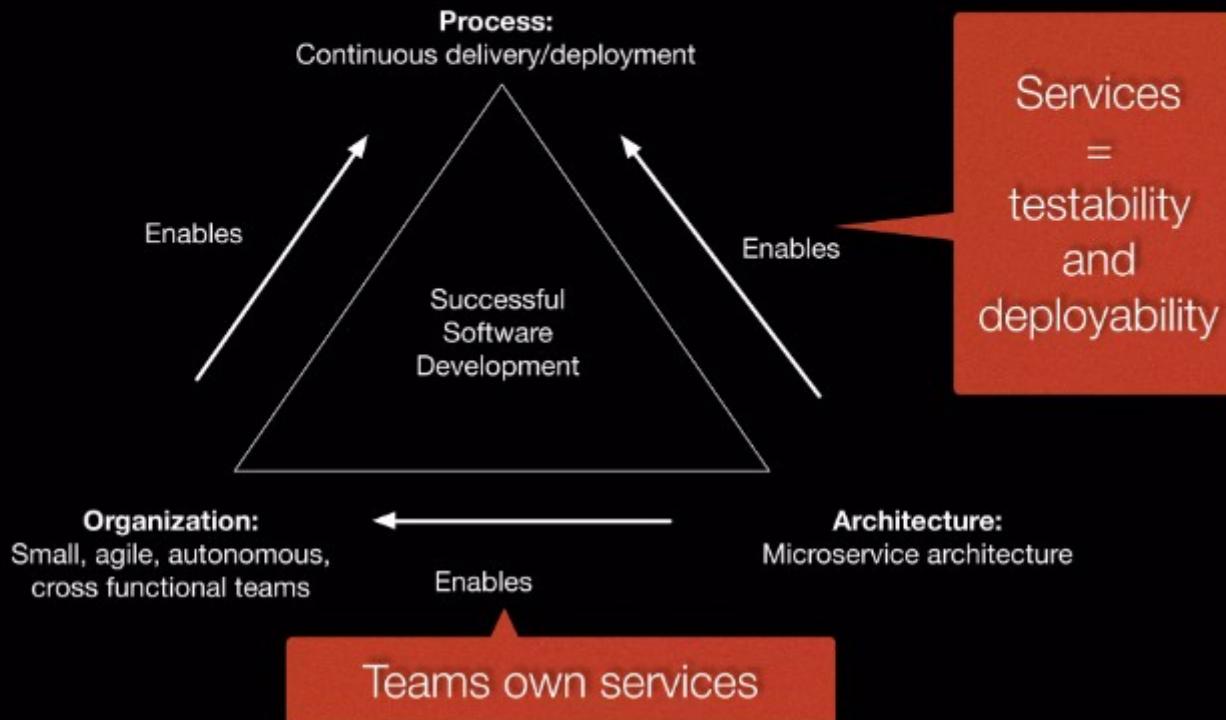
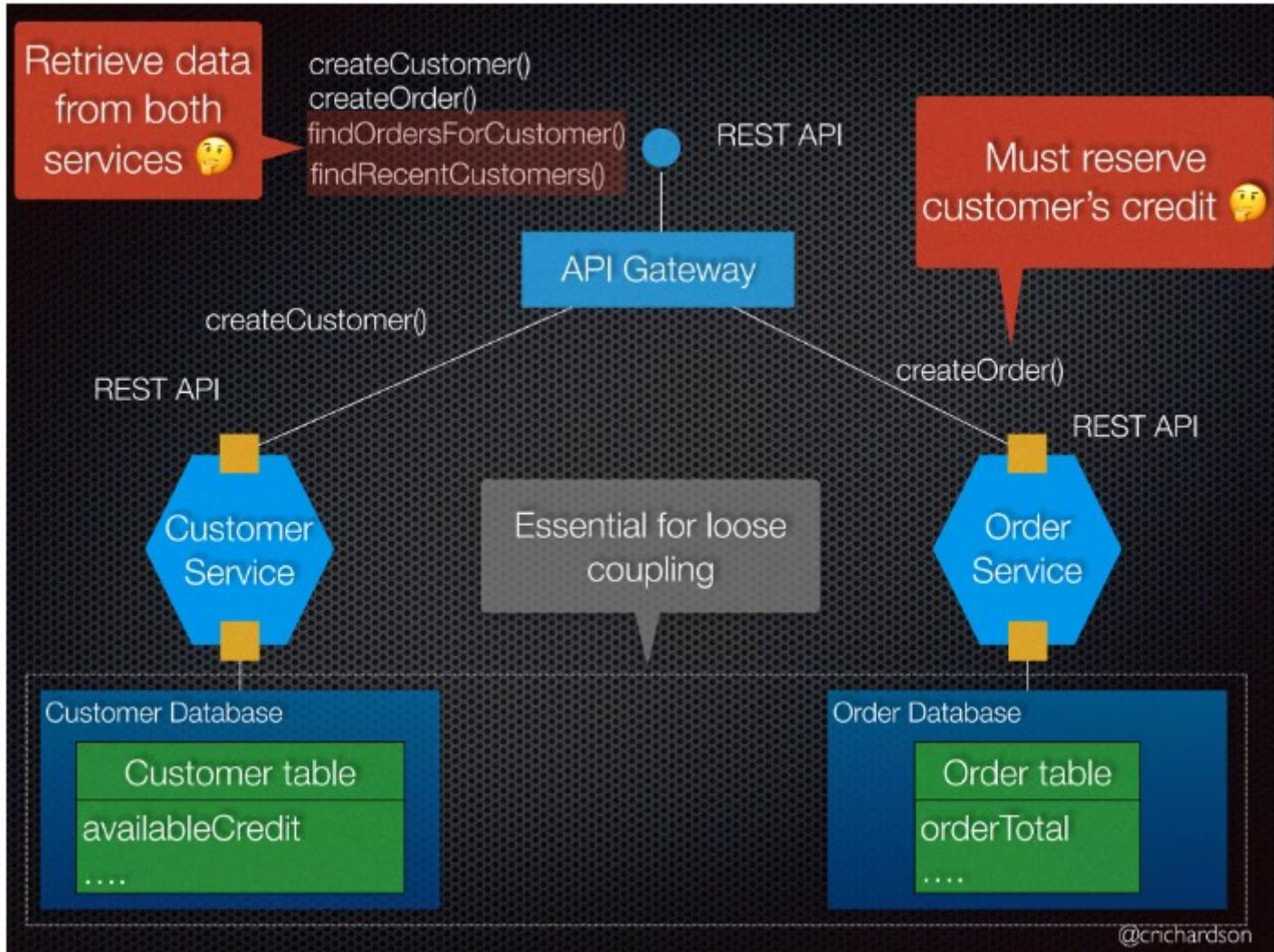


@crichtson

Microservices enable continuous delivery/deployment





No ACID transactions that span services

Distributed transactions

```
BEGIN TRANSACTION
```

```
...
```

```
SELECT ORDER_TOTAL  
FROM ORDERS WHERE CUSTOMER_ID = ?
```

```
...
```

```
SELECT CREDIT_LIMIT  
FROM CUSTOMERS WHERE CUSTOMER_ID = ?
```

```
...
```

```
INSERT INTO ORDERS ...
```

```
...
```

```
COMMIT TRANSACTION
```

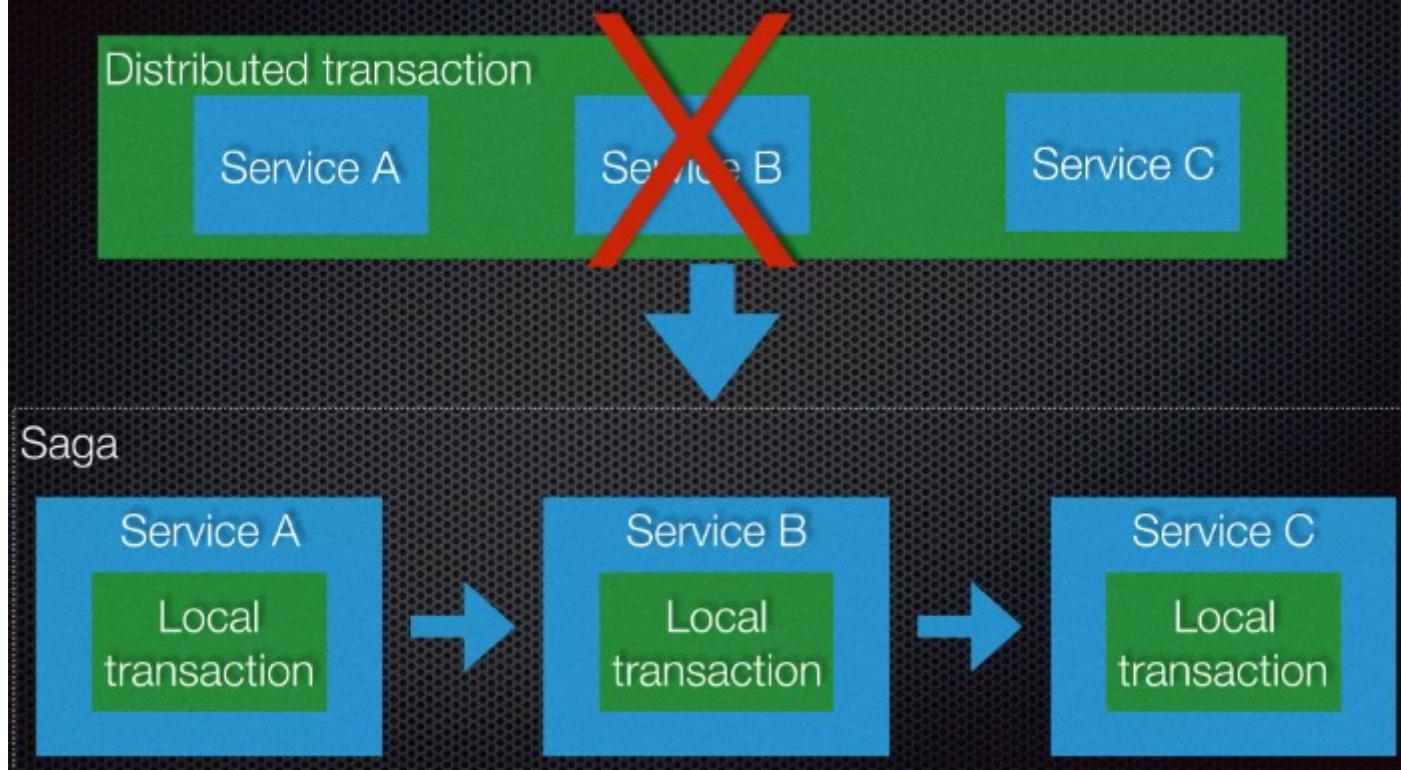
Private to the
Order Service

Private to the
Customer Service

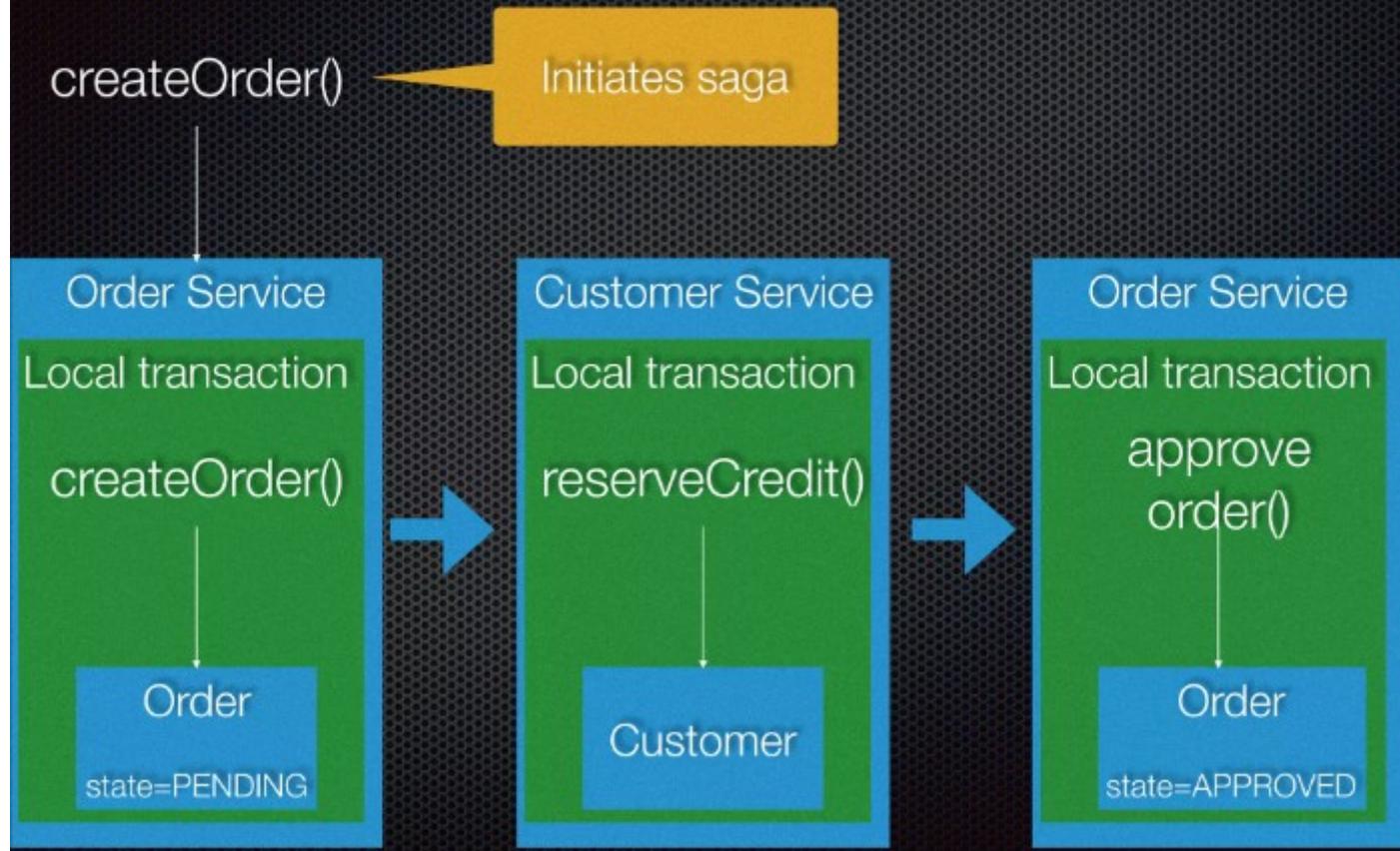
Agenda

- Transactions, queries and microservices
 - Managing transactions with sagas
 - Implementing queries with CQRS
 - Implementing transactional messaging

