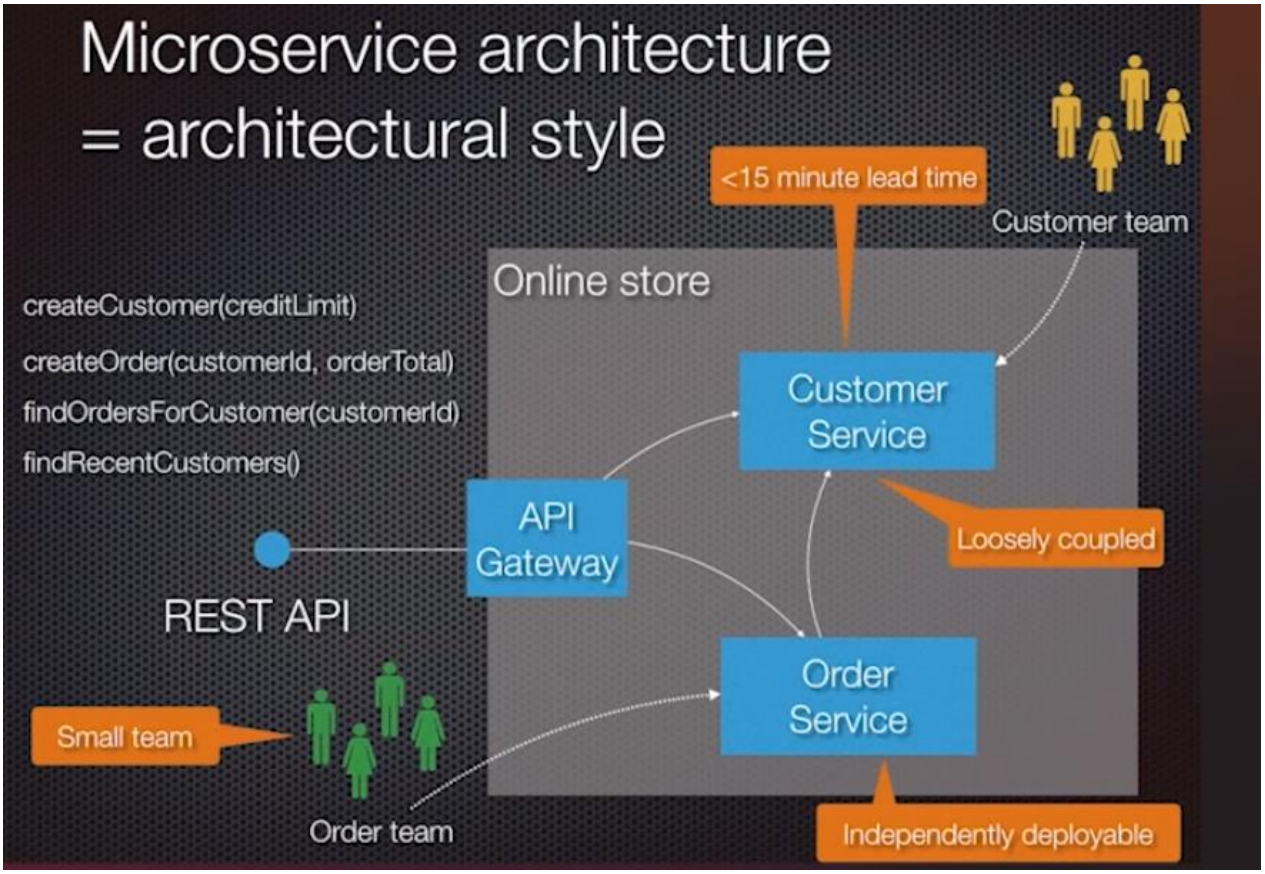
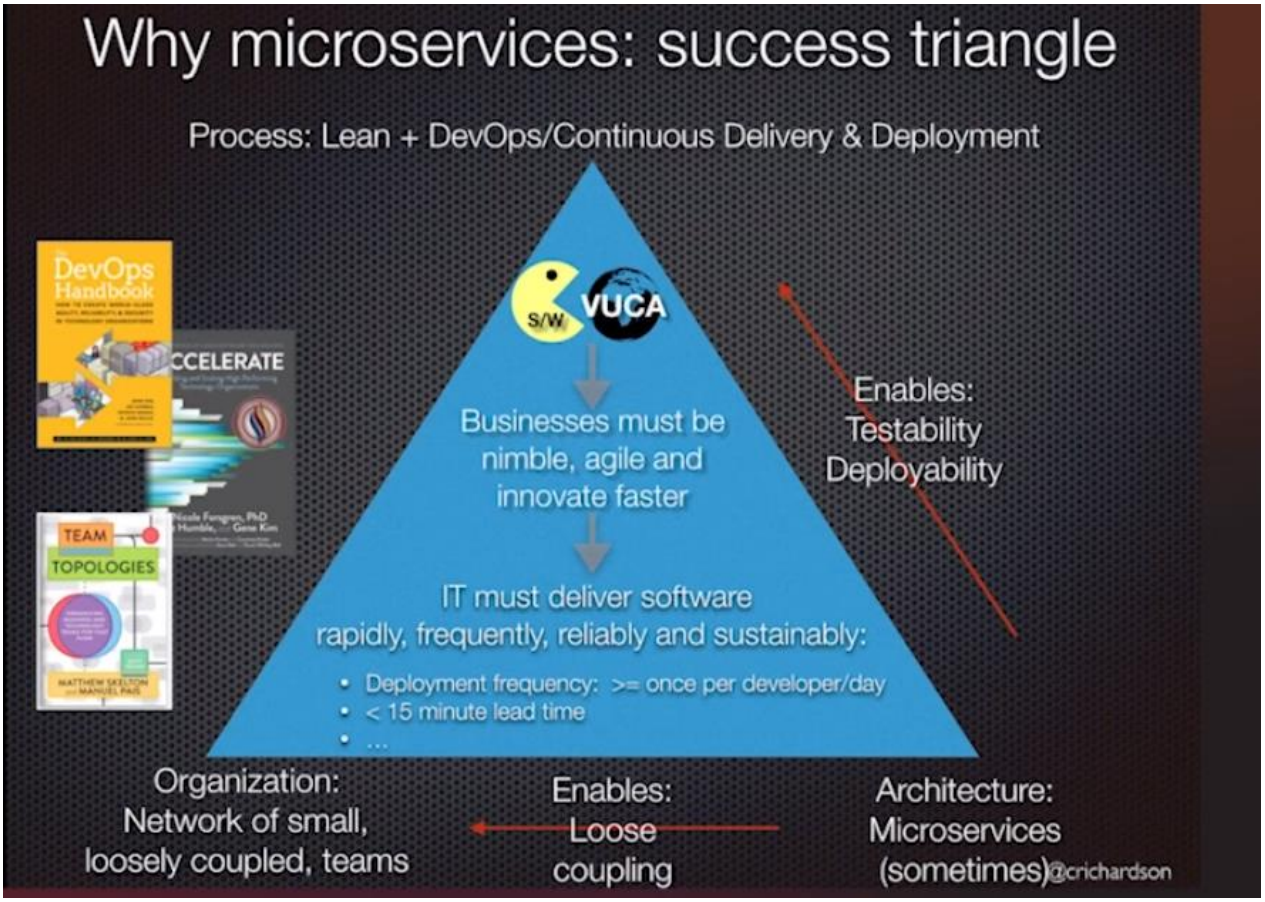
To deliver a large complex application rapidly, frequently and reliably, you often must use the microservice architecture. The microservice architecture is an architectural style that structures the application as a collection of loosely coupled services. One challenge with using microservices is that in order to be loosely coupled each service has its own private database. As a result, implementing transactions and queries that span services is no longer straightforward.

In this EDA Summit presentation, you will learn how event-driven microservices address this challenge. Chris Richardson, creator of microservices.io, describes how to use sagas, which is an asynchronous messaging-based pattern, to implement transactions that span services. You will learn how to implement queries that span services using the CQRS pattern, which maintain easily queryable replicas using events.


Microservice architecture
= architectural style
<15 minute lead time
Customer team
Online store
createCustomer(creditLimit)
createOrder(customerId, orderTotal)
findOrdersForCustomer(customerId)
findRecentCustomers()
Customer Service
API Gateway
Loosely coupled
REST API
Order Service
Small team
Order team
Independently deployable


Why microservices: success triangle
Process: Lean + DevOps/Continuous Delivery & Deployment
S/W
VUCA
Businesses must be nimble, agile and innovate faster
Enables: Testability Deployability
IT must deliver software rapidly, frequently, reliably and sustainably:
Deployment frequency: >= once per developer/day
< 15 minute lead time
...
Organization: Network of small, loosely coupled, teams
Enables: Loose coupling
Architecture: Microservices (sometimes)
@crichardson

