Microservices & Apache Kafka®

Part 2 – Building Event-Driven Services with Apache Kafka

Series Schedule

- Session 1: The Data Dichotomy: Rethinking the way we treat data and services
- **Session 2: Building Event-Driven Services with Apache Kafka**
- Session 3: Putting the ‘Micro’ into Microservices with Stateful Stream Processing

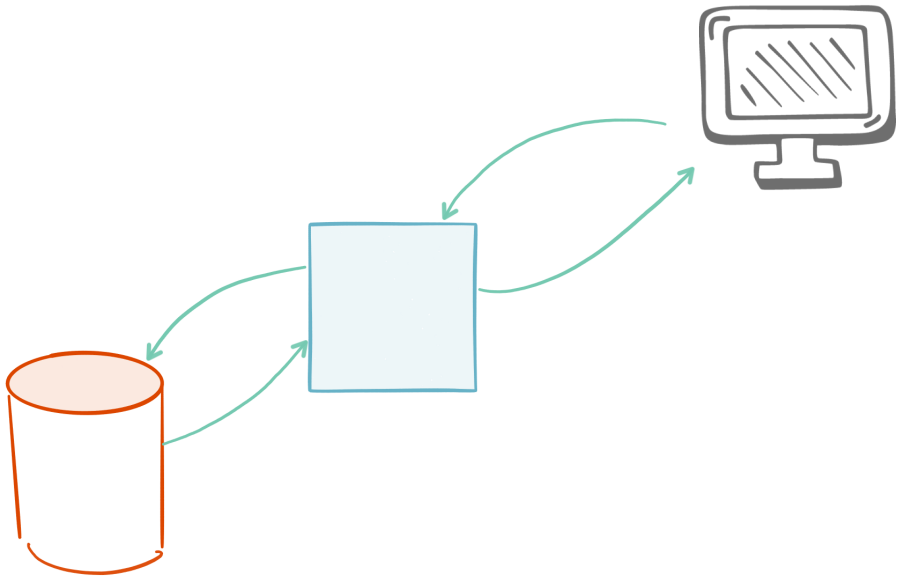
In this talk

1. How we traditionally build services with REST
2. How Event Driven Services are different
3. Building an Immutable, Shared Narrative.
4. Leveraging Materialized Views.
5. Relating to CQRS & Event Sourcing
6. Pulling it all together

Companies evolve in different ways

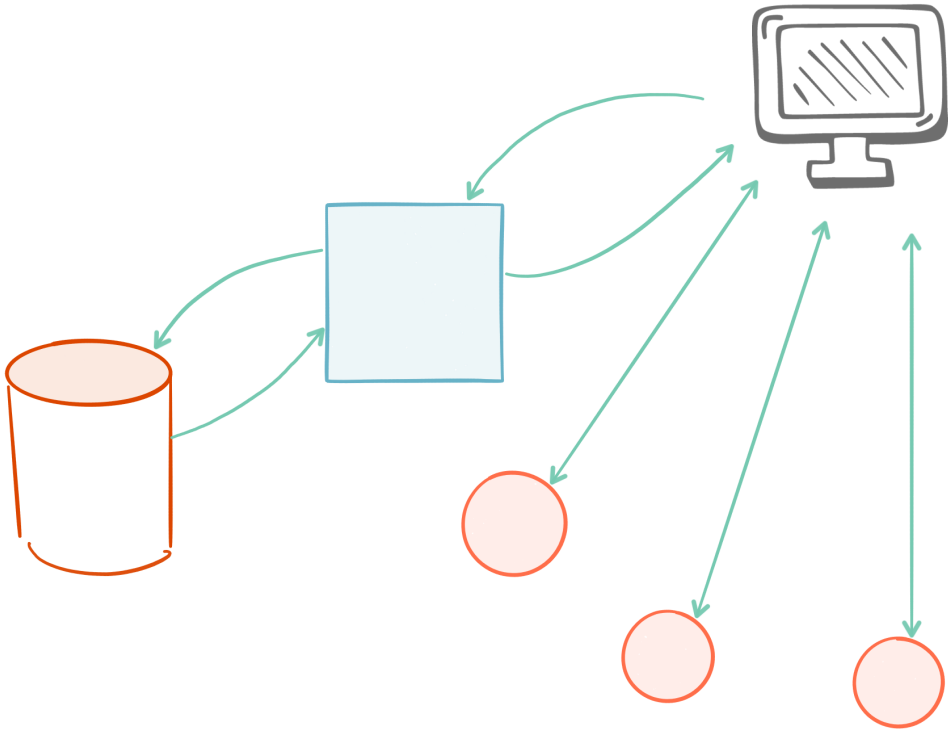
- Internet companies
- Enterprise companies

Internet companies typically grow from front-facing websites.

