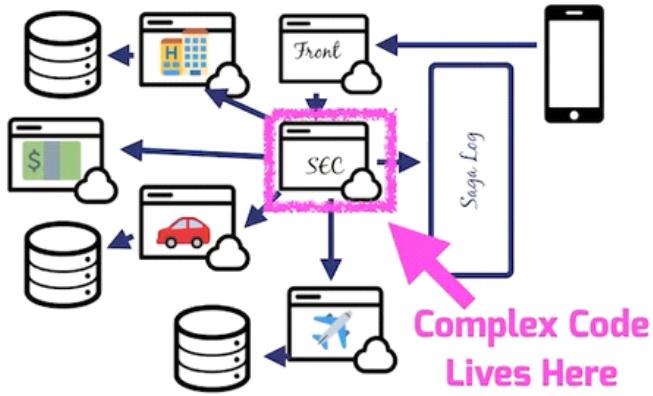


заново пытаемся купить билет на самолет

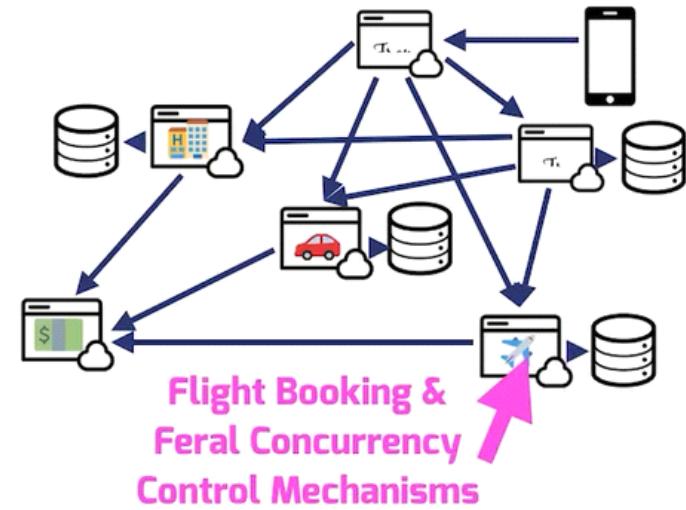




как запрограммировать saga execution coordinator



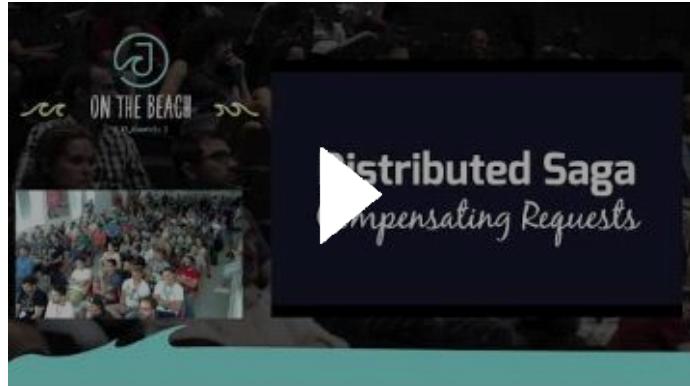
в каждом микросервисе нужно сделать механизм который запустит компенсирующую сагу



sagas (with parallel processes)

14 декабря 2020 г. 12:27

[Distributed Sagas: A Protocol for Coordinating Microservices - Caitie McCaffrey - JOTB17](#)



onenote:///C:/Users/trans/Qsync/vova_from_onenote\tf_agonote_v1\SYSTEMDESIGN\microservices.io.one#sagas§ion-id={AAA08E86-29AC-41DC-957C-60E279549374}&page-id={52157417-F1F7-4AA8-99B9-936253E5ED38}&end

A saga is a sequence of local transactions that are coordinated using messaging.

последовательность локальных транзакций на основе сообщений

- Это последовательность локальных транзакций, каждая из которых обновляет данные в одном сервисе, задействуя знакомые фреймворки и библиотеки для ACID-транзакций, упомянутые ранее
- Это позволяет приложению поддерживать согласованность данных в нескольких сервисах без использования распределенных транзакций.
- Повествование — это последовательность локальных транзакций, которые координируются с помощью сообщений. Каждая локальная транзакция обновляет данные лишь в одном сервисе. При этом все изменения фиксируются, поэтому, если повествование нужно откатить из-за нарушения бизнес-правила, оно должно выполнить компенсирующие транзакции, чтобы явно отменить внесенные изменения.
- <https://microservices.io/patterns/data/saga.html>

Реализация каждой бизнес-операции, охватывающей несколько сервисов, - это сага

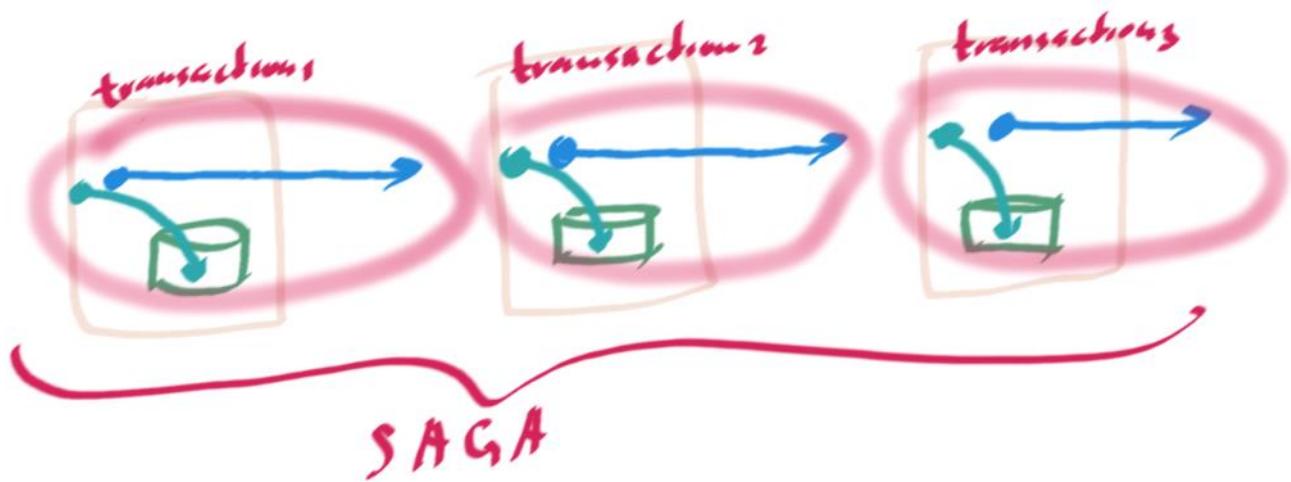
- Реализация каждой бизнес-операции, охватывающей несколько сервисов, - это сага. Сага - это последовательность локальных транзакций. Каждая локальная транзакция обновляет базу данных и публикует сообщение или событие для запуска следующей локальной транзакции в саге. Если локальная транзакция терпит неудачу из-за нарушения бизнес-правила, то сага выполняет серию компенсирующих транзакций, которые отменяют изменения, сделанные предыдущими локальными транзакциями.

разработчик должен разработать компенсирующие транзакции, которые явно отменяют изменения, сделанные ранее в саге

- Модель программирования более сложная. Например, разработчик должен разработать компенсирующие транзакции, которые явно отменяют изменения, сделанные ранее в саге.
- , поскольку каждая локальная транзакция фиксирует свои изменения, для отката повествования необходимо использовать компенсирующие транзакции

микросервис должен атомарно(те в одной локальной транзакции) обновлять свою базу данных и публиковать event/сообщение

- Чтобы быть надежным, служба должна атомарно обновлять свою базу данных и публиковать сообщение / событие. Он не может использовать традиционный механизм распределенной транзакции, охватывающий базу данных и брокер сообщений



Завершение одной локальной транзакции приводит к выполнению следующей

- Системная операция инициирует первый этап повествования. Завершение одной локальной транзакции приводит к выполнению следующей. В разделе 4.2 вы увидите, как координация этих этапов реализуется с помощью асинхронных сообщений.
- Сервис публикует сообщение по завершении локальной транзакции. Это инициирует следующий этап повествования и позволяет не только добиться слабой связанности участников, но и гарантировать полное выполнение повествования.

гарантирует выполнение всех этапов повествования, даже если один или несколько участников оказываются недоступными

- Важным преимуществом асинхронного обмена сообщениями является то, что он гарантирует выполнение всех этапов повествования, даже если один или несколько участников оказываются недоступными.
- Даже если получатель временно недоступен, брокер буферизирует сообщение до того момента, когда его можно будет доставить.

реализация саги разбросана между сервисами. Из-за этого разработчикам иногда трудно понять, как работает то или иное повествование.

- Они сложнее для понимания. В отличие от оркестрации хореография не описывает повествование на каком-то одном участке кода — его реализация разбросана между сервисами. Из-за этого разработчикам иногда трудно понять, как работает то или иное повествование.

вся координирующая логика находится внутри микросервисов.

- это значительно усложняет бизнес-логику
- те весь бизнес процесс не в едином месте а разбит на микросервисы (каждый микросервис реализует часть бизнес-процесса и в то же время должен представлять его целиком)

Существует риск жесткого связывания

- Каждый участник повествования должен подписаться на все события, которые на него влияют. Например, сервис Accounting интересуют все события, приводящие к выставлению счета или возмещению средств на банковской карте. В итоге возникает риск того, что ему придется обновляться синхронно с жизненным циклом заказа, который реализован сервисом Order

Хореография может хорошо работать с простыми повествованиями, но, учитывая ее недостатки, в более сложных случаях лучше использовать оркестрацию.

Я рекомендую использовать оркестрацию для любых повествований, за исключением самых простых.

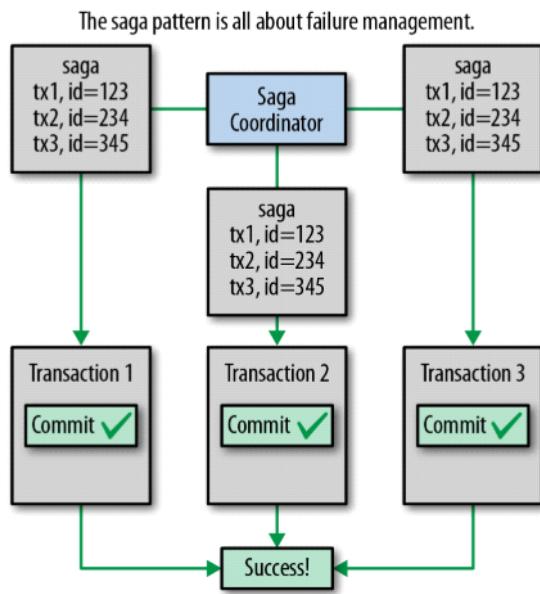
saga - Суть идеи состоит в том, что одну длительную распределенную транзакцию можно рассматривать как композицию нескольких быстрых локальных транзакционных шагов

- Суть идеи состоит в том, что одну длительную распределенную транзакцию можно рассматривать как композицию нескольких быстрых локальных транзакционных шагов.
- Каждый шаг транзакции сопряжен с компенсирующим реверсивным действием (реверсирование с точки зрения бизнес-семантики, без обязательного сброса состояния компонента), так что вся распределенная транзакция может быть отменена в случае сбоя путем выполнения компенсирующего действия каждого шага.
- В идеале эти шаги должны быть коммутируемыми, чтобы их можно было выполнять параллельно.
(коммутативный закон сложения: $m + n = n + m$)
 - Saga - отличный инструмент для обеспечения атомарности длительных транзакций.
 - Но важно понимать, что это не решение для изоляции .
 - Одновременно выполняемые саги потенциально могут влиять друг на друга и вызывать ошибки.
 - Если это приемлемо, это зависит от варианта использования.
 - Если это неприемлемо, вам нужно использовать другую стратегию, например, убедиться, что сага не охватывает несколько границ согласованности, или просто использовать другой шаблон или инструмент для работы.
- The essence of the idea is that one long-running distributed transaction can be seen as the composition of multiple quick local transactional steps. Every transactional step is paired with a compensating reversing action (reversing in terms of business semantics, not necessarily resetting the state of the component), so that the entire distributed transaction can be reversed upon failure by running each step's compensating action. Ideally, these steps should be commutative so that they can be run in parallel.
- Одним из преимуществ этого метода (см. Рис. 6-3) является то, что он **eventually consistent** в конечном

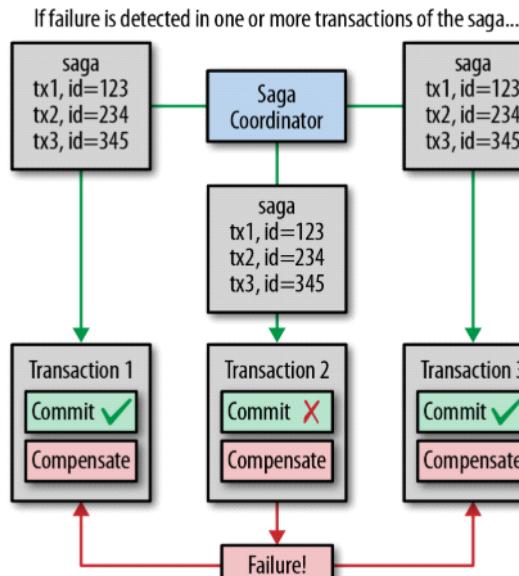
итоге согласован и хорошо работает с разделенными и асинхронно взаимодействующими компонентами, что делает его отличным подходом для архитектур, управляемых событиями и сообщениями.

saga pattern - is all about failure management

- long-running distributed workflows across multiple services



A saga knows all of the steps involved in a long running transaction.



...all transactions can be undone using compensating actions.

в микросервисной архитектуре транзакции охватывают несколько сервисов, каждый из которых имеет свою БД.

- В таких условиях приложение должно использовать более продуманный механизм работы с транзакциями.

The `createOrder()` operation reads from Consumer Service and updates data in Order Service, Kitchen Service, and Accounting Service.

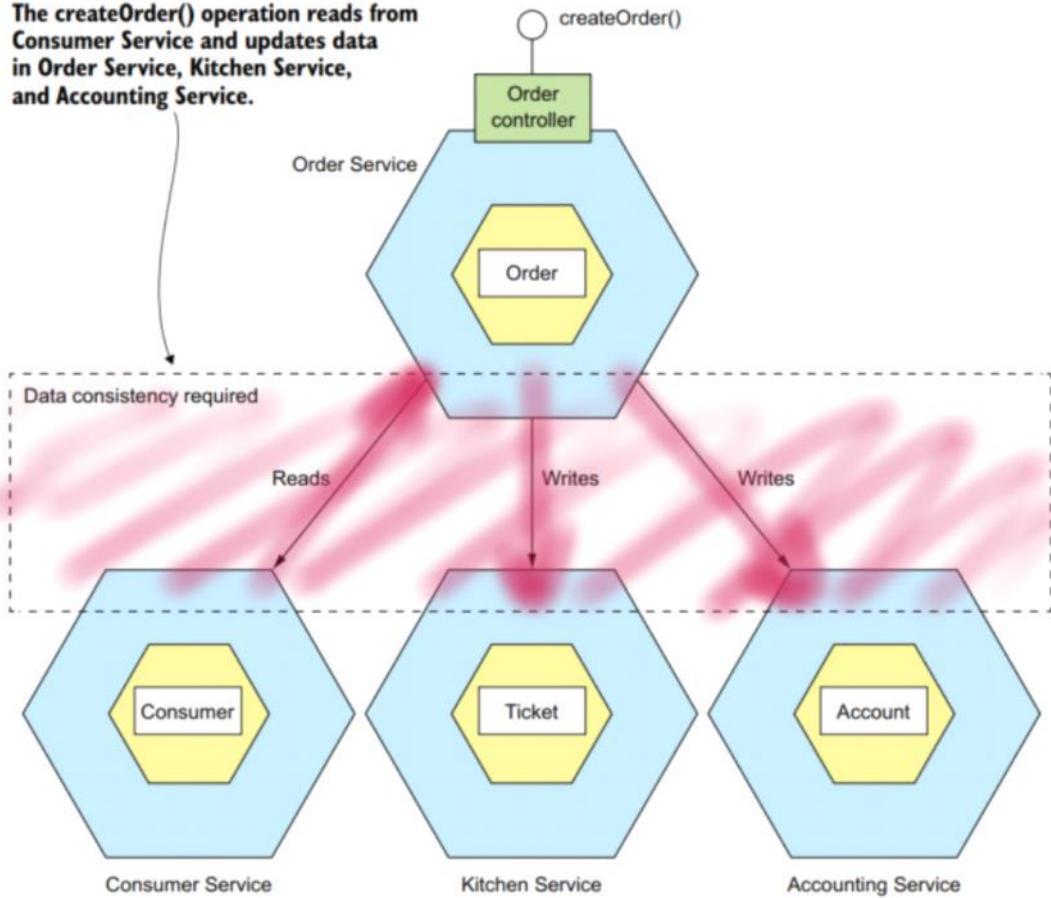


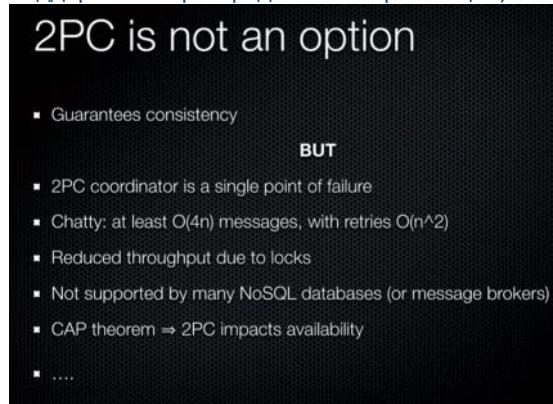
Figure 4.1 The `createOrder()` operation updates data in several services. It must use a mechanism to maintain data consistency across those services.