Loose coupling is essential

Services collaborate, e.g. Order Service must reserve customer credit

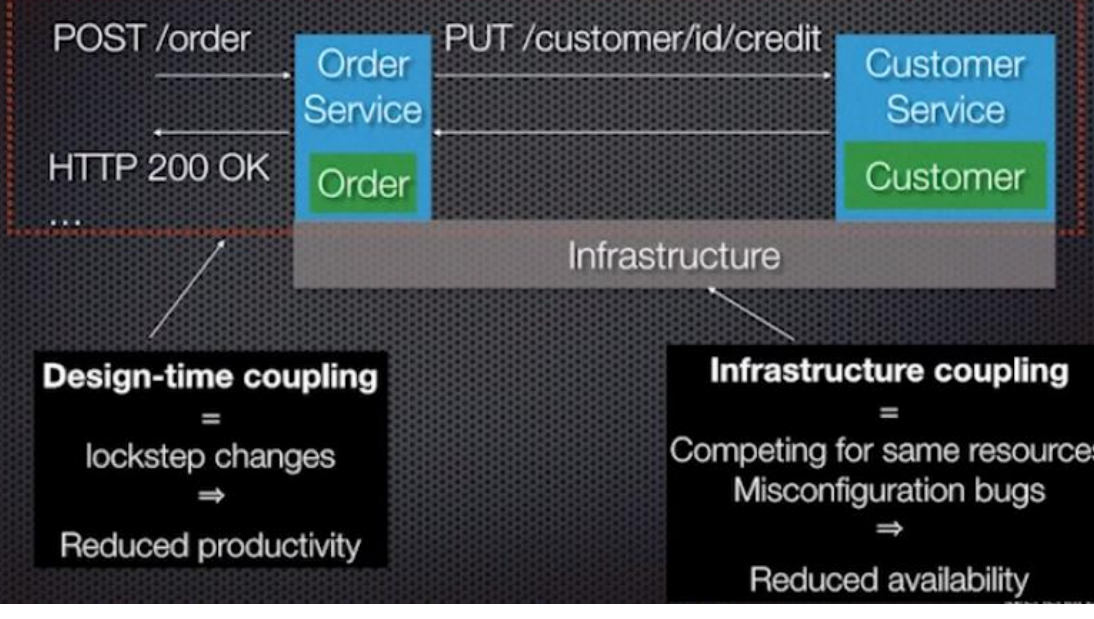
⇒

Some coupling - degree of connectedness - is inevitable

BUT

You must design services to be loosely coupled

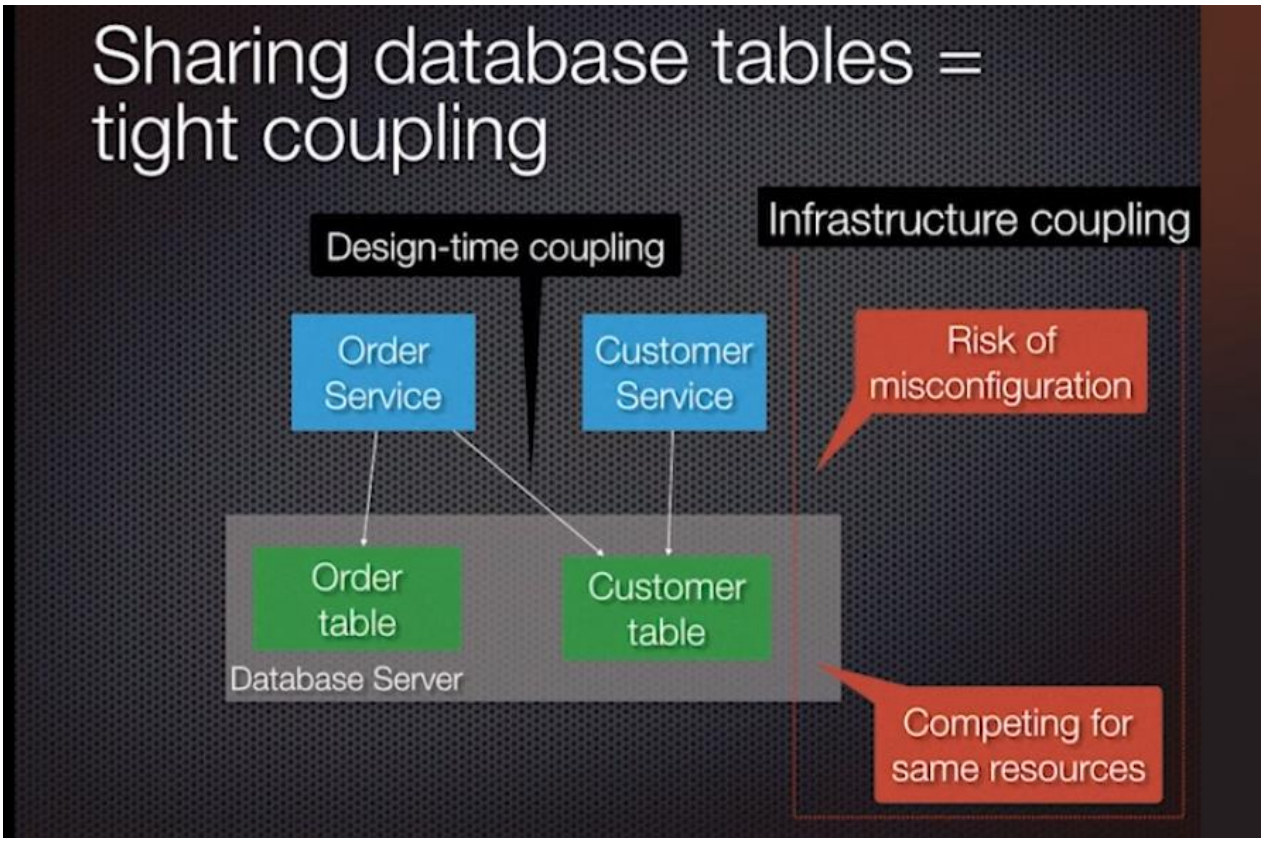
Runtime coupling
=
Both services must be available
⇒
Reduced availability

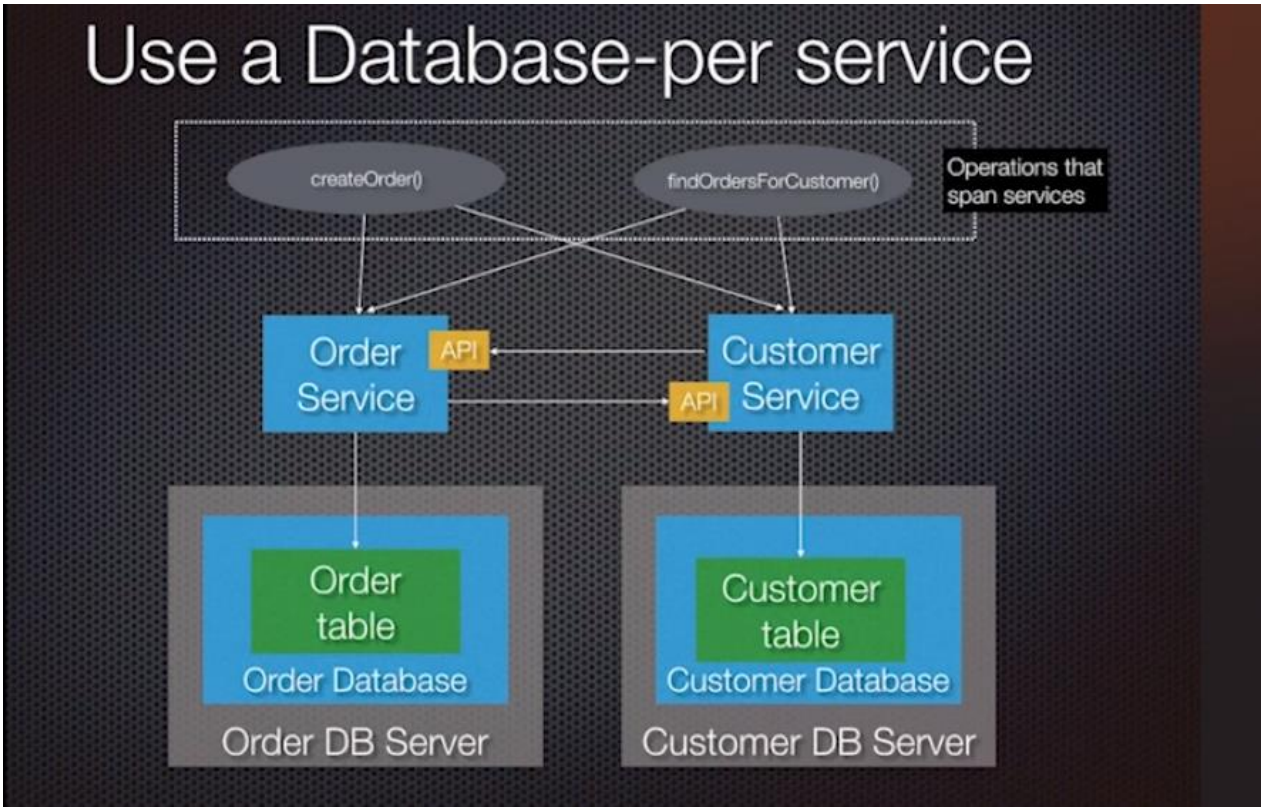


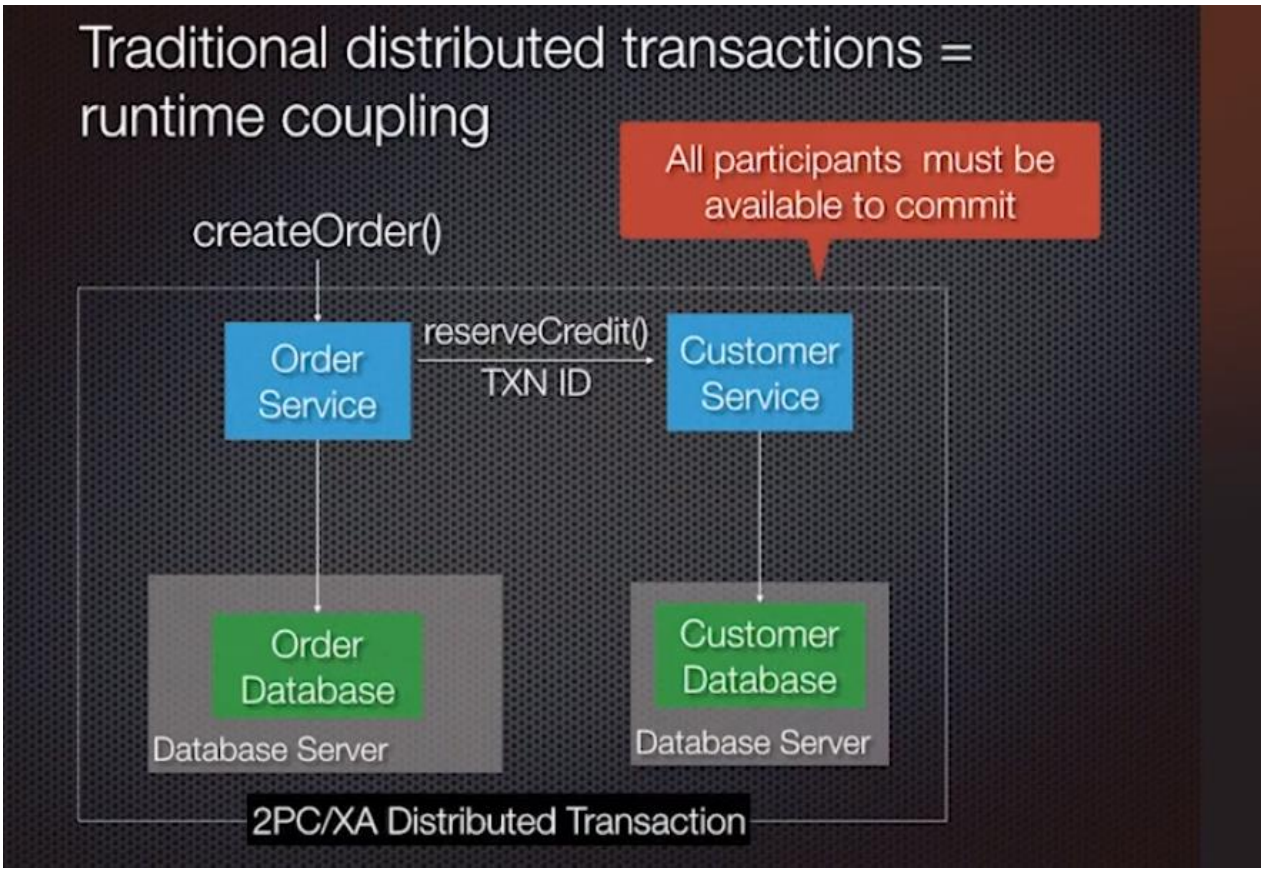
Design-time coupling
=
lockstep changes
⇒
Reduced productivity

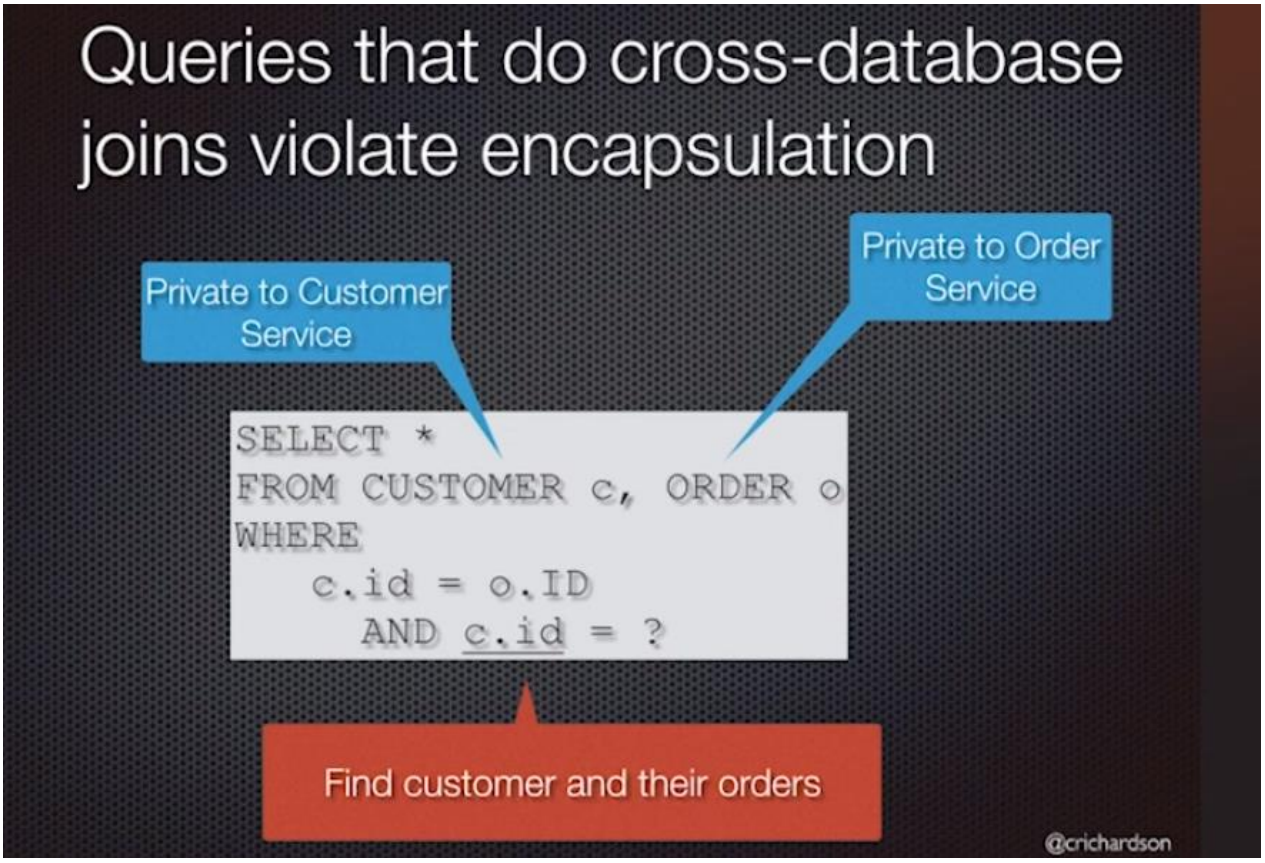
Infrastructure coupling
=
Competing for same resources
Misconfiguration bugs
⇒
Reduced availability