Use Sagas instead of 2PC



Create Order Saga

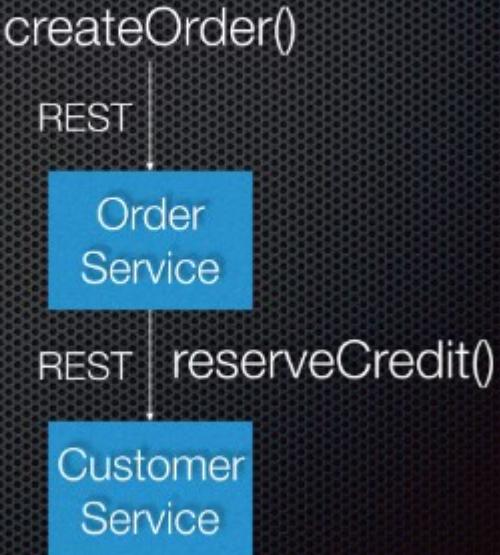


Saga design challenges

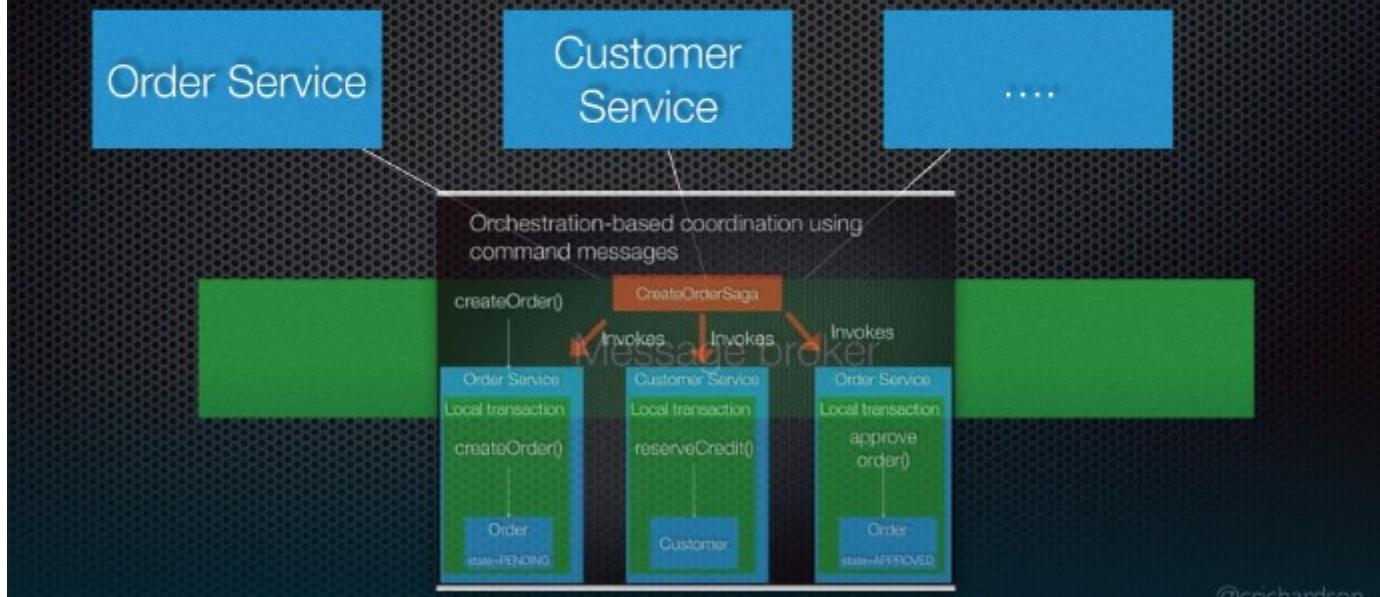
- API design
 - Synchronous REST API initiates asynchronous saga
 - When to send back a response?
- Rollback ⇒ compensating transactions
- Sagas are ACD - No I
 - Sagas are interleaved ⇒ anomalies, such as lost updates
 - Must use countermeasures

How do the saga participants communicate?

- Synchronous communication, e.g. REST = temporal coupling
- Client and server need to be both available
- Customer Service fails ⇒ retry provided it's idempotent
- Order Service fails ⇒ Oops



Collaboration using asynchronous, broker-based messaging



About the message broker

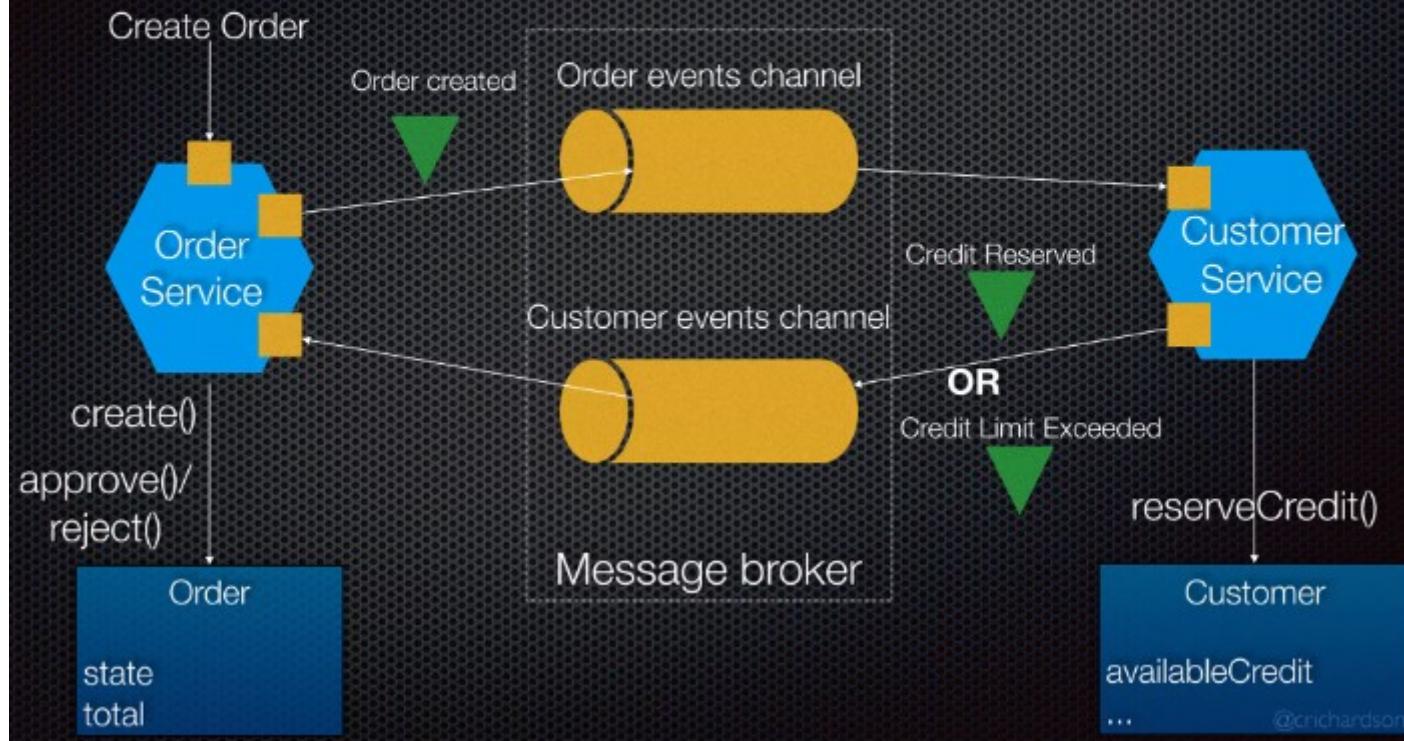
- At least once delivery
 - Ensures a saga completes when its participants are temporarily unavailable
 - Ordered delivery
 - Mechanism for scaling consumers that preserves ordering e.g.
 - Apache Kafka consumer group
 - ActiveMQ message group
 - ...

Choreography: **distributed** decision making

vs.