sagas (общие заметки)

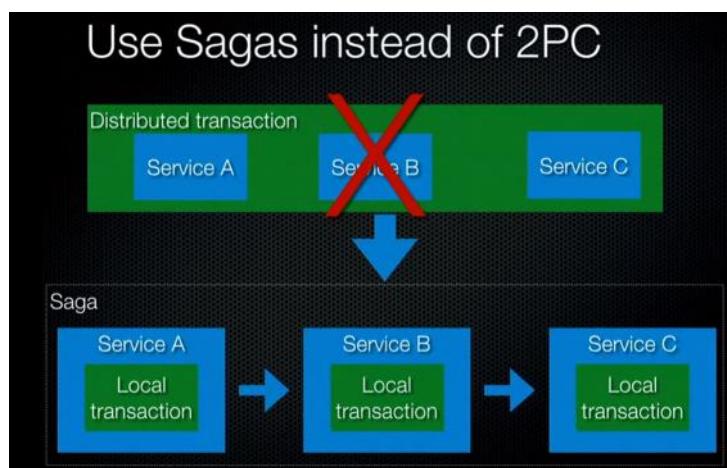
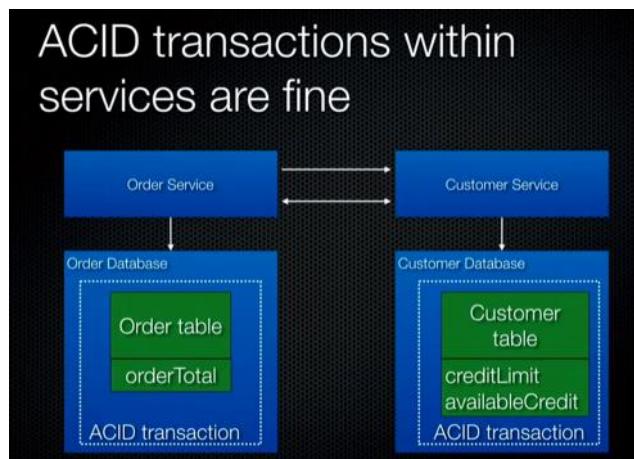
13 октября 2020 г. 14:27

<https://vasters.com/archive/Sagas.html>
<https://theburningmonk.com/2017/07/applying-the-saga-pattern-with-aws-lambda-and-step-functions/>
<https://queue.acm.org/detail.cfm?id=3025012>

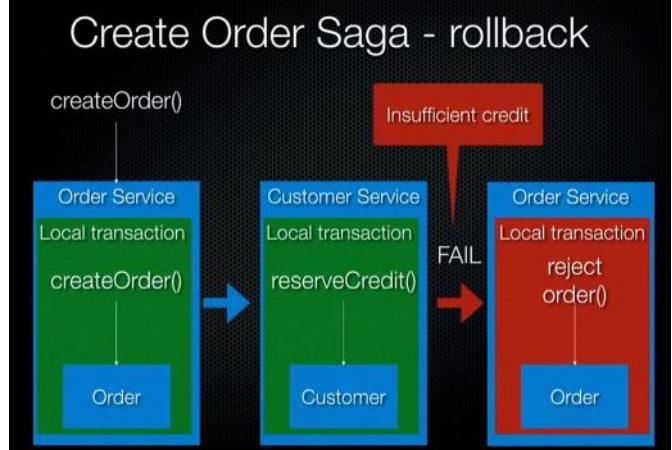
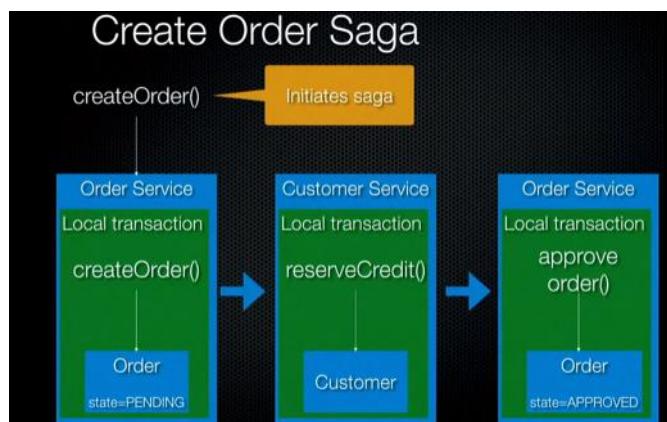
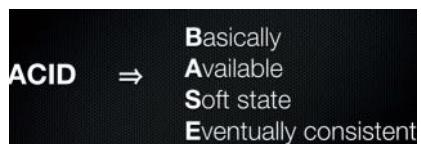
2PC (2 phase commit) распределенные транзакции негодятся так как единый координатор это источник ошибок (и могут быть блокировки и не все поддерживают распределенные транзакции)



ACID транзакции внутри одного микросервиса это ОК



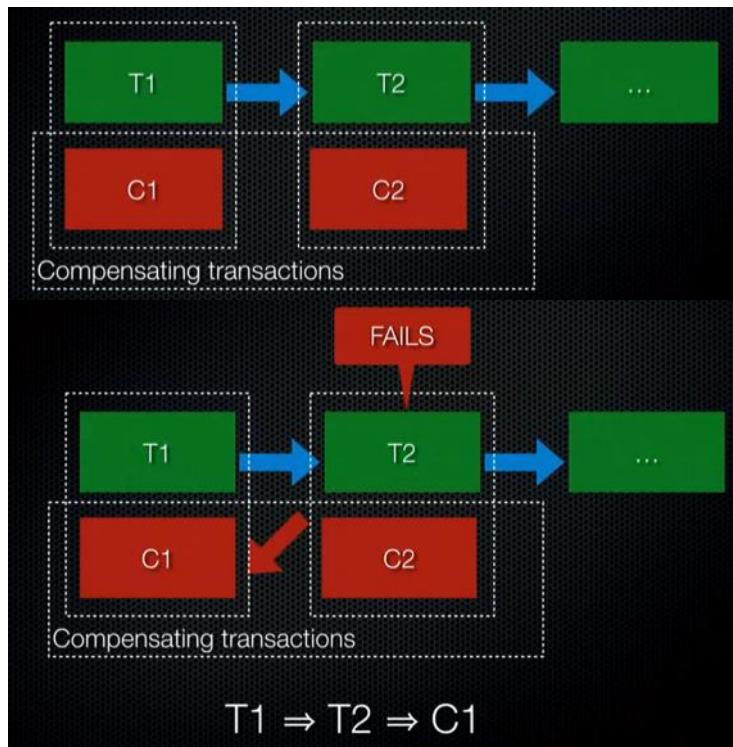
ACID-транзакции внутри саги те между микросервисами



как аналог rollback в саге нужно сделать компенсирующие механизмы

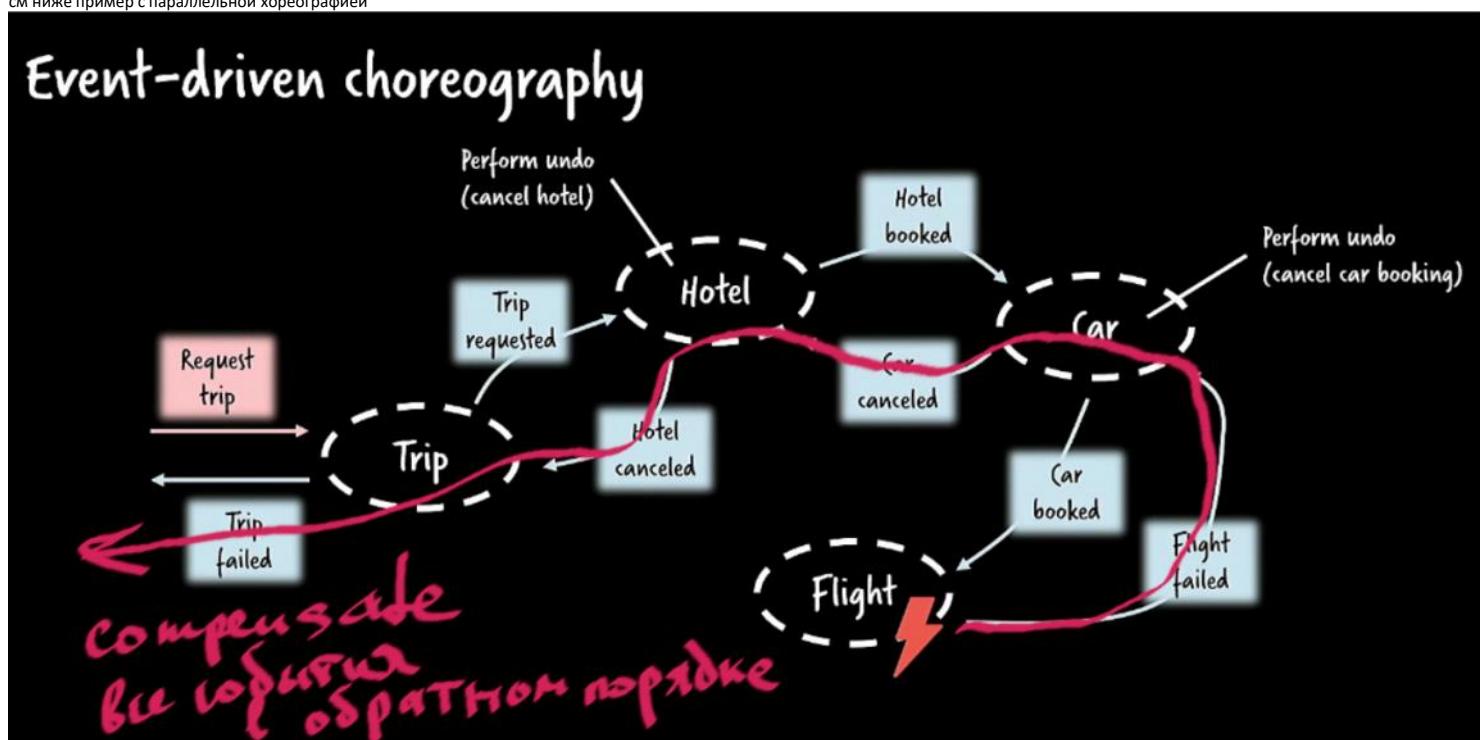


- Use compensating transactions
- Developer must write application logic to “rollback” eventually consistent transactions
- Careful design required!

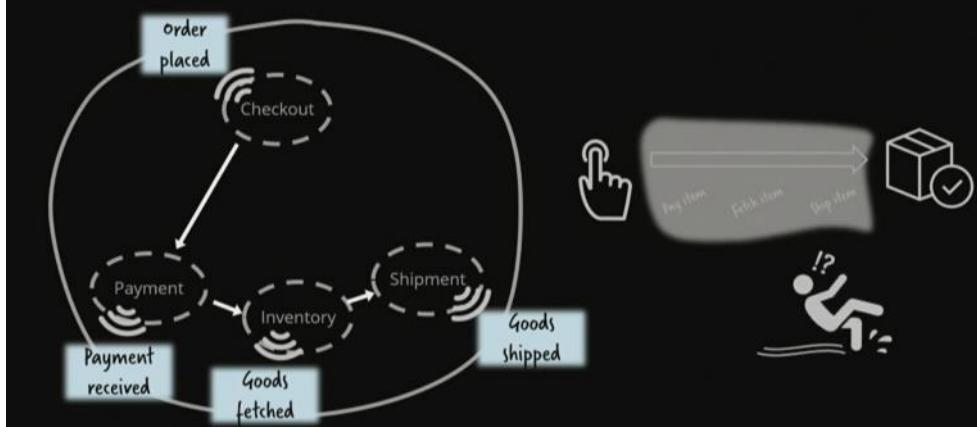


пример1 - сервисы хореографируются **последовательно**
см ниже пример с параллельной хореографией

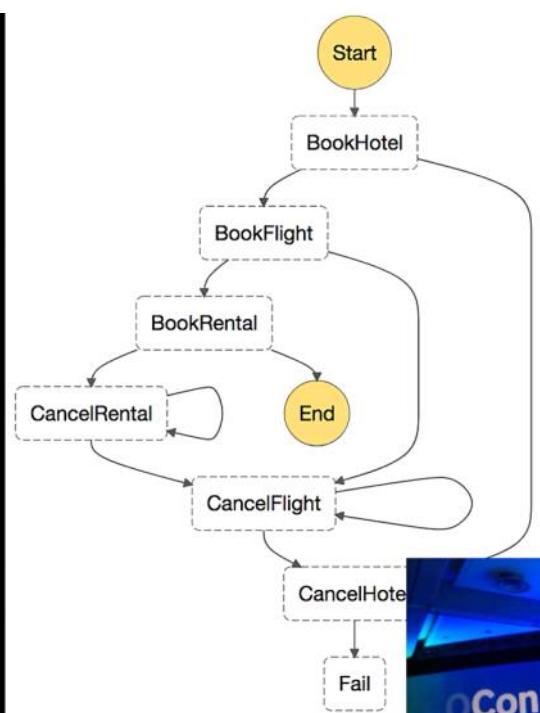
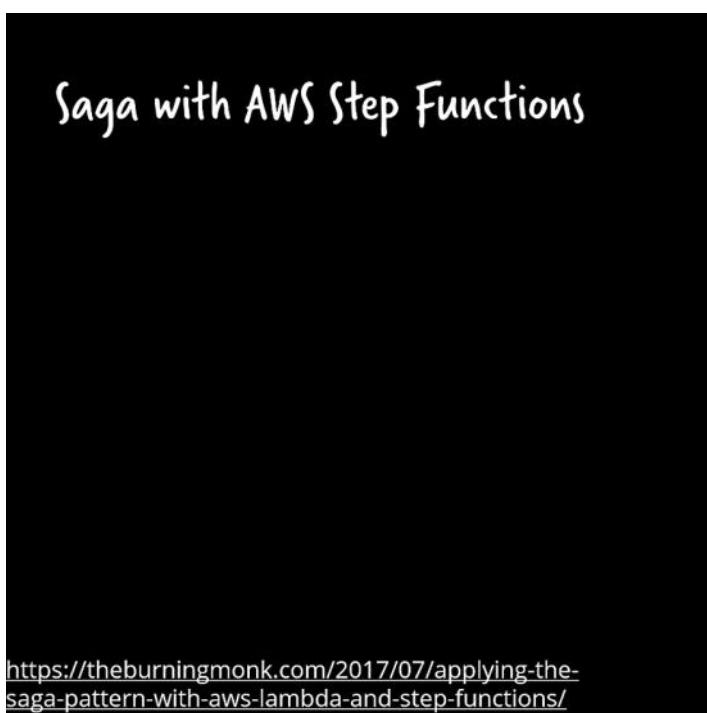
Event-driven choreography



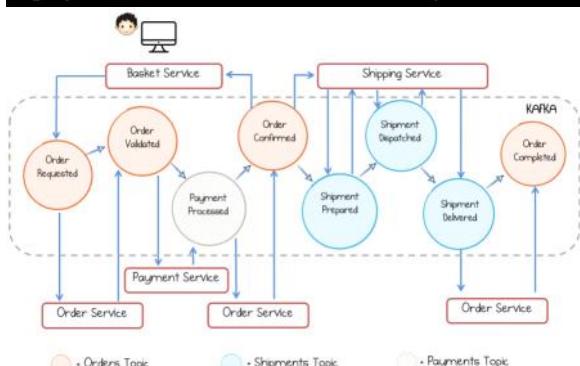
Peer-to-peer event chains



Saga with AWS Step Functions



<https://theburningmonk.com/2017/07/applying-the-saga-pattern-with-aws-lambda-and-step-functions/>