- Distributed data challenges in a microservice architecture


Sharing database tables = tight coupling
Design-time coupling
Infrastructure coupling
Order Service
Customer Service
Risk of misconfiguration
Order table
Customer table
Database Server
Competing for same resources


Use a Database-per service
createOrder()
findOrdersForCustomer()
Operations that span services
Order Service
API
Customer Service
API
Order table
Order Database
Order DB Server
Customer table
Customer Database
Customer DB Server

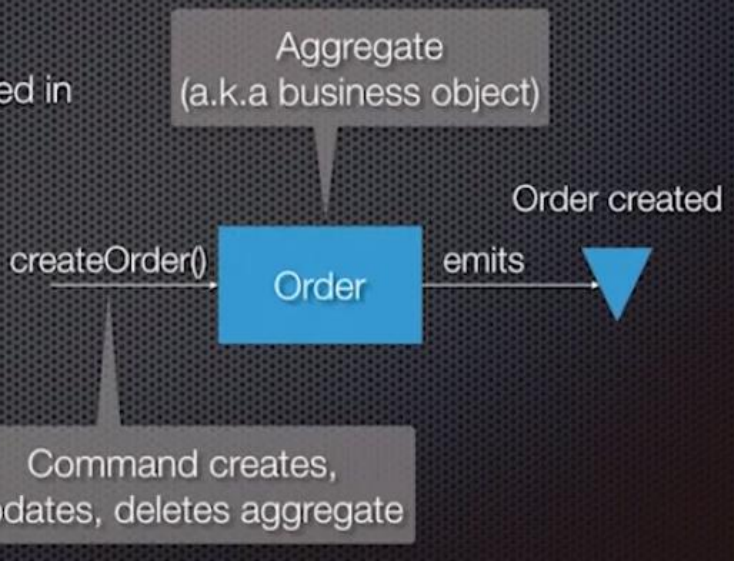

Traditional distributed transactions = runtime coupling
All participants must be available to commit
createOrder()
Order Service
reserveCredit()
TXN ID
Customer Service
Order Database
Database Server
Customer Database
Database Server
2PC/XA Distributed Transaction


Queries that do cross-database joins violate encapsulation
Private to Customer Service
Private to Order Service
SELECT *
FROM CUSTOMER c, ORDER o
WHERE
c.id = o.ID
AND c.id = ?
Find customer and their orders
@crichardson

Events to the rescue

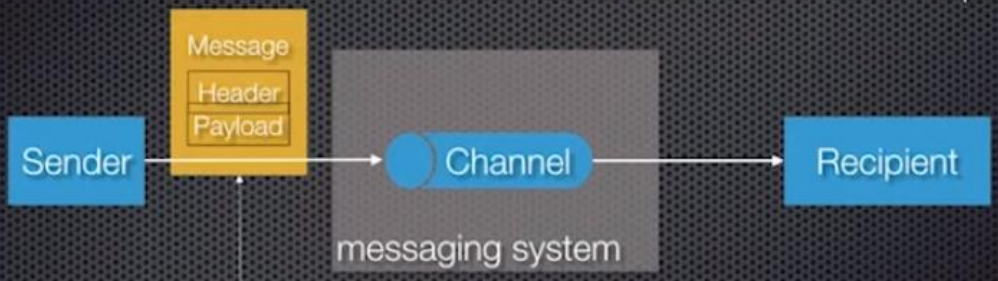
An event is something notable that has occurred in a domain, e.g.

- Order Created
- Order Confirmed
- Order Cancelled
- ..

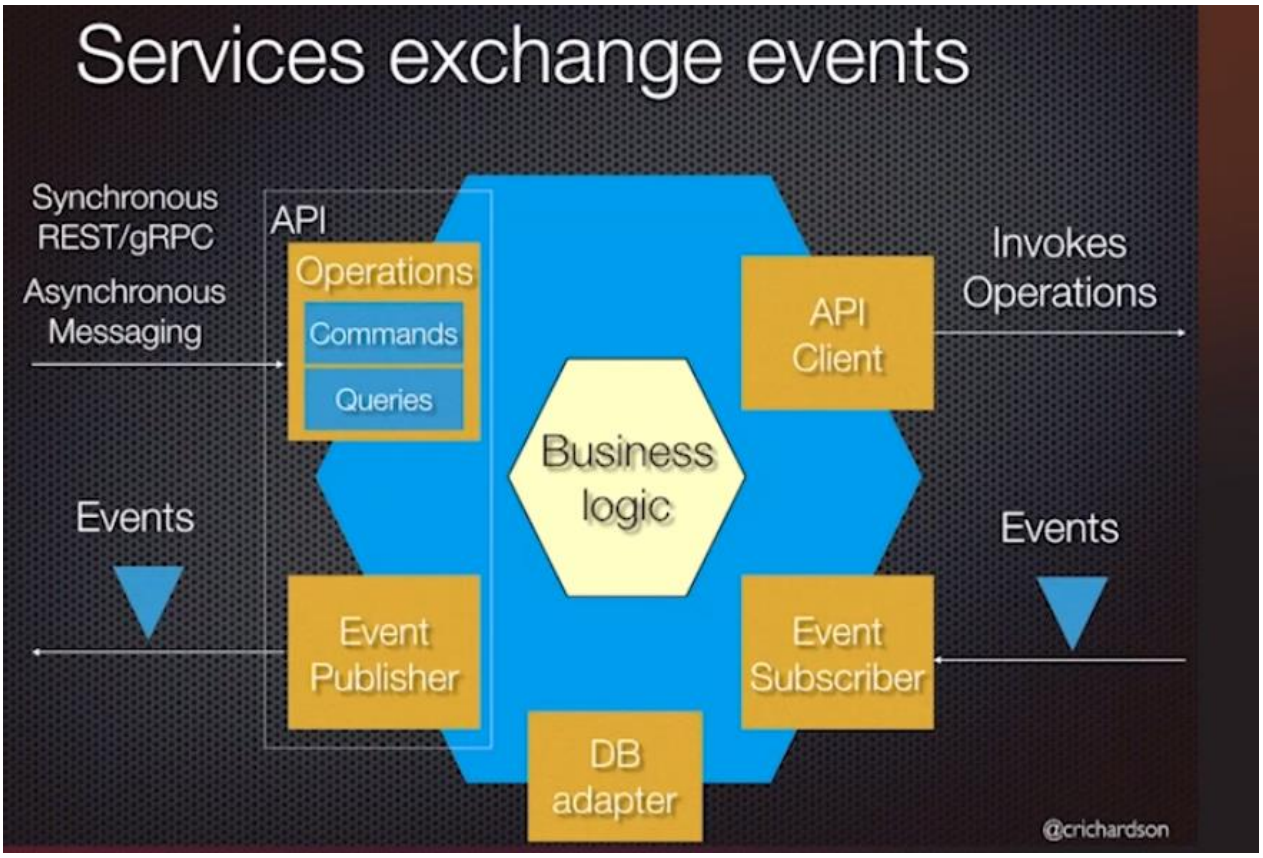


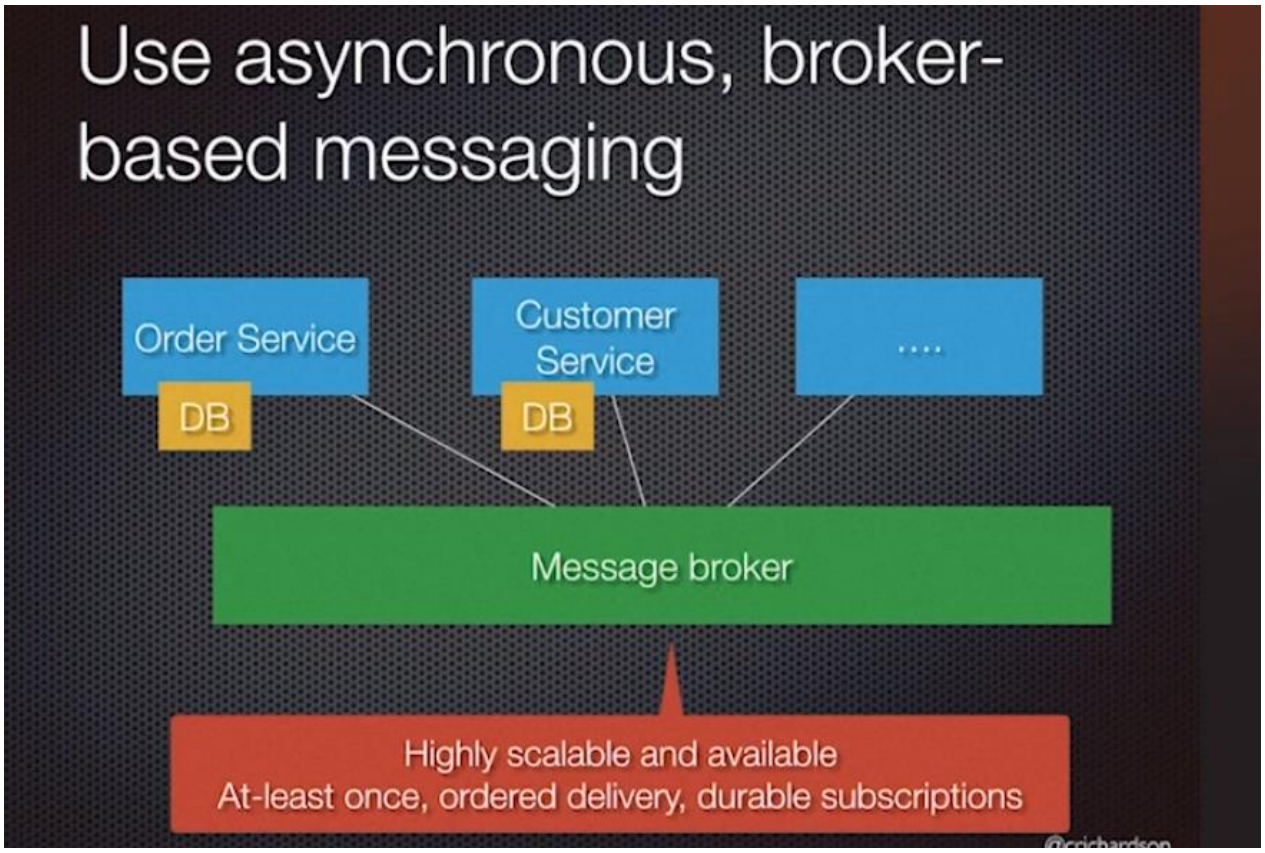
An event is a type of message

Channel types:
- Point-to-point - deliver to one recipient
- Publish-subscribe - deliver to all recipients

