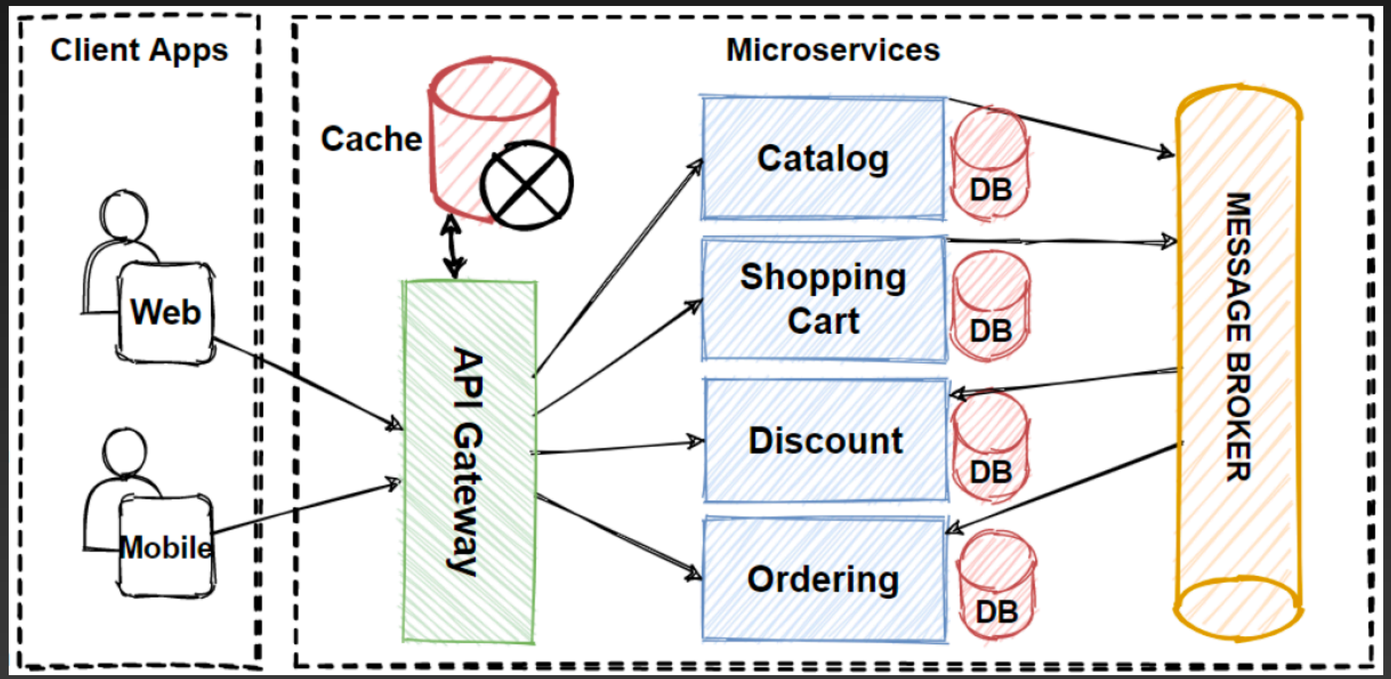
Microservices and Domain-Driven Design

By Nanthakarn Limkool - 6210545505

Microservices

What is microservices?

- Microservices is a suite of small services that independent from each other, but can be used together to create a single application.