у нас есть только два варианта ответа на ошибку: компенсировать и извиниться



Sept 25-26, 2015
thestrangeloop.com

Without cross-service transactions:

A

Compensating transactions

≈ abort/rollback at app level

C

Apologies

detect & fix constraint violations
after the fact, rather than preventing them

при компенсации заказ не дуаляется а делается статус "rejected по техническим причинам"

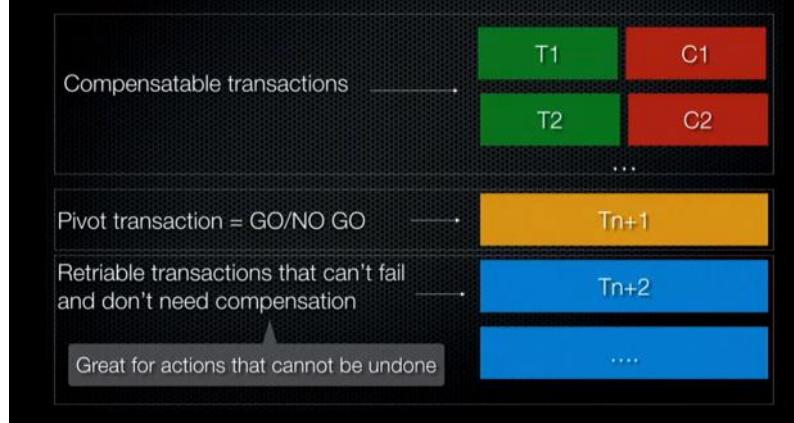
пот если упадет компенсационная транзакция то уже непонятно что делать

Writing compensating transactions isn't always easy

- Write them so they will always succeed
 - If a compensating transaction fails ⇒ no clear way to recover
- Challenge: Undoing changes when data has already been changed by a different transaction/saga
 - More on this later
- Actions that can't be undone, e.g. sending email
 - Reorder saga and move to end

сага состоит из трех фаз

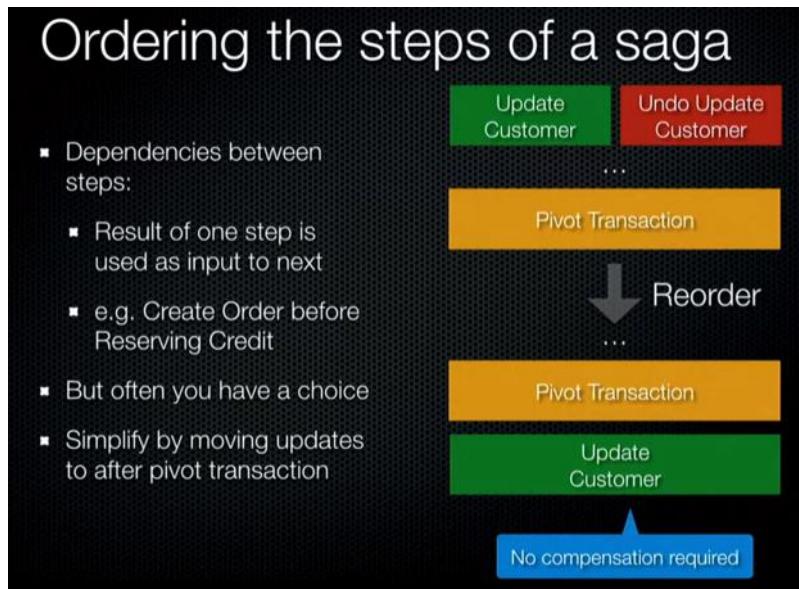
Saga structure: 3 phases



Structure of Create Order Saga



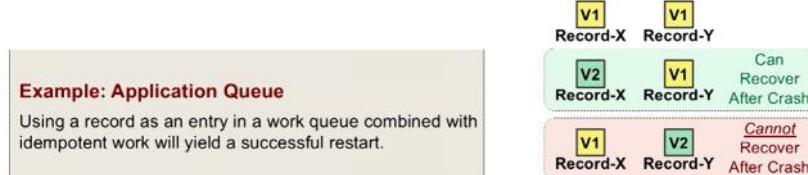
(1) для упрощения саги можно переставить транзакции местами
- тогда можно избавится от некоторых компенсирующих ветвей (как на картинке ниже)



Careful Replacement (Record Writes)

• Careful Replacement for Record Writes

- Update to records in pre-SQL databases needed careful ordering
- In many cases, an update to one record (say Record-X) before updating Record-Y allows the application to recover after failure

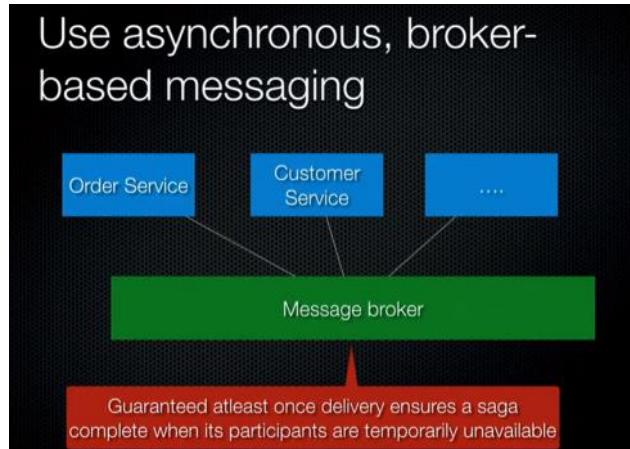


(2) semantic lock - я могу явно ввести правило что например PENDING ORDER нельзя перевести в статус CANCELED

Use countermeasures to prevent anomalies

- **Use semantic lock** to prevent lost updates
 - e.g. a PENDING order cannot be cancelled
- **Reorder saga** to eliminate dirty reads caused by compensating transactions
 - C_i undoes changes made by $T_i \Rightarrow$ possibility of dirty read
 - Eliminate C_i by moving update to (after) pivot transaction
- **Use commutative updates** to eliminate lost updates caused by compensating transactions
 - Customer.reserveCredit() and releaseCredit()
 - Account.debit() and credit()
- ...

для реализации саги лучше использовать очереди с "at least one delivery"



чем отличается хореография от оркестровки

Choreography: **distributed** decision making

VS.

Orchestration: **centralized** decision making

Benefits and drawbacks of choreography

Benefits

- Simple, especially when using event sourcing
- Participants are loosely coupled

Drawbacks

- Decentralized implementation - potentially difficult to understand
- Cyclic dependencies - services listen to each other's events, e.g. Customer Service **must know** about all Order events that affect credit
- Overloads domain objects, e.g. Order and Customer **know** too much
- Events = indirect way to make something happen

Benefits and drawbacks of orchestration

Benefits

- Centralized coordination logic is easier to understand
- Current state is in database
- Reduced coupling, e.g. Customer knows less
- Reduces cyclic dependencies

Drawbacks

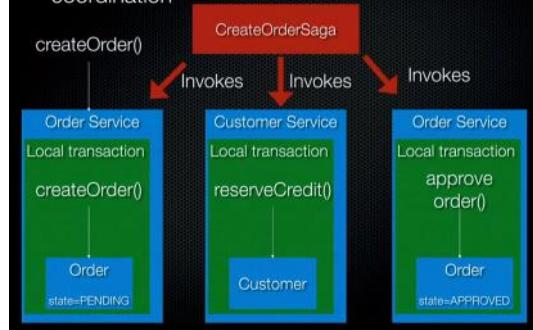
- Risk of smart sagas directing dumb services

- **оркестратор сохраняет состояние процесса в БД**

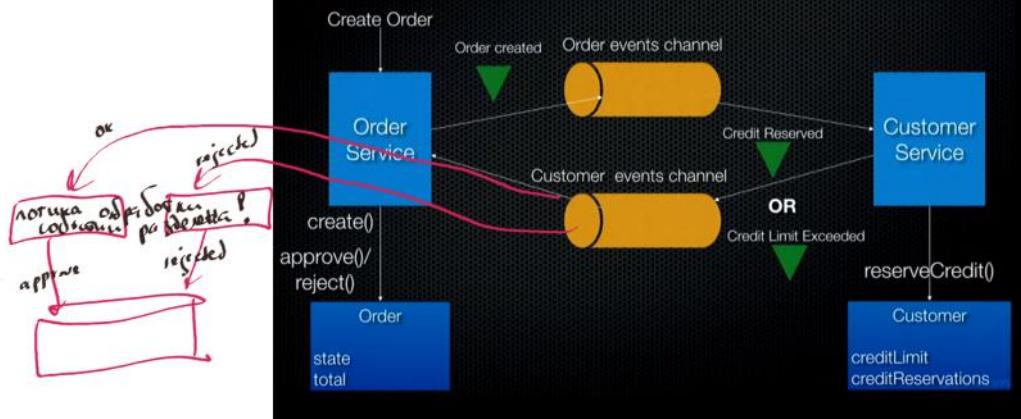
Saga orchestrator behavior

- On create:
 - Invokes a saga participant
 - Persists state in database
 - Wait for a reply
- On reply:
 - Load state from database
 - Determine which saga participant to invoke next
 - Invokes saga participant
 - Updates its state
 - Persists updated state
 - Wait for a reply
 - ...

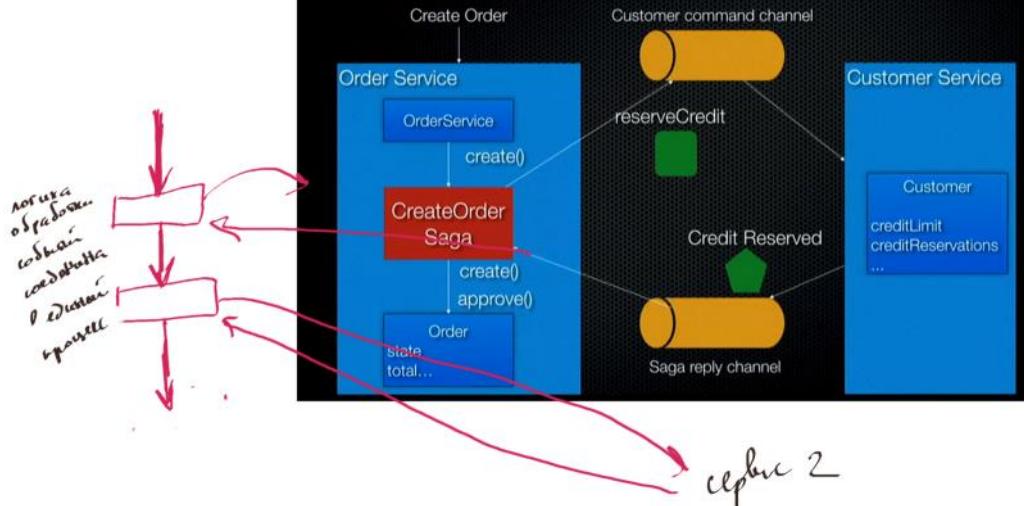
Option #2: Orchestration-based saga coordination



Option #1: Choreography-based coordination using events



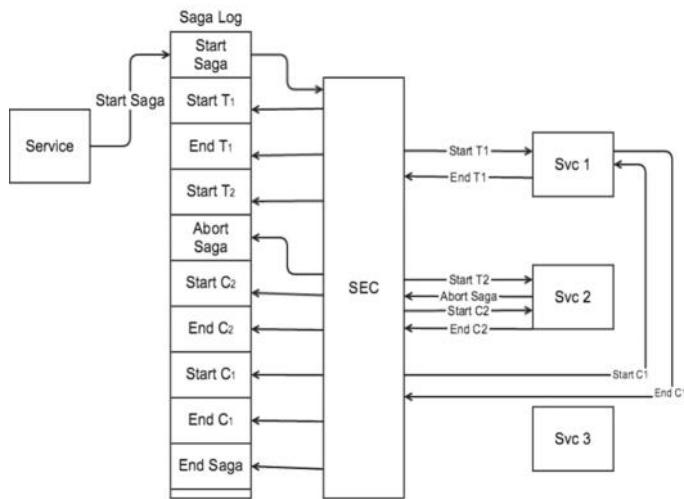
CreateOrderSaga orchestrator



лучше чтобы сага содержала не более 4 шагов, иначе разобраться в ней будет невозможно

If your transaction involves 2 to 4 steps, choreography might be a very good fit.

However, this approach can rapidly become confusing if you keep adding extra steps in your transaction as it is difficult to track which services listen to which events. Moreover, it also might add a cyclic dependency between services as they have to subscribe to one another's events.



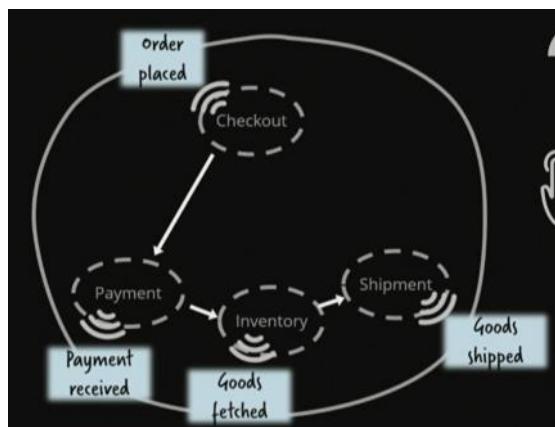
Un-Safe State

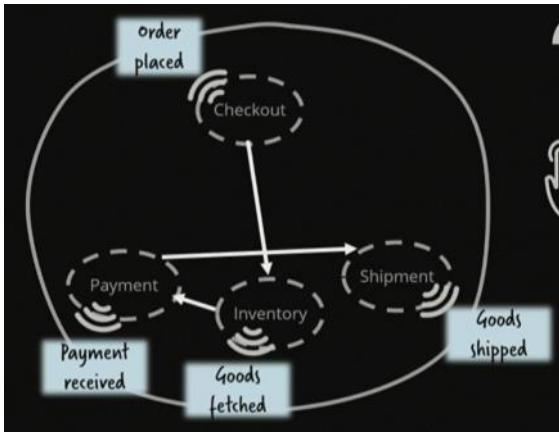
- Start Ti logged, no End Ti logged
 - Abort Saga
 - Start Compensating Requests

Request Messaging Semantics

- Sub-Requests (Ti): At Most Once
- Compensating Requests (Ci): At Least Once

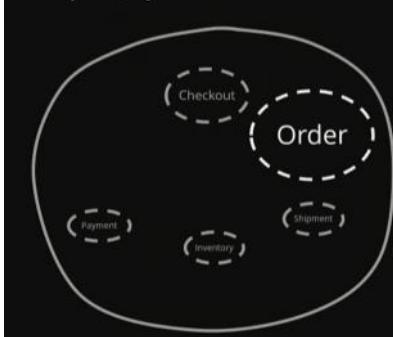
если я захочу поменять порядок событий то придется править все участвующие сервисы





в хореографии нужно избегать централизованного-оркестрирующего сервиса

Danger of god services?



хореография между сервисами а оркестровка внутри сервиса
те эти паттерны работают совместно

основной недостаток ACID транзакций это блокировка всей коммуникации (и большие временные задержки)

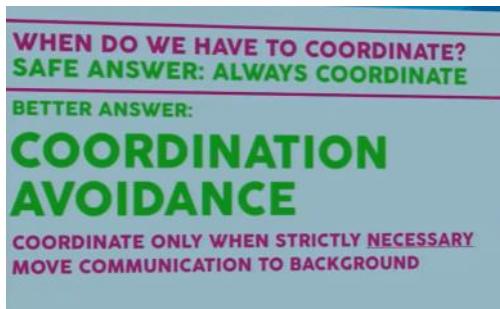
THE SIMPLE ANSWER: SINGLE-SYSTEM IMAGE

Impose a total order on events in the system
Illusion created by a partially ordered protocol



COST:
BLOCKING COMMUNICATION
COORDINATION

координация нужна только когда абсолютно необходима (те когда касается денег и тп)



event cascade (в паттерне event collaboration) - когда одно событие вызывает другие события (и сложно понять что явилось причиной)

Каскад событий - это то, что происходит, когда вы получаете последовательность событий, запускающих другие события. Необычно я проиллюстрирую это абстракцией. Представьте себе три события A, B и C. Когда вы работаете над событием A, вы можете понять, что B является следствием ($A \rightarrow B$), и это хорошо. Кто-то другой работает над B и может понять, что C является следствием ($B \rightarrow C$), и это хорошо. Однако теперь есть каскад $A \rightarrow B \rightarrow C$, который может быть трудно увидеть, потому что вам нужно соединить оба контекста, чтобы увидеть его; когда вы думаете об A, вы не думаете о C, и пока вы работаете над B, вы не думаете об A. В результате этот каскад может привести к неожиданному поведению, которое может быть хорошим, но также может быть проблемой. Трехступенчатый каскад, подобный этому, является простым случаем,

Хотя каждый отдельный компонент проще, сложность взаимодействия будет возрастать. Это усугубляется тем, что эти взаимодействия не понятны при чтении исходного кода. При совместной работе с запросами легко увидеть, что вызов одного компонента вызывает реакцию другого компонента. Даже с полиморфизмом нетрудно разобраться и увидеть результаты. Совместная работа с событиями мало знает, что слушает события до времени выполнения - что на практике означает, что вы можете найти связи между компонентами только в данных конфигурации - и может быть несколько областей данных конфигурации. В результате эти взаимодействия трудно найти, понять и отладить. Здесь очень полезно использовать автоматизированные инструменты, которые могут отображать конфигурацию компонентов во время выполнения, чтобы вы могли видеть, что у вас есть.