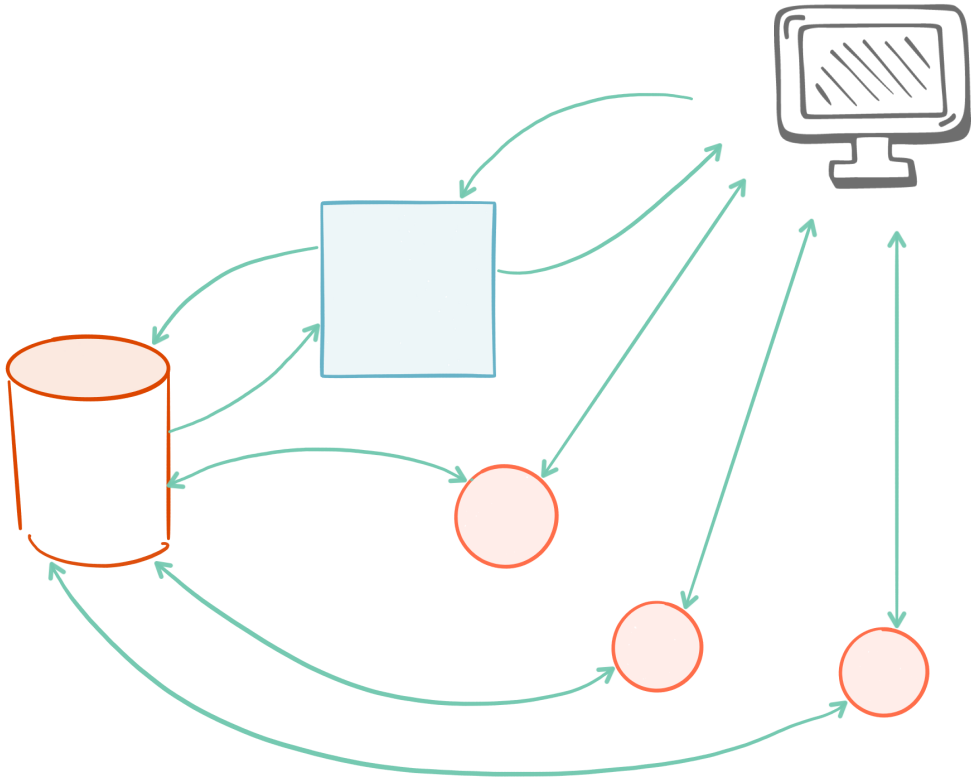


Growing fast

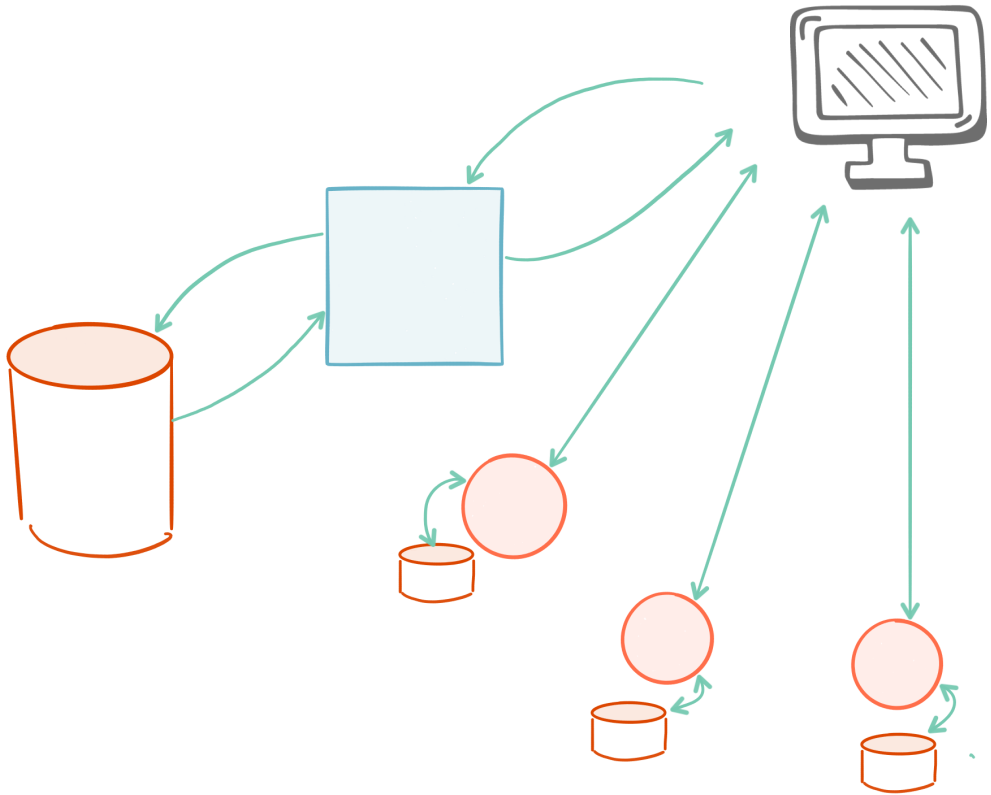
Carve pieces off



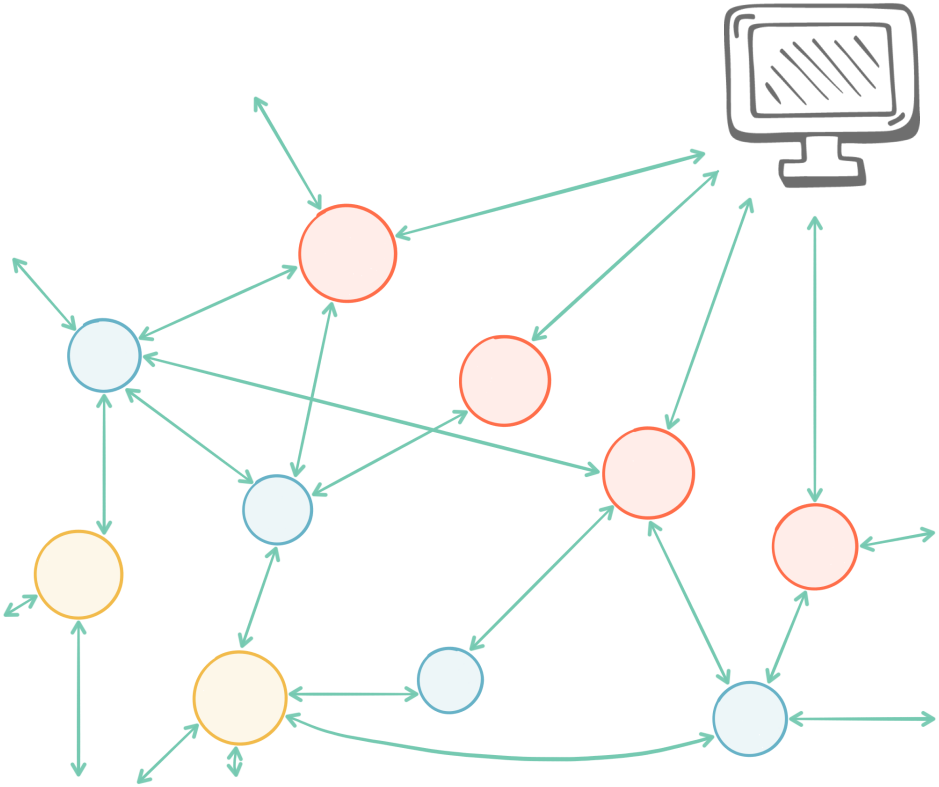
If stateful, they might share a DB



Or they might use their own

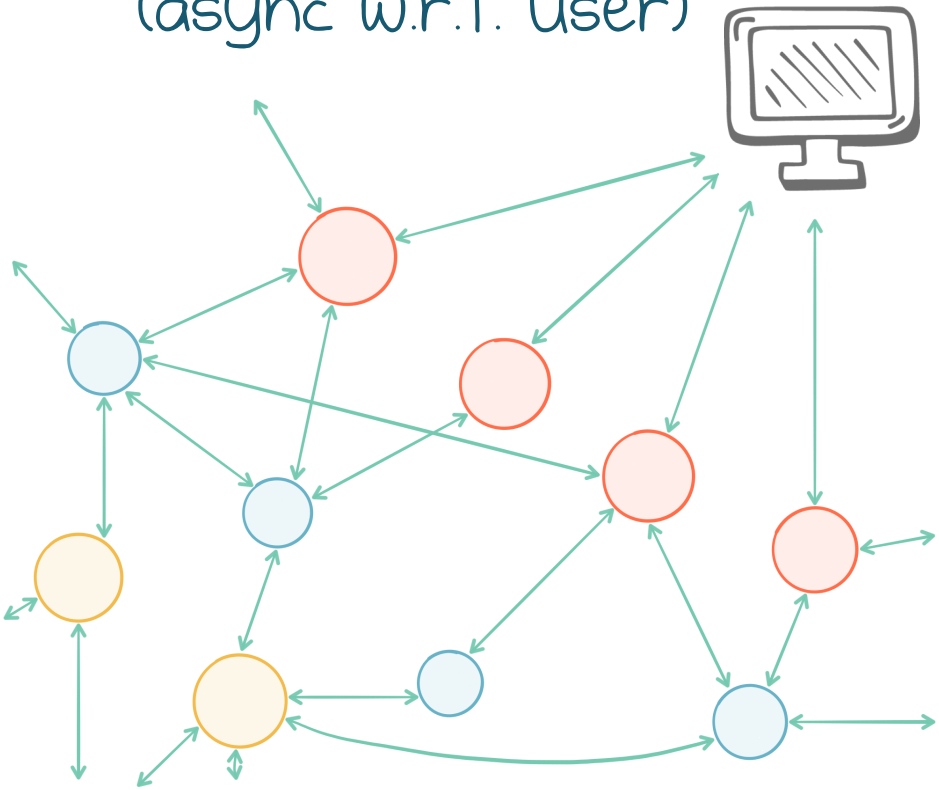


Over time => more complex

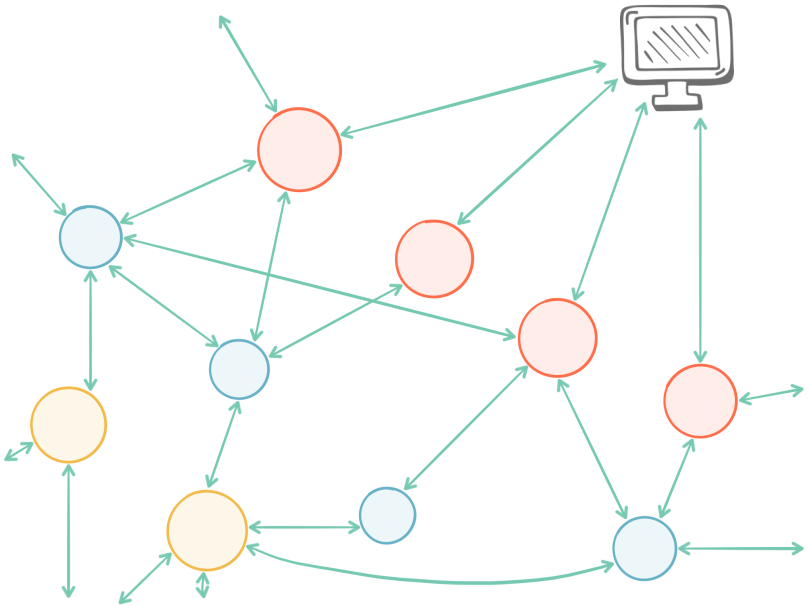


Over time => more async

(async w.r.t. user)



Chained blocking commands

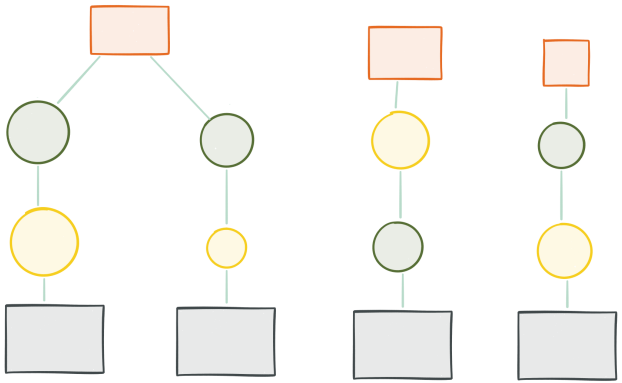


Buffering, handling failure, backpressure, scaling etc all get pushed into service's responsibilities

The synchronous world of request response protocols leads to tight, point-to-point couplings.

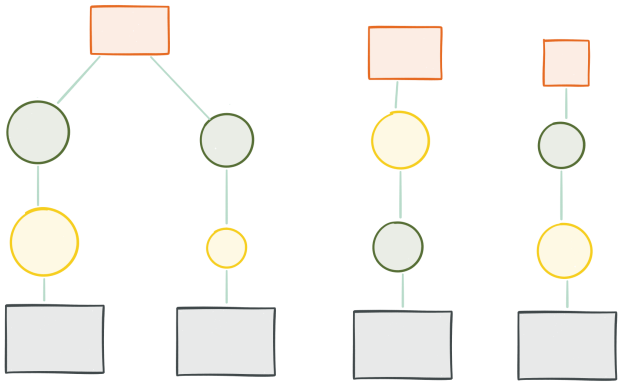
Enterprises are typically different

Enterprise companies form as heterogeneous application conglomerates, often in sylos

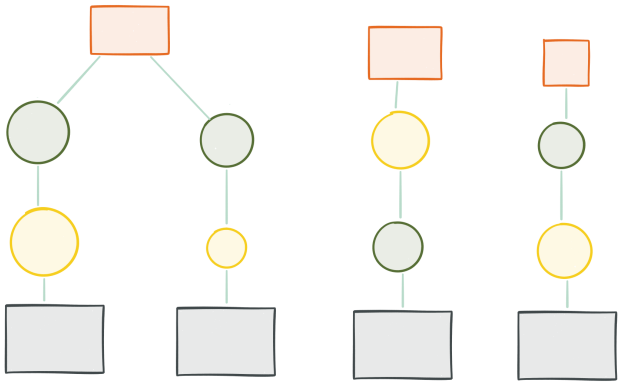


The start life as independent islands

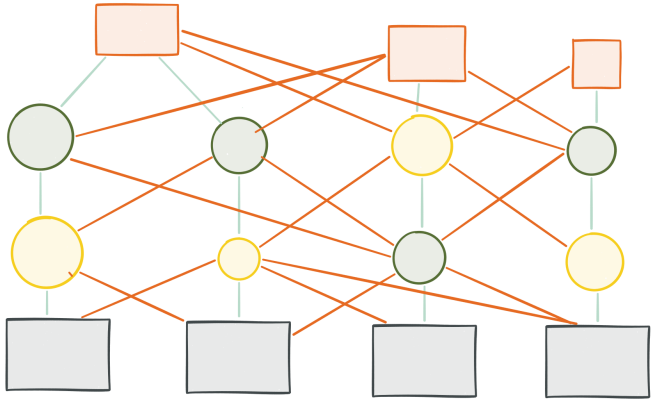
Linked together by files or via messaging



Interactions are async from day-1

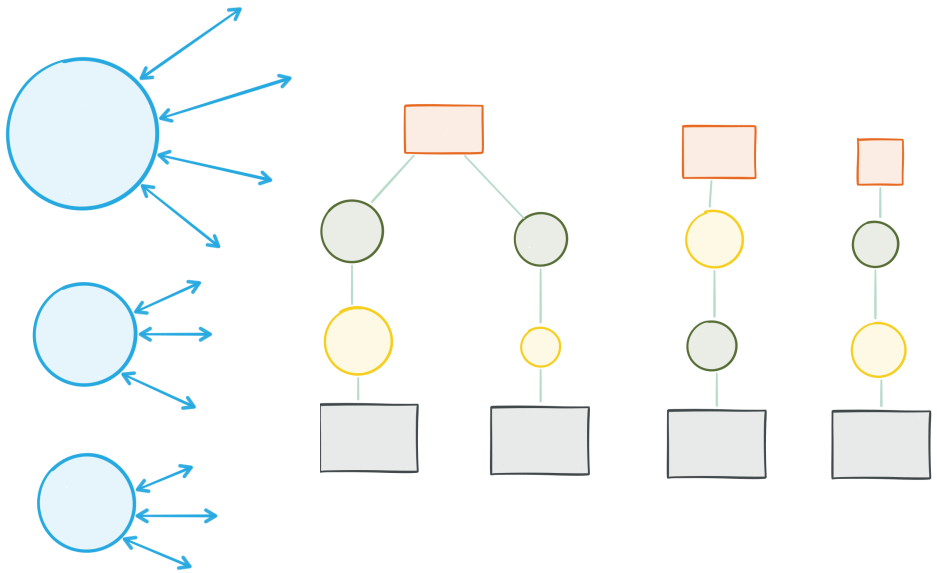


Forwarding of data gets messy: Telephone game



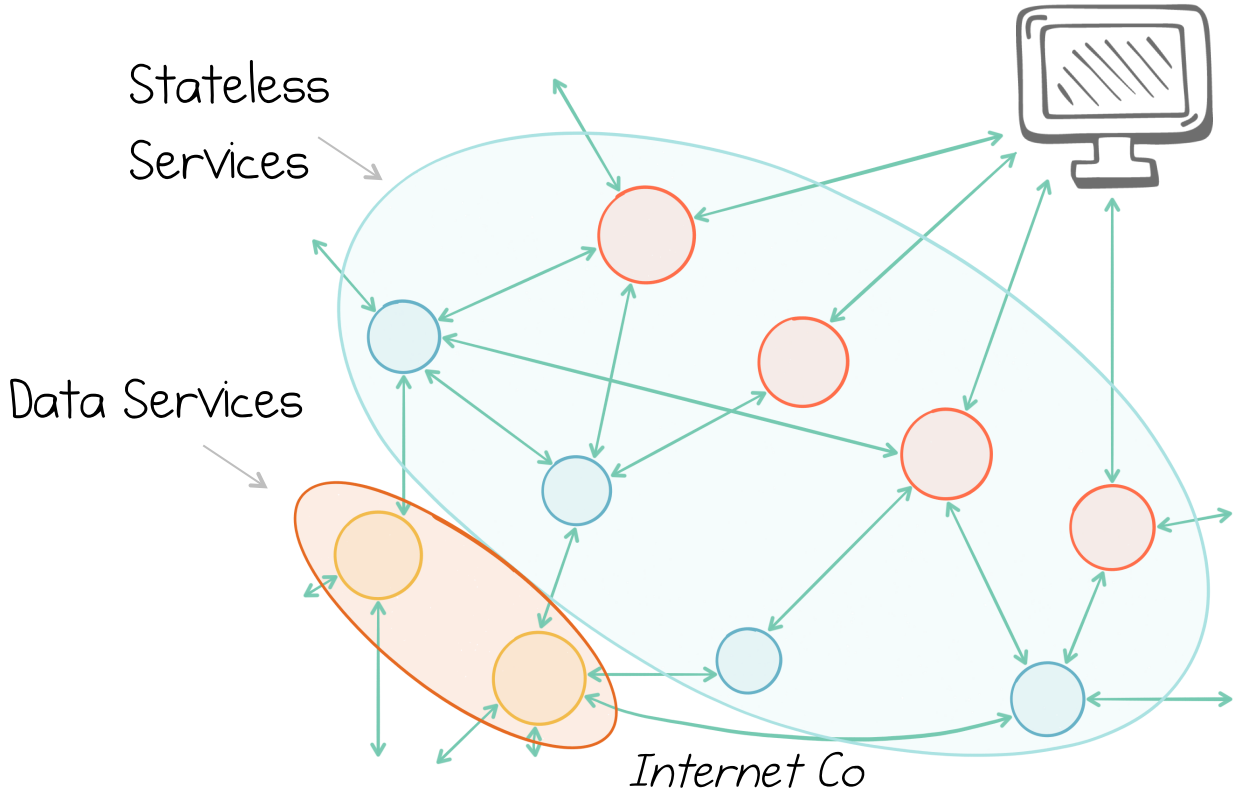
Both these approaches often end up at the same point, but for different reasons