Benefits of Microservices

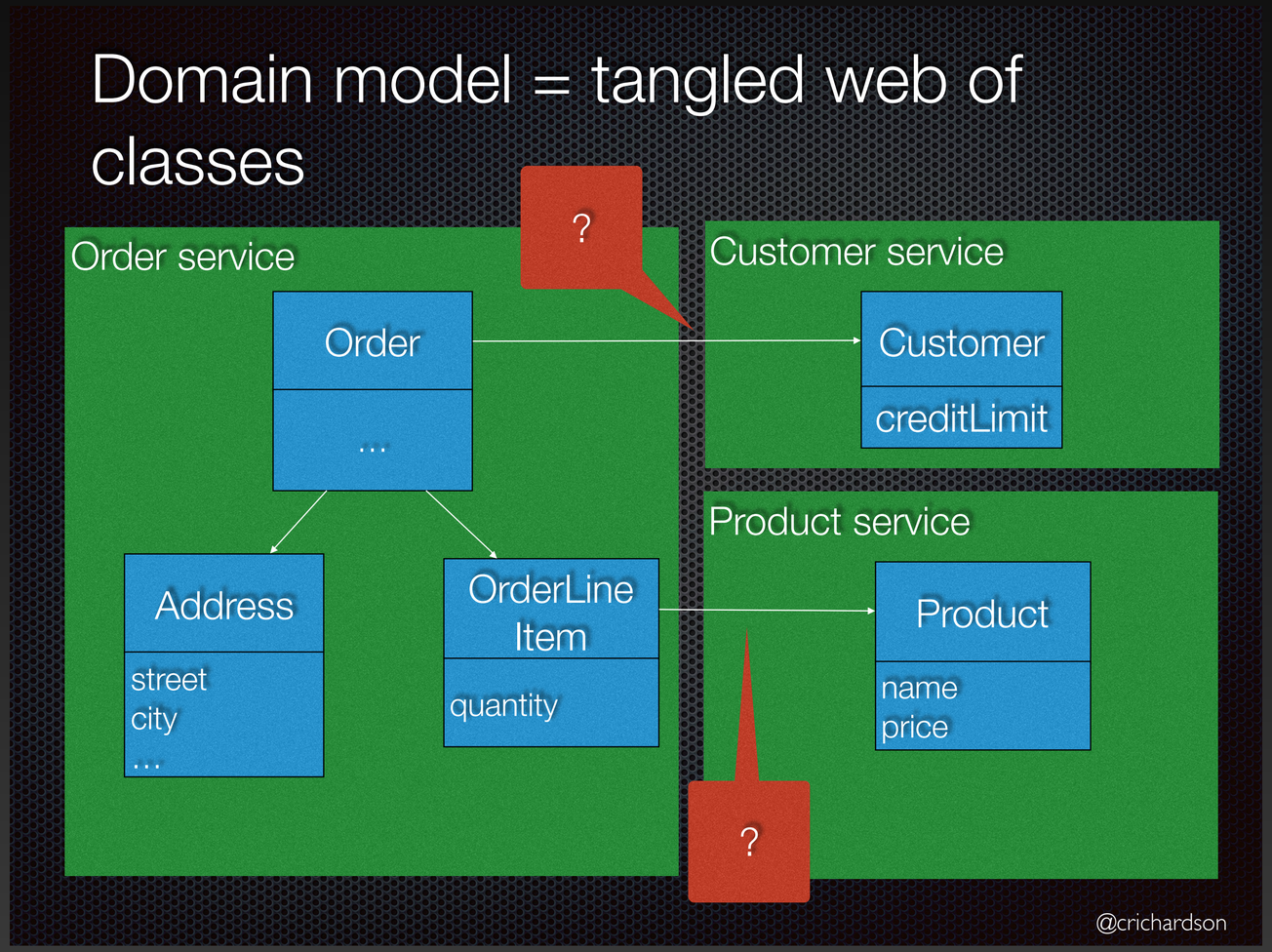
why more and more organizations are moving from monolithic to microservices

- Each developer team can focus on their work
- Highly maintainable and testable
- Loosely coupled
- Independently deployable
- Organized around business capabilities

Challenging Issues
to design and develop microservices

- Tangled web of classes
- ACID transactions violates encapsulation
- ACID transactions requires 2PC, but it is not a possible
- Doesn’t fit with the NoSQL DB transaction model

Challenging Issues
Tangled web of classes



Challenging Issues

ACID transactions violates encapsulation

But it violates encapsulation…

```
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

@crichardson

Challenging Issues

2PC not working, Doesn't fit with the NoSQL DB transaction model