пример "блокировки" в неблокирующей системе

Давайте посмотрим на более реалистичный пример. Рассмотрим систему управления для операционных в больнице. Операционные - это дефицитный и дорогой ресурс, поэтому им нужно тщательно управлять, чтобы извлечь из них максимальную пользу. Операции назначаются заранее, иногда за несколько недель (например, когда мне вынули штифты из исцеленной руки), иногда гораздо быстрее (например, когда у меня была сломана рука, когда меня прооперировали в течение нескольких часов).

Поскольку операционные - это ограниченные ресурсы, имеет смысл, что если операция отменяется или откладывается, комната следует освободить, чтобы ее можно было запланировать для чего-то другого. Таким образом, система планирования помещения должна прослушивать отложенные операции и освобождать комнату, в которой должна была проводиться операция.

Когда пациент поступает на процедуру, в больнице проводится серия предоперационных осмотров. Если одна из этих оценок противоречит операции, операцию следует отложить - по крайней мере, до тех пор, пока врач не осмотрит ее. Таким образом, система планирования операций должна прислушиваться к предоперационным противопоказаниям и откладывать любую операцию, которая противопоказана.

Обе эти причины событий разумны, но вместе они могут иметь непредвиденный эффект. Если кто-то войдет и получит противопоказание, операция откладывается, а комната освобождается. Проблема в том, что врач может просмотреть предоперационную оценку в течение нескольких минут и в любом случае решить продолжить, но тем временем комната была забронирована другой операцией. Это может быть очень неудобно для пациента, которому даже для простой операции придется быстро подготовиться к операции.

Каскады событий хороши тем, что что-то происходит и в результате цепочки локальных логических связей событий происходит что-то косвенное. Каскады событий плохи, потому что что-то происходит, и в результате цепочки локальных логических соединений событий происходит что-то косвенное. Каскады событий обычно выглядят довольно очевидными, когда они описываются таким образом, но пока вы их не увидите, их будет очень трудно обнаружить. Это область, в которой системы визуализации, которые могут строить граф цепочек событий, запрашивая метаданные из самих систем, могут быть очень удобными.

Distributed Saga pattern

7 января 2021 г. 15:27

Divide long-lived, distributed transactions into quick local ones with compensating actions for recovery.

- In other words: Create an ephemeral component to manage the execution of a sequence of actions distributed across multiple components
- термин сага придумал Hector Garcia-Molina, "Sagas," ACM, 1987 (<http://dl.acm.org/citation.cfm?id=38742>).
- [https://docs.microsoft.com/en-us/previous-versions/msp-n-p/jj591569\(v=pandp.10\)?redirectedfrom=MSDN](https://docs.microsoft.com/en-us/previous-versions/msp-n-p/jj591569(v=pandp.10)?redirectedfrom=MSDN)

saga - Суть идеи состоит в том, что одну длительную распределенную транзакцию можно рассматривать как композицию нескольких быстрых локальных транзакционных шагов (каждый из которых только внутри микросервиса)

- In a distributed system, you need to break up transactions involving multiple participants for other reasons: obtaining a shared lock is an expensive operation that can even be impossible in the face of certain failures like network partitions. As we discussed in chapter 8, the key to scalability and loose coupling is to consider each component an island of **delimited consistency**

транзакции только внутри микросервиса

- чтобы осуществить перенос бюджета, ни Алисе, ни Бобу не нужно было знать, как это работает; им нужно было только иметь возможность управлять своим **собственным бюджетом и отвечать на запросы о его сокращении или увеличении**. Это важное преимущество, поскольку оно позволяет отделить описание сложных бизнес-процессов от реализации затронутых компонентов как во время выполнения, так и во время разработки.

compensating transaction - каждая локальная транзакция имеет компенсацию

- You have seen that in order to do this, you have to not only describe the individual transactions that are done on each account but also provide compensating transactions that come into play if the saga needs to be aborted. This is the same as performing a transaction rollback on a RDBMS;

пример: Sam=Saga Alice->Bob

Sam: "Alice, please reduce your budget by \$10,000 if possible."

Alice: "OK. I've done so."

Sam: "Bob, please increase your budget by \$10,000."

Bob: "Thanks, Sam; it's done."

Sam: "Thanks everyone; transfer finished!"

Sam: "Alice, please reduce your budget by \$10,000 if possible."

Alice: "Sorry, Sam, my budget is too low already."

Sam: "Listen up, everyone—this transfer is aborted!"

пример последовательной саги (компенсация срабатывает только у Алисы так как только она успела провести свою транзакцию)

Sam: "Alice, please reduce your budget by \$10,000 if possible."

Alice: "OK. I've done so."

Sam: "Bob, please increase your budget by \$10,000."

Bob: "Sorry, Sam, my project was just canceled—I no longer have a budget."

Sam: "Alice, please increase your budget by \$10,000." **компенсация**"

Alice: "Thanks, Sam; it's done."

Sam: "Listen up, everyone—this transfer is aborted!"

пример параллельной саги (компенсация срабатывает только у Боба так как только он успела провести свою транзакцию)

Sam: "Alice, please reduce your budget by \$10,000 if possible. Bob, please increase your budget by \$10,000."

Bob: "Thanks, Sam. It's done."

Alice: "Sorry, Sam, my budget is too low already."

Sam: "Bob, please reduce your budget by \$10,000."

Bob: "OK. I've done so."

Sam: "Listen up, everyone—this transfer is aborted!"

если сама компенсация упадет с ошибкой, то нужно будет разбирать инцидент вручную

- Конечно, здесь мы предположили, что компенсационные транзакции всегда успешны; но что, если это не так? Что, если в последнем примере Боб уже - и удивительно быстро - потратил 10 000 долларов, пока Сэм ждал ответа Алисы? В этом случае система будет в несогласованном состоянии, которое Сэм не сможет исправить без внешней помощи. Мораль этого продуманного эксперимента заключается в том, что компьютеры и алгоритмы не могут нести ответственность за работу со всеми возможными угловыми случаями, особенно когда речь идет о распределенных системах, где невозможно применить удобные упрощения полностью последовательного локального выполнения. В таких случаях несоответствия должны распознаваться как возможные состояния системы и передаваться вверх: например, чтобы их могли решить люди, которые управляют системой

если порядок выполнения транзакций важен, то можно использовать последовательную сагу (и тогда компенсационные транзакции будут выполняться в строго обратном известном порядке)

- One property of compensating transactions that we have glossed over so far requires a bit of formalism: where transaction T1 takes the component from state S0 to state S1, the compensating transaction C1 takes it from S1 back to S0. We are applying these

transactions within the consistency boundaries of several different components—Alice and Bob, in the earlier example—and within one of these components an execution of a sequence of transactions T1..Tn would take that component from state S0 to Sn. Because transactions do not, in general, commute with each other, taking the system back from state Sn to S0 would require the application of the compensating transactions Cn..C1: the compensating transactions would be applied in reverse order, and this order matters. We have played with the thought of parallelizing parts of the two subconversations (Sam-Alice and Sam-Bob), but you must always be careful to maintain a deterministic and thus reversible order for all transactions that are performed with Alice and Bob individually. For an in-depth discussion of compensation semantics, please refer to Garcia-Molina's "Sagas" paper.

Delimited consistency - означает что консистентность делится

- The Saga pattern is applicable wherever a business process needs to be modeled that involves **multiple independent components**: in other words, wherever a business transaction spans multiple regions of **delimited consistency**. In these cases, the pattern offers **atomicity** (the ability to roll back by applying compensating transactions) and **eventual consistency** (in the sense that application-level consistency is restored by issuing **apologies**; see Pat Helland's aforementioned blog article), but it does not offer isolation
- Это можно проиллюстрировать на примере работы почты в нашем примере приложения Gmail. Поскольку ожидается, что число пользователей будет огромным, вам придется разделить хранилище всей почты на множество разных компьютеров, расположенных в нескольких центрах обработки данных, распределенных по всему миру. Предполагая, что папки пользователя могут быть разделены на несколько компьютеров, перемещение электронного письма из одной папки в другую может означать, что оно перемещается между компьютерами. Он будет либо сначала скопирован в место назначения, а затем удален в источнике, либо помещен в временное хранилище, удален в источнике, а затем скопирован в место назначения.

XA-транзакции и 2PC не годятся

10 января 2021 г. 20:26

* Actual Serial Execution

В самом начале эры баз данных предполагалось, что транзакция БД должна охватывать весь поток действий пользователя

Создатели баз предполагали: было бы удобно сделать весь этот длиннющий процесс одной транзакцией, чтобы фиксировать ее атомарно в случае надобности

- Например, бронирование авиабилета — многоэтапный процесс (поиск маршрутов, цен и свободных мест; выбор маршрута; бронирование мест на каждом из участков маршрута; ввод данных пассажира; оплата). .

К сожалению, люди не так быстро реагируют и принимают решения.

- База, транзакции которой ожидают ввода данных пользователем, должна поддерживать потенциально огромное количество конкурентных транзакций, основная часть из которых простаивает. Большинство БД не способно реализовать это эффективно,

Традиционные реляционные БД НЕ ограничивают длительность выполнения транзакций, поскольку они созданы в расчете на интерактивные приложения, ожидающие ввода данных пользователем.

- Следовательно, одна транзакция может ожидать другую неограниченное время. Даже если постараться максимально сократить длительность всех своих транзакций, когда нескольким транзакциям необходимо обратиться к одному объекту, может сформироваться очередь, поскольку одной транзакции, прежде чем выполнить какие-либо действия, придется ждать завершения нескольких других.

XA-транзакции и 2PC не годятся

- Традиционный подход к обеспечению согласованности данных между несколькими сервисами, БД или брокерами сообщений заключается в применении распределенных транзакций. Стандартом де-факто для управления распределенными транзакциями является X/Open XA (см. <https://ru.wikipedia.org/wiki/XA>). Модель XA использует двухэтапную фиксацию (two-phase commit, 2PC), чтобы гарантировать сохранение или откат всех изменений в транзакции. Для этого требуется, чтобы базы данных, брокеры сообщений, драйверы БД и API обмена сообщениями соответствовали стандарту XA,
XA распределенные транзакции не поддерживаются брокерами(типа кафки) и БД

- Несмотря на внешнюю простоту, распределенные транзакции имеют ряд проблем.
- Многие современные технологии, включая такие базы данных NoSQL, как MongoDB и Cassandra, их не поддерживают.
- Распределенные транзакции не поддерживаются и некоторыми современными брокерами сообщений вроде RabbitMQ и Apache Kafka. Так что, если вы решите использовать распределенные транзакции, многие современные инструменты будут вам недоступны

[XA распределенные транзакции синхронны, что ведет к блокировкам](#)

- Еще одна проблема распределенных транзакций связана с тем, что они представляют собой разновидность синхронного IPC, а это ухудшает доступность. Чтобы распределенную транзакцию можно было зафиксировать, доступными должны быть все вовлеченные в нее сервисы. Как описывалось в главе 3, доступность системы — это произведение доступности всех участников транзакции. Если в распределенной транзакции участвуют два сервиса с доступностью 99,5 %, общая доступность будет 99 %, что намного меньше. Каждый дополнительный сервис понижает степень доступности.

* гарантии транзакций (atomic commit)

4 января 2021 г. 9:44

общее

- Kafka поставляется со встроенными транзакциями почти так же, как и большинство реляционных баз данных. Как мы увидим, реализация совершенно иная, но цель схожа: гарантировать, что наши программы будут давать предсказуемые и повторяемые результаты, даже когда что-то не получается.
- <https://www.confluent.io/blog/transactions-apache-kafka/>

(гарантия атомарности транзакций 1) Messages sent to different topics, within a transaction, will either all be written or none at all

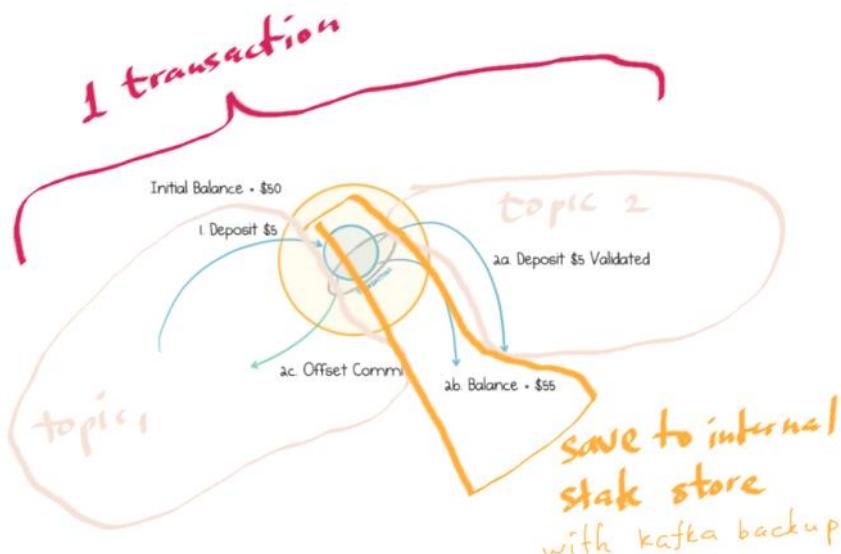
- Сообщения, отправленные в разные темы в рамках транзакции, будут записаны либо полностью, либо вообще не будут.
- Они позволяют атомарно отправлять группы сообщений по разным темам - например, «Заказ подтвержден» и «Уменьшить уровень запасов», что приведет к нарушению согласованности системы в случае успешного выполнения только одного из двух.

(гарантия атомарности транзакций 2) Messages sent to a single topic, in a transaction, will never be subject to duplicates, even on failure.

- Сообщения, отправленные в одну тему в транзакции, никогда не будут дублироваться, даже в случае сбоя.
- Они удаляют дубликаты, из-за чего многие потоковые операции дают неверные результаты (даже такие простые, как подсчет).

в одной транзакции можно записать в свой state store из одного топика + отправить в следующий топик

- Поскольку Kafka Streams использует хранилища состояний, а хранилища состояний поддерживаются темой Kafka, когда мы сохраним данные в хранилище состояний, а затем отправляем сообщение в другую службу, мы можем обернуть все это в транзакцию. Это свойство оказывается особенно полезным.
- As a state store gets its durability from a Kafka topic, we can use transactions to tie writes to the state store and writes to other output topics together. This turns out to be an extremely powerful pattern because it mimics the tying of messaging and databases together atomically, something that traditionally required painfully slow protocols like XA.
- Если транзакции включены в Kafka Streams, все эти операции будут автоматически заключены в транзакцию, гарантируя, что баланс всегда будет атомарно синхронизирован с депозитами

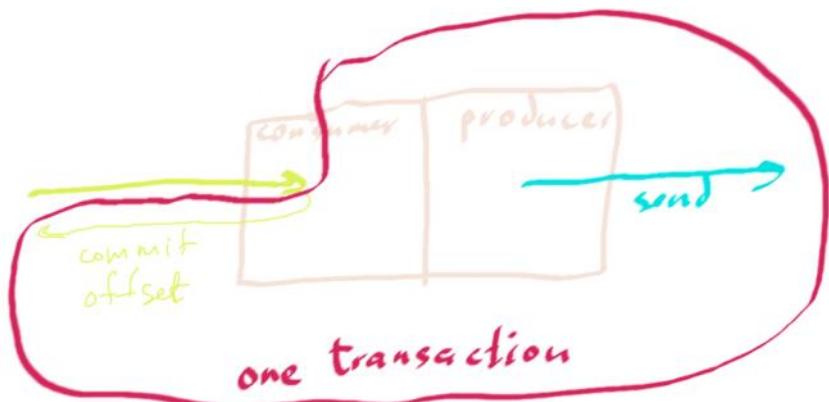


Транзакции в Kafka позволяют создавать длинные цепочки сервисов, в которых обработка каждого шага в цепочке обернута EOS гарантией

- Transactions in Kafka allow the creation of long chains of services, where the processing of each step in the chain is wrapped in exactly-once guarantees. This reduces duplicates, which means services are easier to program and, as we'll see later in this chapter, transactions let us tie streams and state together when we implement storage either through Kafka Streams state stores or using the Event Sourcing design pattern. All this happens automatically if you are using the Kafka Streams API.
- Kafka records the progress that each consumer makes by storing an offset in a special topic, called `consumer_offsets`. So to validate each deposit exactly once, we need to perform the final two actions—
 - o (a) send the “Deposit Validated” message back to Kafka, and
 - o (b) commit the appropriate offset to the `consumer_offsets` topic—as a single atomic unit (Figure 12-3). The code for this would look something like the following

```
//Read and validate deposits
validatedDeposits = validate(consumer.poll(0))

//Send validated deposits & commit offsets atomically
producer.beginTransaction()
producer.send(validatedDeposits)
producer.sendOffsetsToTransaction(offsets(consumer))
producer.endTransaction()
```



в DDD нет распределенных транзакций, есть только транзакции в пределах одного микросервиса(ограниченного контекста или агрегата)

- Let's begin by making one thing clear: transactions are fine within individual services, where we can, and should, guarantee strong consistency. This means that it is fine to use transactional semantics within a single service (the bounded context)—which is something that can be achieved in many ways: using a traditional SQL database like Oracle, a modern distributed SQL database like CockroachDB, or using event sourcing through Akka Persistence. What is problematic is expanding them beyond the single service, as a way of trying to bridge data consistency across multiple services (i.e. bounded contexts).
- The problem with transactions is that their only purpose is to try to maintain the illusion that the world consists of a single globally strongly consistent present—a problem that is magnified exponentially in distributed transactions (XA, Two-phase Commit, and friends). We already have discussed this at length: it is simply not how the world works, and computer science is no different.