Orchestration: **centralized** decision making

Choreography-based Create Order Saga



@crichton

Benefits and drawbacks of choreography

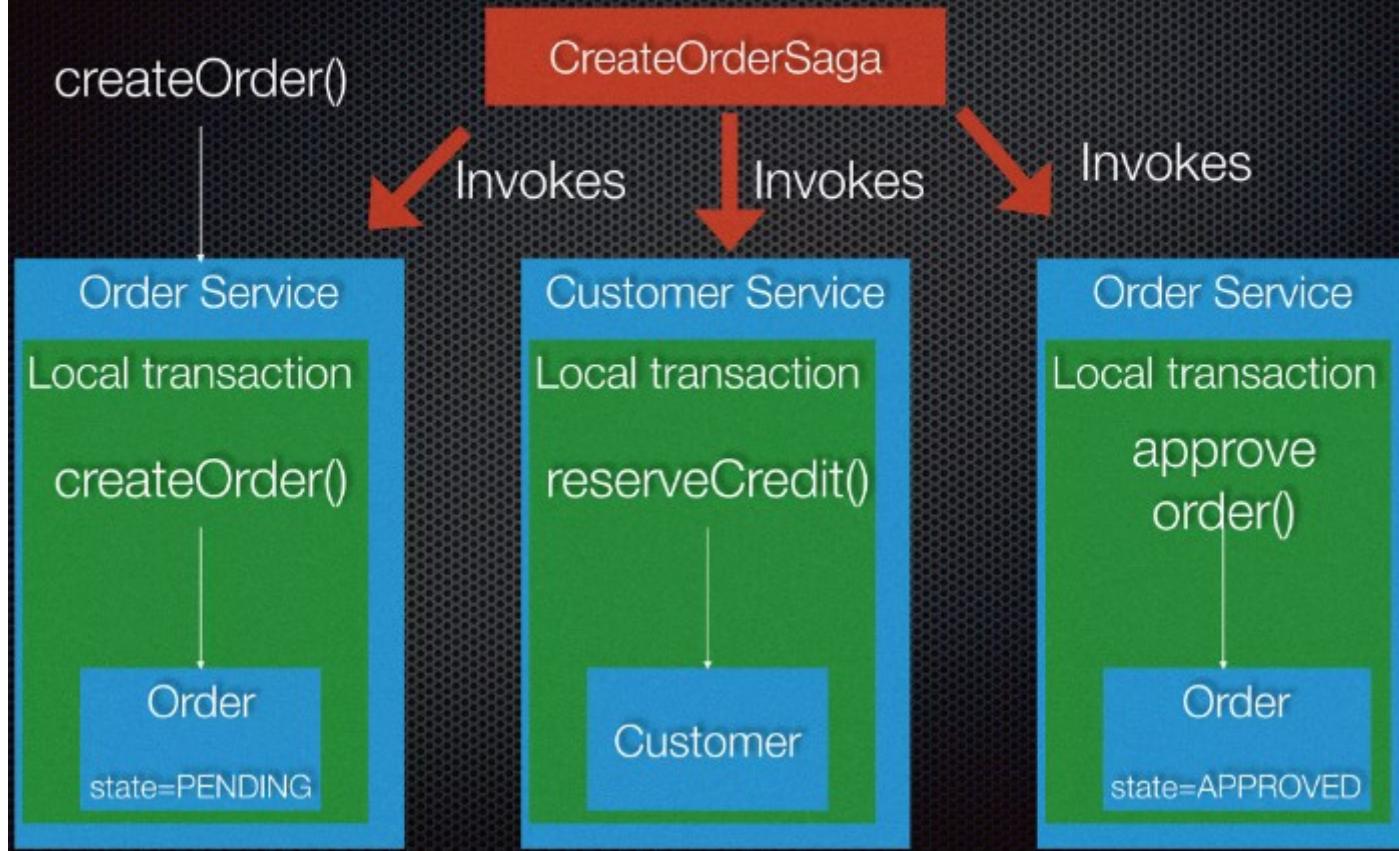
Benefits

- 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

Orchestration-based coordination using command messages



A saga (orchestrator)
is a **persistent object**

that

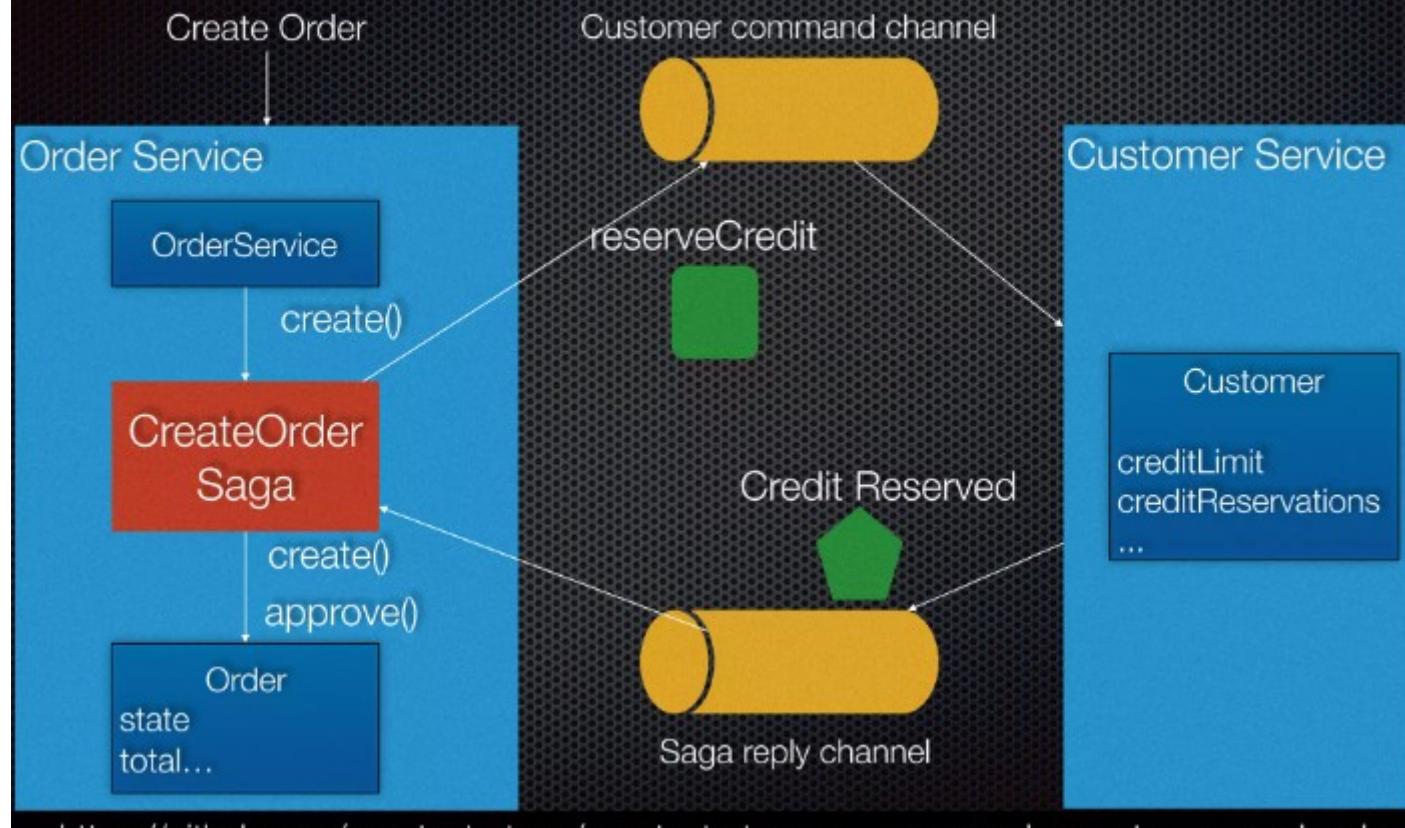
implements a state machine
and

invokes the participants

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
 - ...

CreateOrderSaga orchestrator



Benefits and drawbacks of orchestration

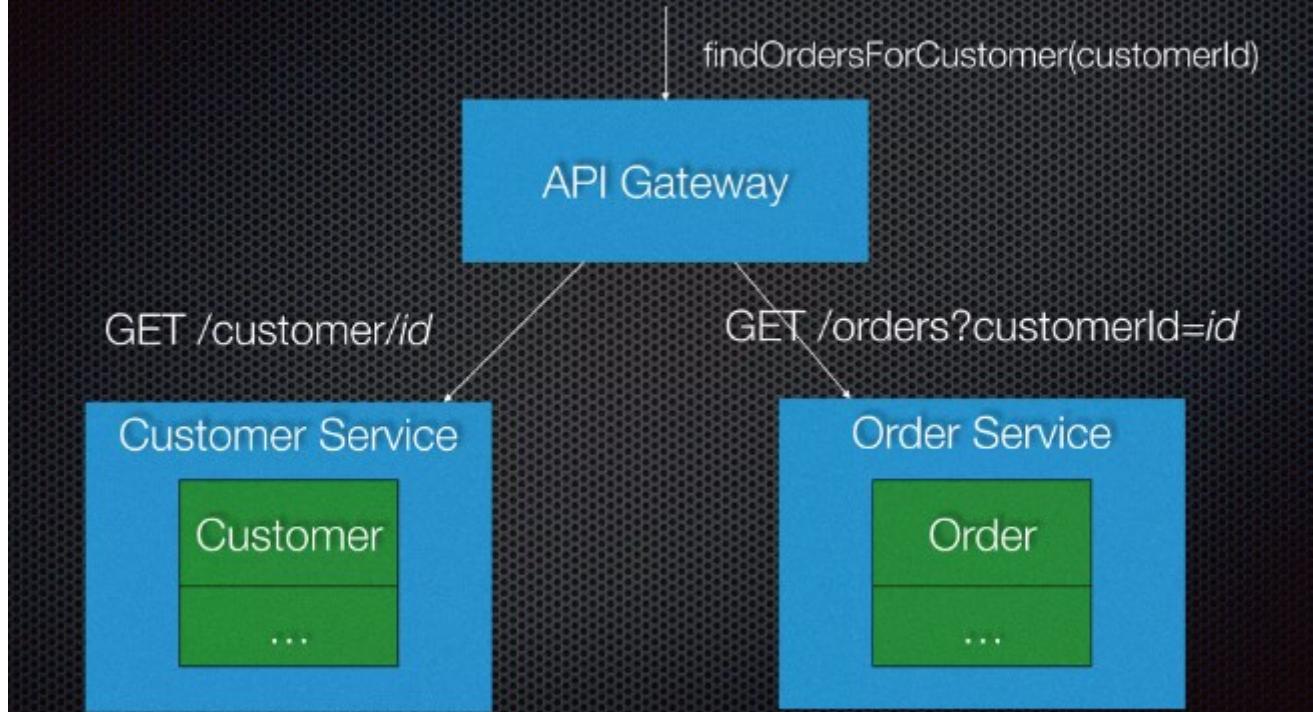
Benefits

- Centralized coordination logic is easier to understand
- Reduced coupling, e.g. Customer Service knows less. Simply has API for managing available credit.
- Reduces cyclic dependencies

Drawbacks

- Risk of smart sagas directing dumb services

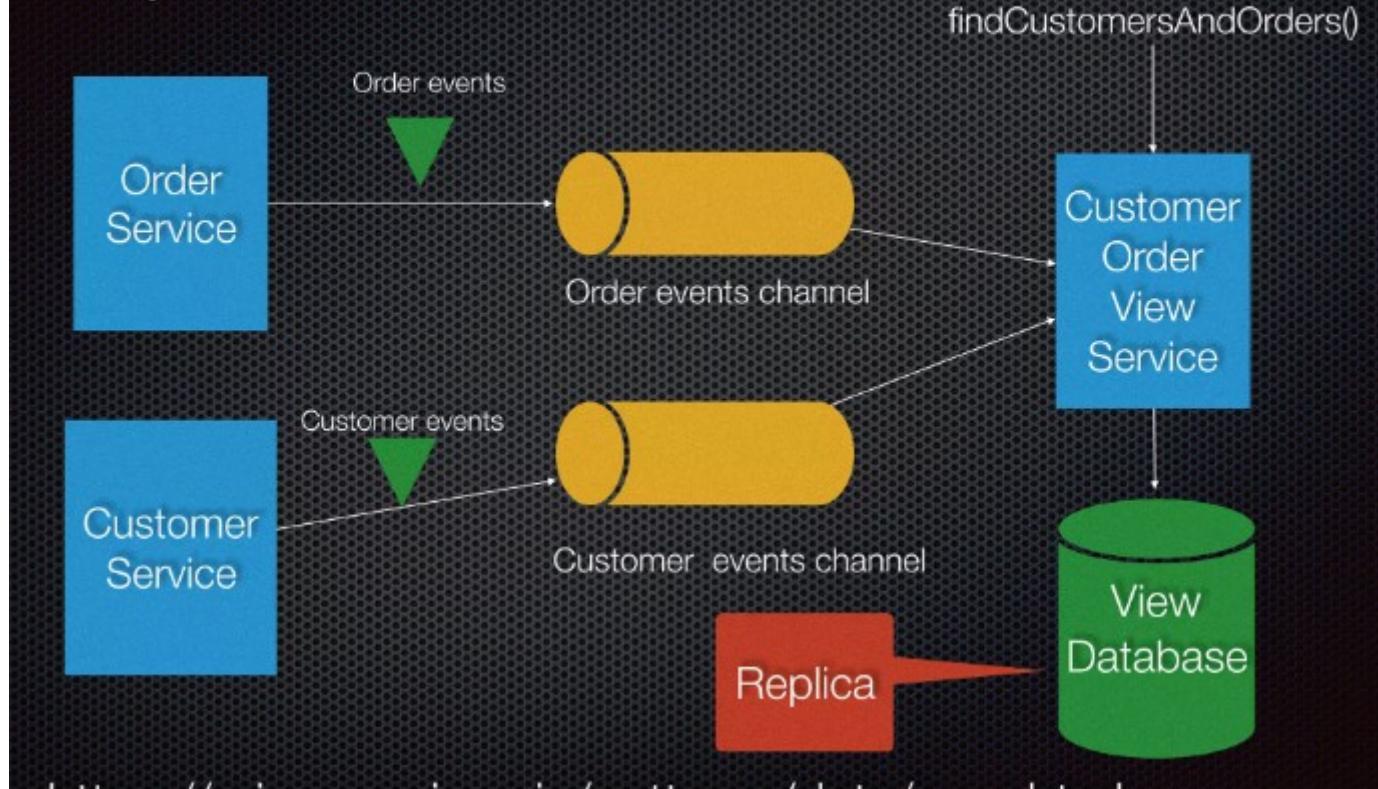
API Composition pattern



API Composition would be inefficient

- 1 + N strategy:
 - Fetch recent customers
 - Iterate through customers fetching their shipped orders
 - Lots of round trips ⇒ high-latency
- Alternative strategy:
 - Fetch recent customers
 - Fetch recent orders
 - Join
 - 2 roundtrips but potentially large datasets ⇒ inefficient

Using events to update a queryable replica = CQRS



<https://microservices.io/patterns/data/cqrs.html>

@crichtson

Persisting a customer and order history in MongoDB

```
{  
    "_id" : "0000014f9a45004b 0a00270000000000",  
    "name" : "Fred",  
    "creditLimit" : {  
        "amount" : "2000"  
    },  
    "orders" : {  
        "0000014f9a450063 0a00270000000000" : (  
            "state" : "APPROVED",  
            "orderId" : "0000014f9a450063 0a00270000000000",  
            "orderTotal" : {  
                "amount" : "1234"  
            }  
        ),  
        "0000014f9a450063 0a00270000000001" : (  
            "state" : "REJECTED",  
            "orderId" : "0000014f9a450063 0a00270000000001",  
            "orderTotal" : {  
                "amount" : "3000"  
            }  
        )  
    }  
}
```