Data is herded into specific services

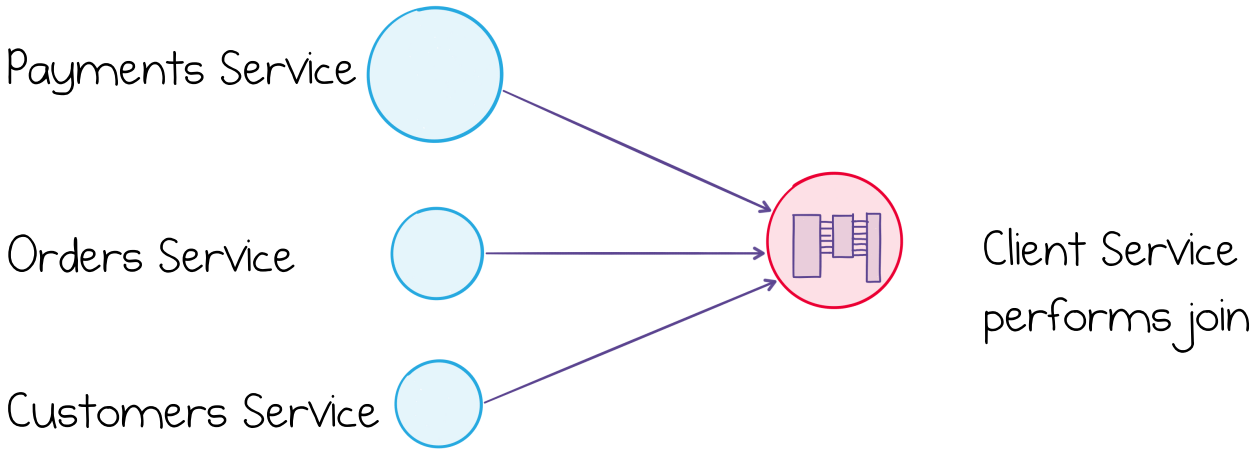


Enterprise

Data is herded into specific services

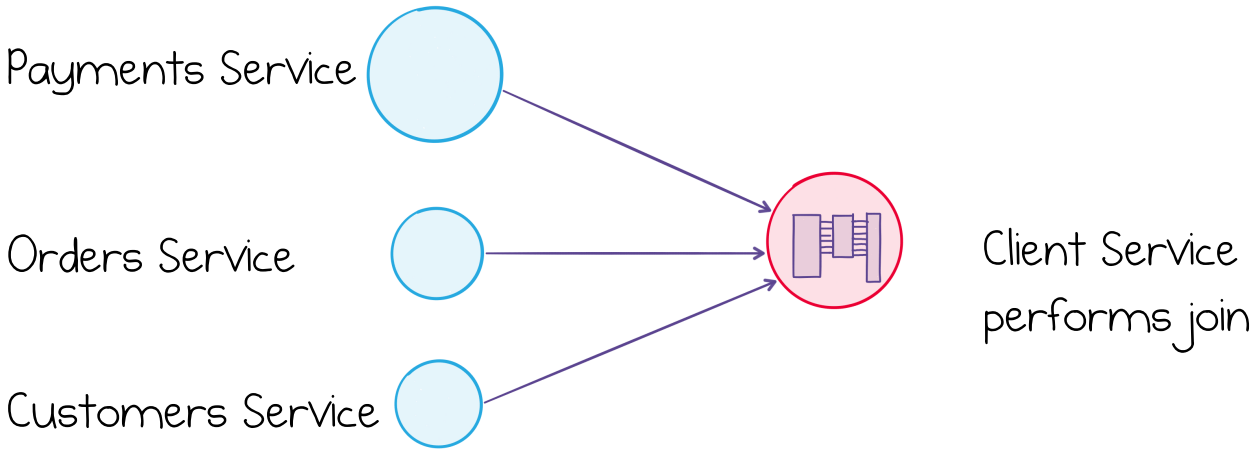


Lots of Data Services
=> Distributed Join Problem



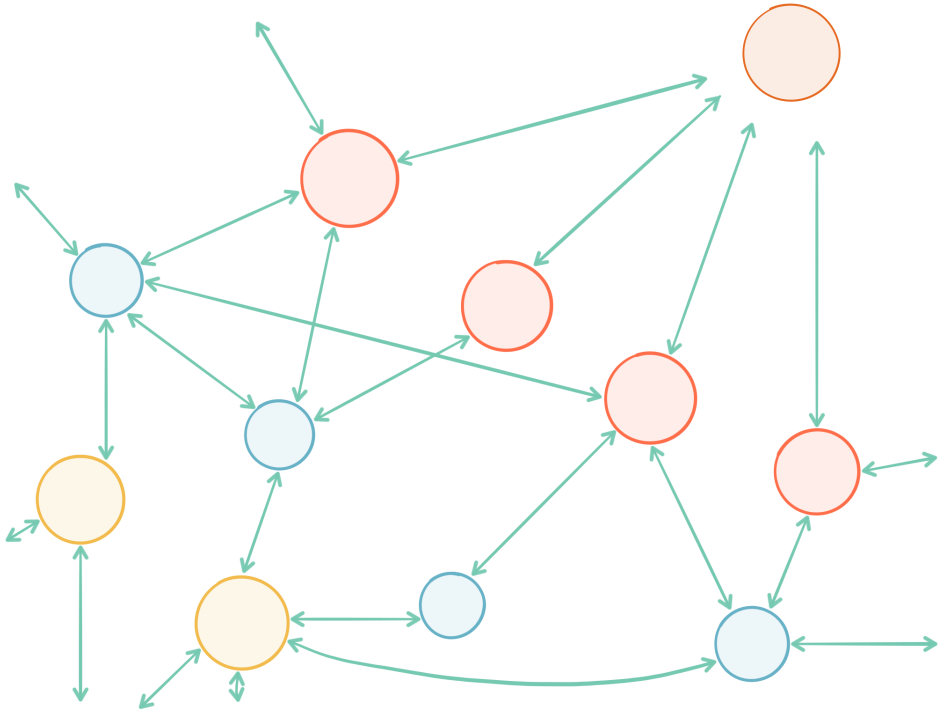
This is a problem databases find hard,
and they're highly tuned for it!

Lots of Data Services
=> Distributed Join Problem



TESTING BECOMES DIFFICULT!

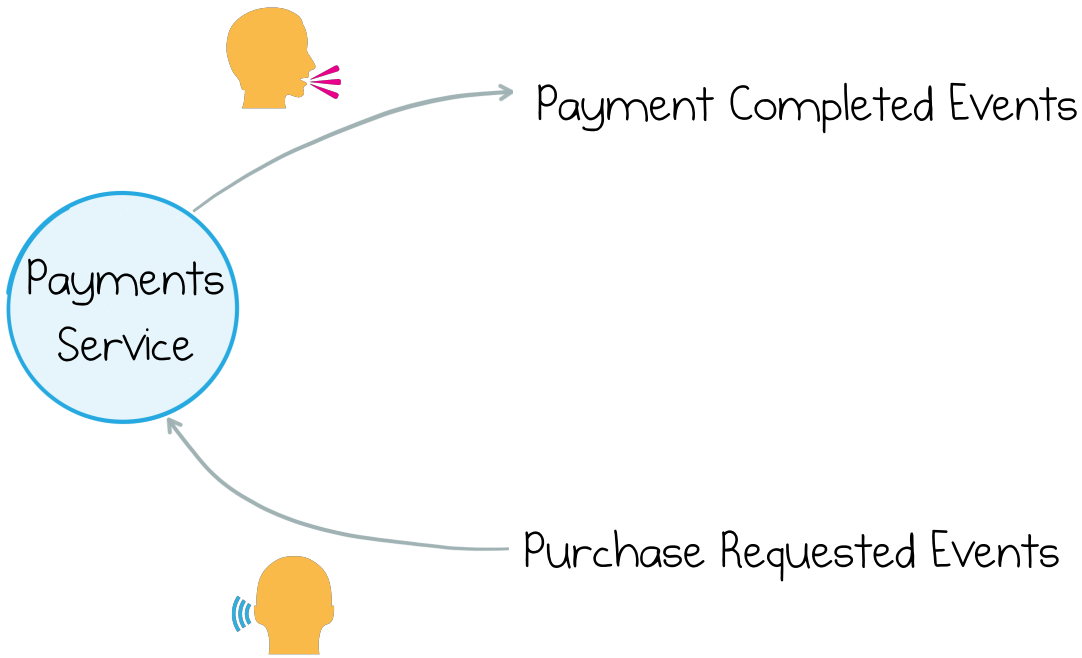
In short: the high degree of "connectedness" makes it hard to evolve independently and to scale



LETS TRY EVENT DRIVEN!