```
BEGIN TRANSACTION
…
SELECT ORDER_TOTAL
 FROM ORDERS WHERE CUSTOMER_ID = ?
…
SELECT CREDIT_LIMIT
FROM CUSTOMERS WHERE CUSTOMER_ID = ?
…
INSERT INTO ORDERS …
…
COMMIT TRANSACTION
```


- Guarantees consistency

BUT

- 2PC is best avoided
- Not supported by many NoSQL databases etc.
- CAP theorem ⇒ 2PC impacts availability
- ….

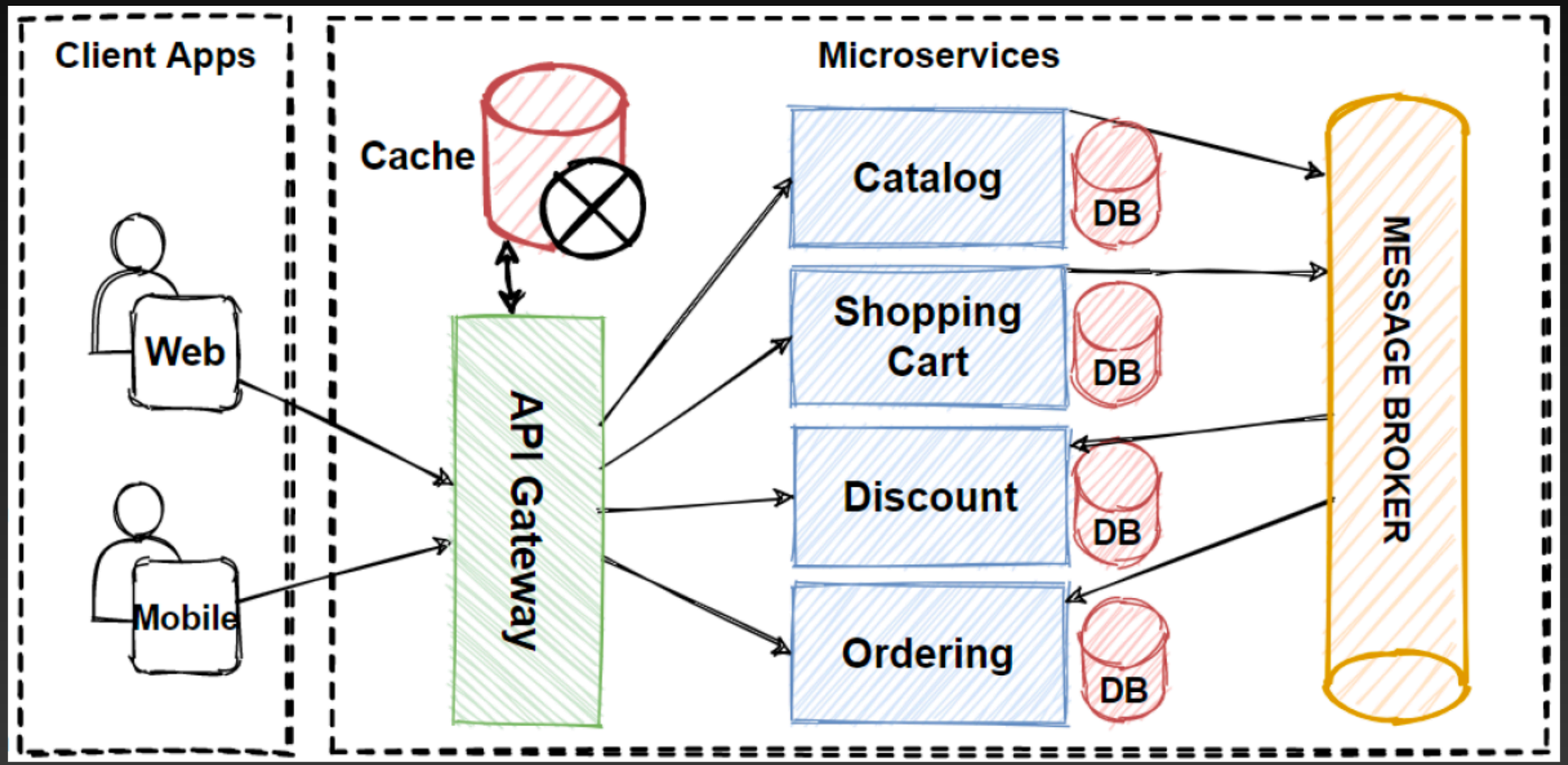
Microservice Architecture

Structure use to manage microservices

- an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.
- these services are built around business capabilities and independently deployable by fully automated deployment machinery.