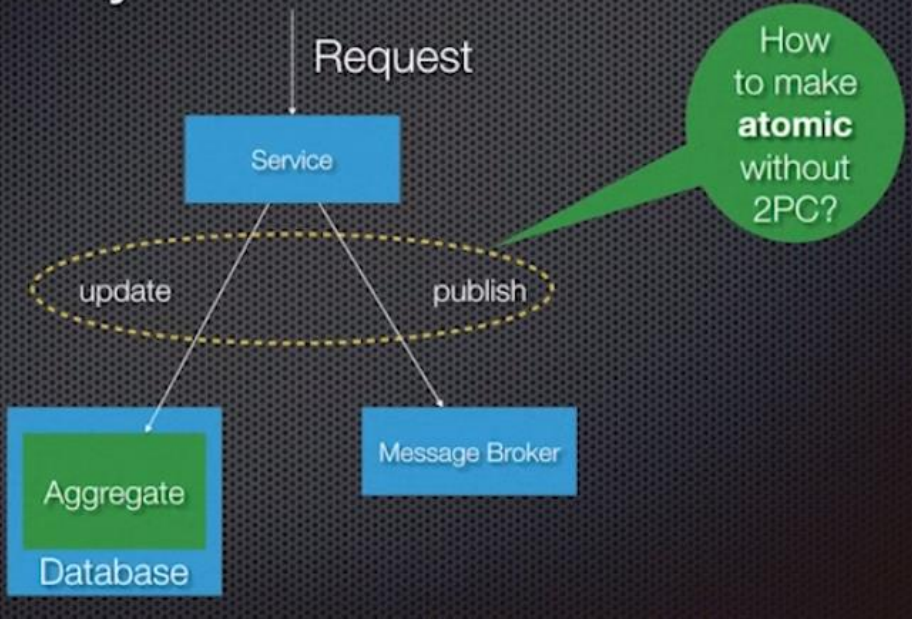


Message types:
- Message - generic
- **Event - something that has happened**
- Command - a request to perform an operation
- Reply - a reply to a command
- ...

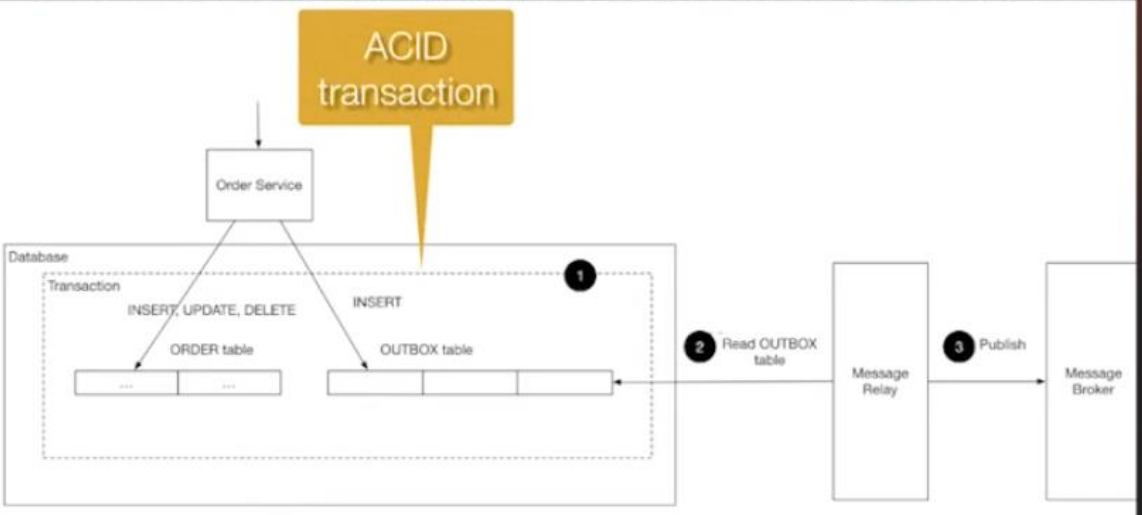

Services exchange events
Synchronous
REST/gRPC
Asynchronous
Messaging
API
Operations
Commands
Queries
Business
logic
API
Client
Invokes
Operations
Events
Events
Event
Publisher
Event
Subscriber
DB
adapter
@crichardson


Use asynchronous, broker-based messaging
Order Service
DB
Customer
Service
DB
....
Message broker
Highly scalable and available
At-least once, ordered delivery, durable subscriptions

Reliable messaging requires atomicity



Atomically updating state and publishing events



https://microservices.io/patterns/data/transactional-outbox.html

- Transactions in a microservice architecture

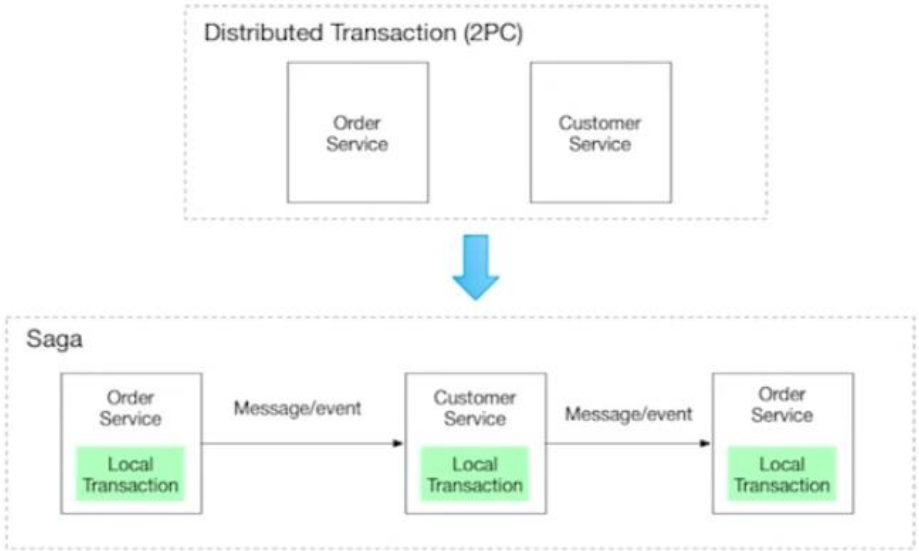
From a 1987 paper

SAGAS

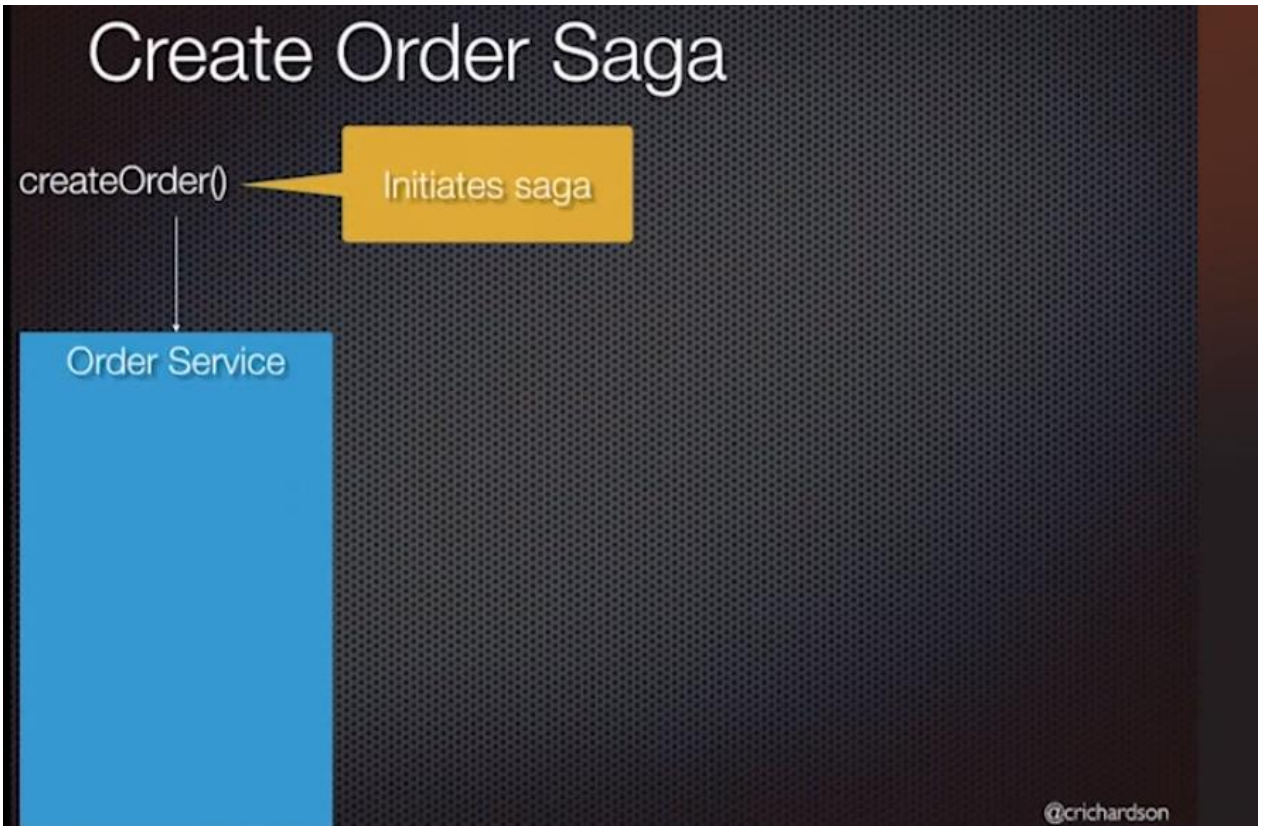
Hector Garcia-Molina
Kenneth Salem

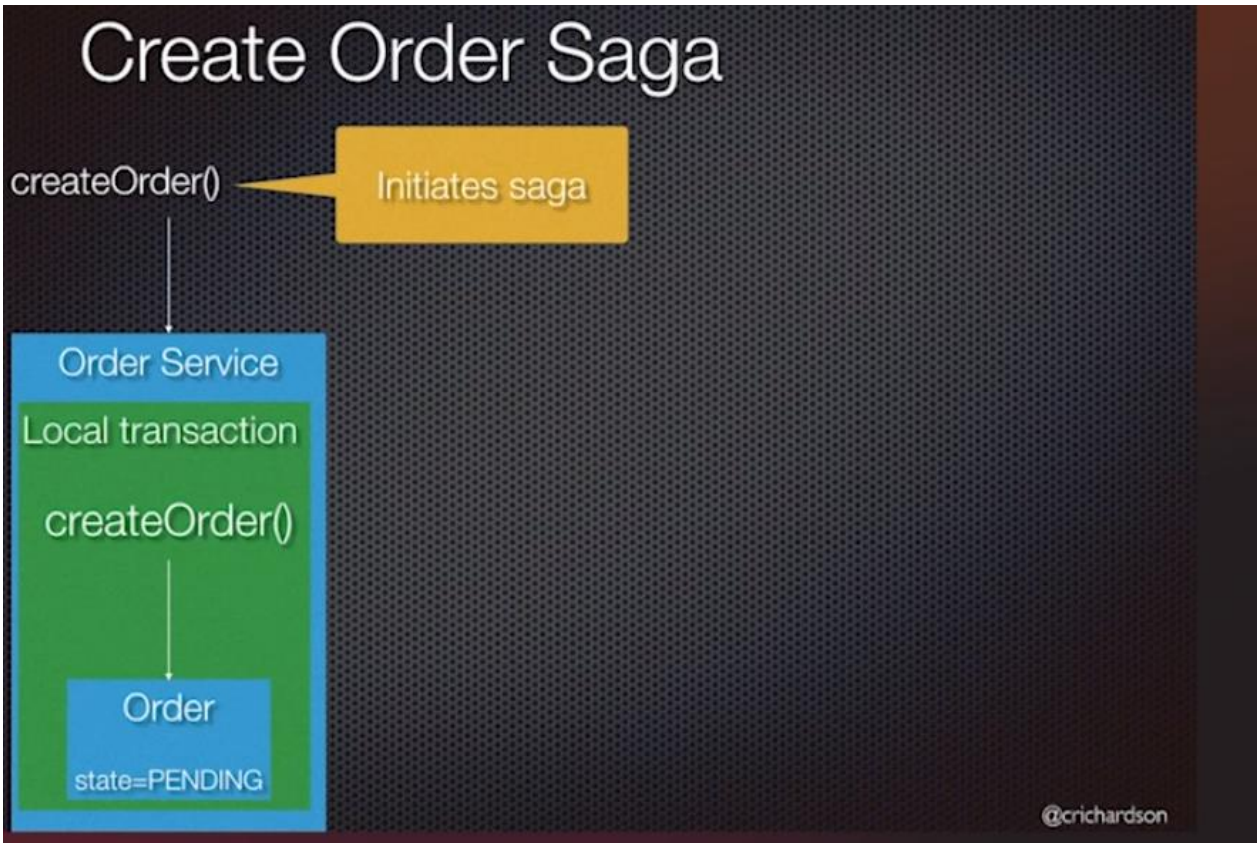
Department of Computer Science
Princeton University
Princeton, N J 08544