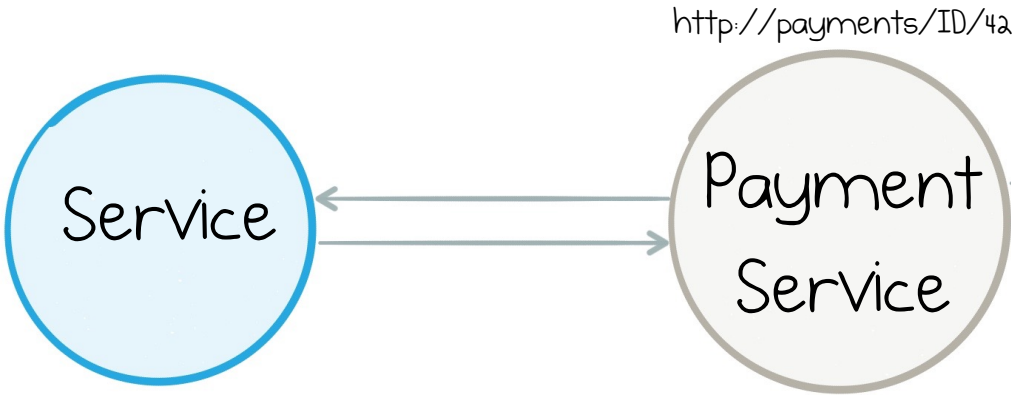
INTERACT THROUGH EVENTS, DON'T TALK TO SERVICES

This



Listen & React to Events

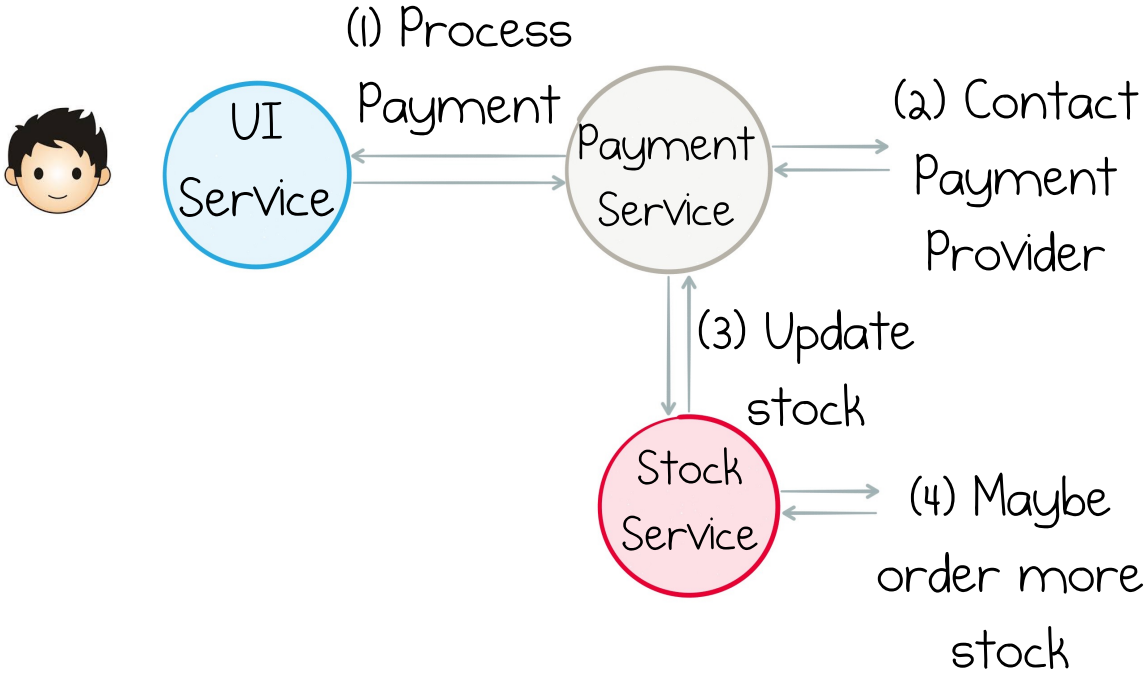
Not this



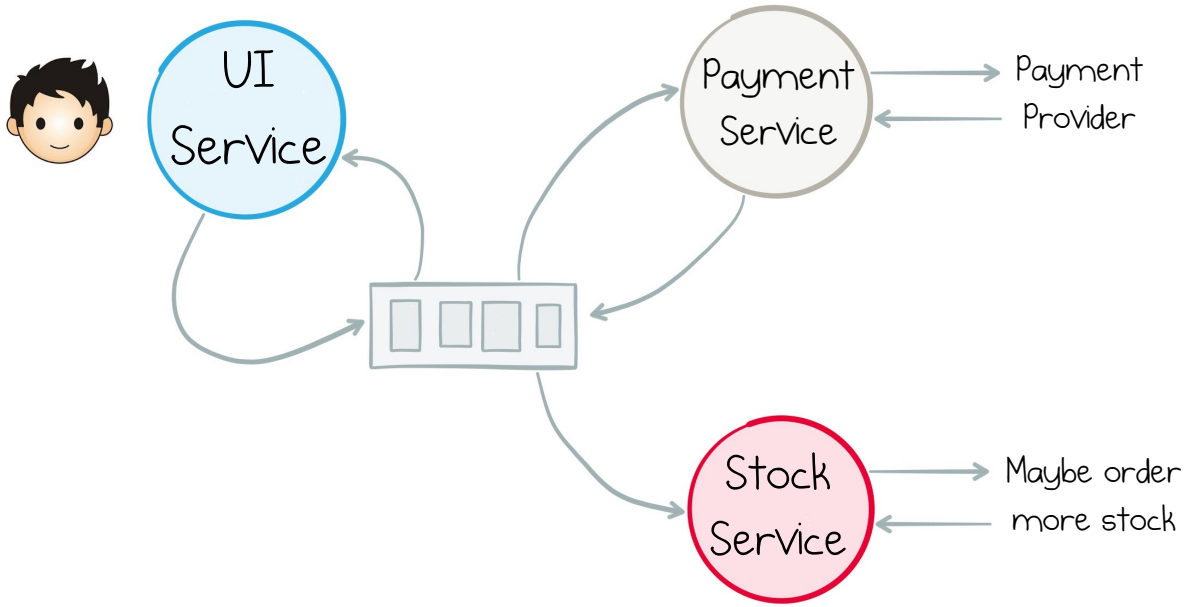
Don't Request/Command from individual services.

Let's take an example...