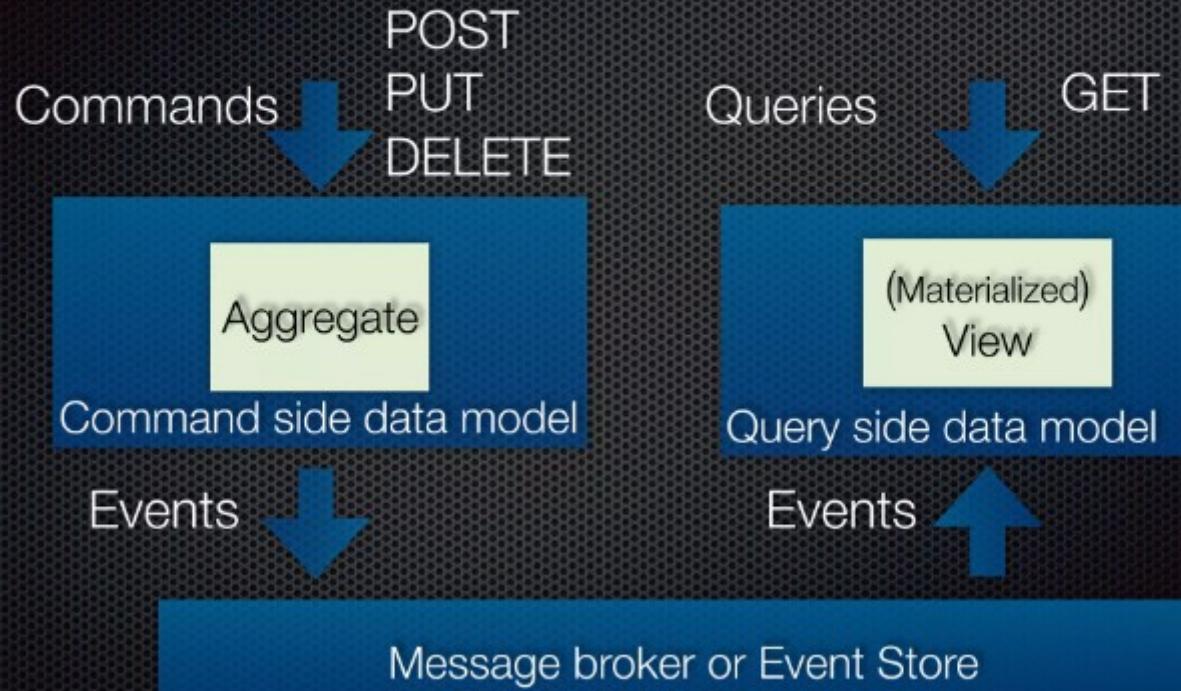
Customer
information

Order
information

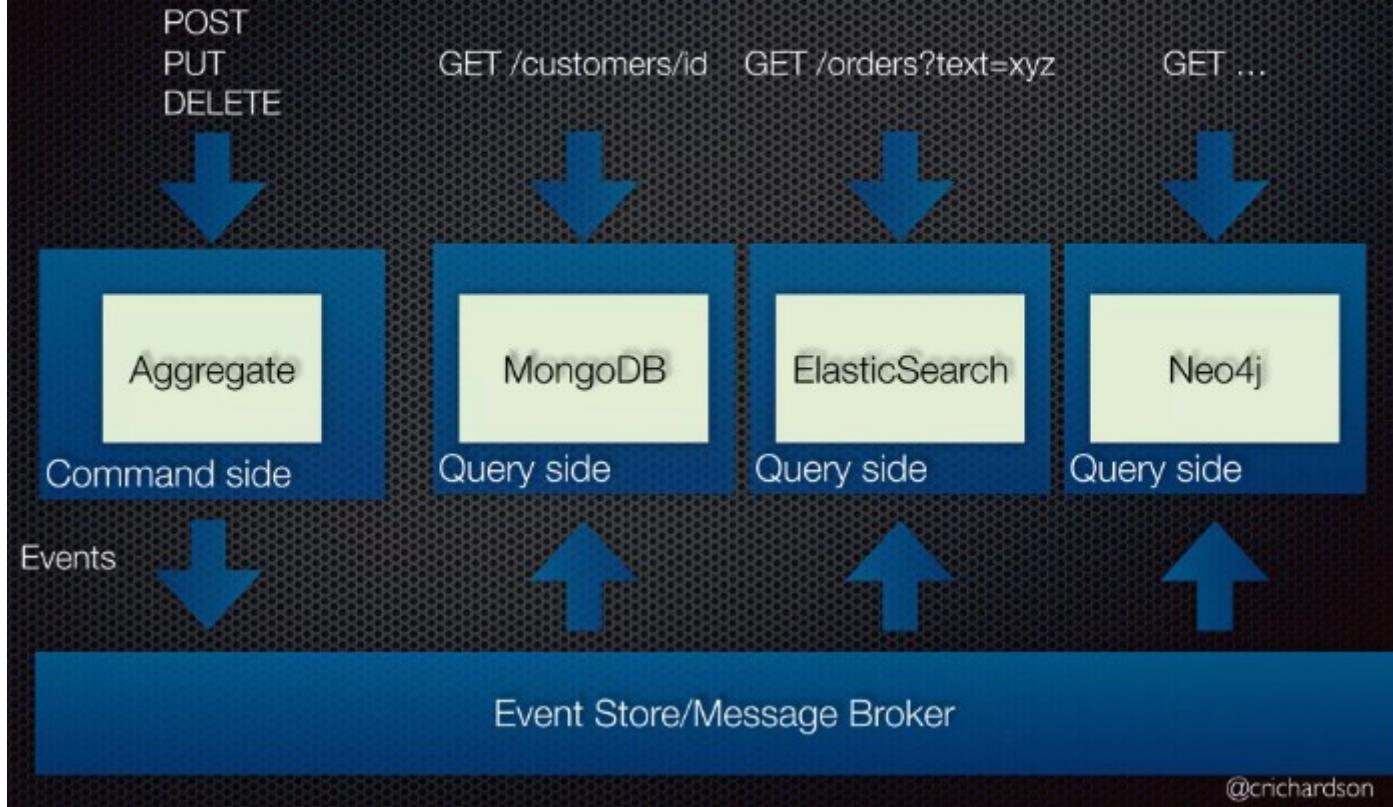
Denormalized = efficient lookup

@crichtson

Command Query Responsibility Segregation (CQRS)



Queries ⇒ database (type)

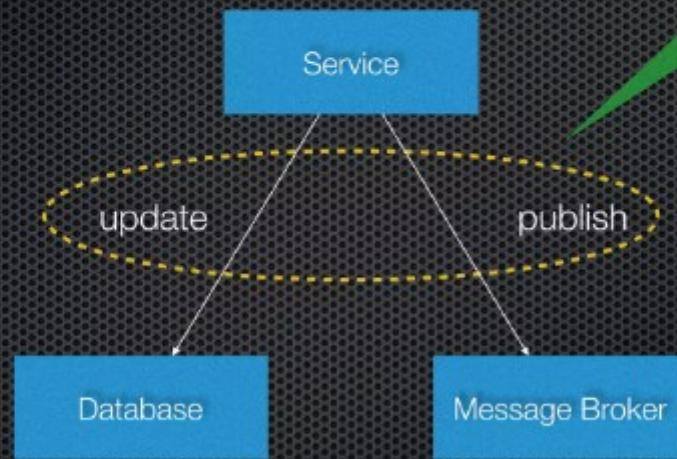


CQRS views are disposable

- Rebuild when needed from *source of truth*
- Using event sourcing
 - (Conceptually) replay all events from beginning of time
- Using traditional persistence
 - “ETL” from *source of truth* databases

Messaging must be transactional

How to
make **atomic**
without 2PC?



CQRS views are disposable

- Rebuild when needed from *source of truth*
- Using event sourcing
 - (Conceptually) replay all events from beginning of time
- Using traditional persistence
 - “ETL” from *source of truth* databases

Event Store/Message Broker

@crichtson

Drawbacks of event sourcing

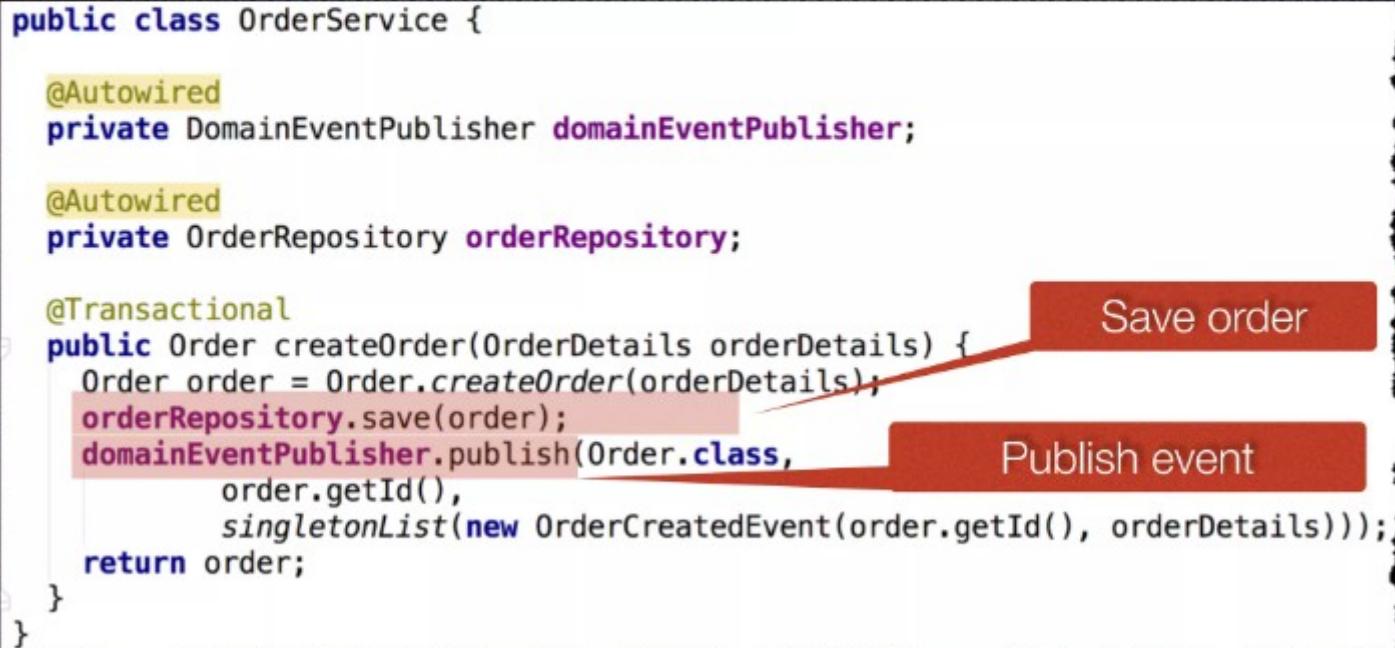
Good fit for choreography-based sagas
BUT orchestration is more challenging



Use event handler to translate event
into command/reply message

Spring Data for JPA example

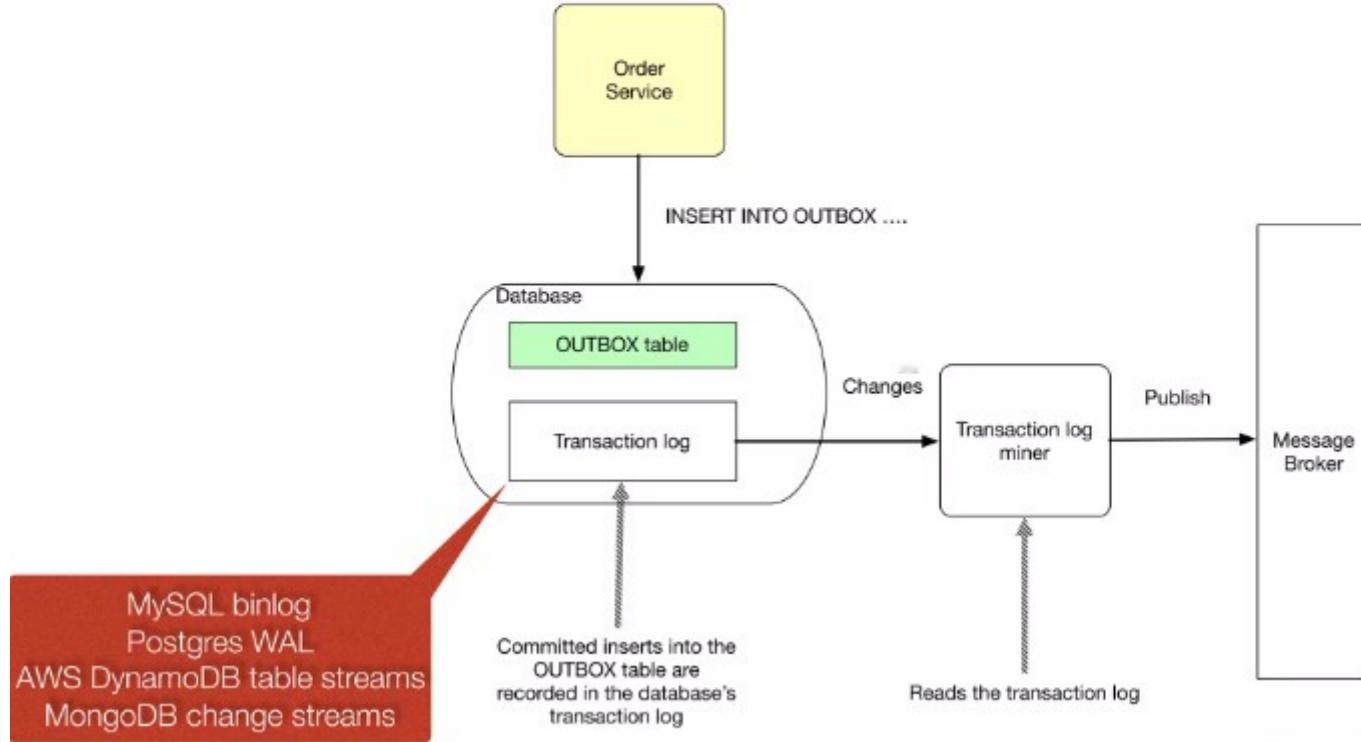
```
public class OrderService {  
  
    @Autowired  
    private DomainEventPublisher domainEventPublisher;  
  
    @Autowired  
    private OrderRepository orderRepository;  
  
    @Transactional  
    public Order createOrder(OrderDetails orderDetails) {  
        Order order = Order.createOrder(orderDetails);  
        orderRepository.save(order);  
        domainEventPublisher.publish(Order.class,  
            order.getId(),  
            singletonList(new OrderCreatedEvent(order.getId(), orderDetails)));  
        return order;  
    }  
}
```

A diagram illustrating the sequence of operations in the code. A red arrow points from the 'Save order' call to the 'orderRepository.save(order)' line. Another red arrow points from the 'Publish event' call to the 'domainEventPublisher.publish...' line.