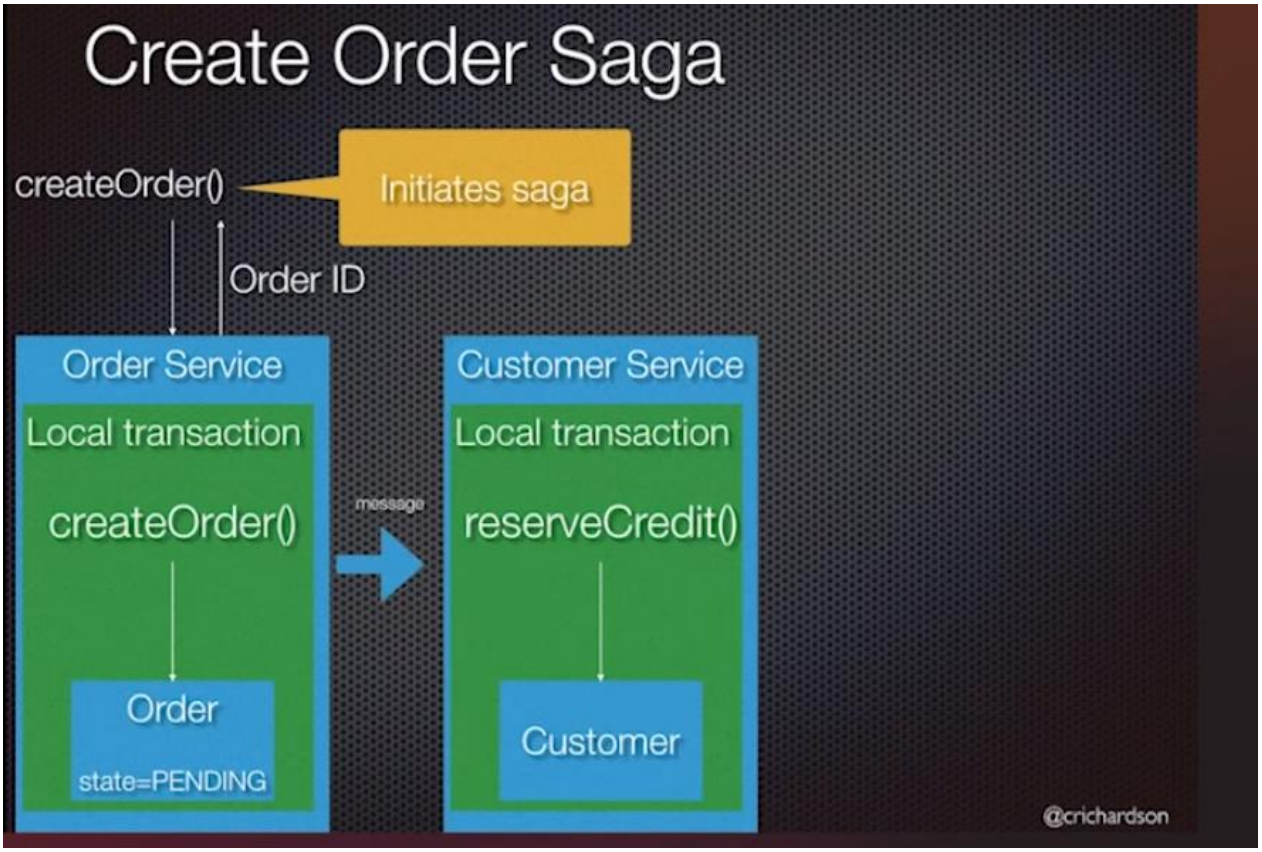
Sagas: event-based transactions

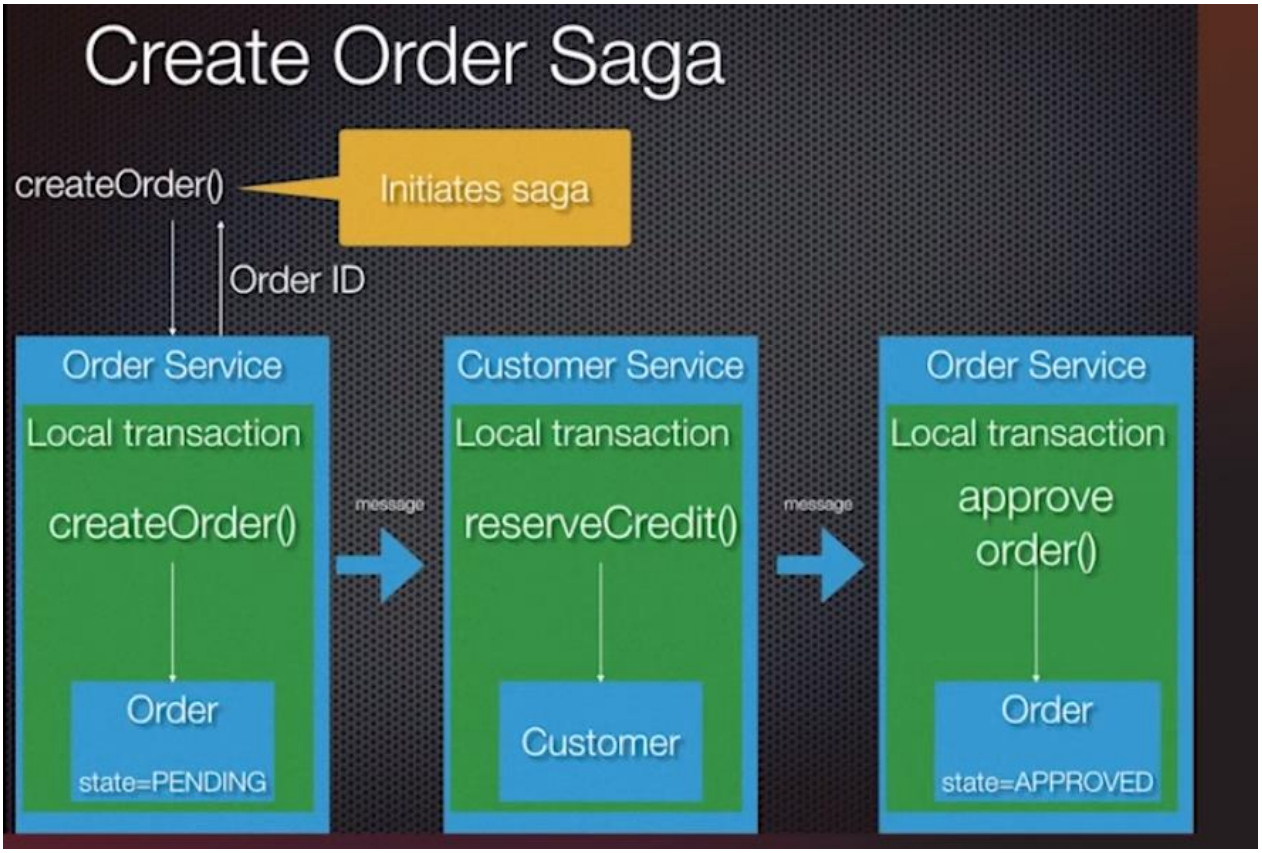


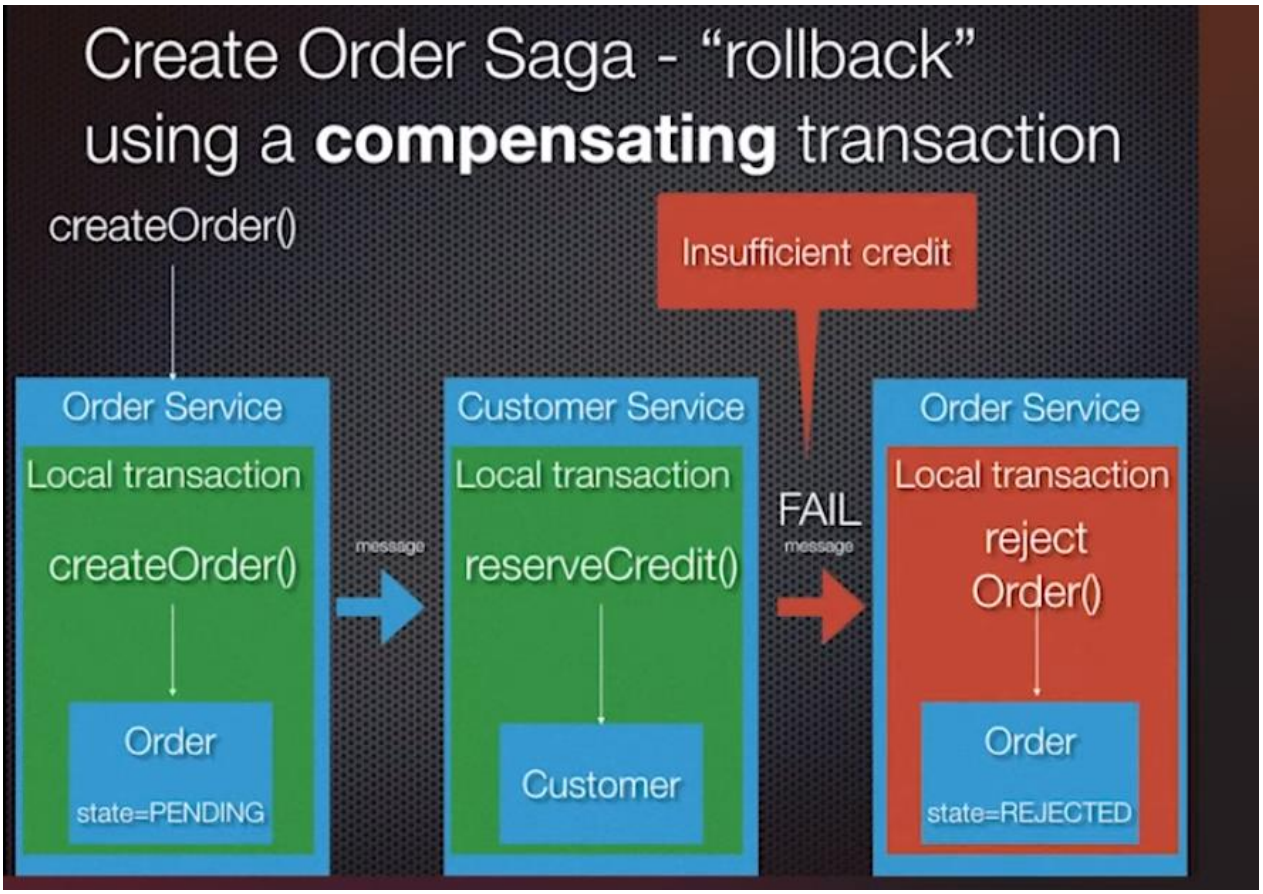
https://microservices.io/patterns/data/saga.html