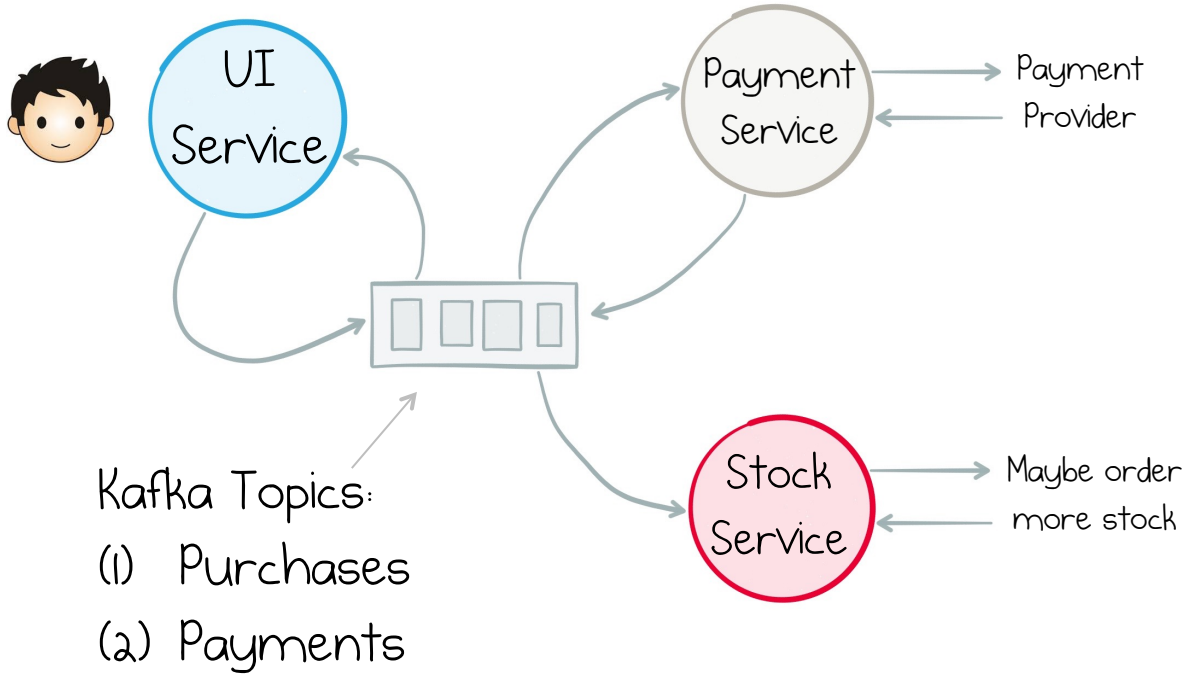
Request Response Example



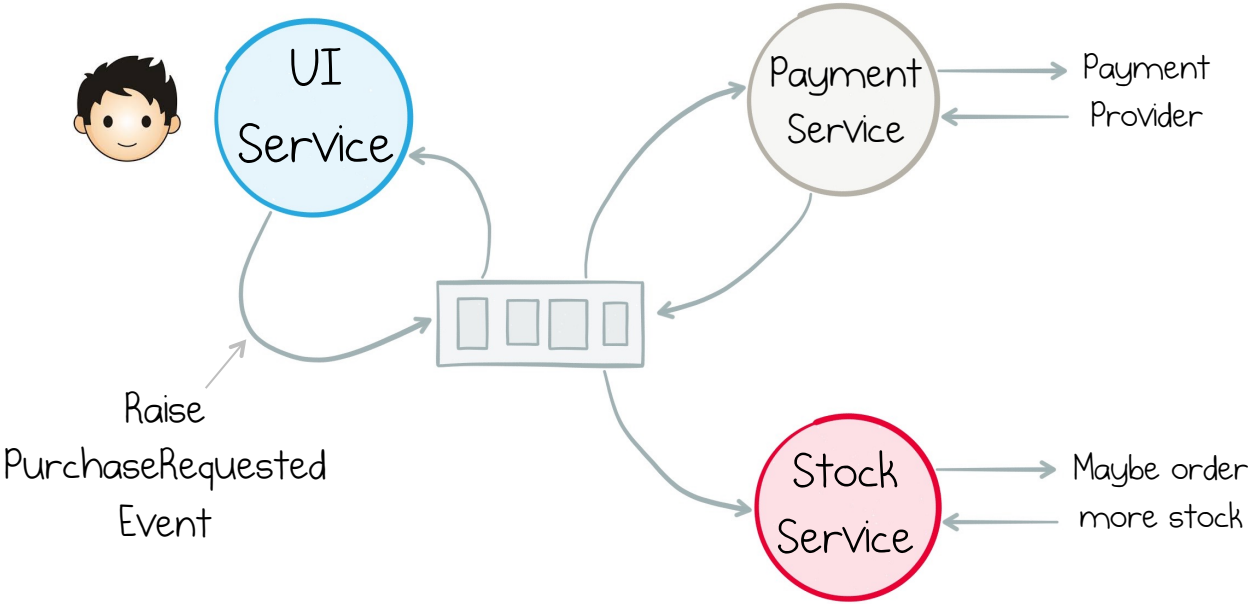
The Event Driven Way



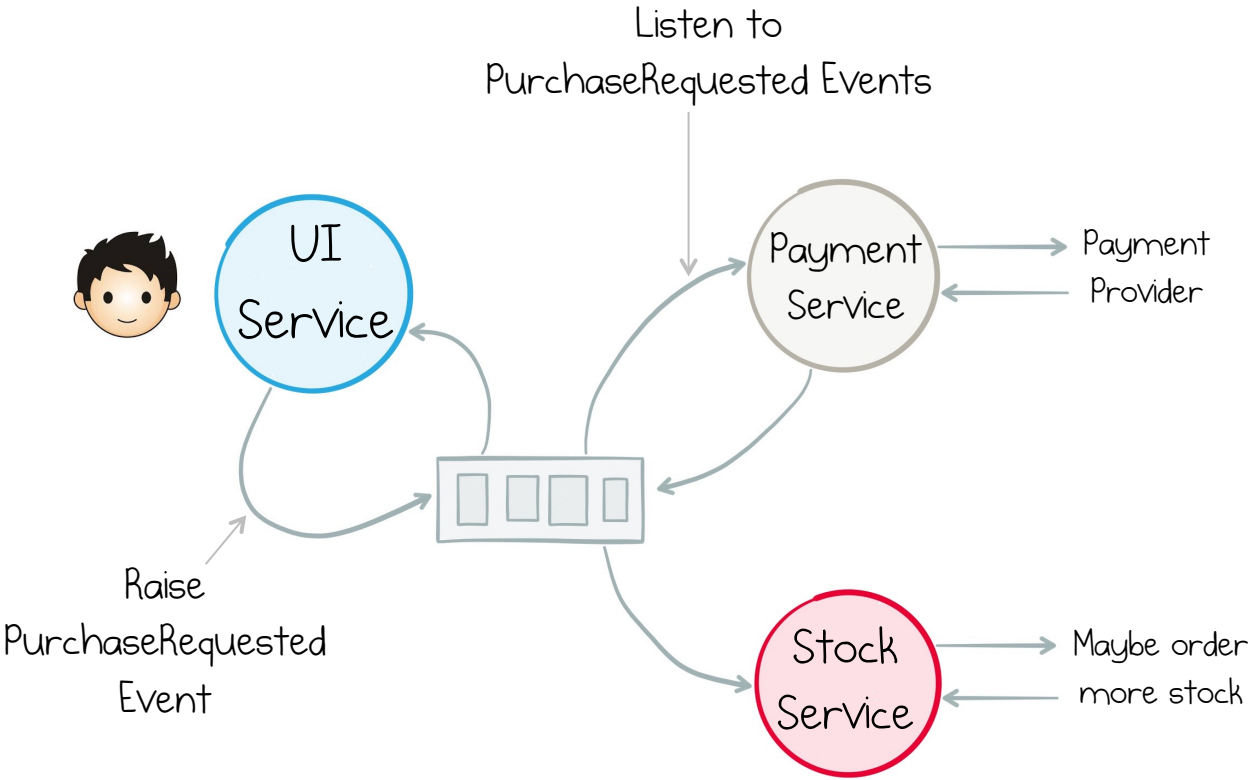
The Event Driven Way



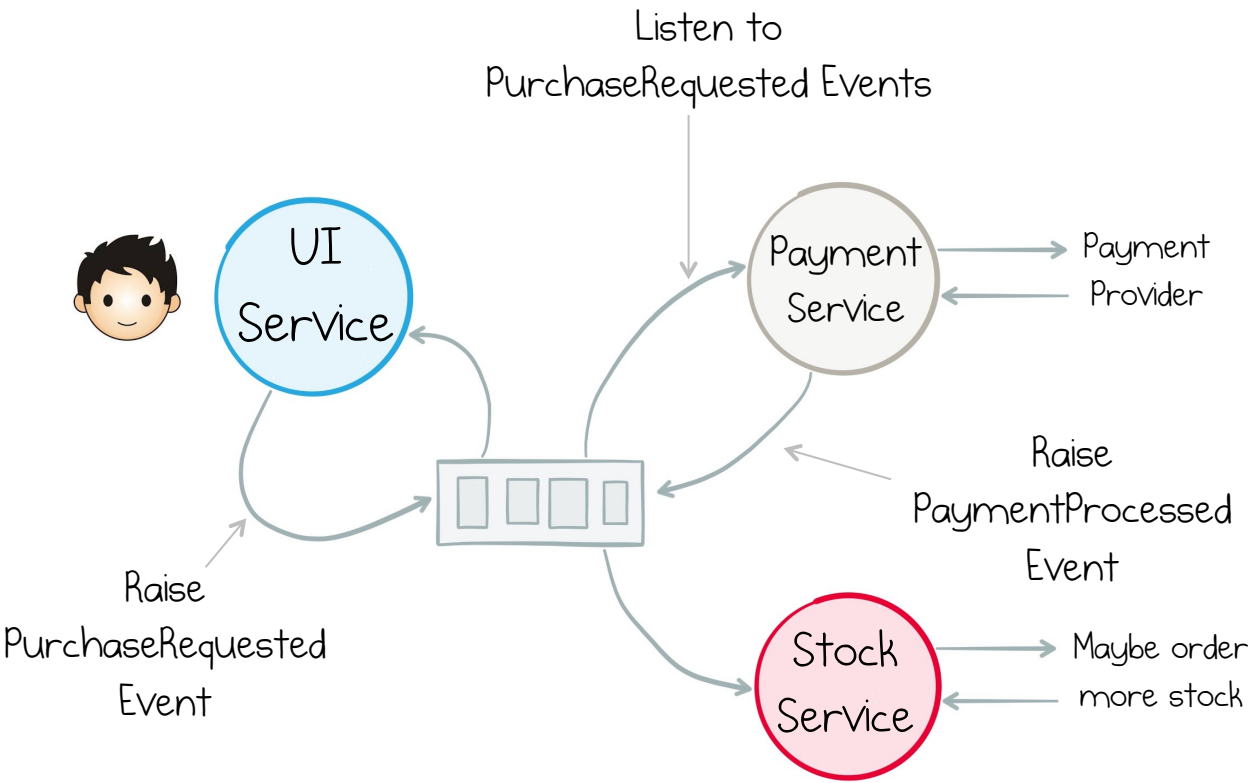
New Order -> PurchaseRequested Event



Payment Service Listens In

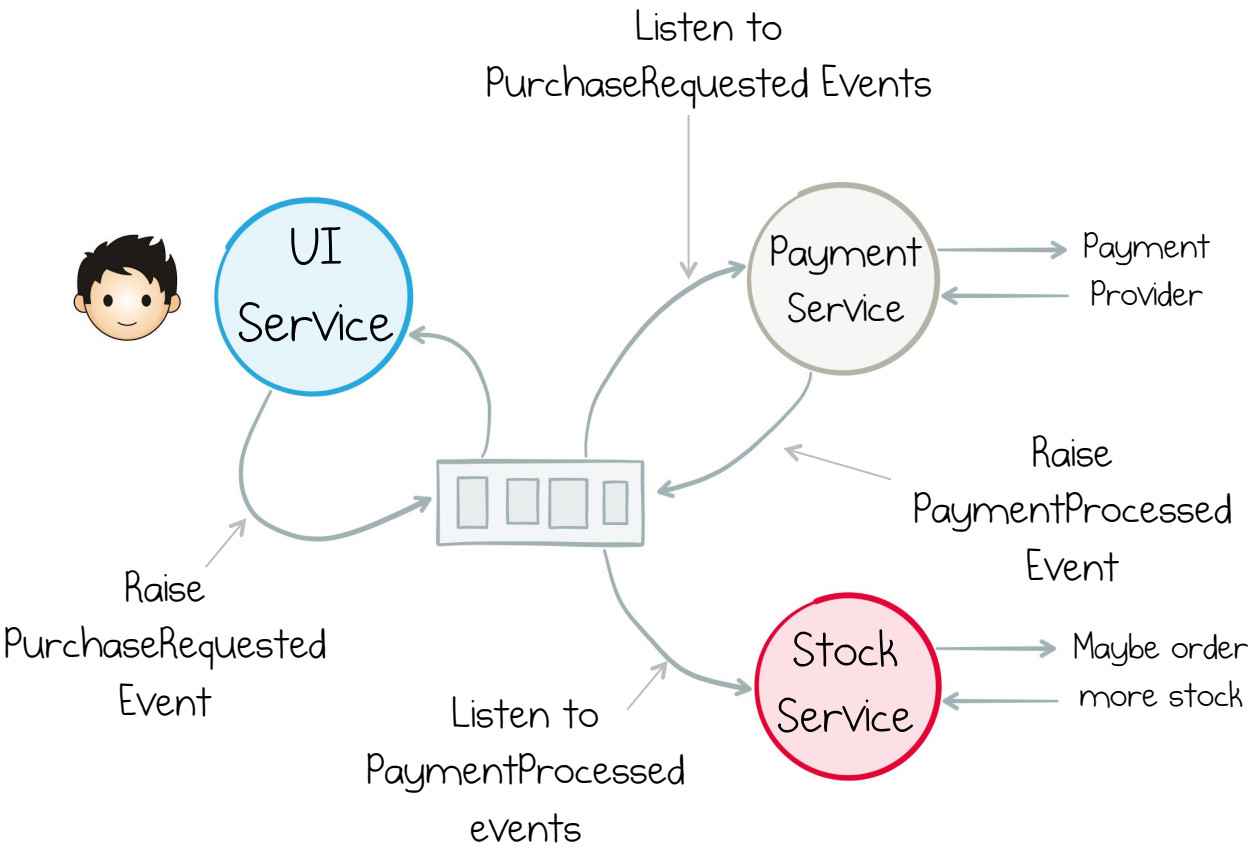


When Complete -> PaymentProcessed

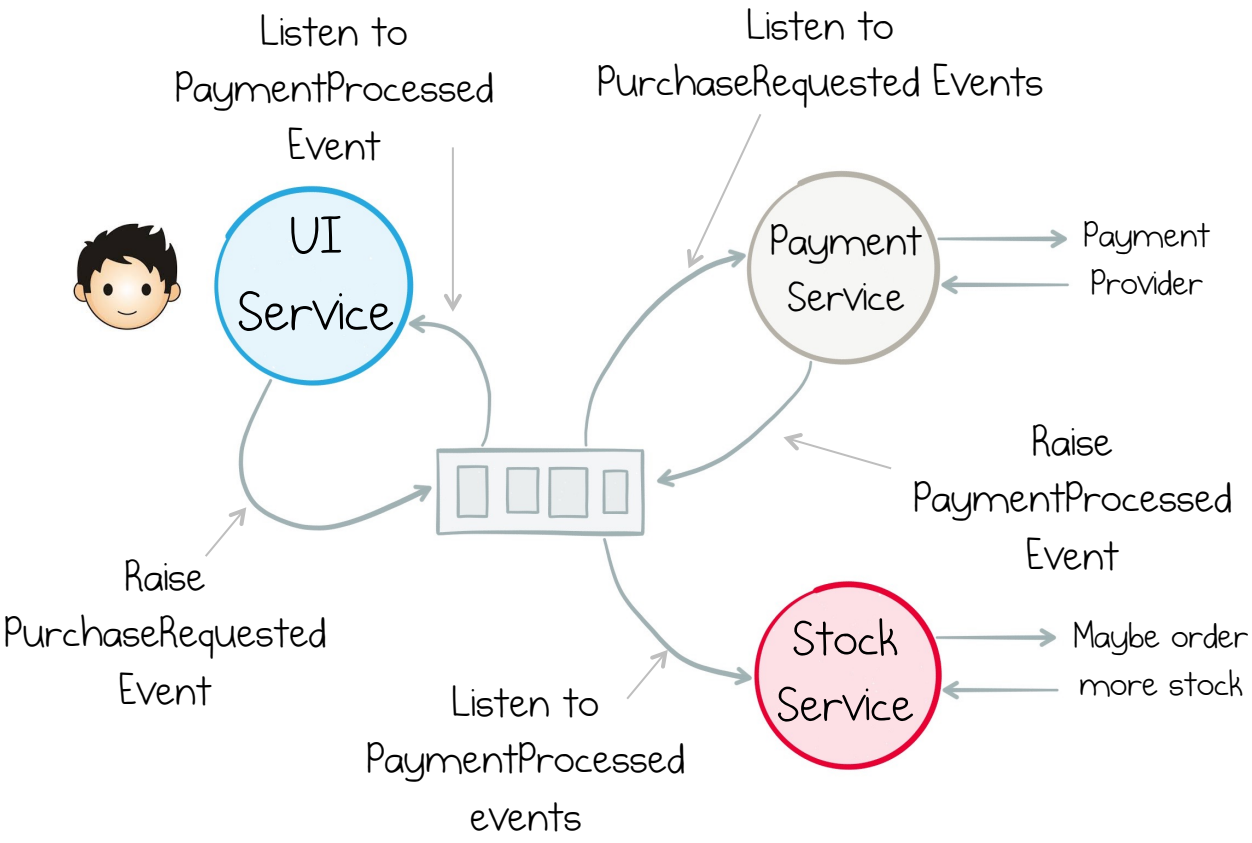


*Some might use the Command/Query Pattern

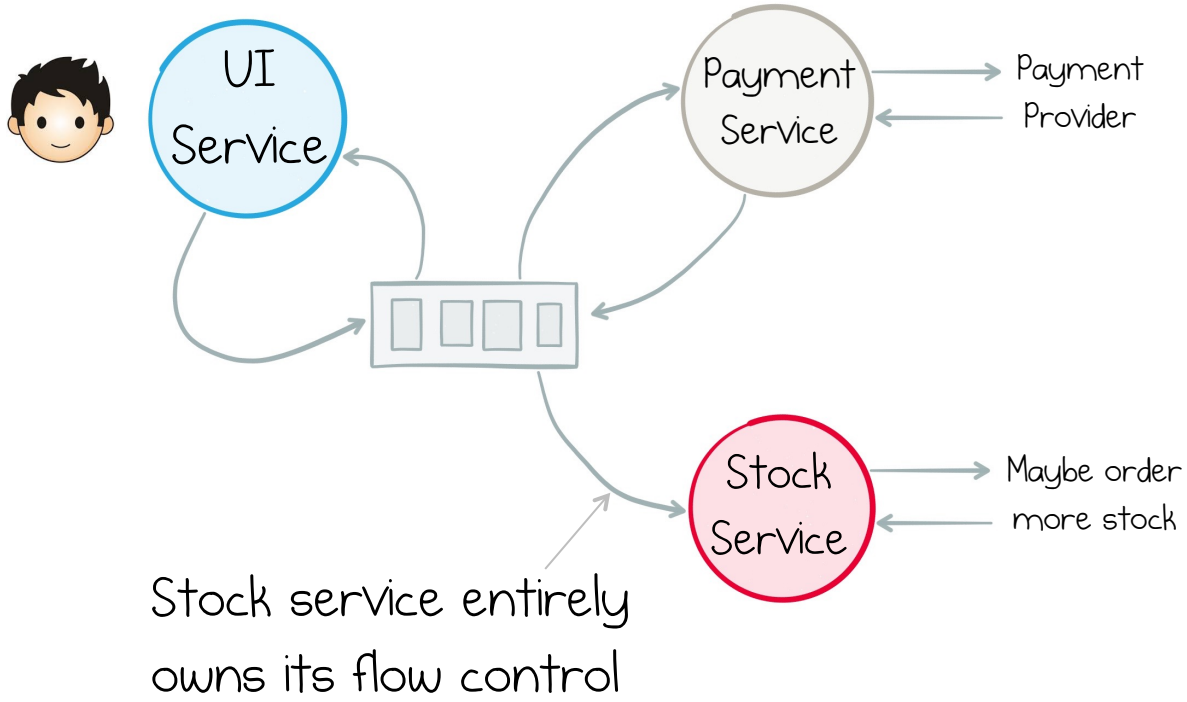
Stock service hooks in too!