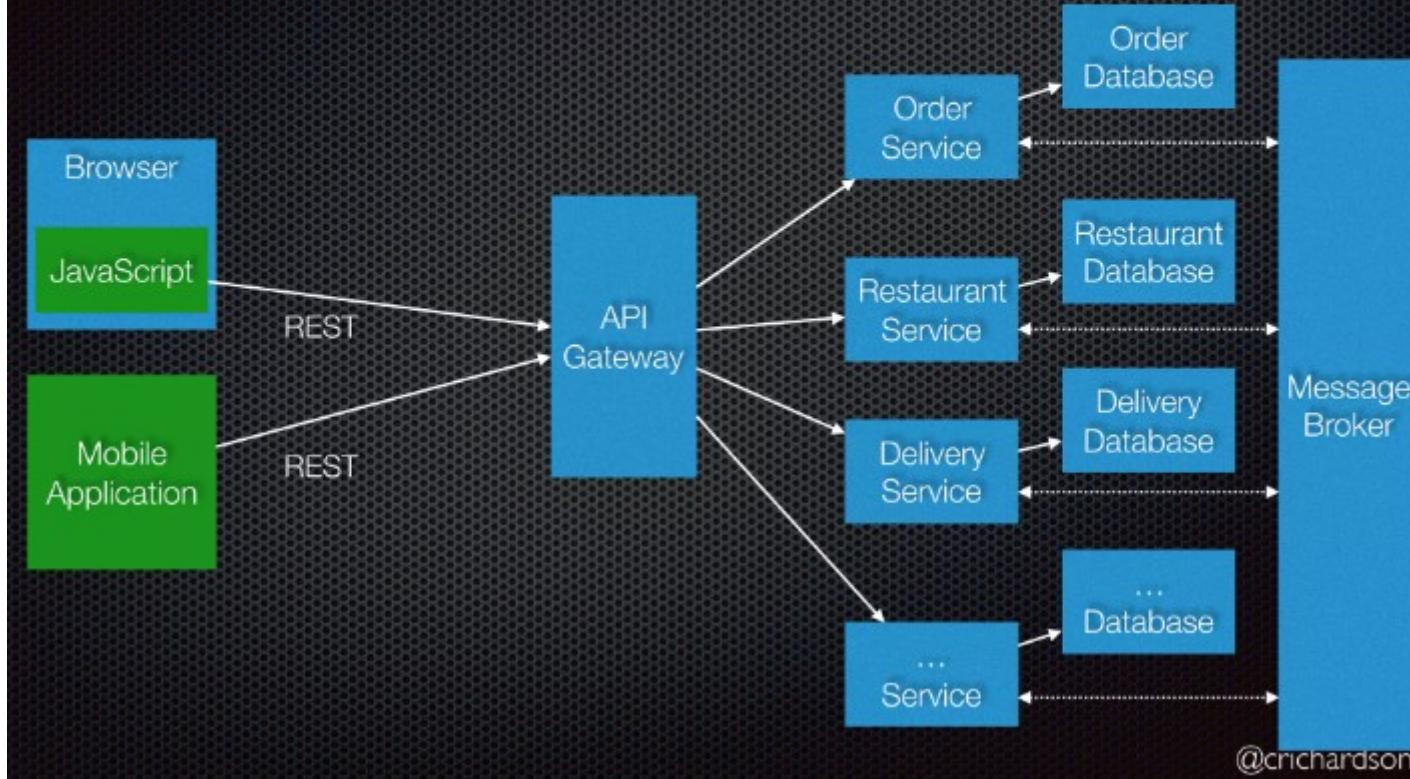
Save order

Publish event

Transaction log tailing

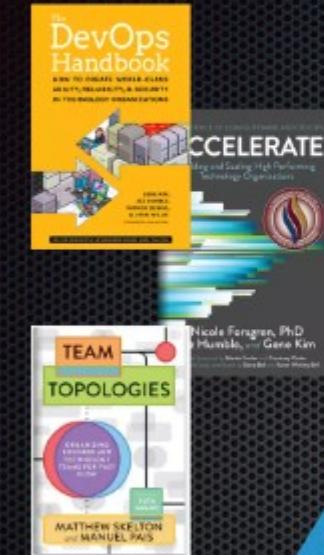


Food to Go: Microservice architecture



Why microservices: success triangle

Process: Lean + DevOps/Continuous Delivery & Deployment



Organization:
Network of small,
loosely coupled, teams

Enables:
Loose
coupling

Architecture:
Microservices
(sometimes)

Businesses must be
nimble, agile and
innovate faster

IT must deliver software
rapidly, frequently, reliably and
sustainably

Enables:
Testability
Deployability

@richardson

Loose coupling is essential

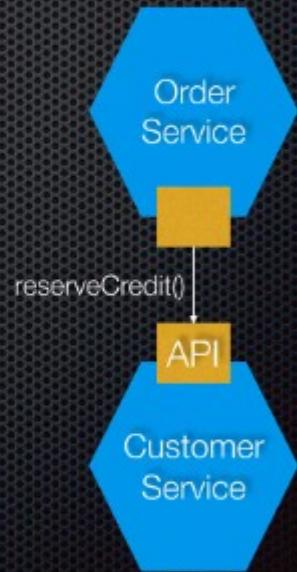
Services collaborate, e.g. Order Service must reserve customer credit



Coupling is inevitable

BUT

Services must be loosely coupled



Runtime coupling

- Order Service cannot respond to a synchronous request (e.g. HTTP POST) until Customer Service responds
- Reduces availability

vs

Design time coupling

- Change Customer Service ⇒ change Order Service
- Reduces development velocity

Neither is guaranteed: you must design your services to be loosely coupled

Degree of coupling

=

f()

- Service boundaries
- Service responsibilities
- Service APIs
- Service collaborations

DRY (Don't repeat yourself) services

"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system"

For example:
Order Total

https://en.wikipedia.org/wiki/Don%27t_repeat_yourself

DRY (Don't repeat yourself) services

"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system"

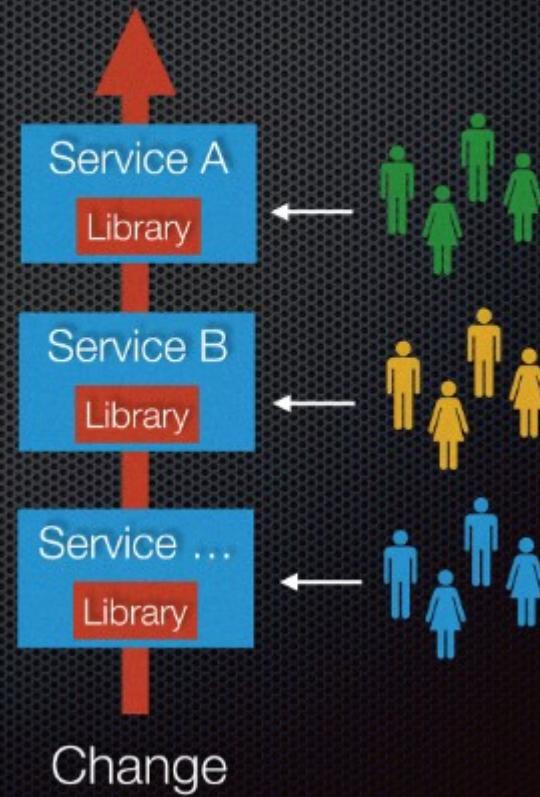
For example:
Order Total

https://en.wikipedia.org/wiki/Don%27t_repeat_yourself

Shared library that calculates Order Total != DRY

Shared libraries containing business logic that changes
⇒ requires multiple services to change/rebuild/redeployed in lock step ✗

Shared utility libraries ✓



Consume as little as possible

What you ignore can't affect you

- Postel's Robustness principle: https://en.wikipedia.org/wiki/Robustness_principle
- Consumer-driven contract tests verify compliance
- Swagger/Protobuf-generated stubs parse everything!

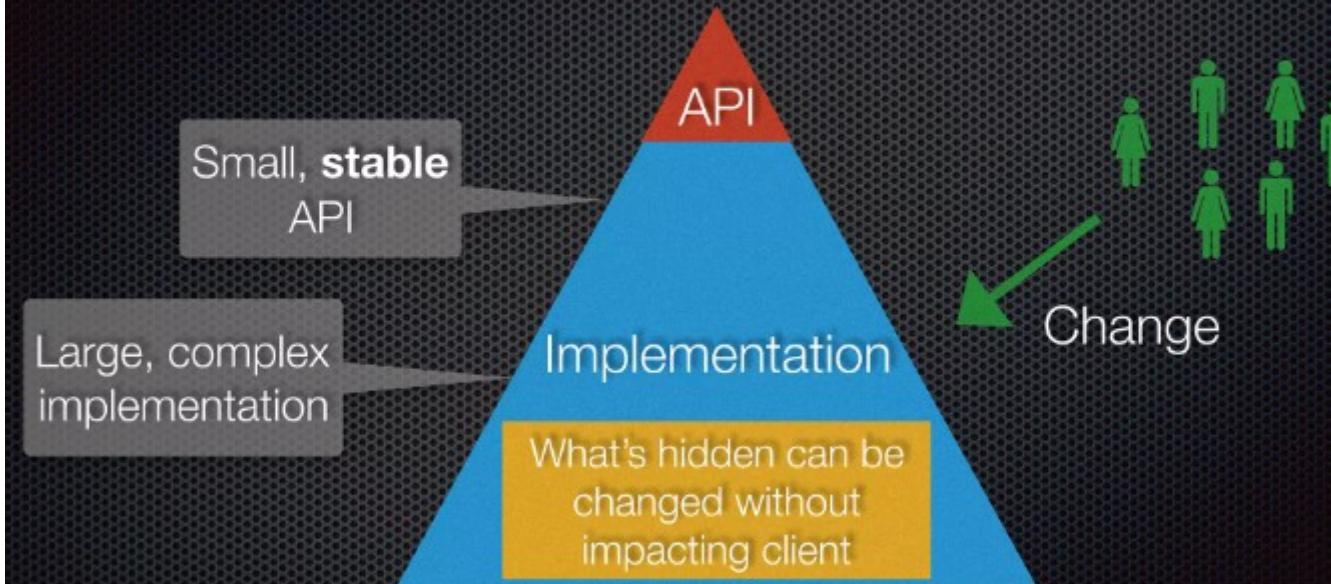
```
{  
...  
"tax": ...  
"serviceFee": ...  
"deliveryFee": ...  
"total": "12.34"  
...  
}
```

Consumer

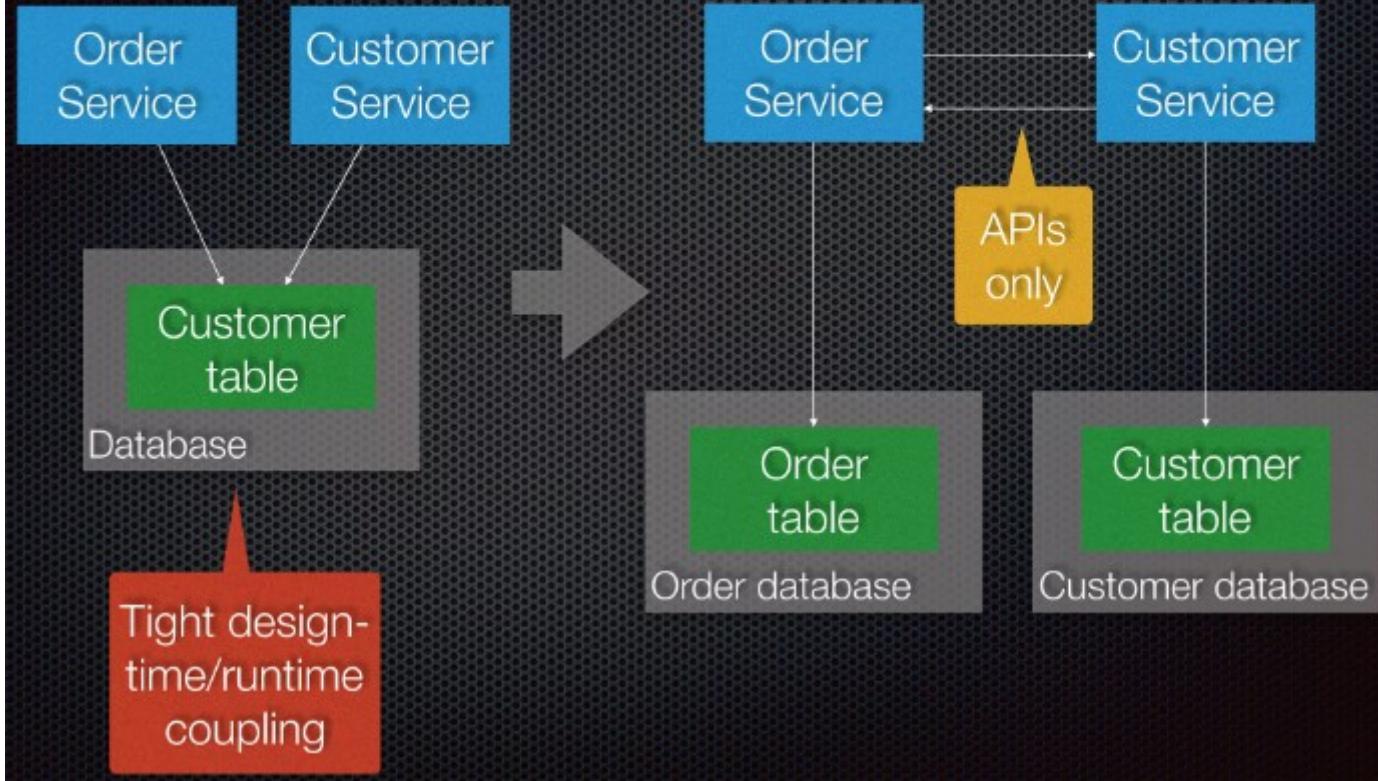
```
class Order {  
    Money total;  
}
```