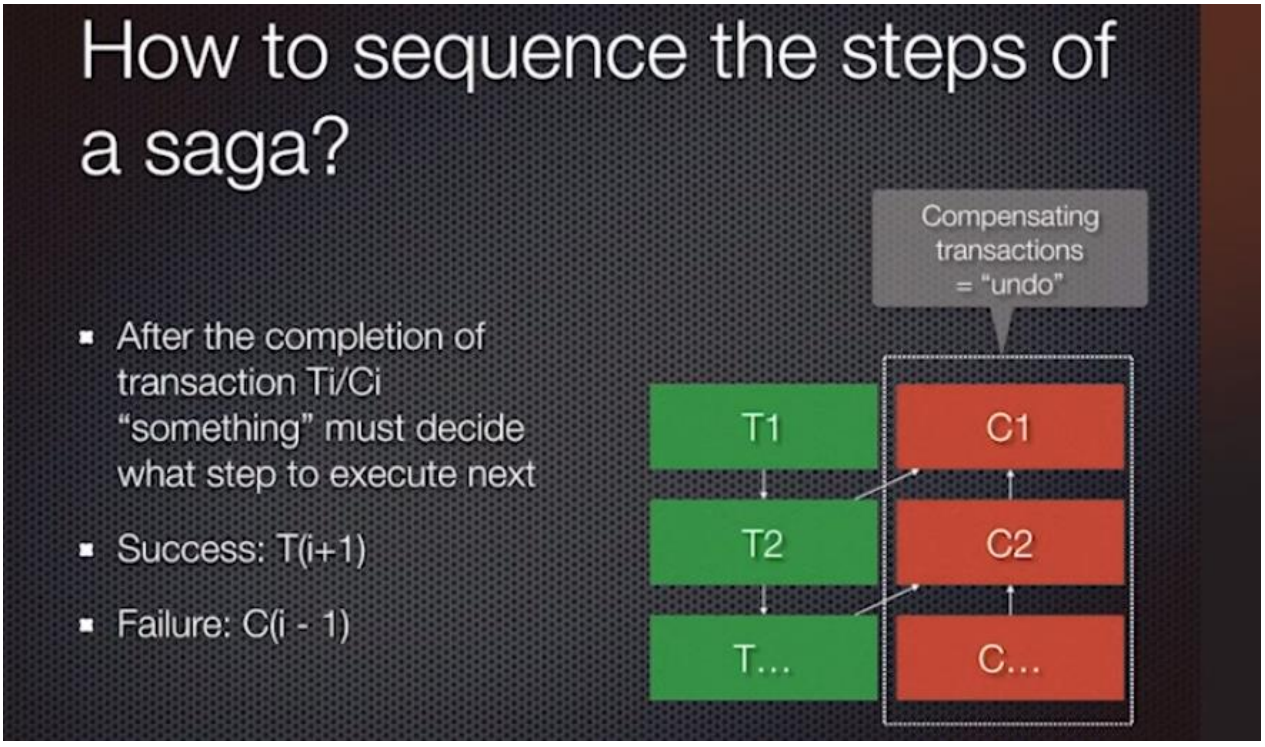

Create Order Saga
createOrder()
Initiates saga
Order Service
@crichardson


Create Order Saga
createOrder()
Initiates saga
Order Service
Local transaction
createOrder()
Order
state=PENDING
@crichardson


Create Order Saga
createOrder()
Initiates saga
Order ID
Order Service
Local transaction
createOrder()
Order
state=PENDING
message
Customer Service
Local transaction
reserveCredit()
Customer
@crichardson


Create Order Saga
createOrder()
Initiates saga
Order ID
Order Service
Local transaction
createOrder()
Order
state=PENDING
message
Customer Service
Local transaction
reserveCredit()
Customer
message
Order Service
Local transaction
approve
order()
Order
state=APPROVED


Create Order Saga - "rollback" using a compensating transaction
createOrder()
Insufficient credit
Order Service
Local transaction
createOrder()
Order
state=PENDING
message
Customer Service
Local transaction
reserveCredit()
Customer
FAIL
message
Order Service
Local transaction
reject Order()
Order
state=REJECTED


How to sequence the steps of a saga?
After the completion of transaction Ti/Ci "something" must decide what step to execute next
Success: T(i+1)
Failure: C(i - 1)
Compensating transactions = "undo"
T1
T2
T…
C1
C2
C…

Choreography: **distributed** decision making

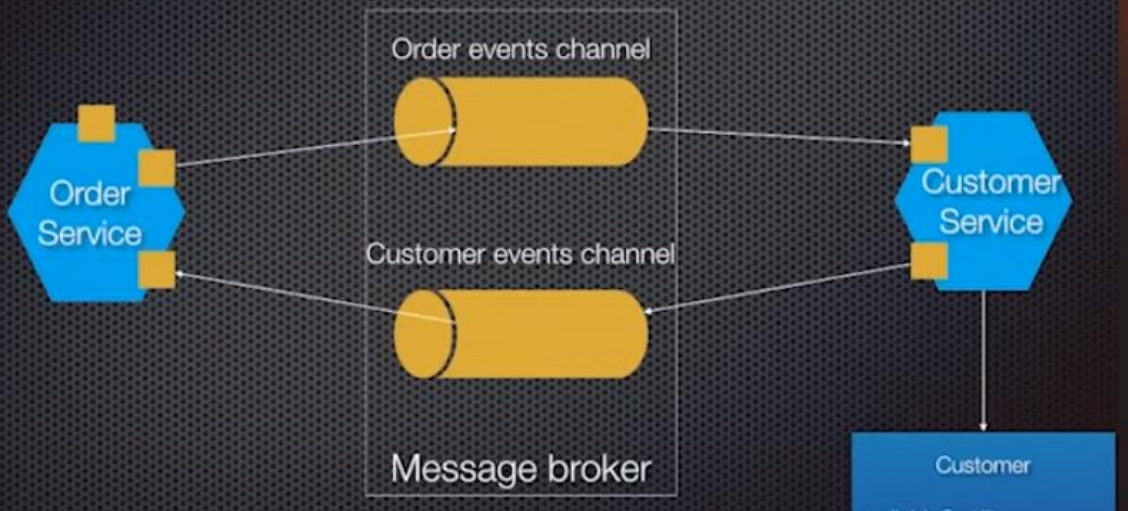
vs.

Orchestration: **centralized** decision making

Choreography = event-driven sagas

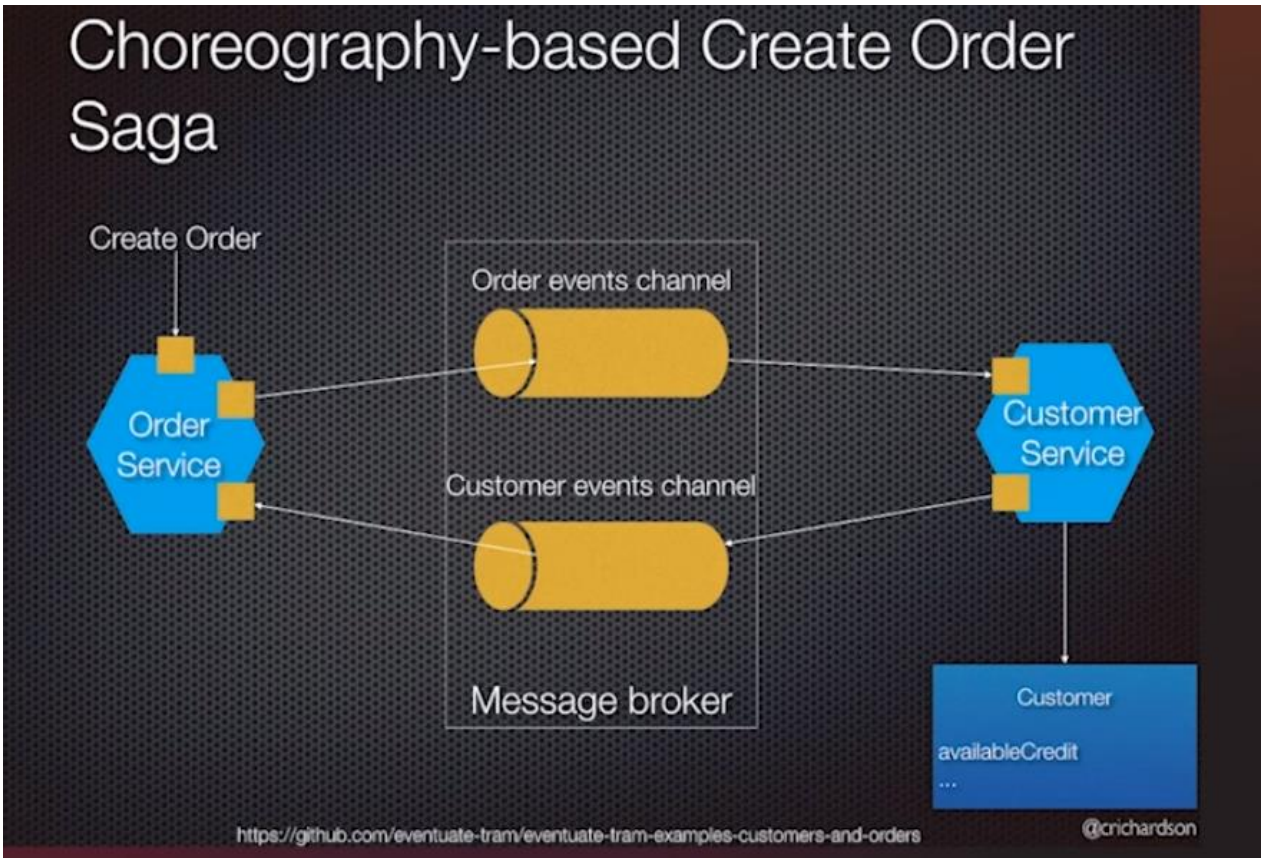
- Coordination logic = code that publishes events + event handlers
- Whenever a saga participant updates a business object, it publishes (domain) events announcing what it **has done**
- Saga participants have event handlers that update business objects => …