Use Distributed Sagas, Not Distributed Transactions

- <http://vasters.com/archive/Sagas.html>
- [Caitie McCaffery - Distributed Sagas: A Protocol for Coordinating Microservices - .NET Fringe 2017](#)

- Этот шаблон был определен Гектором Гарсиа-Молиной в 1987 году как способ сократить период времени, в течение которого база данных должна принимать блокировки. Он не создавался для распределенных систем, но оказалось, что он очень хорошо работает в распределенном контексте.
- Шаблон Saga 8 - это шаблон управления отказами, который обычно используется как альтернатива распределенным транзакциям.
- Это помогает вам управлять длительными бизнес-транзакциями, которые используют компенсирующие действия для управления несоответствиями (сбои транзакций).

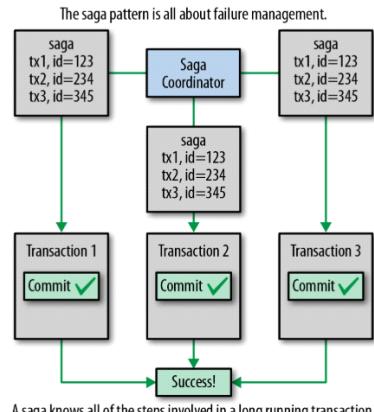
saga - Суть идеи состоит в том, что одну длительную распределенную транзакцию можно рассматривать как композицию нескольких быстрых локальных транзакционных шагов

- Суть идеи состоит в том, что одну длительную распределенную транзакцию можно рассматривать как композицию нескольких быстрых локальных транзакционных шагов.
- Каждый шаг транзакции сопряжен с компенсирующим реверсивным действием (реверсирование с точки зрения бизнес-семантики, без обязательного сброса состояния компонента), так что вся распределенная транзакция может быть отменена в случае сбоя путем выполнения компенсирующего действия каждого шага.
- В идеале эти шаги должны быть коммутируемыми, чтобы их можно было выполнять параллельно. (коммутативный закон сложения: $m + n = n + m$)
 - o Saga - отличный инструмент для обеспечения атомарности длительных транзакций.

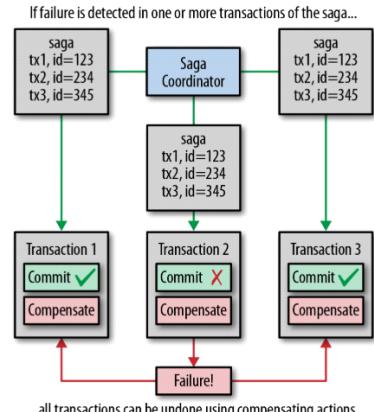
- Но важно понимать, что это не решение для изоляции .
- Одновременно выполняемые саги потенциально могут влиять друг на друга и вызывать ошибки.
 - Если это приемлемо, это зависит от варианта использования.
 - Если это неприемлемо, вам нужно использовать другую стратегию, например, убедиться, что сага не охватывает несколько границ согласованности, или просто использовать другой шаблон или инструмент для работы.
- The essence of the idea is that one long-running distributed transaction can be seen as the composition of multiple quick local transactional steps. Every transactional step is paired with a compensating reversing action (reversing in terms of business semantics, not necessarily resetting the state of the component), so that the entire distributed transaction can be reversed upon failure by running each step's compensating action. Ideally, these steps should be commutative so that they can be run in parallel.
- Одним из преимуществ этого метода (см. Рис. 6-3) является то, что он **eventually consistent** в конечном итоге согласован и хорошо работает с разделенными и асинхронно взаимодействующими компонентами, что делает его отличным подходом для архитектур, управляемых событиями и сообщениями.

saga pattern - is all about failure management

- long-running distributed workflows across multiple services



A saga knows all of the steps involved in a long running transaction.



* Компенсация транзакций

11 января 2021 г. 8:13

Повествование выполняет компенсирующие транзакции в обратном порядке по отношению к исходным

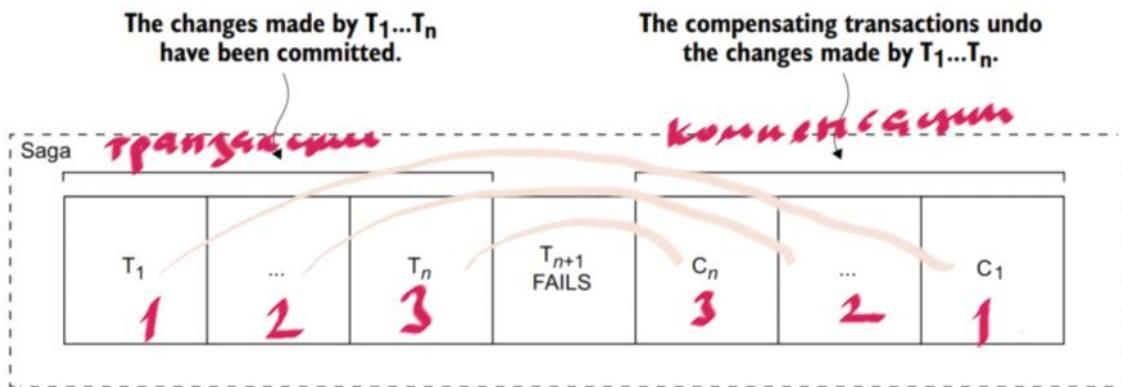


Figure 4.3 When a step of a saga fails because of a business rule violation, the saga must explicitly undo the updates made by previous steps by executing compensating transactions.

Если какая-либо одна локальная транзакция завершится неудачно,

- координирующая логика должна выбрать первого участника и сделать так, чтобы тот выполнил локальную транзакцию. Когда транзакция завершится, механизм координации выберет и вызовет следующего участника. Этот процесс продолжается до тех пор, пока повествование не выполнит все свои этапы.
- Если какая-либо локальная транзакция завершится неудачно, повествование должно выполнить компенсирующие транзакции в обратном порядке.

пример последовательности локальных транзакций саги

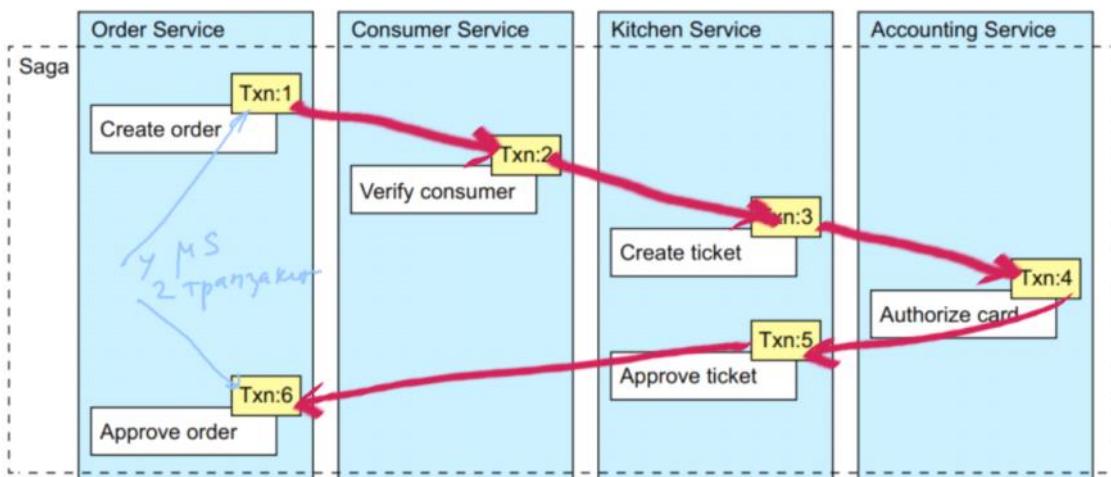


Figure 4.2 Creating an Order using a saga. The `createOrder()` operation is implemented by a saga that consists of local transactions in several services.

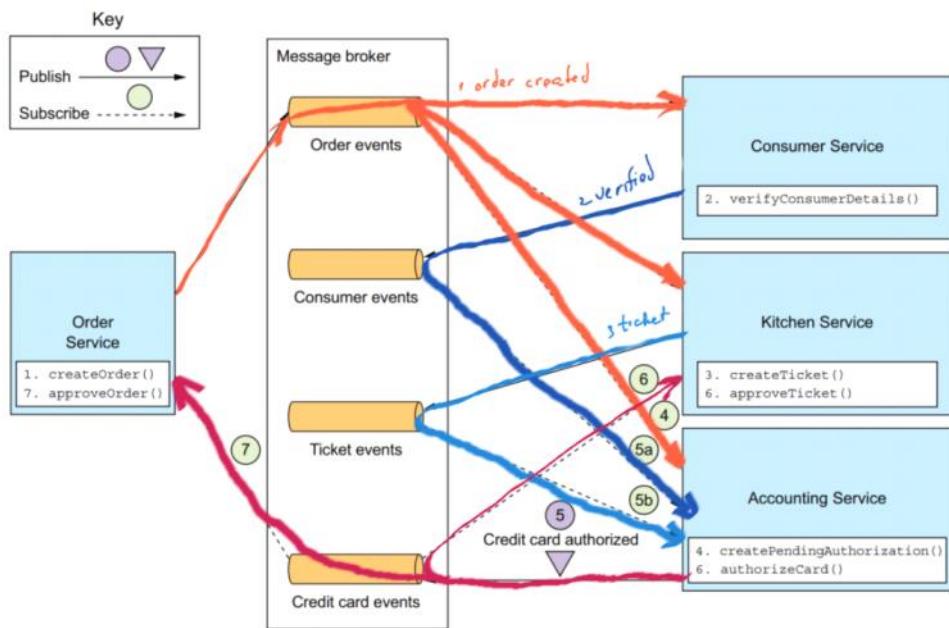
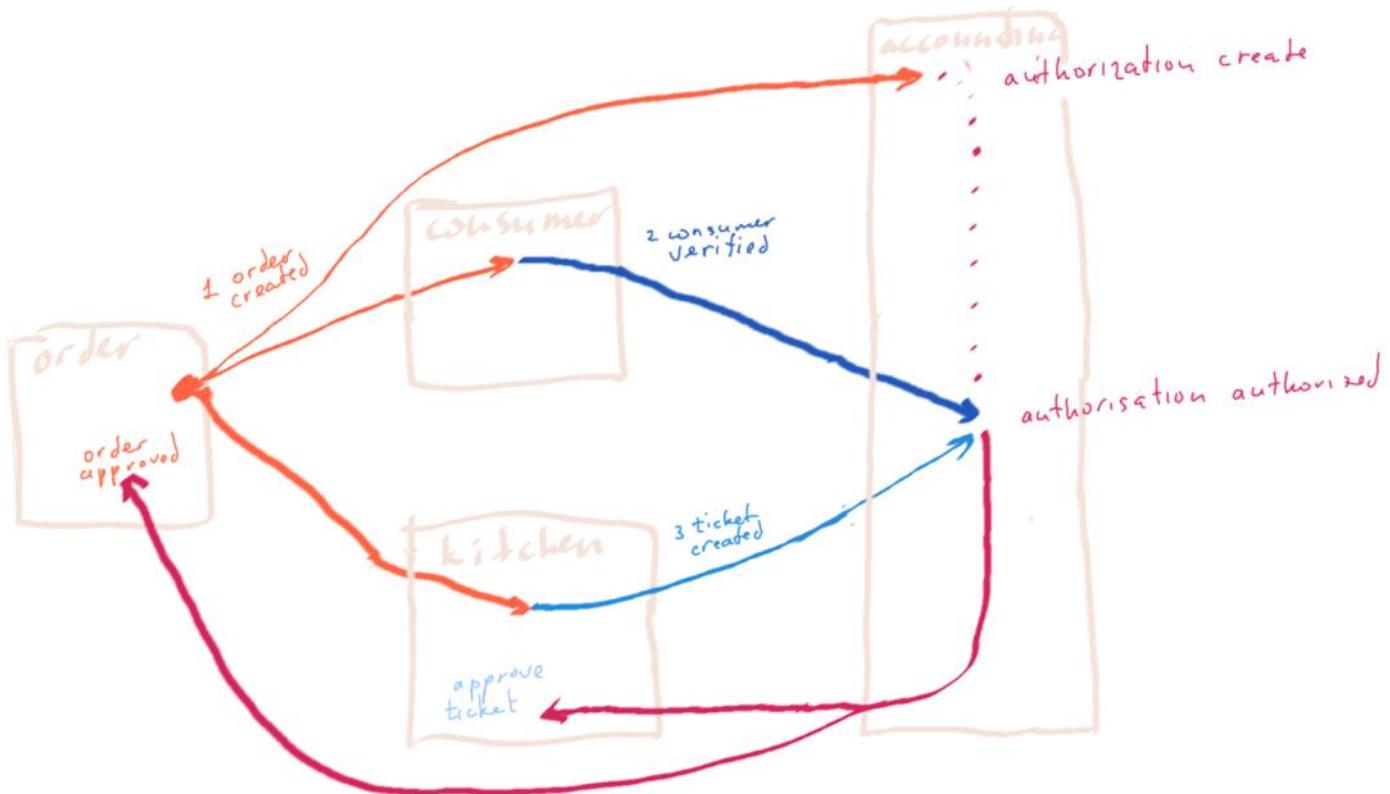
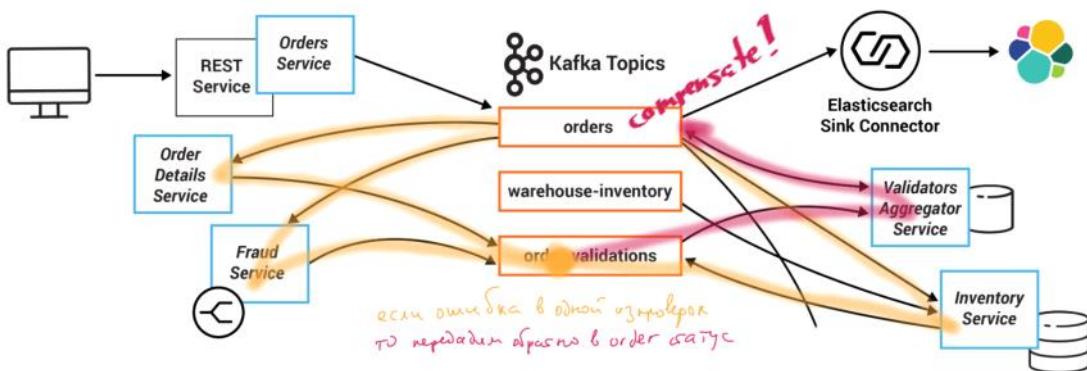
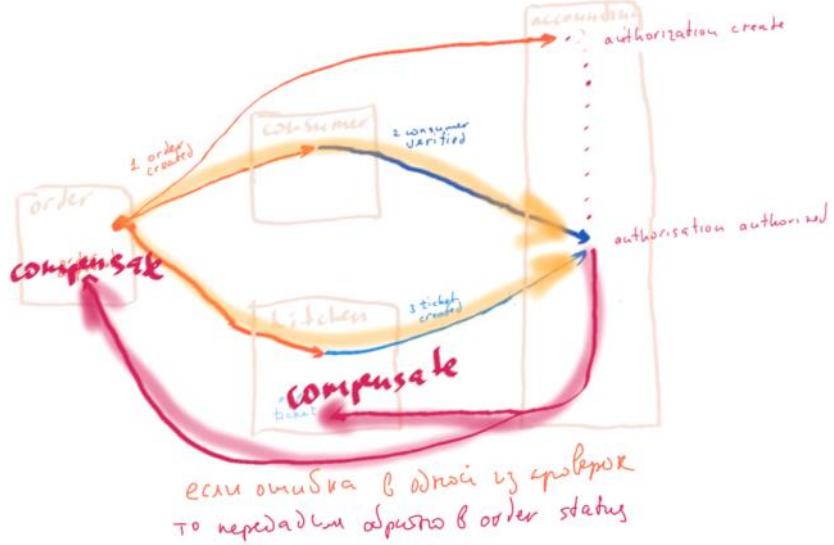


Figure 4.4 Implementing the Create Order Saga using choreography. The saga participants communicate by exchanging events.