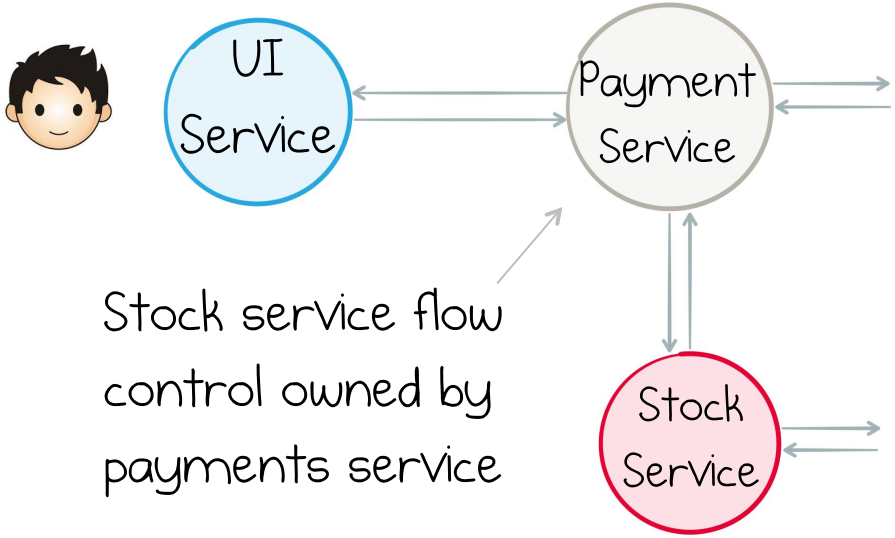
Payment service listens



Flow Control is Receiver Driven

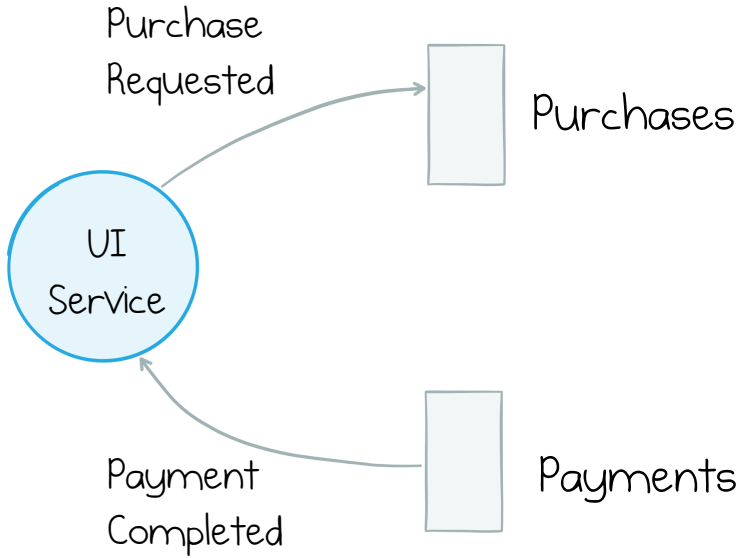


Quite different to Synchronous Model



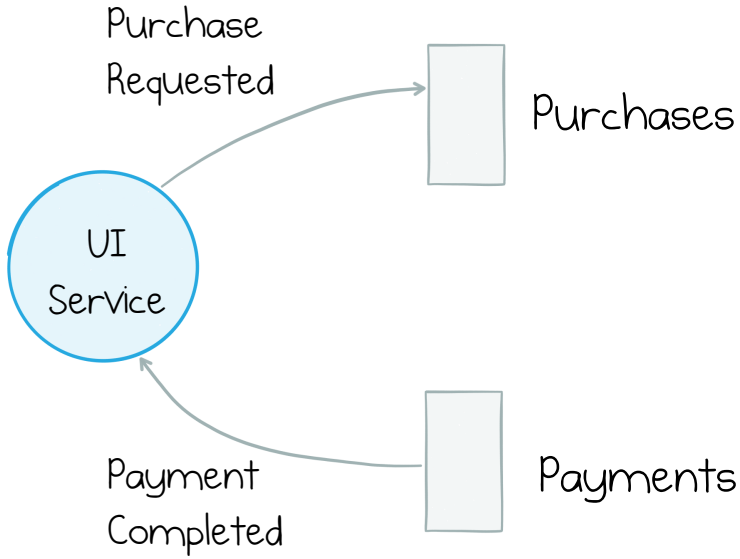
1. Events are Immutable Facts

(state changes taken from the real world and journalled)



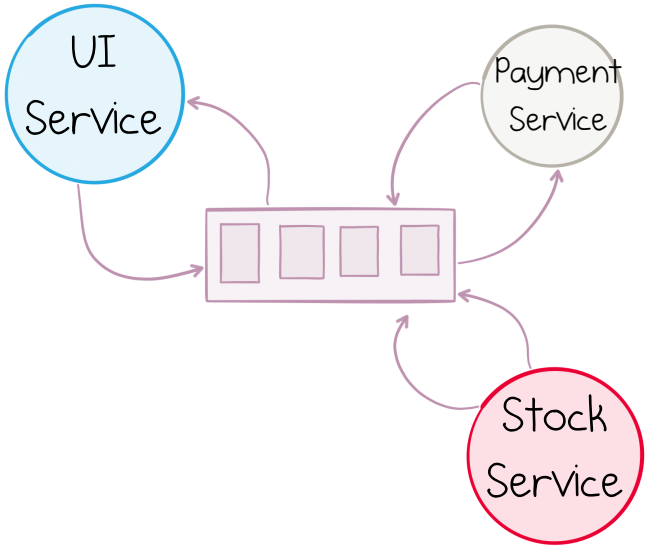
All the benefits of "Event Sourcing"

2. Services couple only to data flows

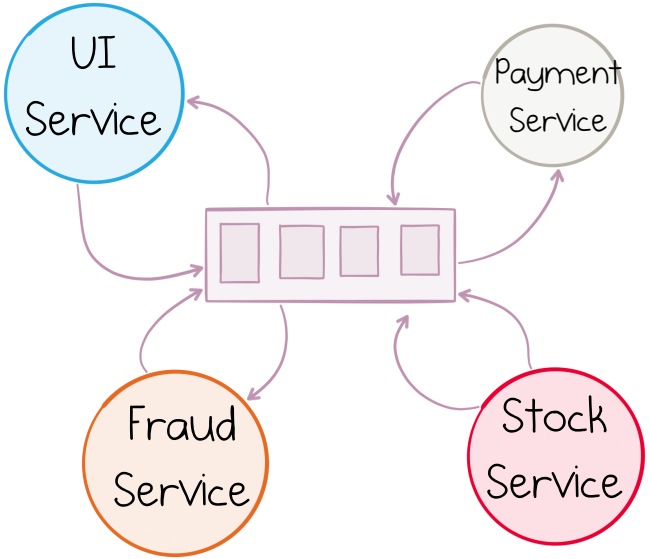


No knowledge of contributing services (RDFC)

Because coupling is only to Events, not services



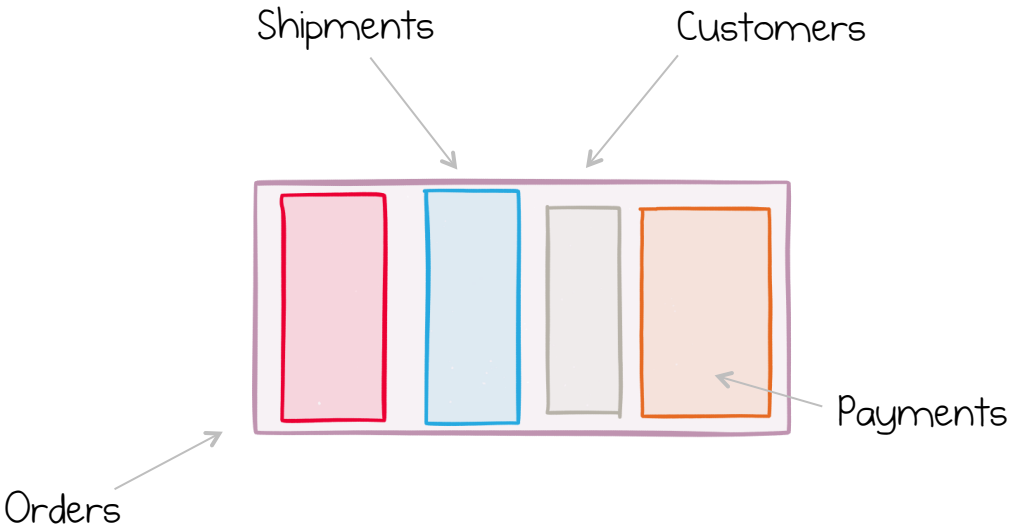
The architecture is "Pluggable"



Communication & State Concepts Merge

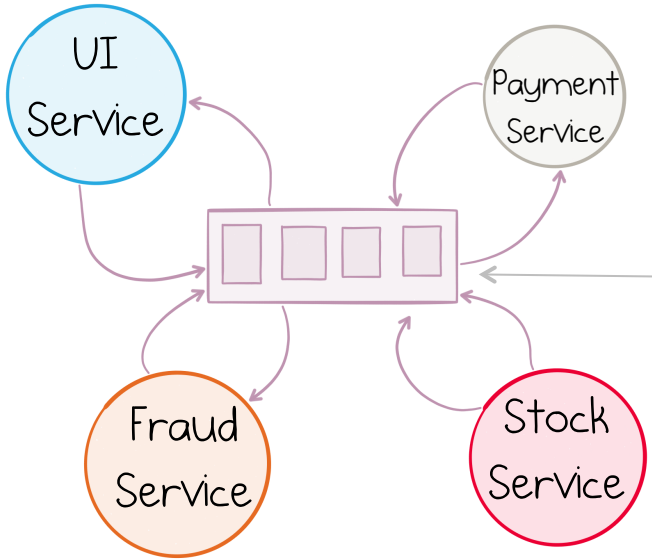
- The concept of Data & Events become one.
- Kafka is both Event Store and Communication Channel

The Events form a Canonical, Shared Narrative



Evolving state of the system over time

Scaling is a concern of the broker, not "upstream" services



Kafka

- Linearly scalable
- Fault Tolerant
- Multi Tennant

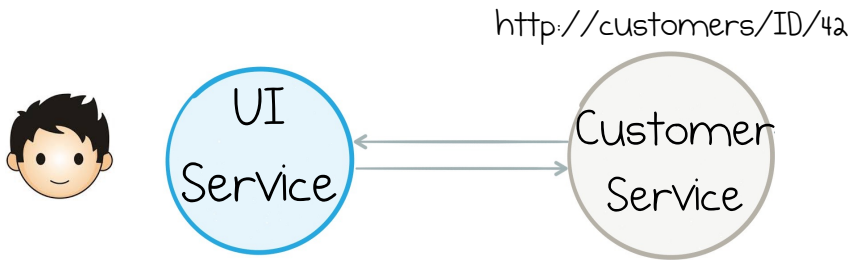
But there is something missing!

How do we look things up?

(i.e. the Q in CQRS)

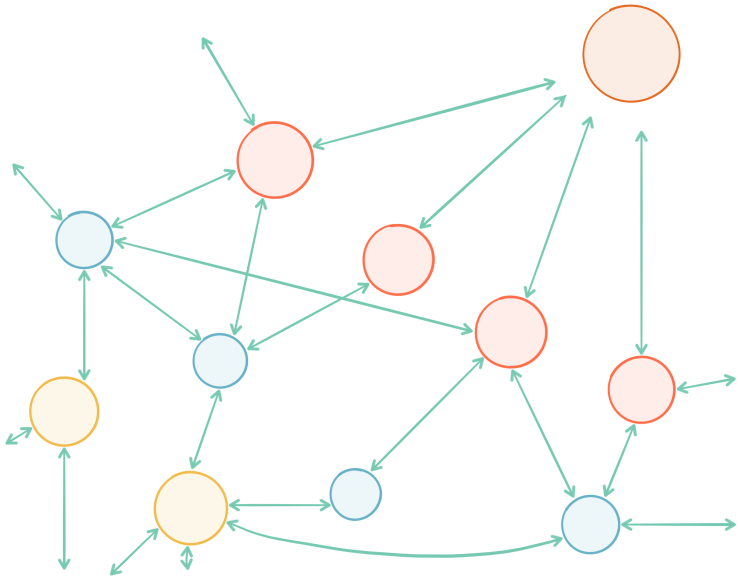
- What is the address for this customer?
- Is this user allowed to view this stock?
- What is the contents of this user's shopping basket?