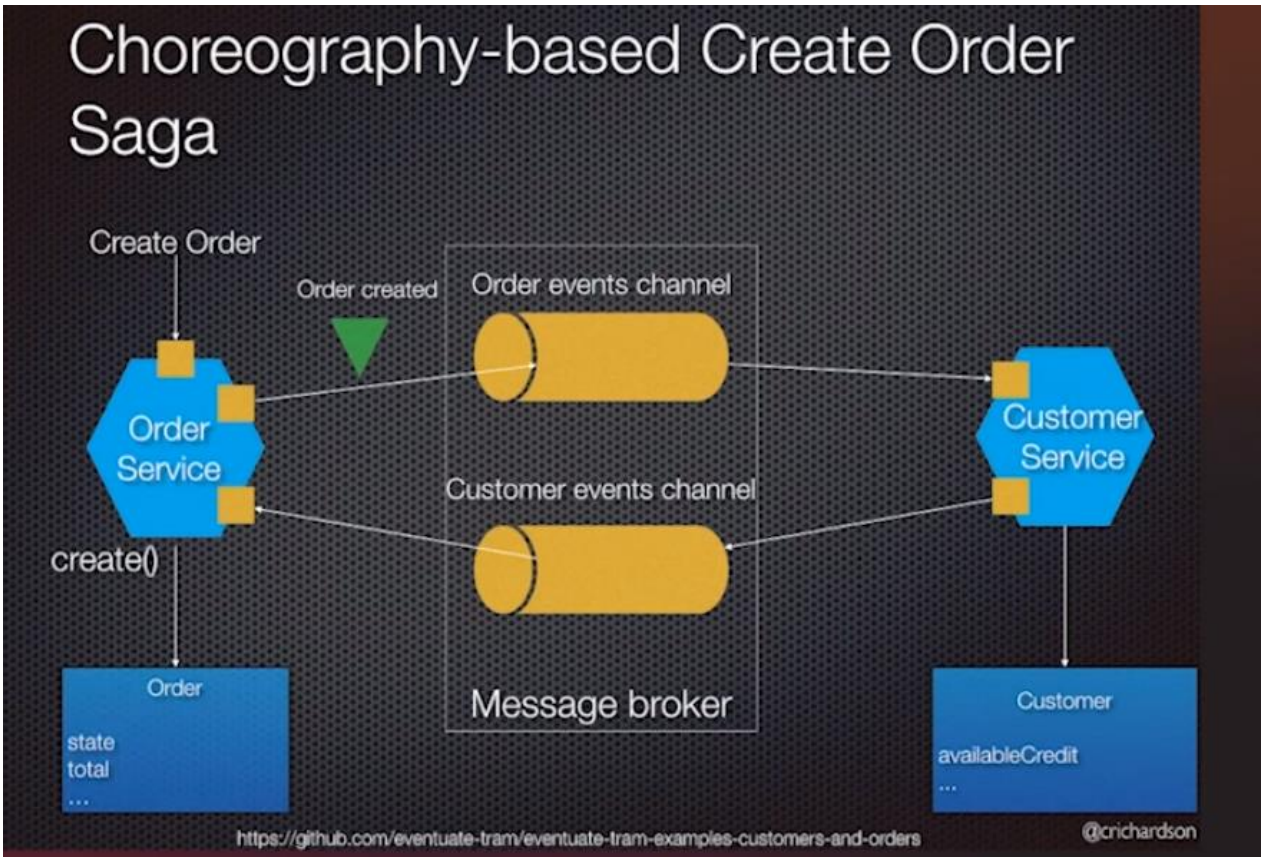
Choreography-based Create Order Saga

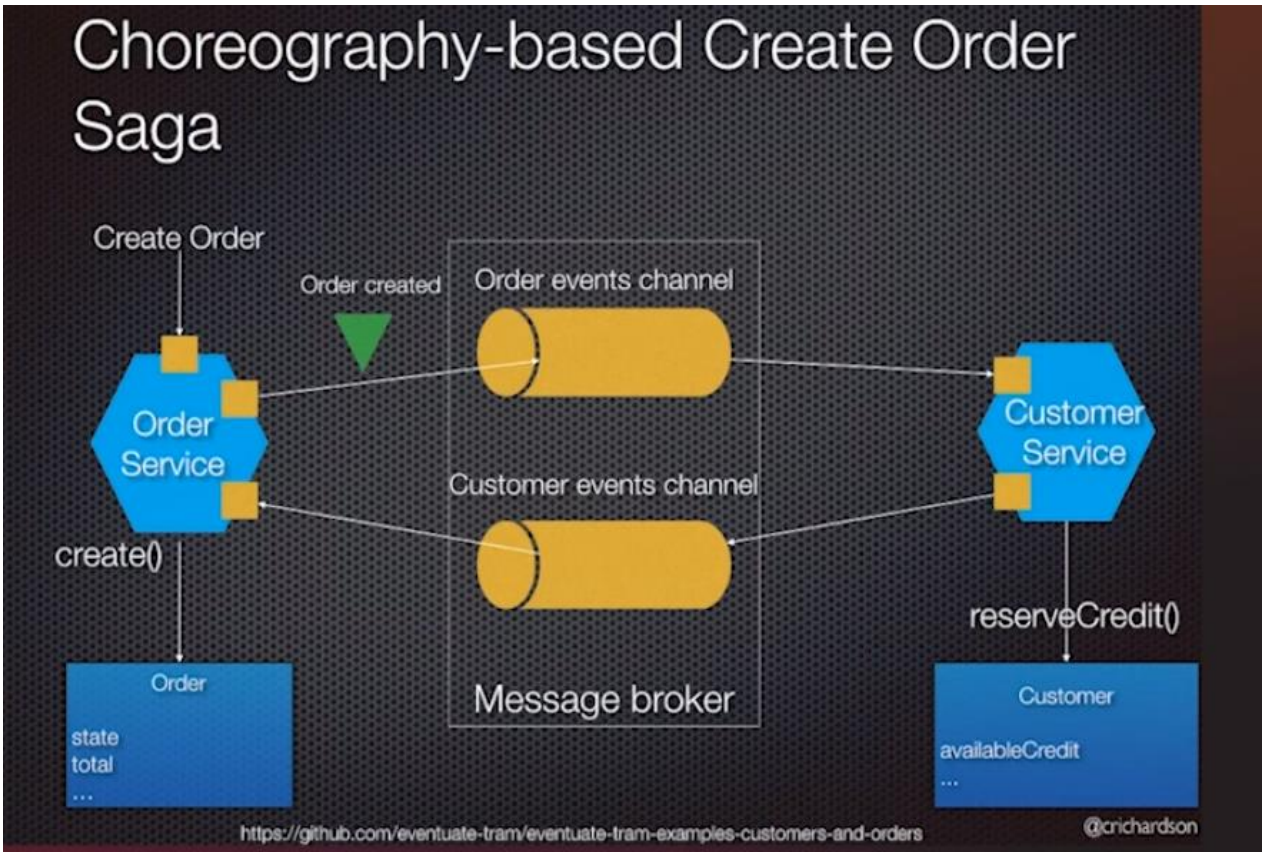


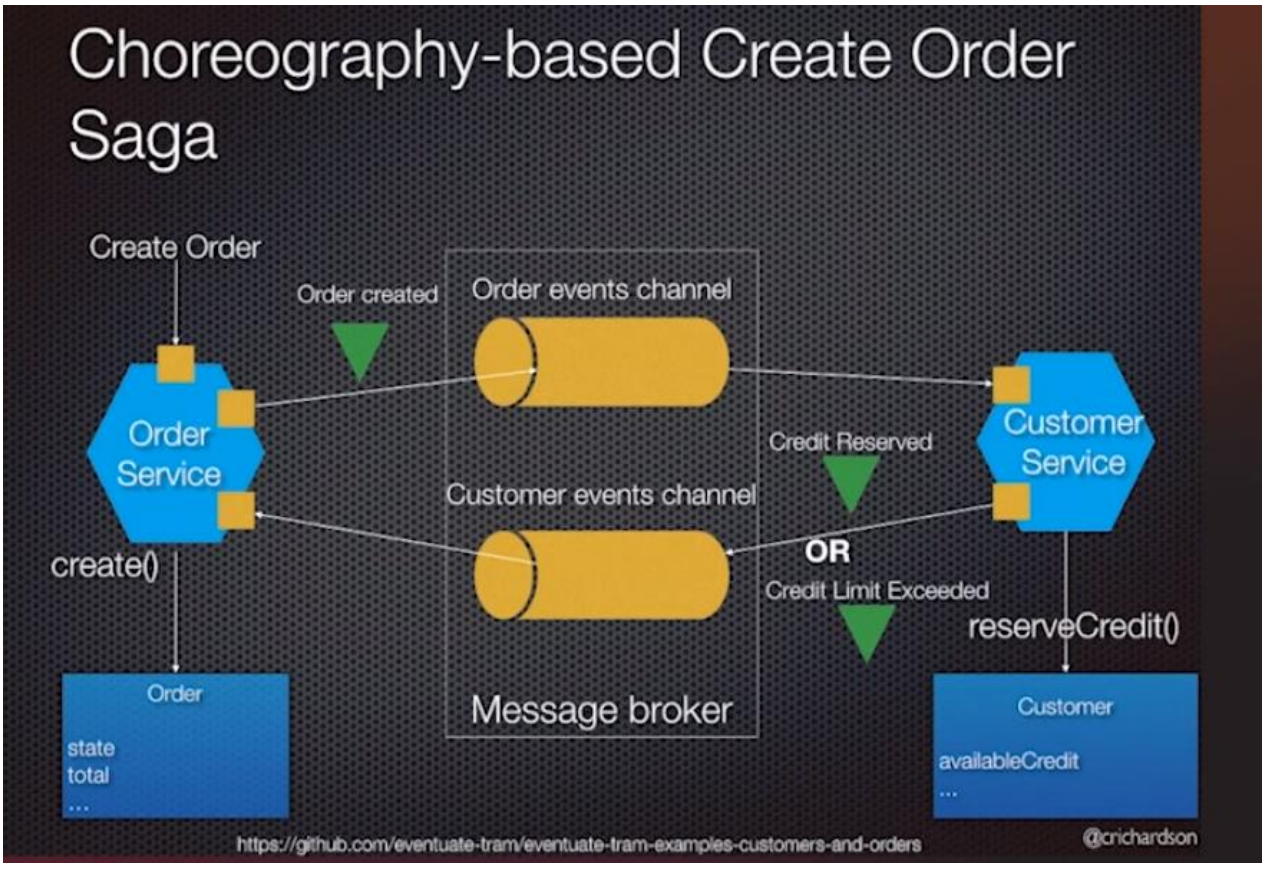
https://github.com/eventuate-tram/eventuate-tram-examples-customers-and-orders