один микросервис может участвовать в двух шагах бизнес-процесса (то иметь две транзакции)

- Например, когда сервис Order получает событие `CreditCardAuthorized`, он должен уметь найти соответствующий заказ (то сопоставить с `order created`). Решением здесь является публикация событий с идентификаторами соответствия, благодаря которым другие участники могут выполнить сопоставление.
- Например, участники повествования Create Order могут использовать параметр `orderId` в качестве ID соответствия, он будет передаваться от одного участника к другому. Сервис Accounting публикует событие `CreditCardAuthorized` с `orderId` из события `TicketCreated`. Когда сервис Order получает событие `CreditCardAuthorized`, он задействует `orderId` для извлечения соответствующего заказа.
- Аналогичным образом сервис Kitchen задействует `orderId` из того же события, чтобы извлечь подходящую заявку.



Каждый этап повествования/саги создает или обновляет ровно один агрегат (благодаря правилу DDD агрегатов "Одна транзакция создает или обновляет один агрегат")

- В этом примере повествование состоит из трех транзакций. Первая транзакция обновляет агрегат X в сервисе A. Две другие происходят в сервисе B: одна транзакция обновляет агрегат Y, а другая — агрегат Z

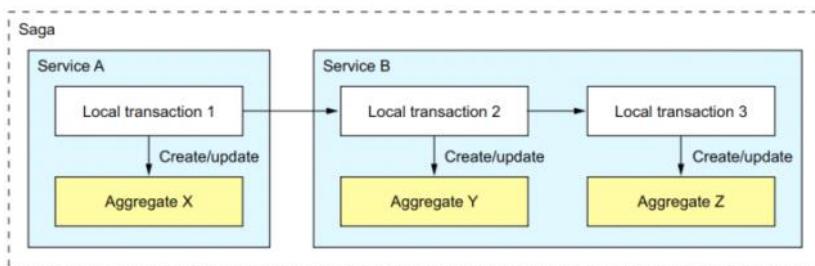


Figure 5.7 A transaction can only create or update a single aggregate, so an application uses a saga to update multiple aggregates. Each step of the saga creates or updates one aggregate.

не всякий этап требует компенсирующей транзакции

Table 4.1 The compensating transactions for the Create Order Saga

Step	Service	Transaction	Compensating transaction
1	Order Service	createOrder()	rejectOrder()
2	Consumer Service	verifyConsumerDetails()	—
3	Kitchen Service	createTicket()	rejectTicket()
4	Accounting Service	authorizeCreditCard()	—
5	Kitchen Service	approveTicket()	—
6	Order Service	approveOrder()	—

пример компенсирующего пути

- 1 Order Service—Create an Order in an APPROVAL_PENDING state.
- 2 Consumer Service—Verify that the consumer can place an order.
- 3 Kitchen Service—Validate order details and create a Ticket in the CREATE_PENDING state.
- 4 Accounting Service—Authorize consumer's credit card, which fails.
- 5 Kitchen Service—Change the state of the Ticket to CREATE_REJECTED.
- 6 Order Service—Change the state of the Order to REJECTED.

compensatable transactions

- первые три этапа повествования Create Order называются доступными для компенсации транзакциями, потому что шаги, следующие за ними, могут отказать.

pivot transaction

- Четвертый этап называется поворотной транзакцией, потому что дальнейшие шаги никогда не отказывают

retriable transactions

- Последние два этапа называются доступными для повторения транзакциями, потому что они всегда заканчиваются успешно.

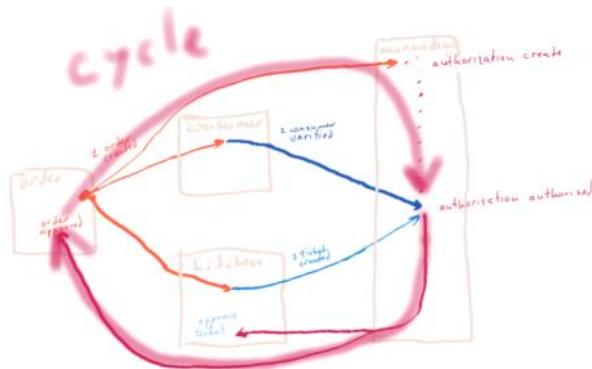
в случае ошибок дотерпим пока бизнес-процесс потока дойдет до конца (**до pivot транзакции**)

- accounting выступает в роли неявного оркестратора
 - например, если consumer service провалил проверку коньюмера, то он не направляет сразу ответ обратно в order service
 - например, если kitchen service провалил проверку order, то он не направляет сразу ответ обратно в order service
- accounting агрегирует пока в нем не соберутся все статусы и уже затем начинает обратно отсылать OK/NOTOK **туда где требуется компенсация** (в order service, kitchen service)

Возникают циклические зависимости между сервисами

Они не всегда являются проблемой, но, как принято считать, их наличие — признак плохого тона.

- Участники повествования подписываются на события друг друга, что часто создает циклические зависимости. Например, если внимательно присмотреться к рис. 4.4, можно заметить такие циклические зависимости, как Order->Accounting->Order. Они не всегда являются проблемой, но, как принято считать, их наличие — признак плохого тона.



компенсация транзакции - может быть бизнес событием

например в реальном мире заинтересованные компании, скорее всего, потребуют плату за отмену бронирования отеля

- Однако в реальном мире заинтересованные компании, скорее всего, потребуют плату за отмену бронирования.
- Благодаря этому, отмена - это не только техническое событие, происходящее в фоновом режиме, такое как откат транзакции, но и бизнес-процесс

compensation transaction is just a normal service call.

- A compensation transaction is just a normal service call. Technical as well as business reasons can lead to the use of mechanisms such as compensation transactions for microservices

* Choreography || Orchestration

12 января 2021 г. 9:41

two ways of coordination sagas

- Для координации этапов повествования можно применять либо хореографию, либо оркестрацию.
- В повествованиях, основанных на хореографии, локальная транзакция публикует события, которые заставляют других участников выполнить свои локальные транзакции.
- При использовании оркестрации централизованный оркестратор рассыпает участникам сообщения с инструкциями, какие локальные транзакции нужно выполнить

(A) Choreography

- each local transaction publishes domain events that trigger local transactions in other services
- Хореография — это один из способов реализации повествований. Она не предусматривает центрального координатора, который выдает участникам команды. Вместо этого участники подписываются на события друг друга и реагируют соответствующим образом

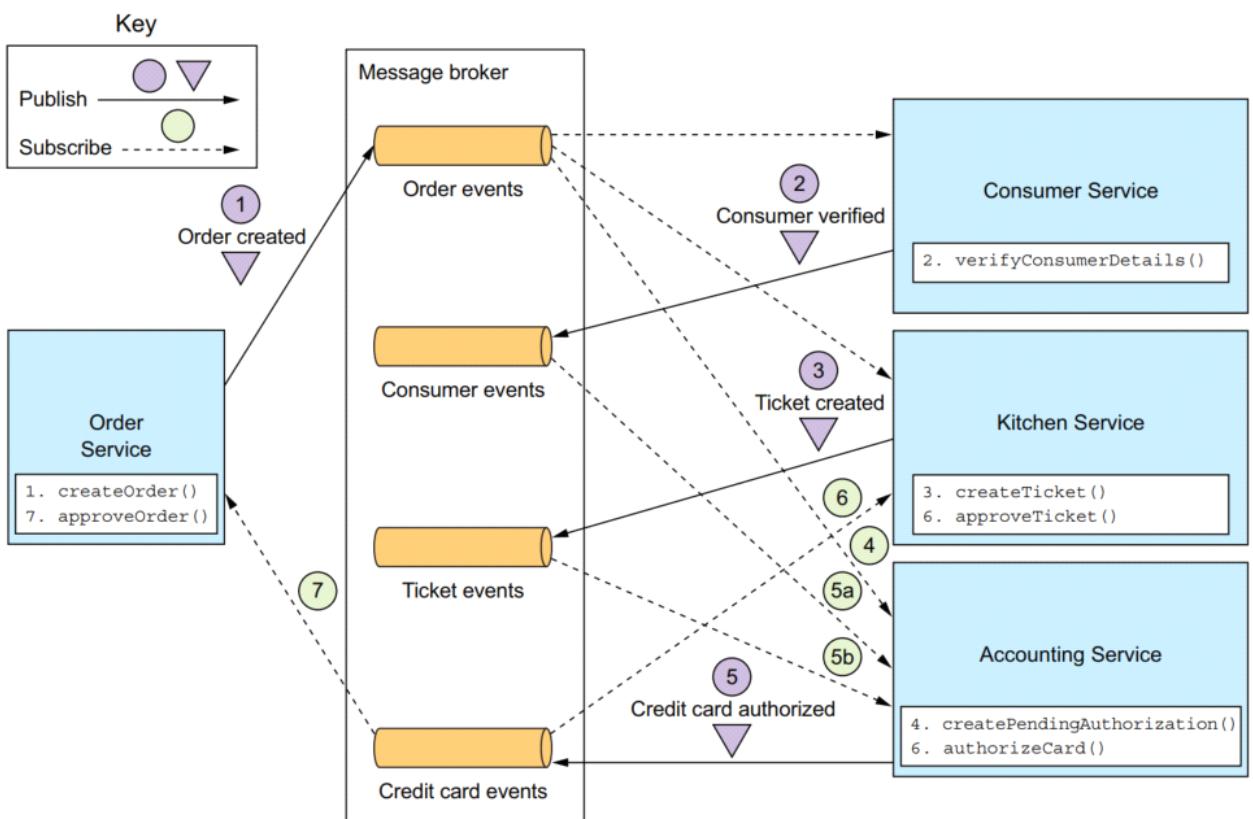


Figure 4.4 Implementing the Create Order Saga using choreography. The saga participants communicate by exchanging events.

The happy path through this saga is as follows:

- 1 Order Service creates an Order in the APPROVAL_PENDING state and publishes an OrderCreated event.
- 2 Consumer Service consumes the OrderCreated event, verifies that the consumer can place the order, and publishes a ConsumerVerified event.
- 3 Kitchen Service consumes the OrderCreated event, validates the Order, creates a Ticket in a CREATE_PENDING state, and publishes the TicketCreated event.
- 4 Accounting Service consumes the OrderCreated event and creates a CreditCardAuthorization in a PENDING state.
- 5 Accounting Service consumes the TicketCreated and ConsumerVerified events, charges the consumer's credit card, and publishes the CreditCardAuthorized event.
- 6 Kitchen Service consumes the CreditCardAuthorized event and changes the state of the Ticket to AWAITING_ACCEPTANCE.
- 7 Order Service receives the CreditCardAuthorized events, changes the state of the Order to APPROVED, and publishes an OrderApproved event.

The Create Order Saga must also handle the scenario where a saga participant rejects the Order and publishes some kind of failure event. For example, the authorization of the consumer's credit card might fail. The saga must execute the compensating transactions to undo what's already been done. Figure 4.5 shows the flow of events when the AccountingService can't authorize the consumer's credit card.

The sequence of events is as follows:

- 1 Order Service creates an Order in the APPROVAL_PENDING state and publishes an OrderCreated event.
- 2 Consumer Service consumes the OrderCreated event, verifies that the consumer can place the order, and publishes a ConsumerVerified event.
- 3 Kitchen Service consumes the OrderCreated event, validates the Order, creates a Ticket in a CREATE_PENDING state, and publishes the TicketCreated event.
- 4 Accounting Service consumes the OrderCreated event and creates a CreditCardAuthorization in a PENDING state.
- 5 Accounting Service consumes the TicketCreated and ConsumerVerified events, charges the consumer's credit card, and publishes a Credit Card Authorization Failed event.
- 6 Kitchen Service consumes the Credit Card Authorization Failed event and changes the state of the Ticket to REJECTED.
- 7 Order Service consumes the Credit Card Authorization Failed event and changes the state of the Order to REJECTED.

(B) Orchestration

- an orchestrator (object) tells the participants what local transactions to execute
- Оркестрация — это еще один способ реализации повествований. Она подразумевает определение класса-оркестратора, единственной задачей которого является рассылка инструкций участникам. Оркестратор взаимодействует с участниками в стиле «команда/асинхронный ответ». Чтобы выполнить этап повествования, он шлет участнику командное сообщение, объясняя, какую операцию тот должен выполнить. После выполнения операции участник возвращает оркестратору сообщение с ответом. Оркестратор обрабатывает это сообщение и решает, какой этап повествования нужно выполнить дальше.

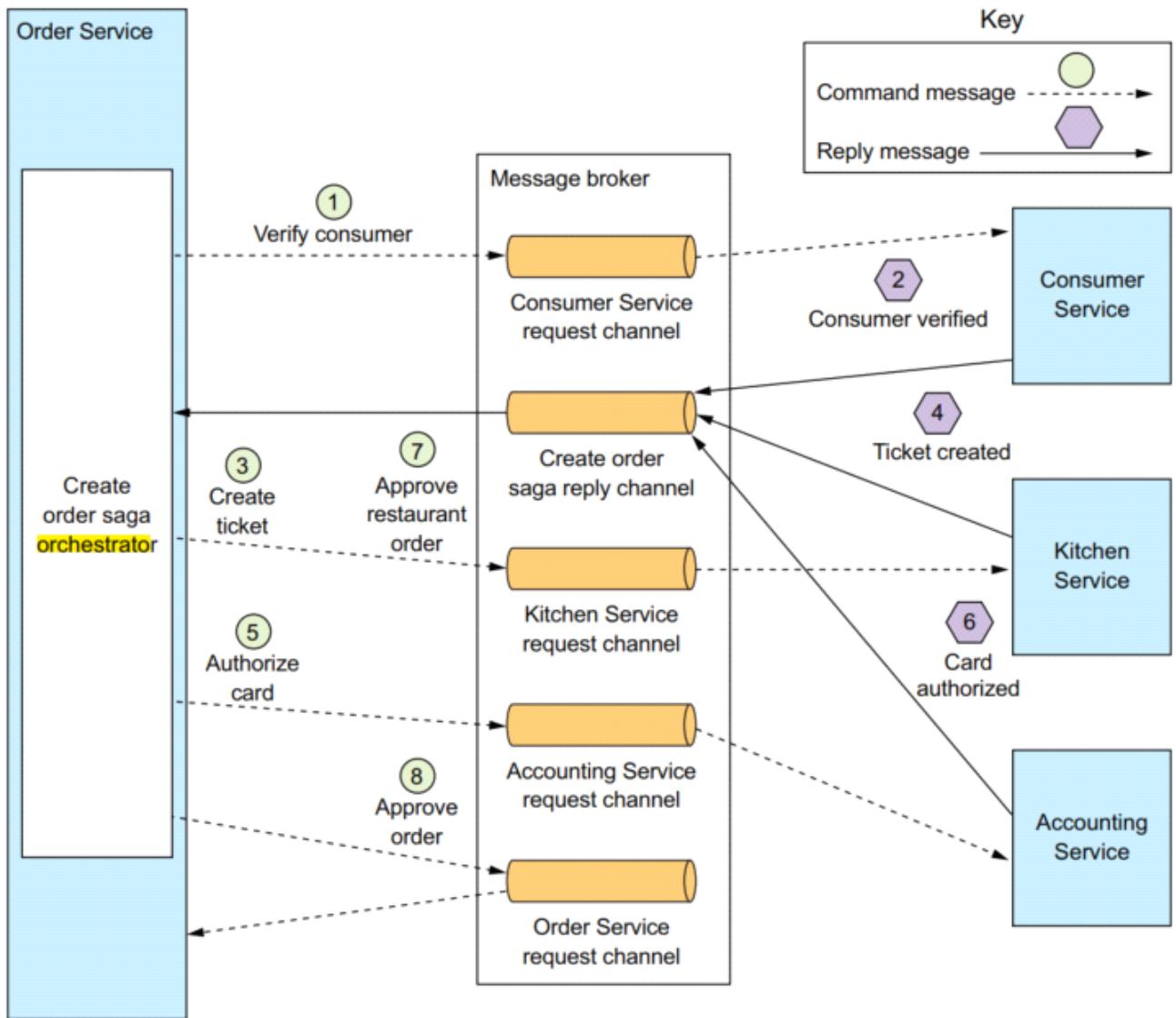


Figure 4.6 Implementing the Create Order Saga using orchestration. Order Service implements a saga orchestrator, which invokes the saga participants using asynchronous request/response.