- there is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

Microservice Architecture

Structure use to manage microservices



API Gateway

How API Gateway can act as the Facade design pattern?

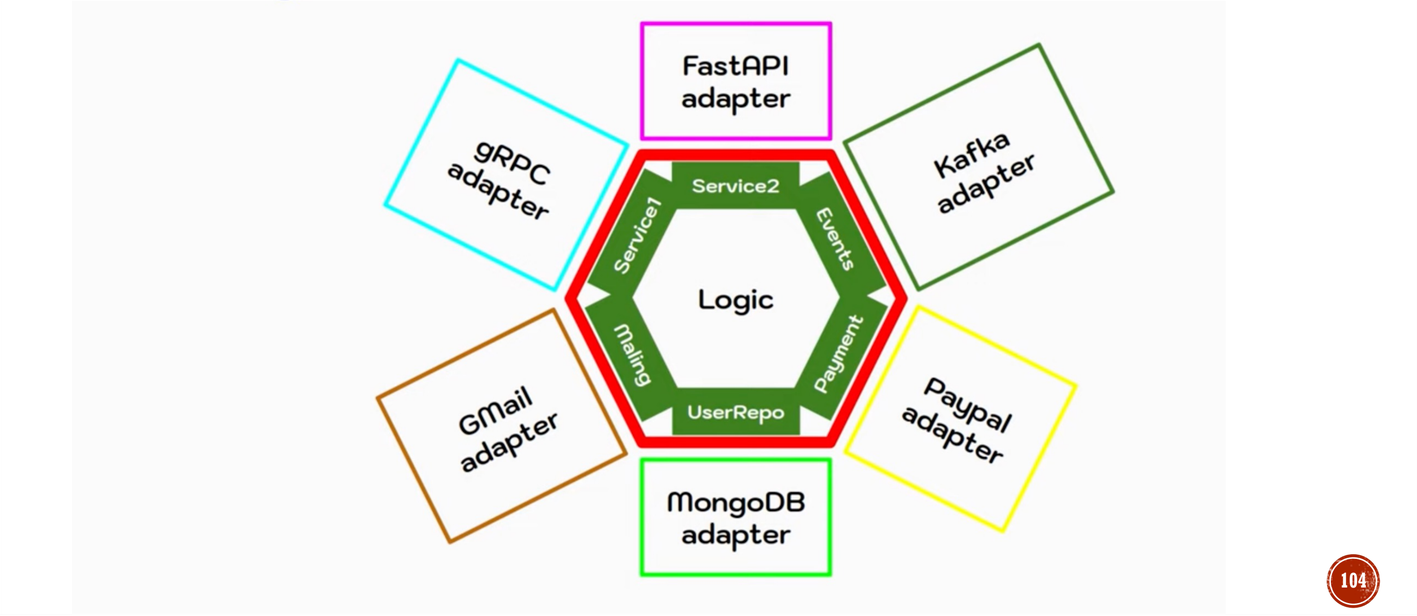
- API Gateway is a centralized management of service
  - Hexagonal Archiecture

- Facade design pattern (commonly used in OOP)
  - Create an interface, and implemented classes
  - Import those classed to use in another class (facade class)

Hexagonal Archiecture

Clean Architecture for API Gateway

- Hexagonal Archiecture is an architecture of programming that solves sensitivity problems when multiple services are used as the same file