Icebergs: expose as little as possible

Twilio: sendSms(from, to, message)



Use a database-per-service



Essential: Use resilience patterns for synchronous communication

4. Resilience patterns

name	how does it work?	description	slogans	links
Retry	repeats failed executions	Many faults are transient and may self-correct after a short delay.	"Insert coin to try again", "Maybe it's just a blip"	overview , documentation
Circuit Breaker	temporary blocks possible failures	When a system is seriously struggling, failing fast is better than making clients wait.	"Stop doing it if it hurts", "Give that system a break", "Baby, don't hurt me, no more"	overview , documentation , Feign , Retrofit
Rate Limiter	limits executions/period	Prepare for a scale and establish reliability and HA of your service.	"That's enough for this minute!", "Well, it'll work next time"	overview , documentation , Feign , Retrofit
Time Limiter	limits duration of execution	Beyond a certain wait, a success result is unlikely.	"Don't wait forever"	
Bulkhead	limits concurrent executions	Resources are isolated into pools so that if one fails, the others will continue.	"One fault shouldn't sink the whole ship", "Please, please, not all at once."	overview , documentation
Cache	memorizes a successful result	Some proportion of requests may be similar.	"You've asked that one before"	
Fallback	provides an alternative result for failures	Things will still fail - plan what you will do when that happens.	"Degrade gracefully", "A bird in the hand is worth two in the bush"	Try:recover , Spring , Feign

Above table is based on [Polly: resilience policies](#).

<https://github.com/resilience4j/resilience4j#resilience-patterns>

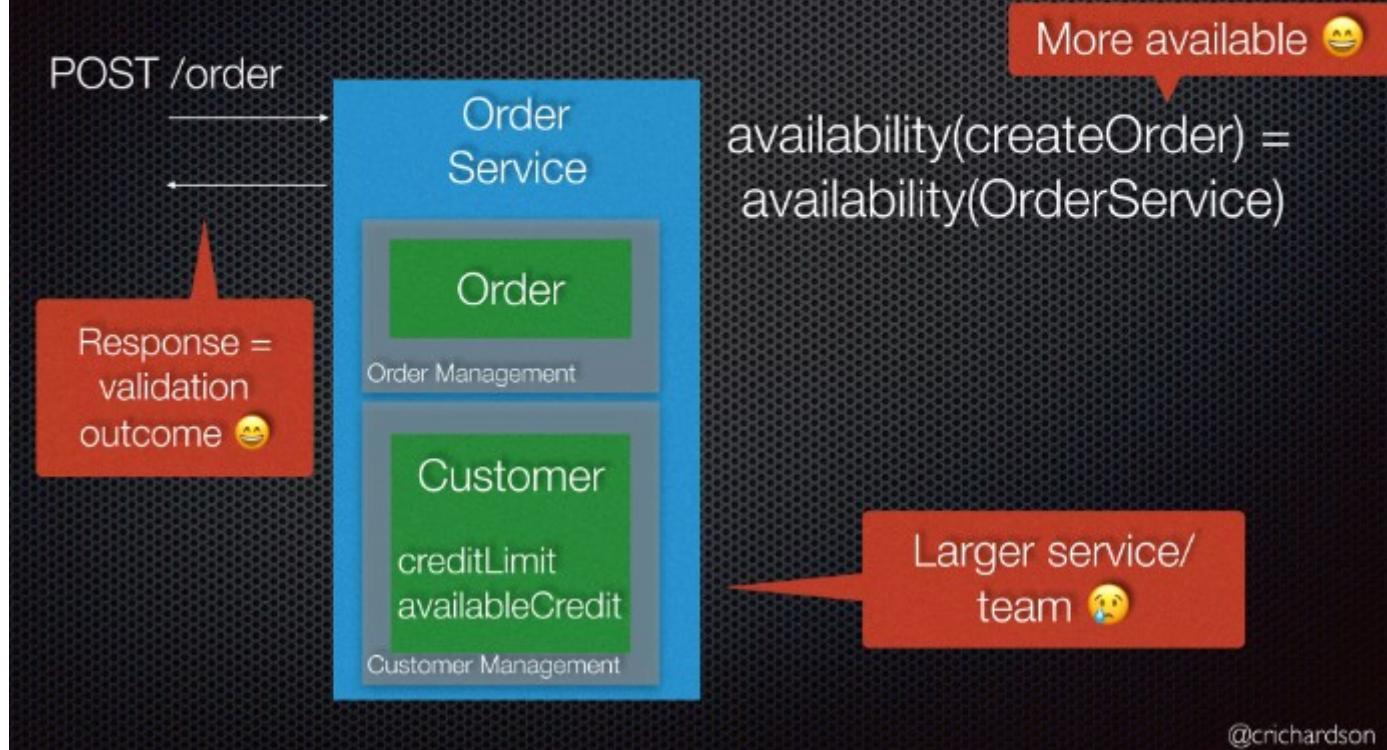
@crichtson

Self-contained service:

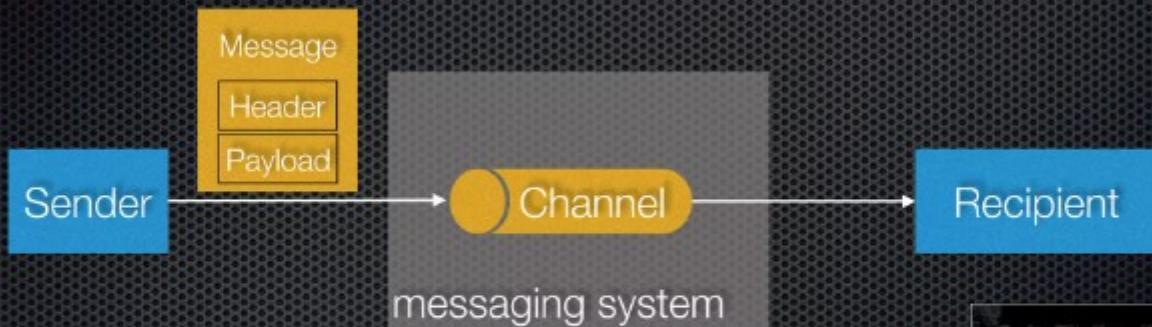
Can handle a synchronous request without waiting for a response from another service

<https://microservices.io/patterns/decomposition/self-contained-service.html>

Improving availability: replace service with module

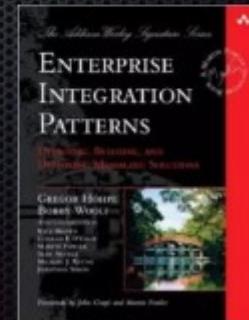


Use asynchronous messaging



Channel types:

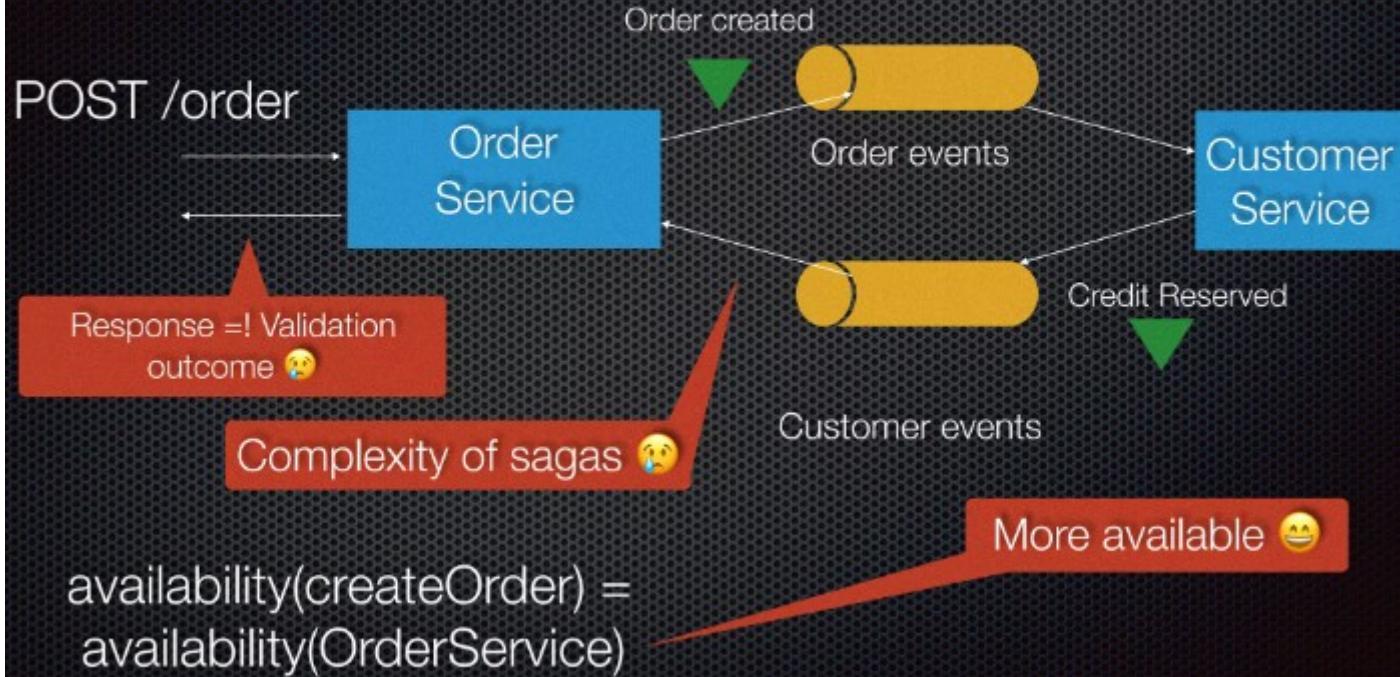
- Point-to-point
- Publish/Subscribe



<http://bit.ly/books-eip>

@crichtson

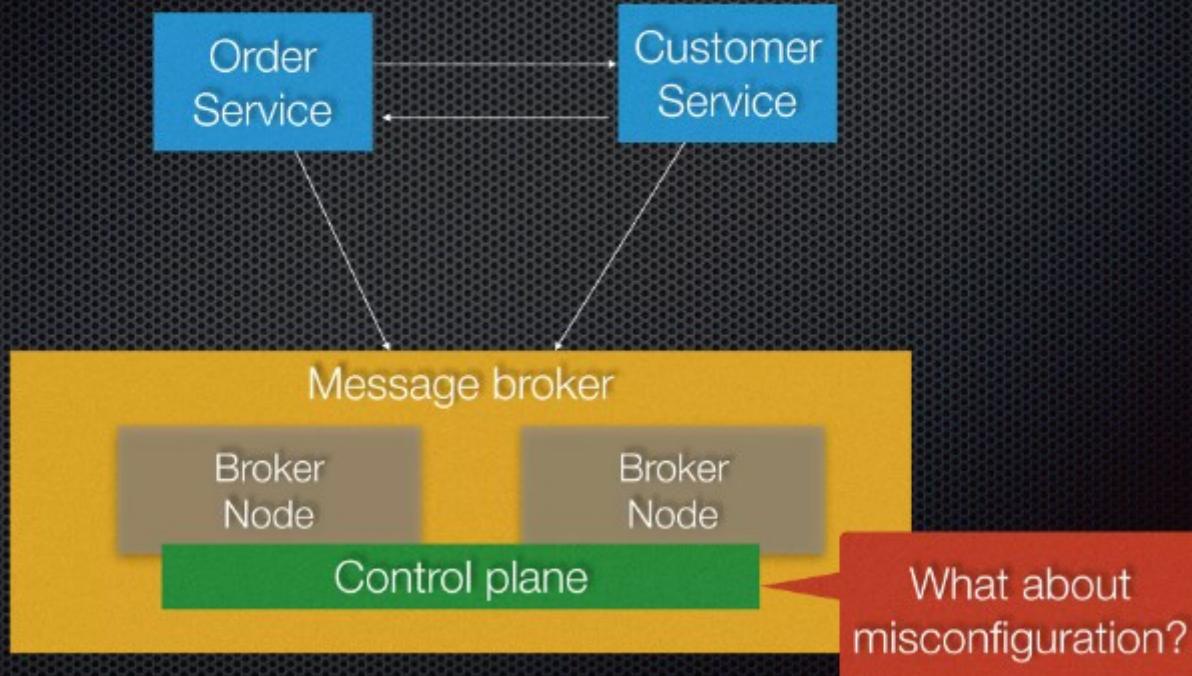
Improving availability: sagas



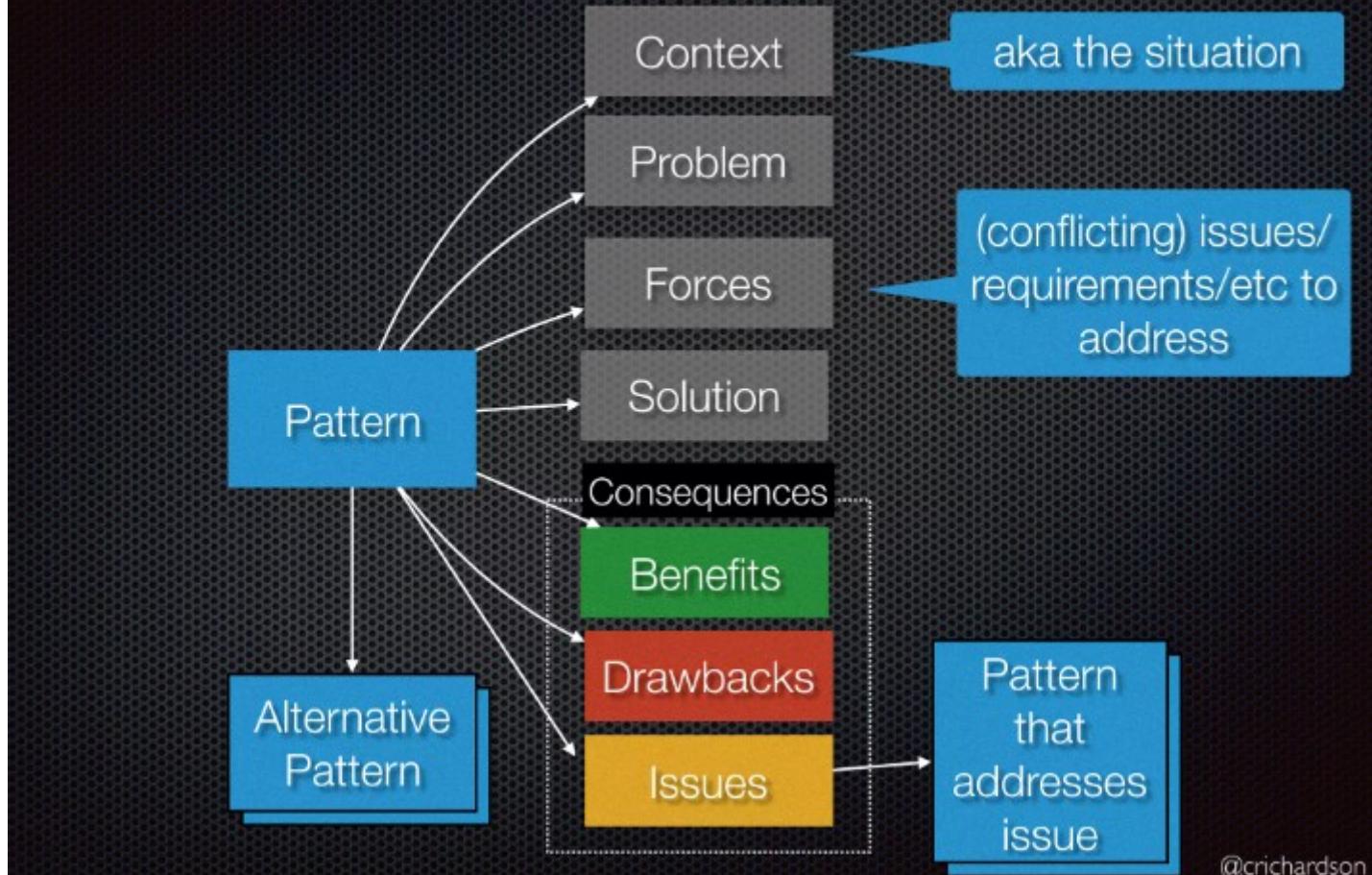
<https://microservices.io/patterns/data/saga.html>

@crichtson

Message brokers need to be shared!

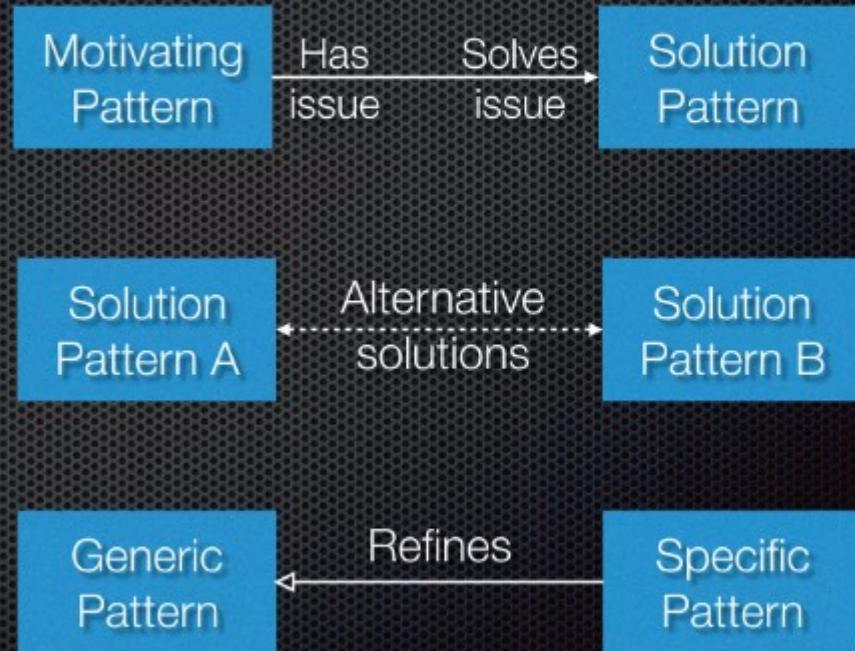


The structure of a pattern



Pattern language

A collection of related patterns that solve problems in a particular domain



The pattern language guides you when developing an architecture

- What architectural decisions you must make (a.k.a. problems to solve)
- For each decision/problem:
 - Available patterns
 - Trade-offs of each pattern

The context, problem and forces

- Context: IT must deliver software rapidly, frequently, reliably and sustainably
- Problem: what's the architecture?
- Forces
 - Simplicity
 - Testability
 - Deployability
 - Modularity / Loose coupling
 - Evolvability (of technology stack)

Pattern: microservice architecture

An architectural style
that structures an application as a set of deployable/ executable units, a.k.a. services

A service is:

- Highly maintainable and testable
- Minimal lead time (time from commit to deploy)
- Loosely coupled
- Independently deployable
- Implements a business capability
- Owned/developed/tested/deployed by a small team

Benefits and drawbacks

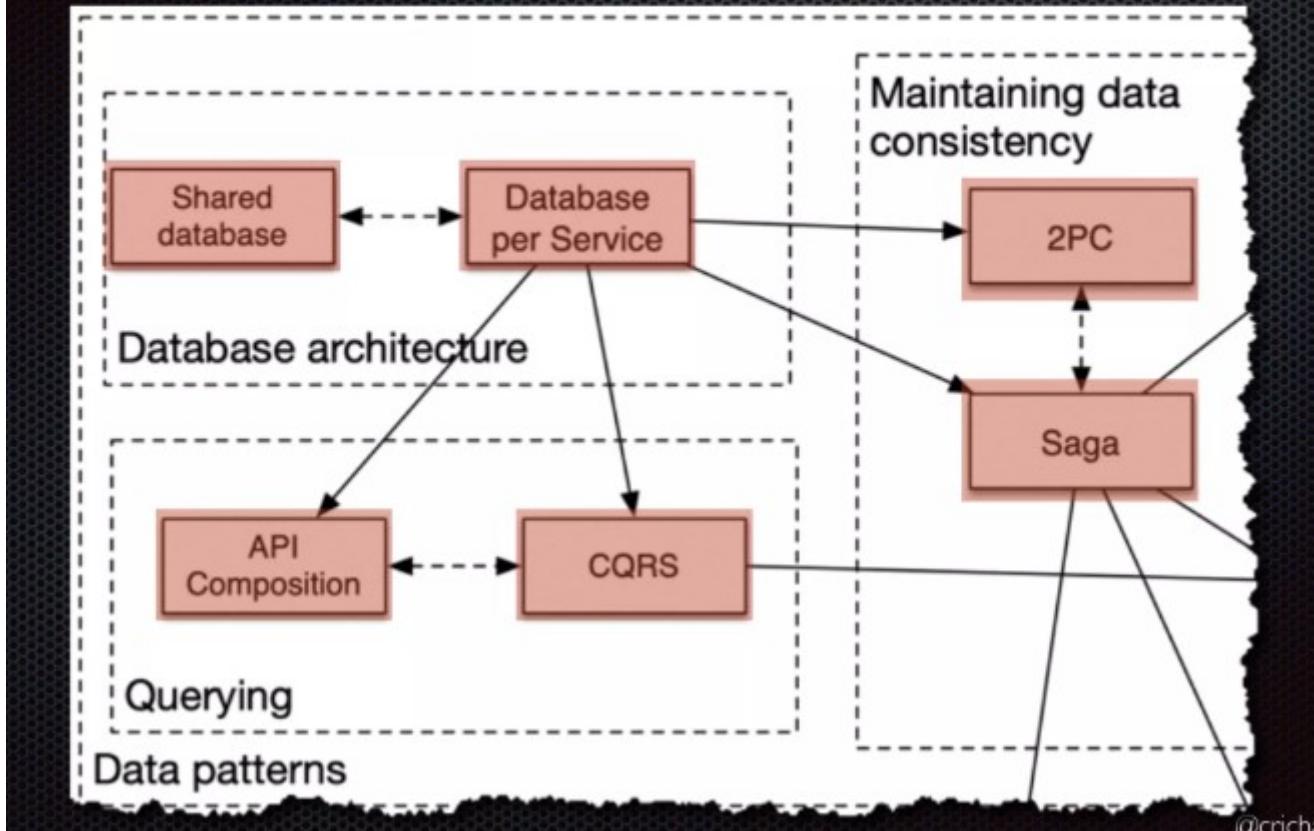
Benefits

- Multiple technology stacks/ incremental evolution
- Teams can deploy changes independently
- Fast deployment pipeline
- Improved fault isolation

Drawbacks

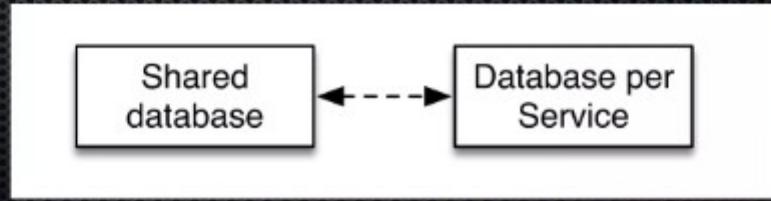
- Potential complexity of distributed architecture: eventual consistency, inter-service communication
- More complex operations: observability and troubleshooting

Applying the data patterns



What's the database architecture?

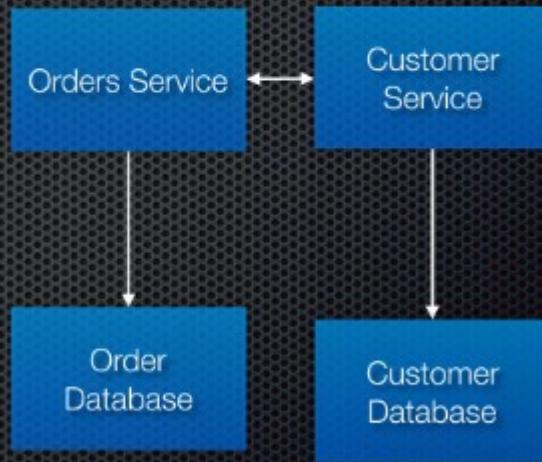
- Context:
 - You have applied the Microservice Architecture pattern
- Forces:
 - Services must be loosely coupled - both design-time and runtime



@crichardson

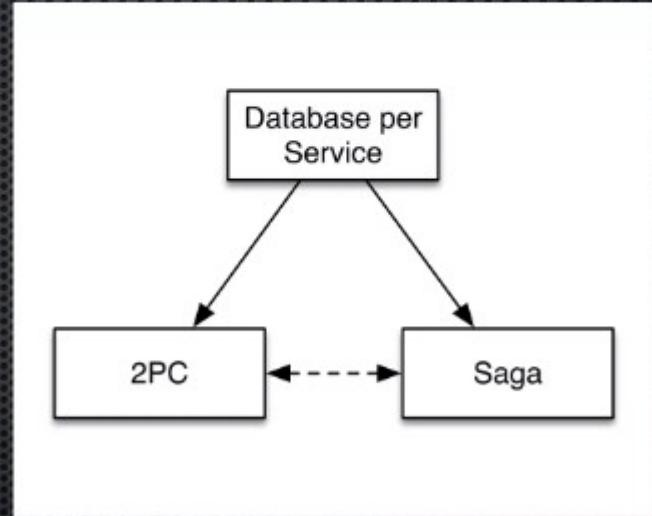
Pattern: Database per Service

- Benefits:
 - Loose design-time coupling
 - Loose runtime coupling
- Drawbacks:
 - More complex, eventual consistency



How to implement transactions?