Order Service first creates an Order and a Create Order Saga orchestrator. After that, the flow for the **happy path** is as follows:

- 1 The saga orchestrator sends a Verify Consumer command to Consumer Service.
- 2 Consumer Service replies with a Consumer Verified message.
- 3 The saga orchestrator sends a Create Ticket command to Kitchen Service.
- 4 Kitchen Service replies with a Ticket Created message.
- 5 The saga orchestrator sends an Authorize Card message to Accounting Service.
- 6 Accounting Service replies with a Card Authorized message.
- 7 The saga orchestrator sends an Approve Ticket command to Kitchen Service.
- 8 The saga orchestrator sends an Approve Order command to Order Service.

В конце БП order шлет сообщение сам себе

- Обратите внимание на то, что на последнем этапе оркестратор шлет командное сообщение сервису Order, компонентом которого он сам является. В принципе, повествование Create Order могло бы подтвердить заказ, обновив его напрямую. Но, чтобы оставаться

последовательным, оно обращается с сервисом Order просто как с еще одним участником.

[Вся координирующая логика повествования находится в оркестраторе](#)

- Благодаря этому доменные объекты становятся проще и им не нужно знать о повествованиях, в которых они участвуют. Например, при использовании оркестрации класс Order ничего не знает о повествованиях, поэтому имеет более простую модель конечного автомата. Во время выполнения повествования Create Order он переходит напрямую из состояния APPROVAL-PENDING в состояние APPROVED. Класс Order не обладает никакими промежуточными состояниями, которые соответствуют этапам повествования. Это значительно упрощает бизнес-логику.

[риск избыточной централизации бизнес-логики в оркестраторе](#)

- При этом оркестрация имеет один недостаток — риск избыточной централизации бизнес-логики в оркестраторе. В результате получается архитектура, в которой умный оркестратор командует глупыми сервисами. К счастью, этой проблемы можно избежать, если проектировать оркестраторы так, чтобы они отвечали лишь за последовательное выполнение действий и не содержали никакой дополнительной бизнес-логики.

[реализация оркестровки с помощью STATE MACHINE](#)

- Конечный автомат — это хорошая модель для оркестратора повествования.
- Он состоит из набора состояний и переходов между ними, которые инициируются с помощью событий.
- У каждого перехода может быть какое-то действие, которое в контексте повествования означает вызов участника.
- Переходы между состояниями инициируются завершением локального перехода, выполненного участником повествования.
- Текущее состояние и конкретный результат локального перехода определяют следующий переход и действие, которое нужно выполнить (если такое имеется)

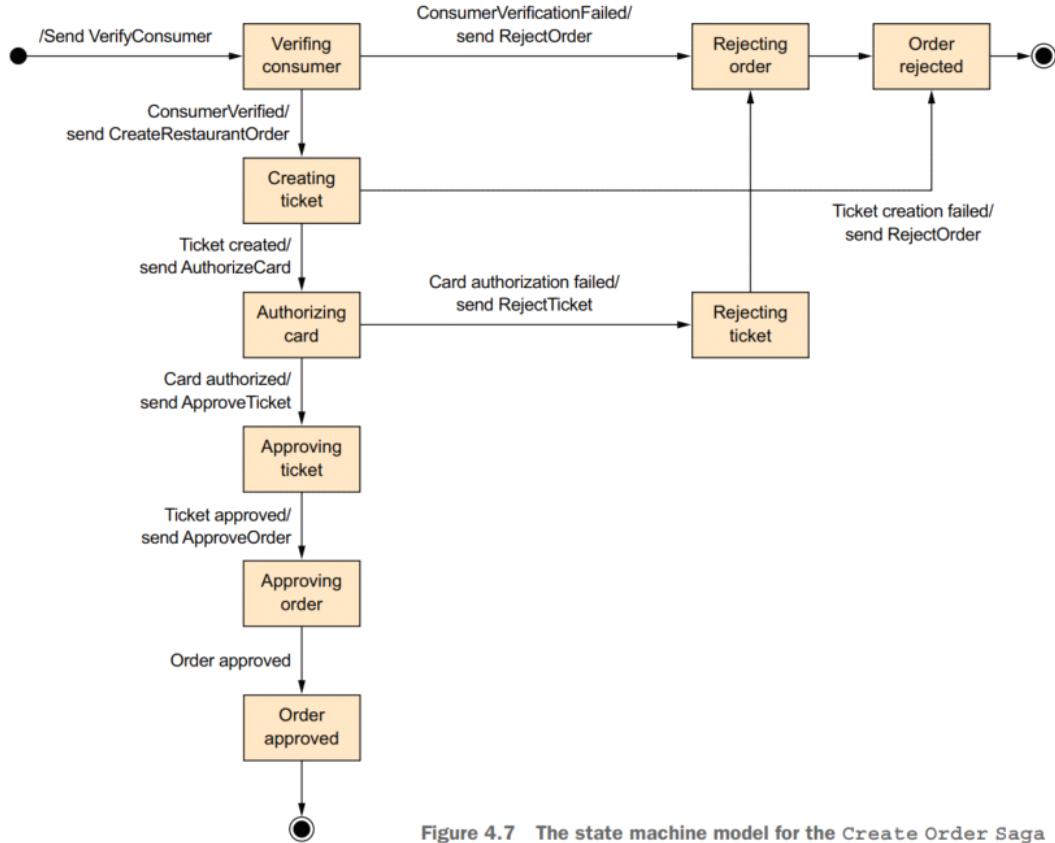


Figure 4.7 The state machine model for the Create Order Saga

Figure 4.7 shows the state machine model for the Create Order Saga. This state machine consists of numerous states, including the following:

- **Verifying Consumer**—The initial state. When in this state, the saga is waiting for the Consumer Service to verify that the consumer can place the order.
- **Creating Ticket**—The saga is waiting for a reply to the Create Ticket command.
- **Authorizing Card**—Waiting for Accounting Service to authorize the consumer's credit card.
- **Order Approved**—A final state indicating that the saga completed successfully.
- **Order Rejected**—A final state indicating that the Order was rejected by one of the participants.

* изолированность: аномалии и контрмеры

12 января 2021 г. 10:41

saga | saga не зватает поддержки изолированности

- Им не хватает поддержки изолированности, которая есть в ACID-транзакциях. В итоге приложение должно использовать так называемые контрмеры — методики проектирования, которые устраняют или снижают влияние аномалий конкурентности, вызванных нехваткой изолированности.
- Одна из проблем повествований связана с тем, что по своей природе они являются ACD (Atomicity, Consistency, Durability — «атомарность, согласованность, долговечность»).
- Я покажу, как использовать контрмеры, чтобы устранили или снизить влияние аномалий конкурентности, вызванных нехваткой изолированности между повествованиями.

isolation гарантирует, что результат параллельного выполнения нескольких ACID-транзакций будет таким же, как при некоем последовательном выполнении

- локальные обновления в микросервисах сразу же распространяются в виде событий по паутине микросервисов

- База данных дает иллюзию того, что каждая ACID-транзакция имеет эксклюзивный доступ к информации.
- Изолированность намного упрощает написание бизнес-логики, которая выполняется конкурентно.
- Разработчик обязан писать свои повествования так, чтобы избежать этих аномалий или минимизировать их влияние на бизнес (см контрмеры)
- Трудность в ходе работы с повествованиями состоит в том, что им не хватает изолированности ACID-транзакций. Дело тут в следующем: обновления, которые выполняет каждая локальная транзакция, сразу же фиксируются и становятся доступными любым другим повествованиям. Такое поведение может создать две проблемы.
 - Во-первых, другие повествования могут изменить данные, к которым обращается используемое в данный момент повествование.
 - Во-вторых, другие повествования могут читать данные в процессе их обновления, что чревато несогласованностью.

заменить параллельные транзакции на последовательные

- Нехватка изолированности может вызвать аномалии (этот термин встречается в литературе по базам данных). Так называется ситуация, когда параллельное выполнение транзакций дает иные результаты, чем последовательное

существует конфликт между высокой доступностью данных и изоляцией

- на первый взгляд отсутствие изолированности кажется неприемлемым. Но на практике разработчики часто уменьшают изолированность для получения более высокой производительности (так как параллельное выполнение всегда быстрее последовательного)
- разработчики транзакций должны выбрать как можно меньшую защиту от аномалий изоляции, чтобы получить как можно более высокую доступность

(1) anomaly: Lost updates

- Потеря обновлений — одно повествование перезаписывает изменения, внесенные другим, не читая их при этом
 - 1. Первый этап повествования Create Order создает заказ.
 - 2. Пока это повествование выполняется, повествование Cancel Order отменяет заказ.
 - 3. На завершающем этапе повествование Create Order подтверждает заказ.В этой ситуации Create Order игнорирует и перезаписывает обновление, выполненное повествованием Cancel Order. В итоге приложение FTGO отправит уже отмененный заказ.
- как предотвратить потерю обновлений.

(2) anomaly: Dirty reads

- «Грязное» чтение — транзакция или повествование читают незавершенные обновления другого повествования
- когда первая транзакция обновляет запись без фиксации обновления. После этого вторая транзакция считывает запись. Позднее первое обновление прерывается (или фиксируется), т. е. Вторая транзакция могла прочитать несуществующую версию записи.

(3) anomaly: Fuzzy/nonrepeatable reads

- Нечеткое/неповторяемое чтение — два разных этапа повествования читают одни и те же данные, но получают разные результаты, потому что другое повествование внесло изменения.
- когда первая транзакция читает запись без использования блокировок. Эта запись позже обновляется и фиксируется второй транзакцией до того, как первая транзакция будет зафиксирована или прервана. Другими словами, мы не можем полагаться на то, что прочитали.

каждый тип транзакций играет свою роль в контрмерах

- Особенno важно здесь различие между компенсируемыми и повторяемыми транзакциями. Как вы увидите сами, каждый тип транзакций играет свою роль в контрмерах

(A) Semantic lock—An application-level lock

- Семантическая блокировка — блокировка на уровне приложения.
- При использовании семантической блокировки компенсируемая транзакция устанавливает флаг во всех записях, которые она создает или обновляет. Он говорит о том, что запись не зафиксирована и может измениться. Это может быть либо блокировка, которая закрывает доступ к записи другим транзакциям, либо предупреждение о том, что данную запись следует перепроверять.
 - Флаг сбрасывается либо повторяемой (повествование успешно завершается),
 - либо компенсирующей транзакцией (повествование откатывается обратно).

использование в заказе состояний вида STATE=_PENDING

- Один из возможных подходов — использование в заказе состояний вида *_PENDING, таких как APPROVAL_PENDING.
- Повествование, обновляющее заказы, например Create Order, вначале устанавливает состояние в *_PENDING. Благодаря этому другие транзакции будут знать, что заказ обновляется повествованием, и смогут среагировать соответствующим образом
- semantic lock countermeasure (<https://dl.acm.org/citation.cfm?id=284472.284478>)
- Поле Order.state — отличный пример семантической блокировки. Для ее реализации используются состояния вида *_PENDING, такие как APPROVAL-PENDING и REVISION-PENDING. Всем, кто обращается к заказу, они говорят о том, что в данный момент он обновляется другим повествованием. Например, на первом этапе (который является компенсируемой транзакцией) повествование Create Order создает заказ с состоянием APPROVAL-PENDING.
 - На заключительном этапе (повторяемая транзакция) это поле меняется на APPROVED,
 - а компенсирующая транзакция присваивает ему значение REJECTED.

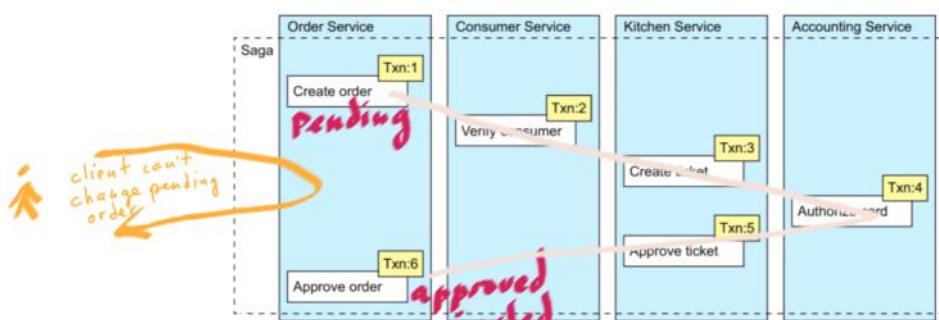


Figure 4.2 Creating an Order using a saga. The createOrder() operation is implemented by a saga that consists of local transactions in several services.

кликните повторить попытку позже когда заказ будет в базе approved/rejected

что делать с заблокированной записью: клиент просто повторит попытку позже

- Но управление блокировкой — это лишь половина проблемы. Вам также нужно решить, как каждое отдельное повествование будет обращаться с заблокированной записью. Рассмотрим в качестве примера системную команду cancelOrder(). С ее помощью клиент может отменить заказ, находящийся в состоянии APPROVAL-PENDING. Эту задачу можно решить несколькими способами. Системная команда cancelOrder() может просто отказать и посоветовать клиенту повторить попытку позже. Основное преимущество этого подхода — простая реализация. А недостаток в том, что клиент усложняется за счет логики

повторного вызова.

что делать с заблокированной записью: клиент просто повторит попытку позже: клиент заблокируется, до освобождения блокировки (аналог обычных ACID транзакций)

- В качестве еще одного варианта команда cancelOrder() может сама заблокироваться до снятия блокировки. Преимущество семантических блокировок состоит в том, что они, в сущности, **воссоздают уровень изолированности, обеспеченный ACID-транзакциями**. Повествования, обновляющие одну и ту же запись, сериализуются, что значительно упрощает написание кода.
- Еще одной положительной стороной является то, что **клиенту больше не нужно отвечать за повторные вызоы**.
- Однако при этом приложение должно управлять блокировками. Оно должно также реализовать **алгоритм обнаружения взаимного блокирования**, который откатывает повествование, чтобы снять блокировку, и выполняет его заново

(B) Commutative updates—Design update operations to be executable in any order

- коммутативная операция устраниет апдейты тк мы просто применяем еще одну операцию в любом порядке

- Коммутативные обновления — проектирование операций обновления таким образом, чтобы их можно было выполнить в любом порядке
- Простой и понятной контрмерой является проектирование коммутативных операций обновления. Операции называют коммутативными, если их можно выполнить в любом порядке. В качестве примера можно привести команды debit () и credit () из сервиса Accounting (если не брать во внимание проверки перерасхода средств). Это полезная контрмера, так как она устраняет множество обновлений.
- Представьте себе сценарий, в котором повествование нужно откатить после того, как компенсируемая транзакция уже сняла (или возместила) средства со счета. Компенсирующая транзакция может просто возместить (или снять) нужную сумму, чтобы отменить обновление. Возможность того, что это перезапишет обновления, сделанные другими повествованиями, отсутствует.

(C) Pessimistic view—Reorder the steps of a saga to minimize business risk

- Пессимистическое представление — **перестановка этапов повествования для минимизации бизнес-рисков**

? непонятен пример

consumers have a credit limit. In this application, a saga that cancels an order consists of the following transactions:

- Consumer Service—Increase the available credit.
- Order Service—Change the state of the Order to cancelled.
- Delivery Service—Cancel the delivery.

Let's imagine a scenario that interleaves the execution of the Cancel Order and Create Order Sagas, and the Cancel Order Saga is rolled back because it's too late to cancel the delivery. It's possible that the sequence of transactions that invoke the Consumer Service is as follows:

- 1 Cancel Order Saga—Increase the available credit.
- 2 Create Order Saga—Reduce the available credit.
- 3 Cancel Order Saga—A compensating transaction that reduces the available credit.