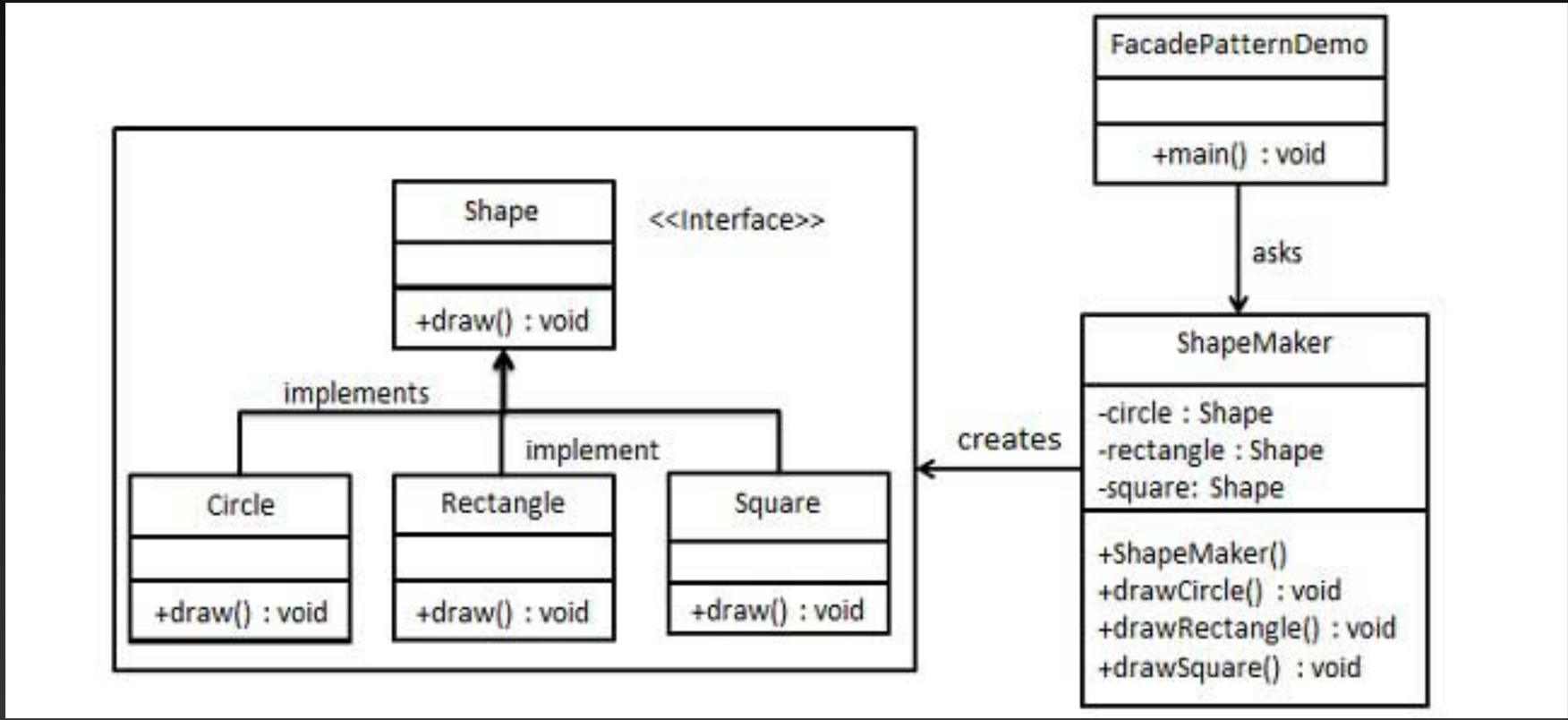


Hexagonal Archiecture
Clean Architecture for API Gateway

- To describe how it works. For example,
  - if your application uses database mongoDB for a long time
  - then you want to change it to dynamoDB, normally you will need to find where mongoDB related methods are calling and replace it with dynamoDB methods.
  - But in Hexagonal Architecture, you created an abstract for NoSQL at first
  - then create adapter (in this case ‘MongodbAdapter’) class based on the that abstract, so when you using it in any file and want