- Context:
 - You have applied the Microservice architecture and Database per service patterns
- Problem:
 - How to implement transactions that span services/databases?
- Forces:
 - Services must be loosely coupled



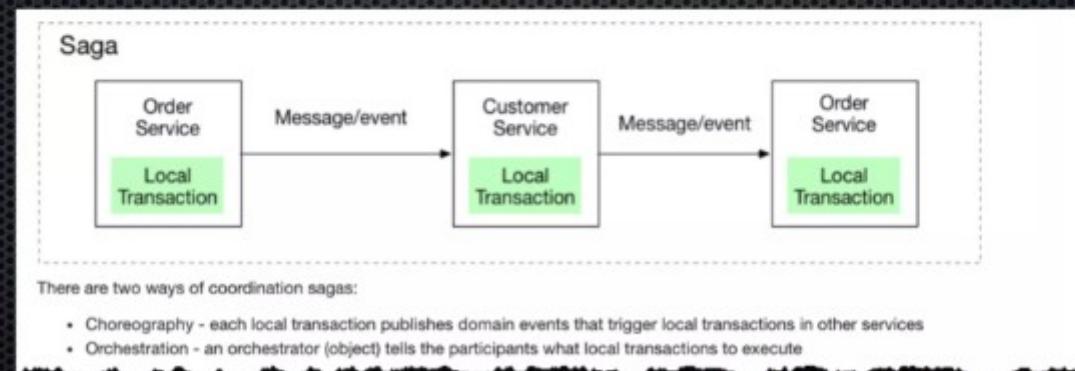
Pattern: Saga

Benefits

- Loose runtime coupling

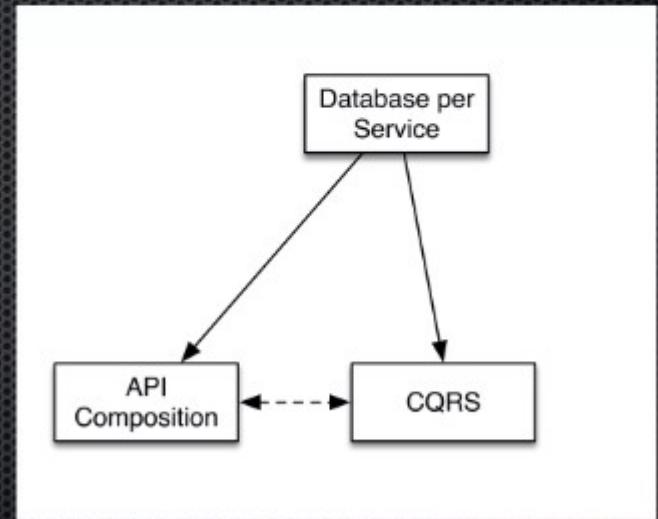
Drawbacks

- Complexity of eventual consistency



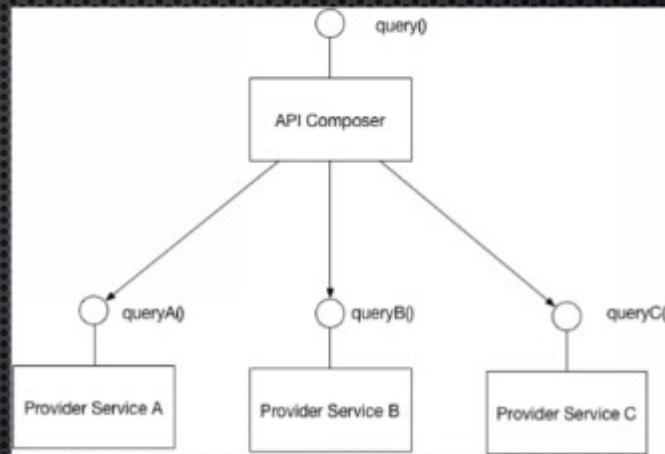
How to implement queries?

- Context:
 - You have applied the Microservice architecture and Data per service patterns
- Problem:
 - How to implement queries that span services?
- Forces:
 - Services must be loosely coupled



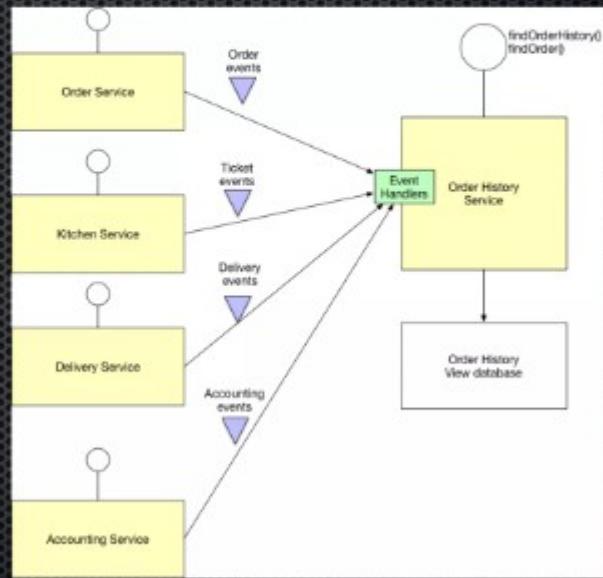
Pattern: API Composition

- Benefits:
 - Simple
- Drawbacks:
 - Tight runtime coupling
 - Potentially inefficient
 - Not transactionally consistent



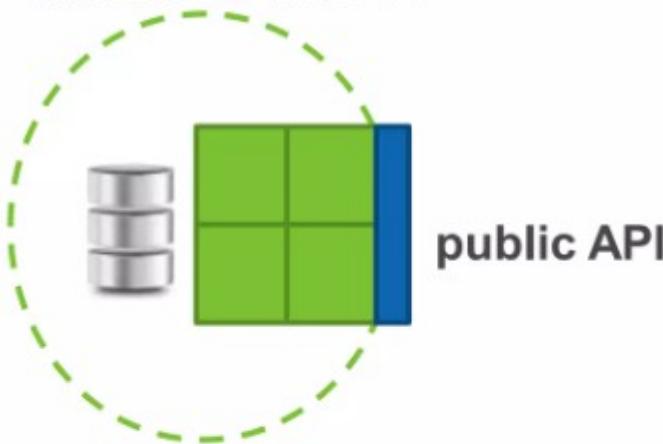
Pattern: CQRS

- Benefits:
 - Loose runtime coupling
 - Efficient queries
- Drawbacks:
 - Cost and complexity of replication
 - Not transactionally consistent



Principle 1: Microservices only rely on each other's public API (Evolve API in backward-compatible way...and document!)

Microservice A



Version 1.0.0

storeRestaurant (*id, name, cuisine*)

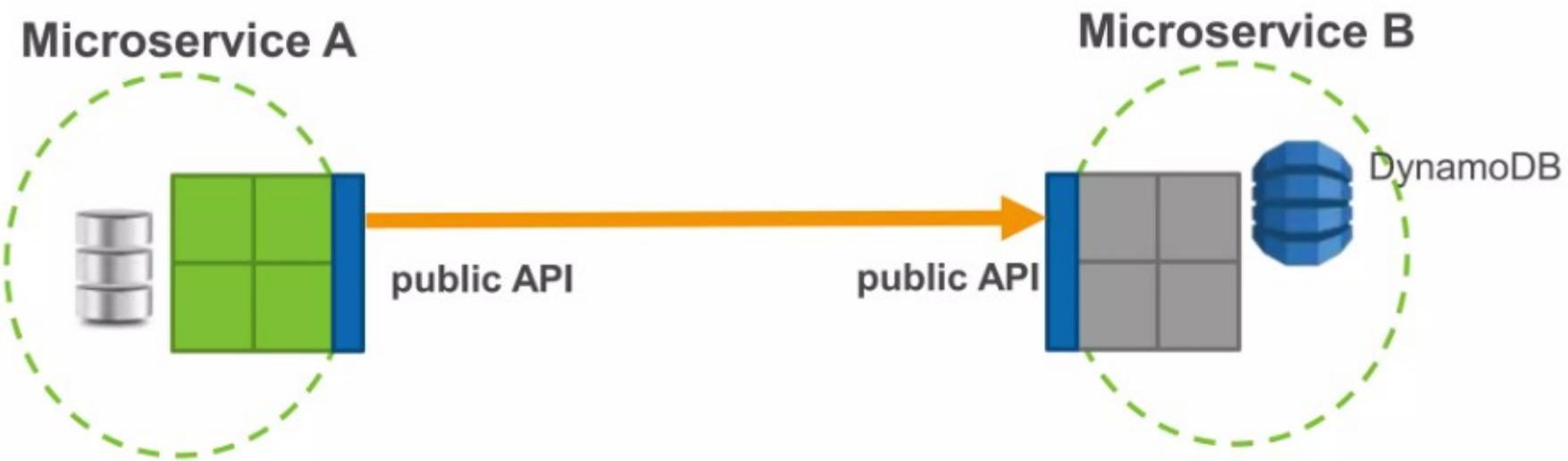
Version 1.1.0

storeRestaurant (*id, name, cuisine*)
storeRestaurant (*id, name, arbitrary_metadata*)
addReview (*restaurantId, rating, comments*)

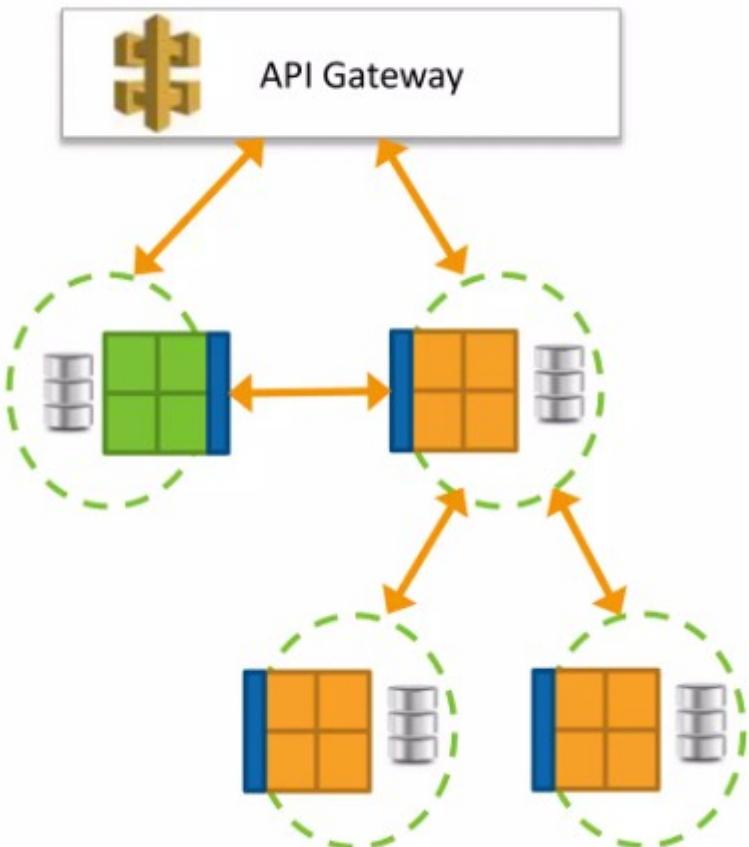
Version 2.0.0

storeRestaurant (*id, name, arbitrary_metadata*)
addReview (*restaurantId, rating, comments*)

Principle 2: Use the right tool for the job (Embrace polyglot persistence)



Principle 3: Secure Your Services



- **Defense-in-depth**
 - Network level (e.g. VPC, Security Groups, TLS)
 - Server/container-level
 - App-level
 - IAM policies
- **Gateway** ("Front door")
- **API Throttling**
- **Authentication & Authorization**
 - Client-to-service, as well as service-to-service
 - API Gateway: custom Lambda authorizers
 - IAM-based Authentication
 - Token-based auth (JWT tokens, OAuth 2.0)
- **Secrets management**
 - S3 bucket policies + KMS + IAM
 - Open-source tools (e.g. Vault, Keywhiz)

Principle 4: Be a good citizen within the ecosystem

Distributed monitoring and tracing

- “Is the service meeting its SLA?”
- “Which services were involved in a request?”
- “How did downstream dependencies perform?”

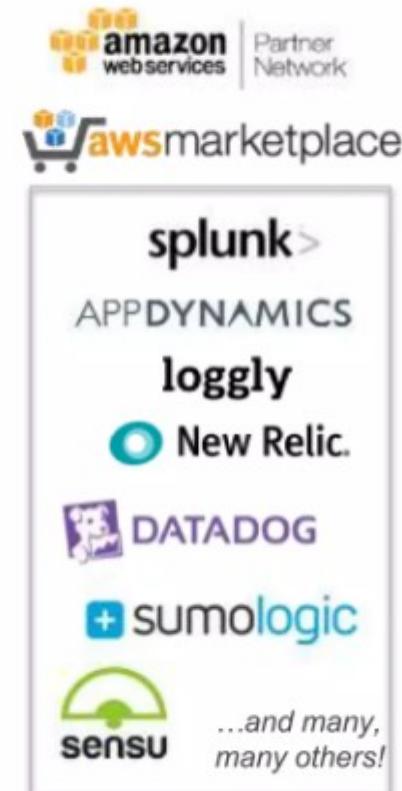
Shared metrics

- e.g. request time, time to first byte

+ User-experience metrics

Distributed tracing

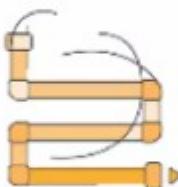
- e.g. AWS X-Ray, Zipkin, OpenTracing, Wingtips



Principle 5: Automate everything



AWS
CodeCommit



AWS
CodePipeline



AWS
CodeDeploy



RDS



DynamoDB



ElastiCache



CloudWatch



Cloud Trail



API Gateway



SQS



SWF



EC2



ECS



Lambda



Auto
Scaling



ELB



Elastic
Beanstalk



SES



SNS



Kinesis