With REST Lookups are natural, if Remote



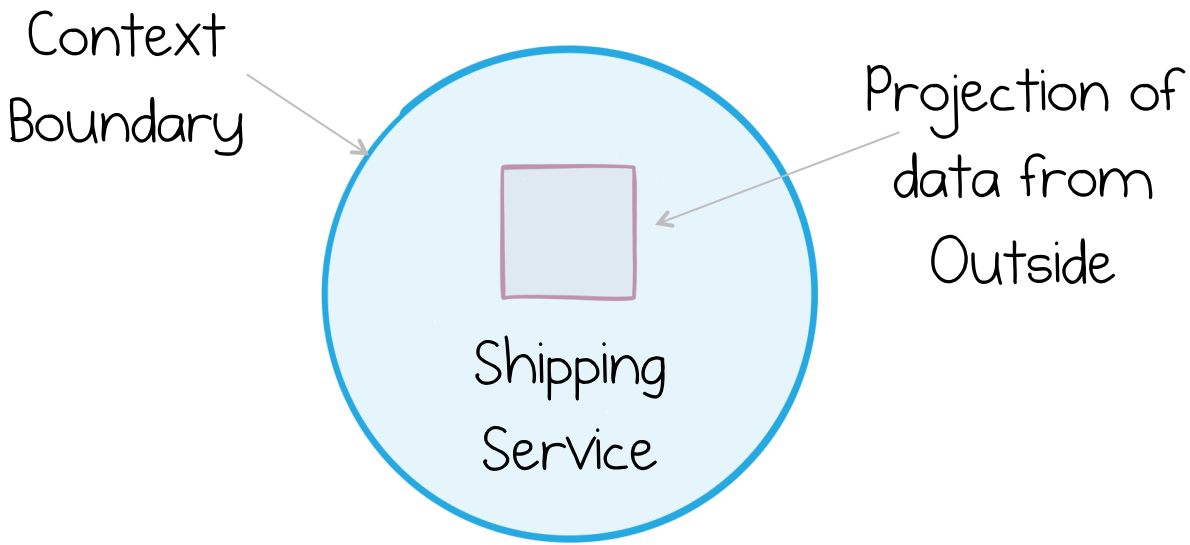
Go to the relevant service and ask!

Which creates the dependency problem

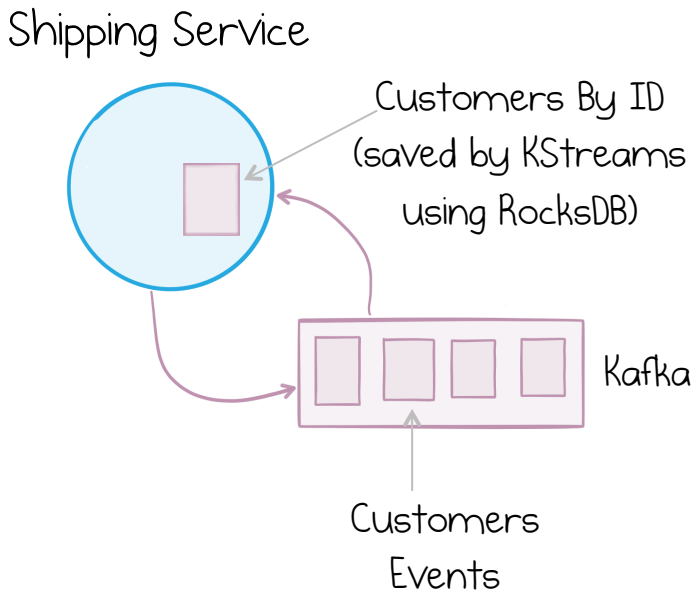


Many services tightly coupled to one another

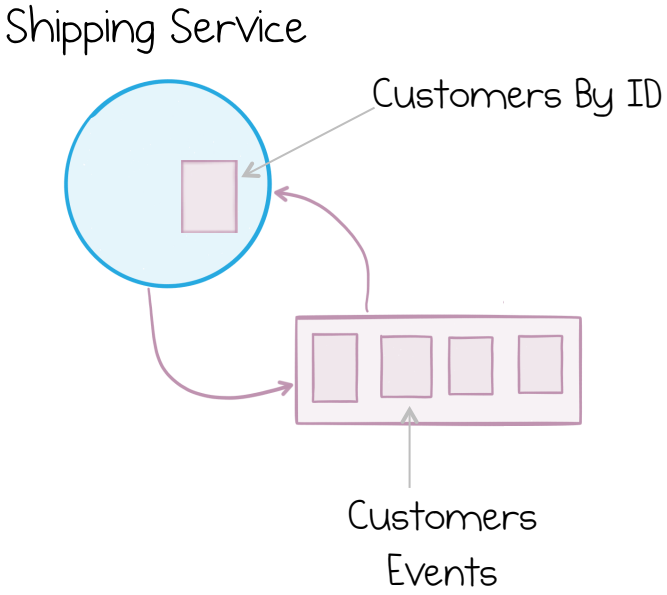
With Event Driven we create "views" or "projections" inside each bounded context



Pattern 1: Local KTables

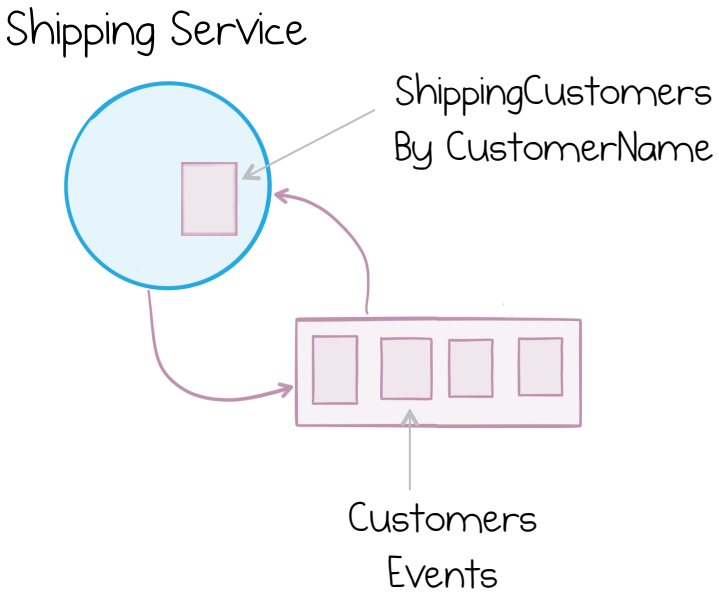


Pattern 1: Local KTables

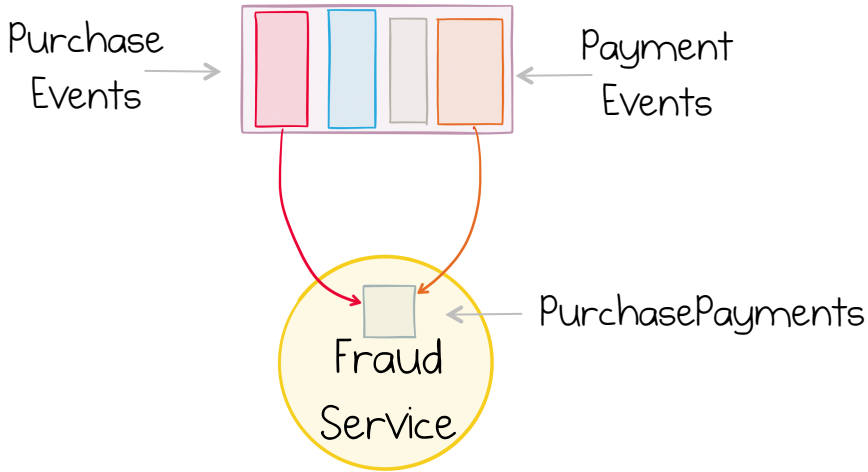



```
KTable customers =
builder.table(
    CustomerId,
    Customer,
    “customers-topic”,
    “customer-store”);

customers = streams.store(
    "customers-store”…);

customer = customers.get(42)
```

As we're using a streaming engine, we can translate into any Domain Model (projection) we wish



Combine different streams from different services

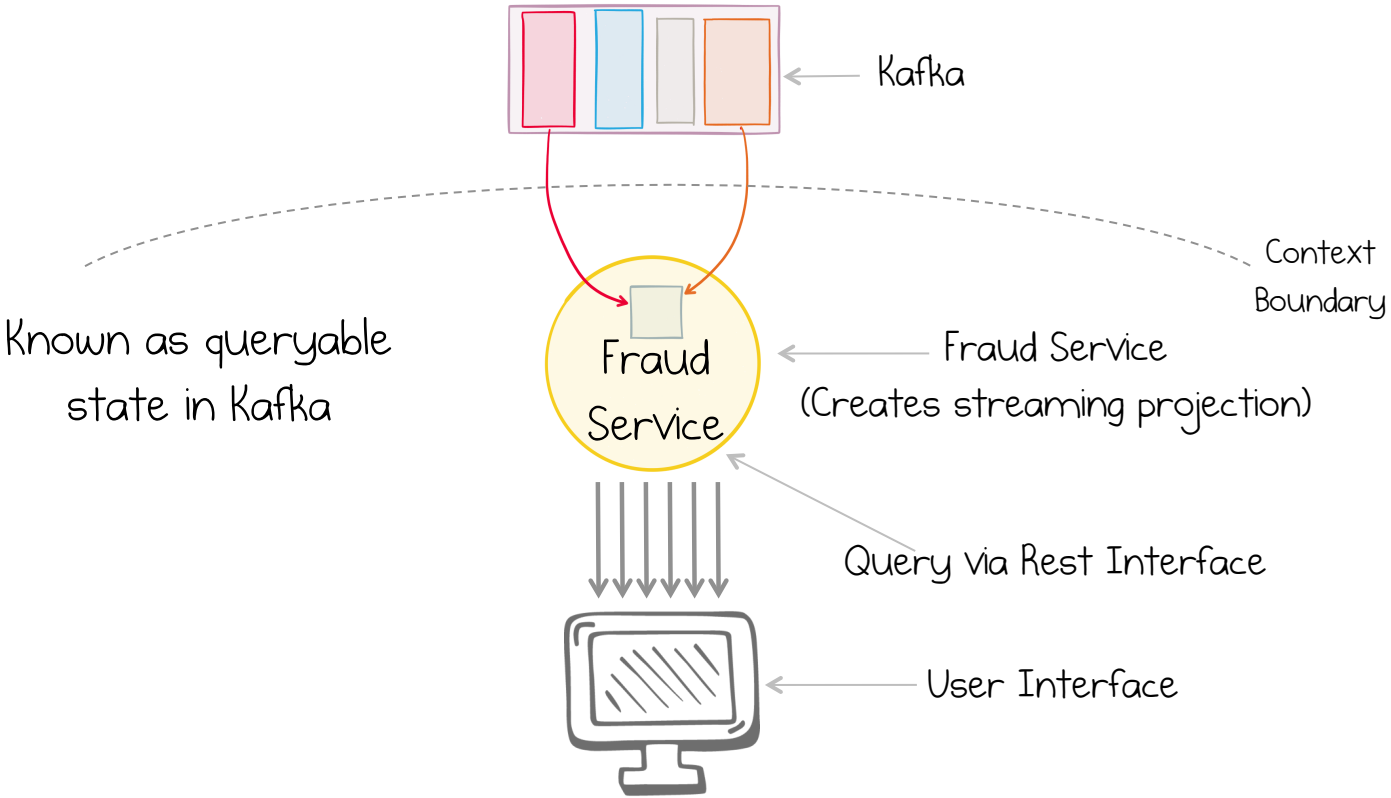



```
payments.join("purchases",..)
.groupByKey()
.reduce(newValueReducer, "PurcahsePaymentsStore")
```