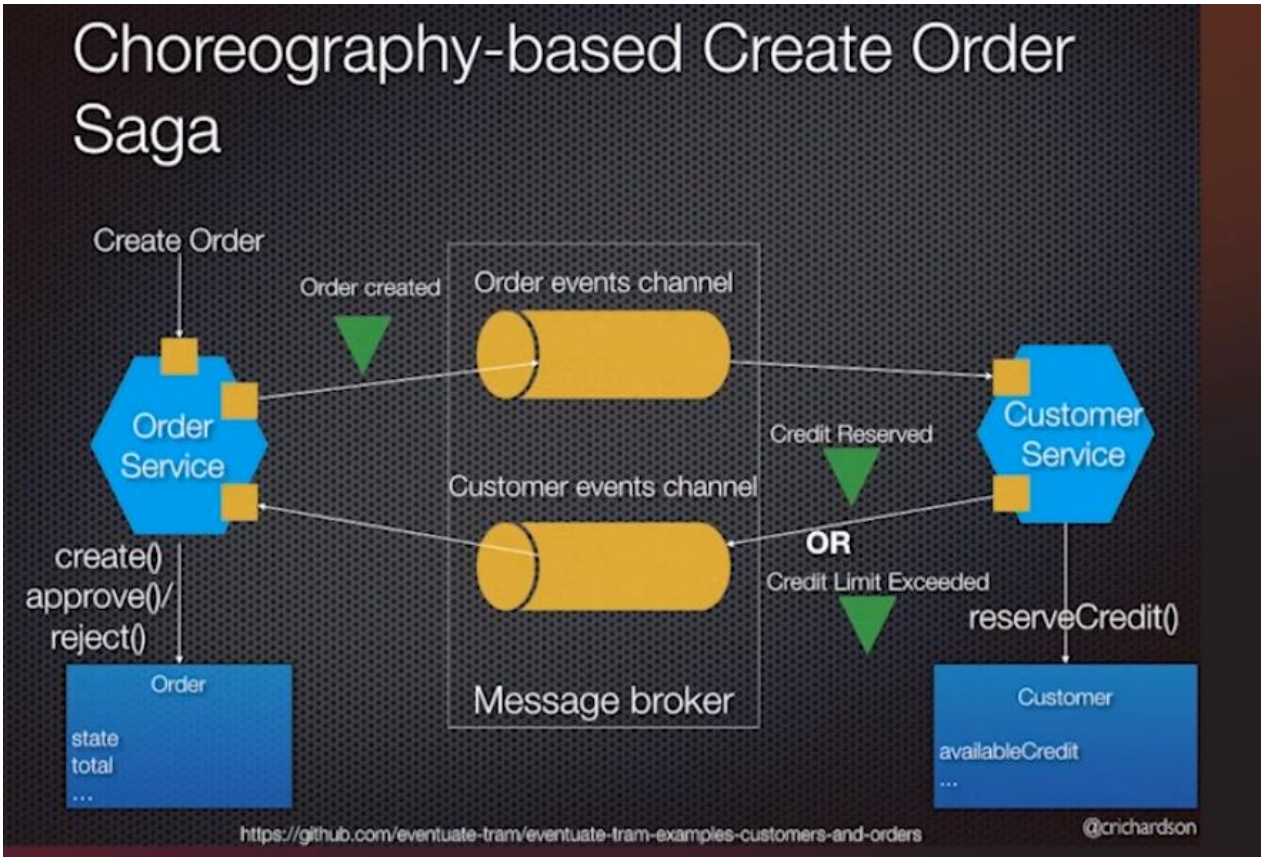
@crichardson

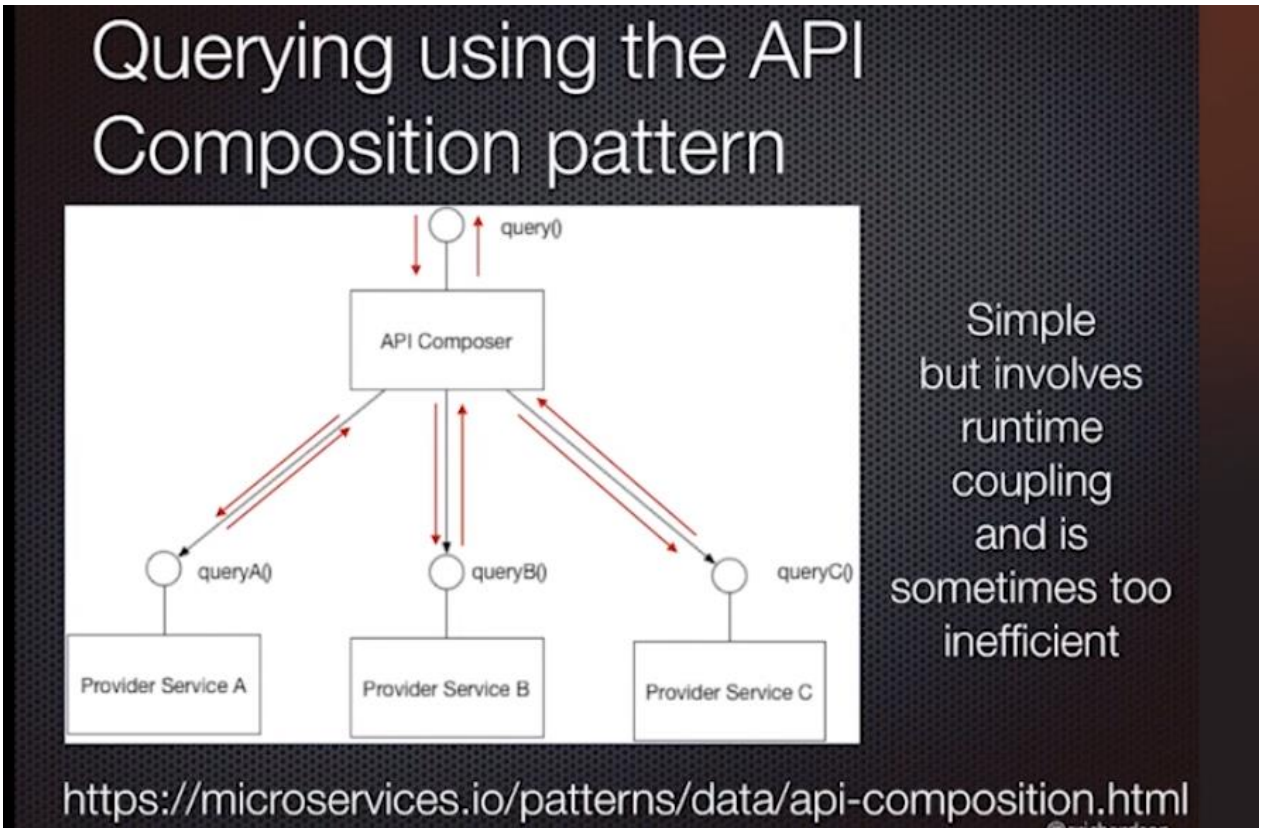

Choreography-based Create Order Saga
Create Order
Order events channel
Order Service
Customer Service
Customer events channel
Message broker
Customer
availableCredit
...
https://github.com/eventuate-tram/eventuate-tram-examples-customers-and-orders
@crichardson


Choreography-based Create Order Saga
Create Order
Order created
Order events channel
Order Service
Customer Service
Customer events channel
create()
Order
state
total
...
Message broker
Customer
availableCredit
...
https://github.com/eventuate-tram/eventuate-tram-examples-customers-and-orders
@crichardson

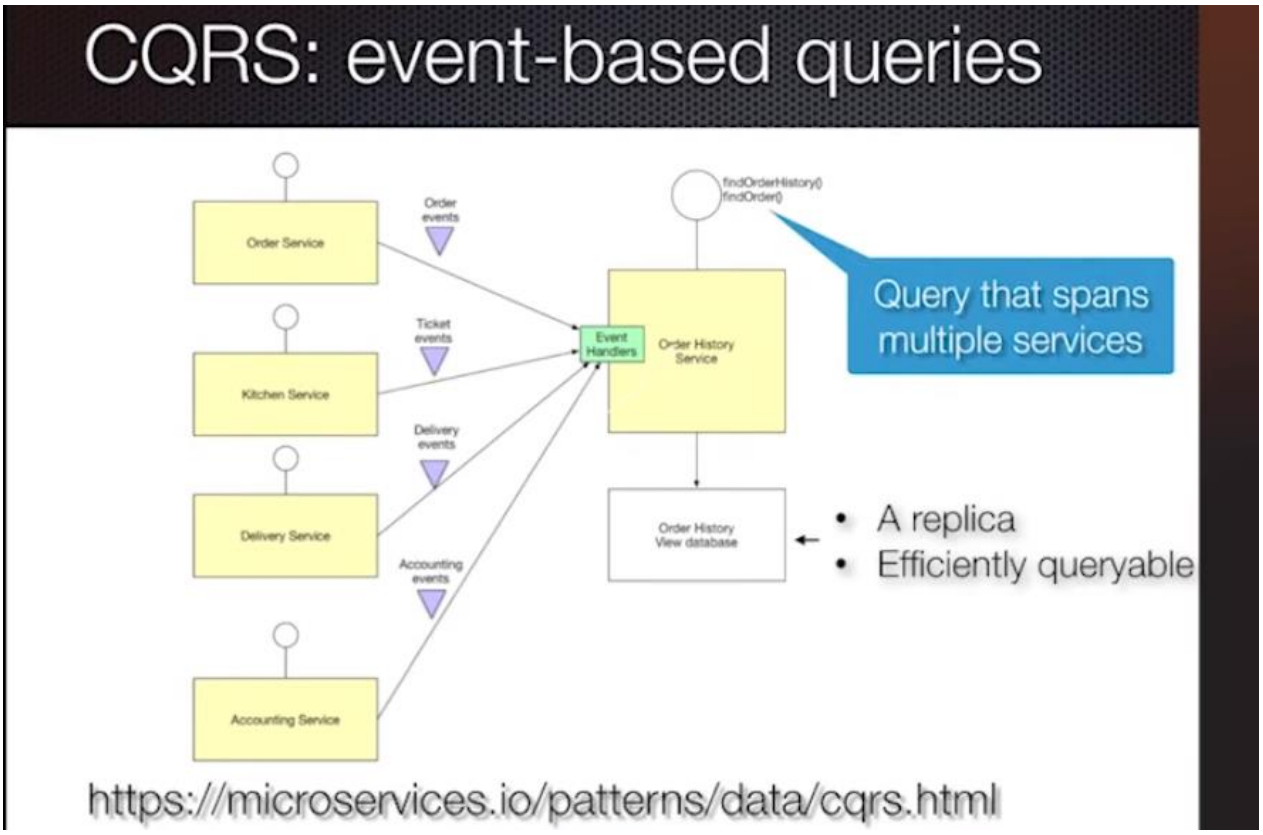

Choreography-based Create Order Saga
Create Order
Order created
Order events channel
Order Service
Customer Service
Customer events channel
create()
reserveCredit()
Order
state
total
...
Message broker
Customer
availableCredit
...
https://github.com/eventuate-tram/eventuate-tram-examples-customers-and-orders
@crichardson


Choreography-based Create Order Saga
Create Order
Order created
Order events channel
Order Service
Credit Reserved
Customer Service
Customer events channel
OR
Credit Limit Exceeded
create()
reserveCredit()
Order
state
total
...
Message broker
Customer
availableCredit
...
https://github.com/eventuate-tram/eventuate-tram-examples-customers-and-orders
@crichardson

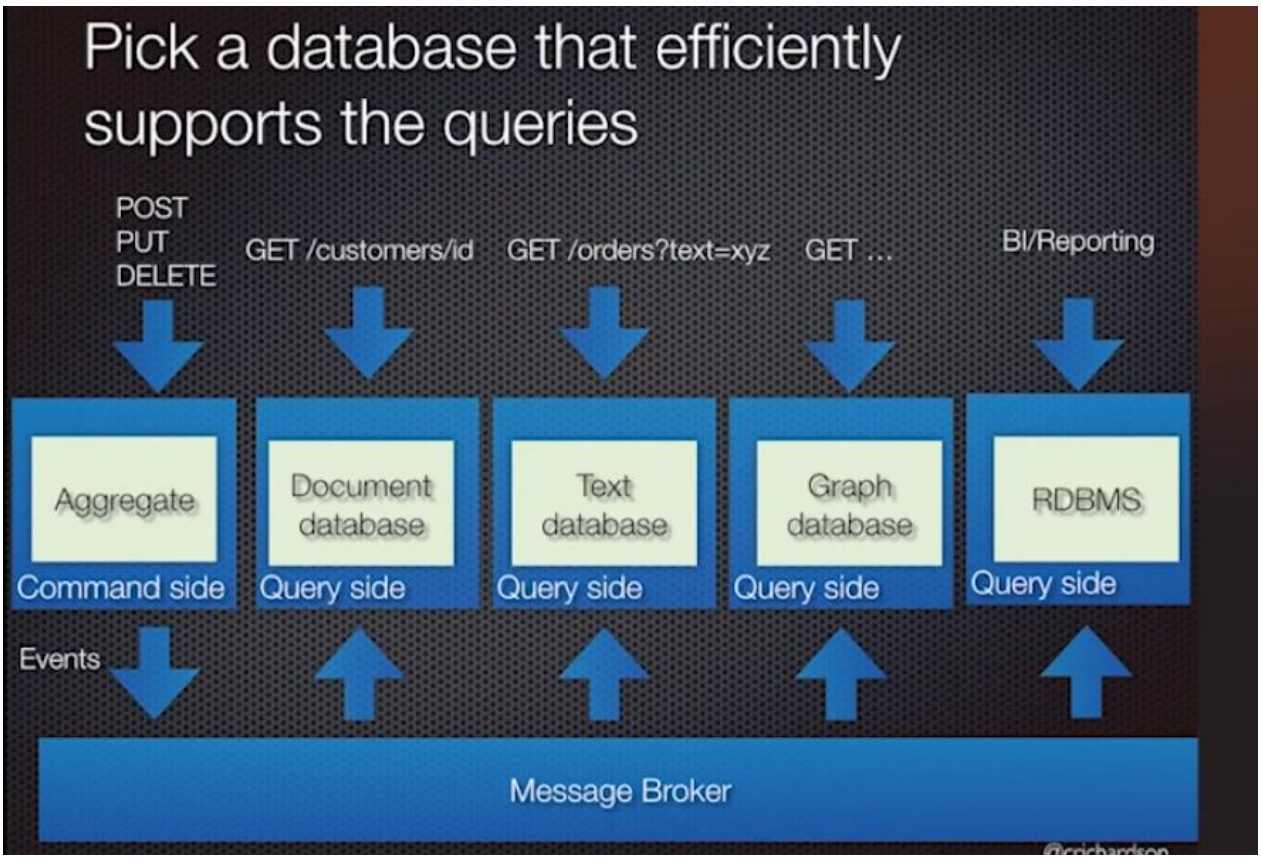




Using circuit breakers and callback strategies in your microservices can help limit the defects of this approach.



For CQRS, you will need separate data models for the querying and updates



Summary

- Rapid, frequent, reliable and sustainable software delivery requires services to be loosely coupled
- Sharing databases = tight design-time coupling ⇒ Database per service
- Traditional distributed transactions = tight runtime coupling
- Use the Saga pattern to implement commands that span services
- Use the CQRS pattern to implement queries that span services