to change it to another adapter class (‘DynamodbAdapter’) you can just update your import section, no need to update all your code line-by-line.

Facade design pattern

Example



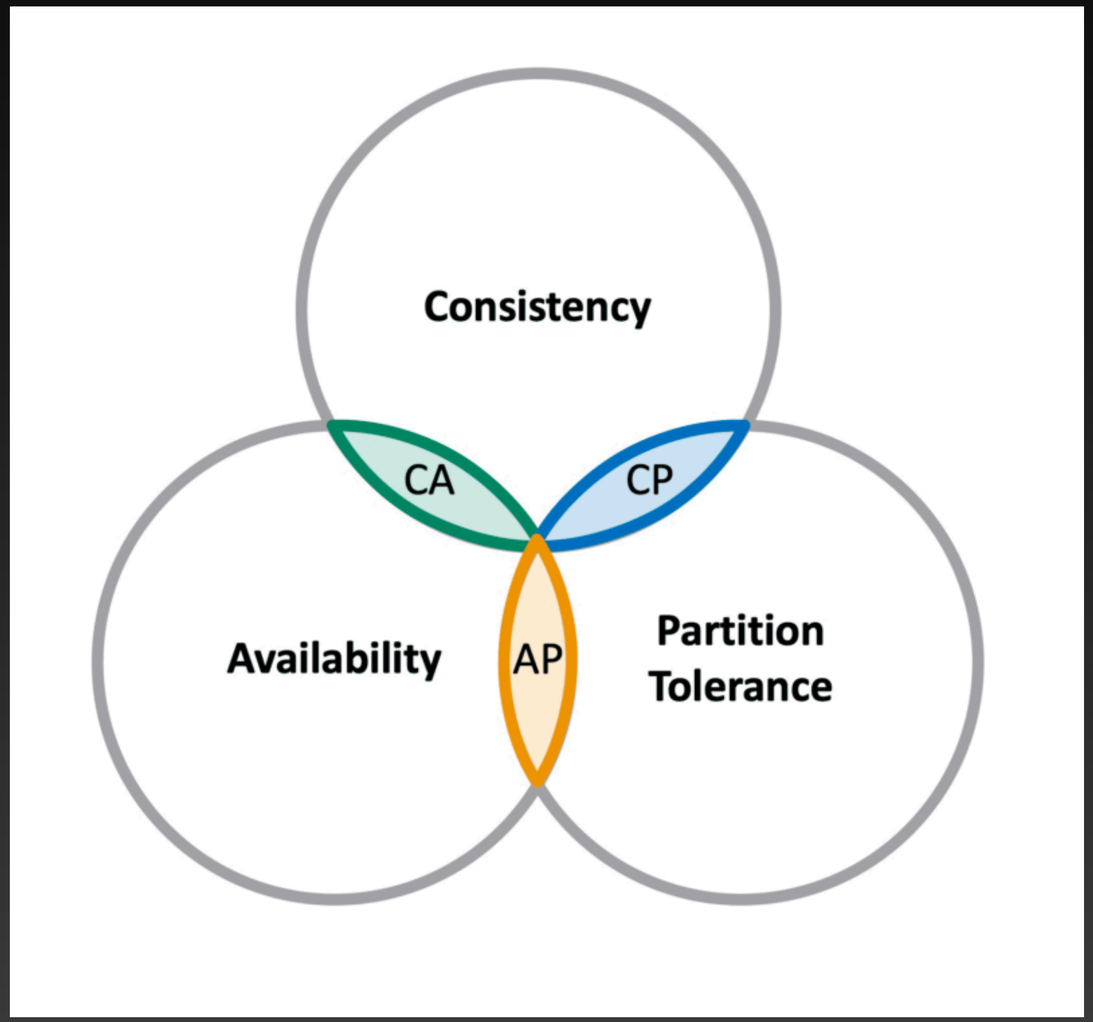
ACID Transactions

Properties of a Transaction

- Atomicity: each statement in a transaction is treated as a single unit
- Consistency: transactions only make changes to tables in predefined, predictable ways
- Isolation: transactions don’t interfere with each other when having multiple users
- Durability: transactions that executed will be saved

CAP Theorem

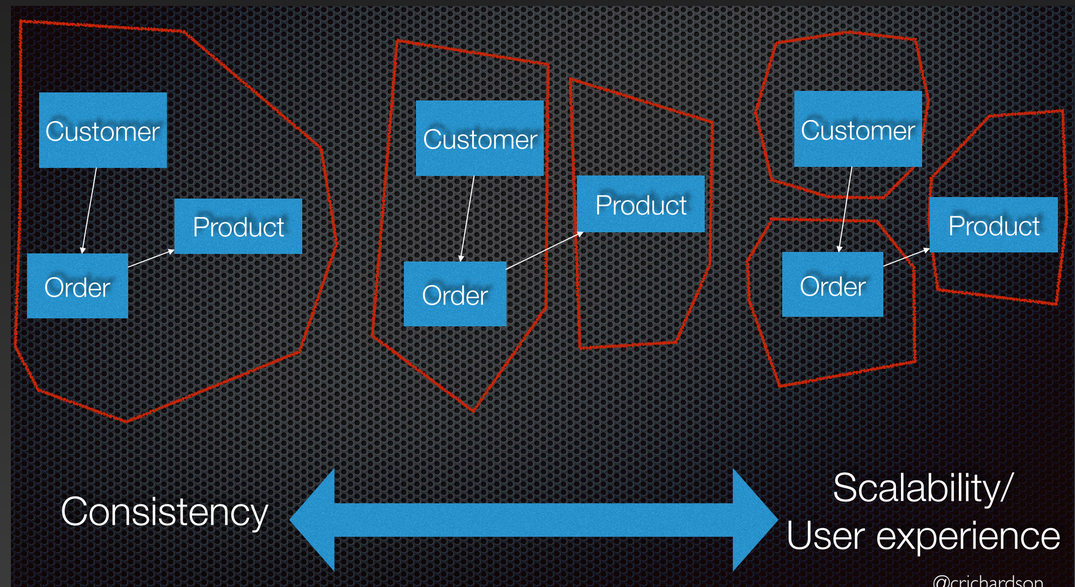