In this scenario, the Create Order Saga does a dirty read of the available credit that enables the consumer to place an order that exceeds their credit limit. It's likely that this is an unacceptable risk to the business.

Another way to deal with the lack of isolation is the *pessimistic view* countermeasure. It reorders the steps of a saga to minimize business risk due to a dirty read. Consider, for example, the scenario earlier used to describe the dirty read anomaly. In that scenario, the Create Order Saga performed a dirty read of the available credit and created an

order that exceeded the consumer credit limit. To reduce the risk of that happening, this countermeasure would reorder the Cancel Order Saga:

- 1 Order Service—Change the state of the Order to cancelled.
- 2 Delivery Service—Cancel the delivery.
- 3 Customer Service—Increase the available credit.

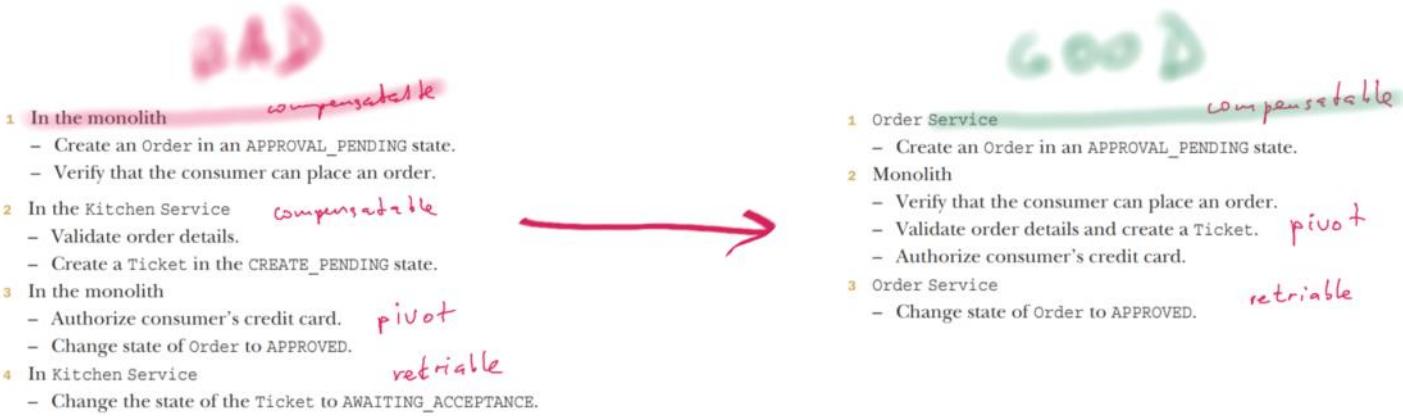
In this reordered version of the saga, the available credit is increased in a retriable transaction, which eliminates the possibility of a dirty read.

Если же все транзакции являются либо pivot, либо retriable, ему никогда не нужно будет ничего компенсировать

- Но извлечение сервисов вполне можно спланировать так, чтобы ваши повествования не **нуждались в реализации компенсирующих транзакций**.
- Дело в том, что это необходимо, только последующие транзакции монолита могут завершиться неудачно
- Если же все транзакции монолита являются либо поворотными, либо повторяемыми, ему никогда не нужно будет ничего компенсировать
- Если все повествования, которые вы напишете при извлечении сервиса, будут иметь такую структуру, вам придется сделать намного меньше изменений в монолите. Более того, извлечение сервисов можно тщательно спланировать в таком порядке, чтобы все транзакции монолита были либо поворотными, либо повторяемыми.

monolith variant 1

- 1 Validate order details.
- 2 Verify that the consumer can place an order.
- 3 Authorize consumer's credit card.
- 4 Create an Order.



Это повествование состоит из трех локальных транзакций: одна в монолите и две в сервисе Order.

- Первая транзакция (в сервисе Order) создает заказ с состоянием APPROVAL-PENDING.
- Вторая транзакция (в монолите) проверяет, может ли клиент размещать заказы, авторизует его банковскую карту и создает заявку.
- Третья (опять в сервисе Order) меняет состояние заказа на APPROVED.
- Транзакция монолита является pivot-поворотной для этого повествования, его точкой невозврата. Если она завершится успешно, повествование доработает до самого конца.
- Проблемы могут возникнуть только с первыми двумя этапами.
- Третья транзакция не может отказать, поэтому монолиту никогда не придется откатывать вторую транзакцию.
- В результате вся сложность поддержки компенсируемых транзакций ложится на сервис Order, который тестировать намного легче, чем монолит

Планирование извлечения сервисов, чтобы избежать реализации компенсирующих транзакций в монолите

Планирование извлечения сервисов, чтобы избежать реализации компенсирующих транзакций в монолите

Как мы только что видели, извлечение сервиса Kitchen требует от монолита реализации компенсирующих транзакций, а извлечение сервиса Order — нет. Это говорит о том, что порядок, в котором извлекаются сервисы, имеет значение. Если тщательно его спланировать, можно избежать внесения масштабных изменений в монолит для поддержки компенсируемых транзакций. Мы можем сделать так, чтобы все транзакции в монолите были либо поворотными, либо повторяемыми. Например, если извлечь из монолита FTGO сначала сервис Order, а затем Consumer, это упростит извлечение сервиса Kitchen. Давайте посмотрим, как это делается.

После извлечения сервиса Consumer команда `createOrder()` использует следующее повествование.

1. Сервис Order — создать заказ с состоянием APPROVAL_PENDING.
2. Сервис Consumer — убедиться в том, что клиент может размещать заказы.
3. Монолит:
 - проверить детали заказа и создать заявку;
 - авторизовать банковскую карту клиента.
4. Сервис Order — изменить состояние заказа на APPROVED.

В этом повествовании транзакция монолита является поворотной. Компенсируемую транзакцию реализует сервис Order.

Вслед за Consumer мы можем извлечь сервис Kitchen. Когда мы это сделаем, команда `createOrder()` будет использовать следующее повествование.

1. Сервис Order — создать заказ с состоянием APPROVAL_PENDING.
2. Сервис Consumer — убедиться в том, что клиент может размещать заказы.
3. Сервис Kitchen — проверить детали заказа и создать заявку с состоянием PENDING.
4. Монолит — авторизовать банковскую карту клиента.
5. Сервис Kitchen — изменить состояние заявки на APPROVED.
6. Сервис Order — изменить состояние заказа на APPROVED.

В этом повествовании транзакция монолита по-прежнему остается поворотной. Компенсируемые транзакции реализуются сервисами Order и Kitchen.

Мы можем продолжить извлечение монолита и извлечь сервис Accounting.

б. Сервис Order — изменить состояние заказа на APPROVED.

В этом повествовании транзакция монолита по-прежнему остается поворотной.
Компенсируемые транзакции реализуются сервисами Order и Kitchen.

Мы можем продолжить рефакторинг монолита и извлечь сервис Accounting. Если сделаем это, команда `createOrder()` будет использовать такое повествование.

1. Сервис Order — создать заказ с состоянием APPROVAL_PENDING.
2. Сервис Consumer — убедиться в том, что клиент может размещать заказы.
3. Сервис Kitchen — проверить детали заказа и создать заявку с состоянием PENDING.

13.3. Проектирование взаимодействия между сервисом и монолитом **523**

4. Сервис Accounting — авторизовать банковскую карту клиента.
5. Сервис Kitchen — изменить состояние заявки на APPROVED.
6. Сервис Order — изменить состояние заказа на APPROVED.

Как видите, тщательное планирование порядка извлечения позволяет избежать использования повествований, которые требуют внесения сложных изменений в монолит. Теперь посмотрим, как обеспечить безопасность при переходе на микросервисы.

(D) **Reread value**— Prevent dirty writes by rereading data to verify that it's unchanged before overwriting it.

разновидность шаблона «Оптимистичная автономная блокировка»

- Повторное чтение значения — предотвращение «грязного» чтения путем повторного считывания данных. Это позволяет убедиться в их неизменности перед тем, как их перезаписывать
- Повторное чтение значения предотвращает потерю обновлений. Повествование, использующее эту контрмеру, повторно считывает запись перед ее обновлением, убеждается в том, что та не изменилась, и только потом обновляет. Если запись изменилась, повествование прекращает работу и, возможно, запускается заново.
- Повествование Create Order может применять эту контрмеру для сценария, в котором заказ отменяется в процессе подтверждения. Транзакция, подтверждающая заказ, проверяет, не изменился ли он с момента создания в текущем повествовании. Не обнаружив изменений, транзакция подтверждает заказ. Но если заказ был отменен, транзакция прерывает повествование, в результате чего выполняются его компенсирующие транзакции.

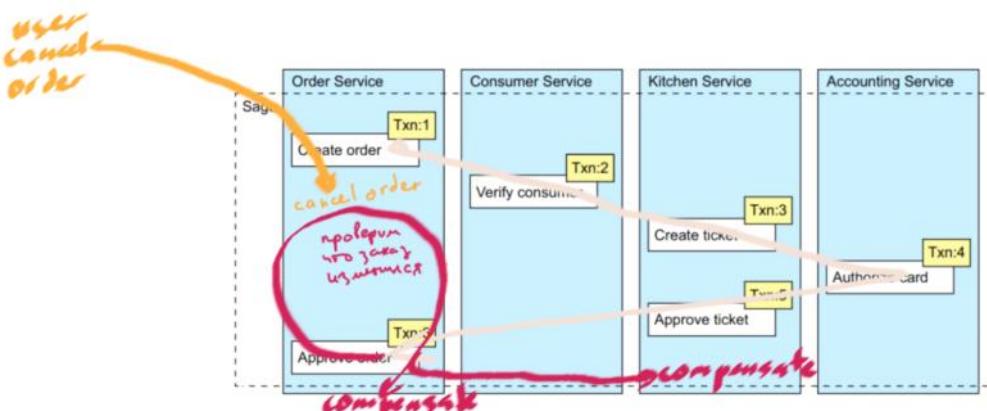


Figure 4.2 Creating an Order using a saga. The `createOrder()` operation is implemented by a saga that consists of local transactions in several services.

- Ситуация, когда два запроса или больше одновременно обновляют один агрегат, не такая уж и редкость. Приложения, которые используют традиционную модель хранения данных, часто применяют оптимистичное блокирование, чтобы транзакции не перезаписывали изменения друг друга. Чтобы определить, изменился ли агрегат с тех пор, как он был прочитан, оптимистичное блокирование задействует столбец с версией. Приложение накладывает корень агрегата на таблицу со столбцом `VERSION`, который инкрементируется при каждом обновлении записи. Приложение обновляет агрегат с помощью выражения `UPDATE`, например
- Выражение `UPDATE` будет успешным, только если версия не изменилась с момента, когда приложение в последний раз считывало агрегат. Если агрегат считывается двумя транзакциями, успешно завершится только та, которая первой выполнит обновление. Вторая будет отменена, поскольку номер версии изменился, благодаря этому она не перезапишет

изменения, внесенные первой транзакцией.

```
UPDATE AGGREGATE_ROOT_TABLE  
SET VERSION = VERSION + 1  
WHERE VERSION = <original version>
```

(E) Version file—Record the updates to a record so that they can be reordered.

способ превращения некоммутативных обновлений в коммутативные

- Файл версий — ведение записей об обновлениях, чтобы их можно было менять местами сначала записать все операции в history file те в файл содержащий историю всех операций
- рассмотрим сценарий, в котором повествования Create Order и Cancel Order выполняются параллельно. Если здесь не применяется семантическая блокировка, существует вероятность того, что повествование Cancel Order отменит авторизацию банковской карты заказчика до того, как Create Order ее авторизует. Чтобы справиться с этими перепутанными запросами, сервис Accounting может записывать операции по мере поступления и затем выполнять их в правильном порядке. В рассматриваемом случае он сначала запишет запрос Cancel Authorization. Затем, получив запрос Authorize Card (он просмотрит файл истории операций и далее решит что делать с новой операцией), просто пропустит авторизацию банковской карты, так как у него уже есть информация о получении запроса Cancel Authorization.

(F) By value—Use each request's business risk to dynamically select the concurrency mechanism.

например только для финансовых операций допускаются XA-транзакции

- По значению — использование бизнес-рисков каждого запроса для динамического выбора механизма конкурентности
- это стратегия выбора механизмов конкурентности на основе бизнес-рисков. Приложение, которое ее применяет, использует свойства всех запросов, чтобы сделать выбор между повествованиями и распределенными транзакциями.
 - Таким образом, запросы с низким уровнем риска выполняются в виде повествований и, возможно, с помощью контрмер, описанных в предыдущем разделе.
 - Но запросы с повышенным риском (например, связанные с большими суммами денег) задействуют распределенные транзакции. Благодаря этой стратегии приложение может динамически искать баланс между бизнес-рисками, доступностью и масштабируемостью.

* STRUCTURE OF A SAGA

12 января 2021 г. 12:23

Table 4.1 The compensating transactions for the Create Order Saga

Step	Service	Transaction	Compensating transaction
1	Order Service	createOrder()	rejectOrder()
2	Consumer Service	verifyConsumerDetails()	—
3	Kitchen Service	createTicket()	rejectTicket()
4	Accounting Service	authorizeCreditCard()	—
5	Kitchen Service	approveTicket()	—
6	Order Service	approveOrder()	—

Compensable transactions:
Must support roll back

Step	Service	Transaction	Compensation Transaction
1	Order Service	createOrder()	rejectOrder()
2	Consumer Service	verifyConsumerDetails()	-
3	Kitchen Service	createTicket()	rejectTicket()
4	Accounting Service	authorizeCreditCard()	-
5	Restaurant Order Service	approveRestaurantOrder()	-
6	Order Service	approveOrder()	-

Pivot transactions:
The saga's go/no-go transaction.
If it succeeds, then the saga runs to completion.

Retriable transactions:
Guaranteed to complete

Figure 4.8 A saga consists of three different types of transactions: compensatable transactions, which can be rolled back, so have a compensating transaction, a pivot transaction, which is the saga's go/no-go point, and retriable transactions, which are transactions that don't need to be rolled back and are guaranteed to complete.

пример компенсирующего пути

- 1 Order Service—Create an Order in an APPROVAL_PENDING state.
- 2 Consumer Service—Verify that the consumer can place an order.
- 3 Kitchen Service—Validate order details and create a Ticket in the CREATE_PENDING state.
- 4 Accounting Service—Authorize consumer's credit card, which fails.
- 5 Kitchen Service—Change the state of the Ticket to CREATE_REJECTED.
- 6 Order Service—Change the state of the Order to REJECTED.

compensatable transactions - их можно откатить при наличии компенсирующих

транзакций

- если операцию ненужно изменять она все равно считается компенсируемой
- первые три этапа повествования Create Order называются доступными для компенсации транзакциями, потому что шаги, следующие за ними, могут отказать
- Транзакции, доступные для компенсации, — транзакции, которые потенциально можно откатить с помощью компенсирующих транзакций.
- В рамках саги Create Order компенсируемыми транзакциями являются этапы `createOrder()`, `verifyConsumerDetails()` и `CreateTicket()`.
 - У операций `createOrder()` и `CreateTicket()` есть компенсирующие транзакции, которые отменяют их обновления.
 - Операция `verifyConsumerDetails()` выполняет лишь чтение, поэтому ее не нужно компенсировать

pivot transaction - решающий момент саги

- Четвертый этап называется поворотной транзакцией, потому что дальнейшие шаги никогда не отказывают
- Поворотная транзакция — решающий момент в повествовании. Если поворотная транзакция фиксируется, повествование отработает до конца. Поворотная транзакция может оказаться недоступной ни для компенсации, ни для повторения. Это также может быть последняя компенсируемая или первая повторяемая транзакция
- `authorizeCreditCard()` — это поворотная транзакция в данном повествовании. Если банковскую карту заказчика удается авторизовать, завершение повествования гарантировано.

retriable transactions - их не нужно откатывать, и они всегда завершаются

- Последние два этапа называются доступными для повторения транзакциями, потому что они всегда заканчиваются успешно
- Транзакции, доступные для повторения, — транзакции, идущие за поворотной. Всегда завершаются успешно
- За поворотной транзакцией следуют операции `approveTicket()` и `approveOrder()`, которые можно повторить

* транзакции и DDD

25 января 2021 г. 11:02

каждая операция в интерфейсе соответствует только одной транзакции

- The domainbased design within a microservice ensures that each operation at the interface only corresponds to one transaction.
- Доменная структура микросервиса гарантирует, что каждая операция в интерфейсе соответствует только одной транзакции

Транзакции, которые содержат как операции чтения, так и записи, сложно реализовать.

- Операции чтения и записи могут быть реализованы в различных микросервисах. Это может означать, что очень сложно объединить операции в одну транзакцию, поскольку транзакции между микросервисами обычно невозможны.