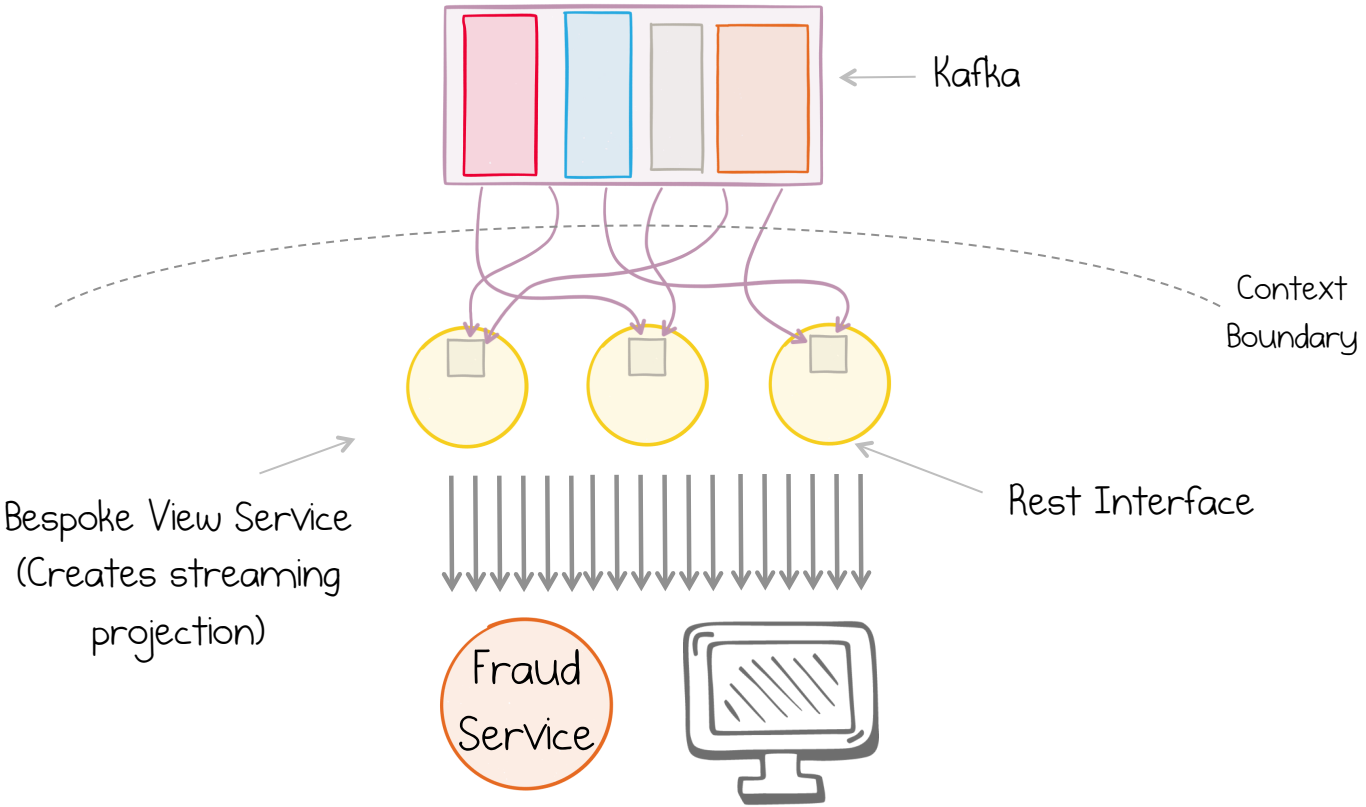
Pattern 1: Local KTables

- Simple to create and query
- Local to each service so fast
- Powerful DSL for transformation
- Inbuilt high availability
- No external database
- Controlled entirely within service's bounded context
- KTable and GlobalKTable allow scale out

Pattern 2(a): Queryable Interface



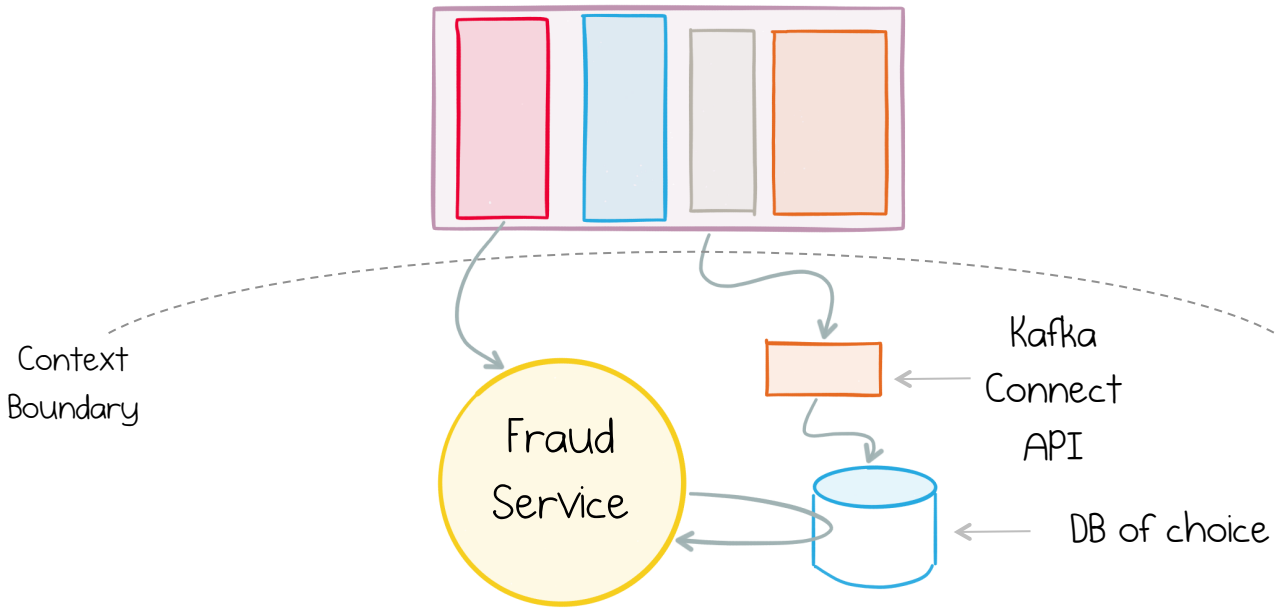
Pattern 2(b): Query Service



Pattern 2: Exposing Materialised Projections

- Similar to Pattern 1 but with a "Query Interface" (using Kafka's Queryable State feature)
- Commonly used with UI's

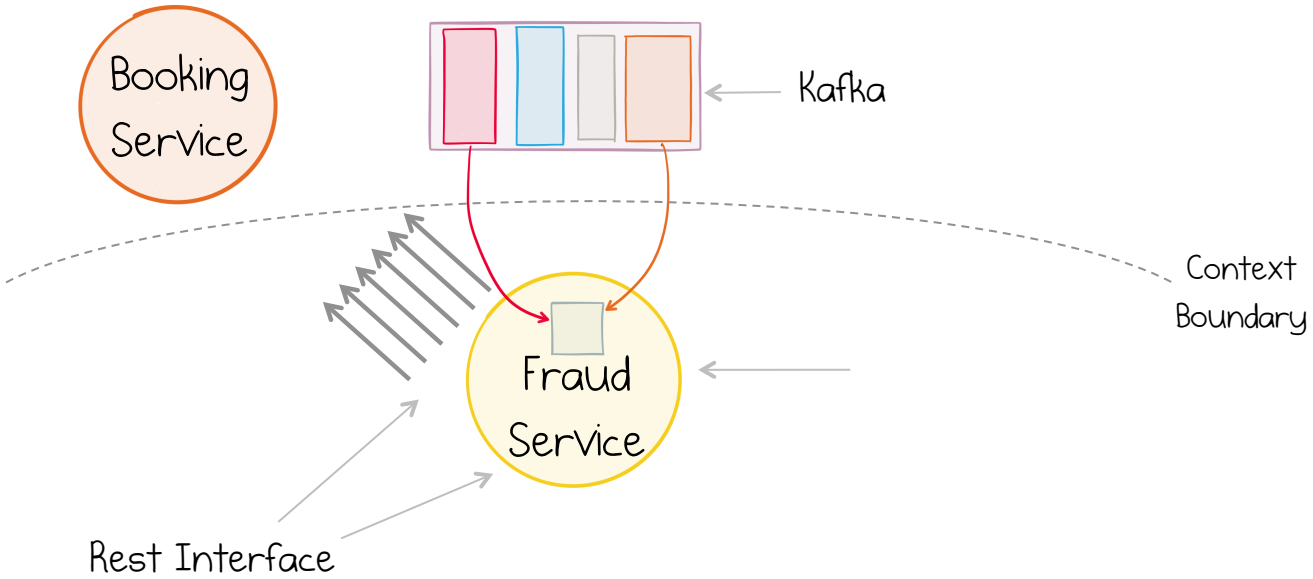
Pattern 3: External DB



Pattern 3: External DB

- Again similar, but heavier weight
- Use when you cannot pre-compute the view (i.e. to do ad hoc queries)
- Trick: start with only the data (fields) you need today. You can always go back for more.

Pattern 4: Hybrid



NB this breaks the async model, but can be a useful compromise in some cases

Pattern 4: Hybrid

- Quite common in practice
- Allows you to break out of the async world
- Use sparingly, but **do use** when appropriate (e.g. login service)

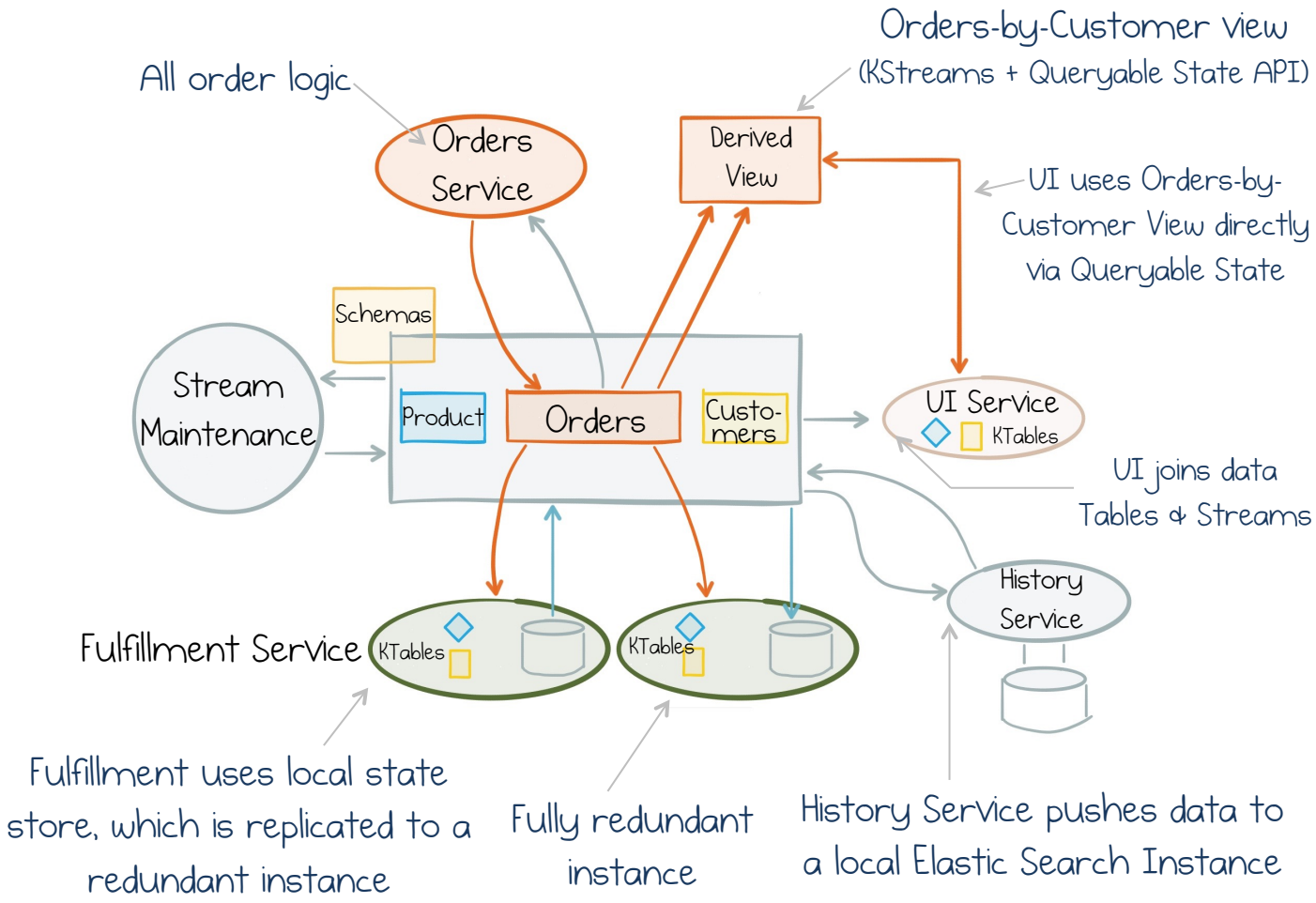
Benefits

- Forms a central, immutable narrative
- Communication Protocol and Event Store become one
- Better decoupling (RDFC)
- Excellent scalability
- Several approaches for managing queries, lightweight => heavy weight
- Event sourcing & CQRS at its core

INTERACT THROUGH EVENTS, DON'T TALK TO SERVICES

BUT IT'S OK TO BREAK THE RULES!

Pulling it all together


All order logic
Orders Service
Orders-by-Customer view
(KStreams + Queryable State API)
Derived View
UI uses Orders-by-Customer View directly via Queryable State
Schemas
Stream Maintenance
Product
Orders
Custo-mers
UI Service
KTables
UI joins data Tables & Streams
History Service
Fulfillment Service
KTables
KTables
Fulfillment uses local state store, which is replicated to a redundant instance
Fully redundant instance
History Service pushes data to a local Elastic Search Instance

Stay in touch!

Online Talks

cnfl.io/online-talks