Trade off among Availability, Consistency, Partitioning



- Consistency: every read receives the most recent write or an error
- Availability: every request receives a (non-error) response, without the guarantee that it contains the most recent write.
- Partition tolerance: the system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes.

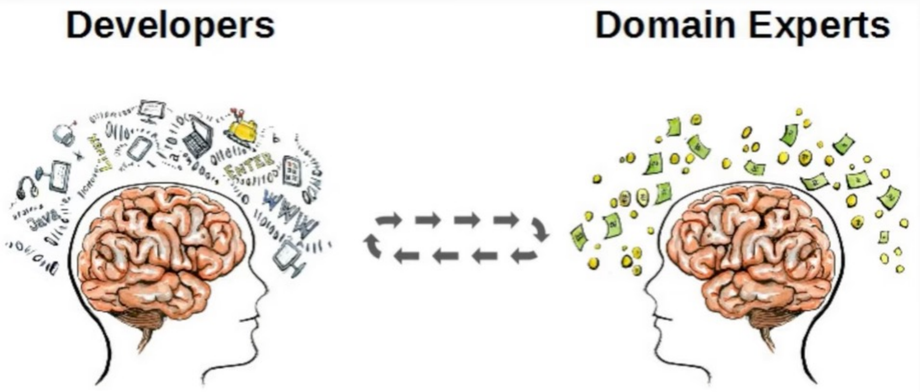
Domain-Driven Design
Business Logic Design of microservices

- DDD: Domain-Driven Design
- DDD is a design of conceptual grouping microservices (Aggregate) based on business logic
- (Aggregate Granularity) To balancing between Consistency and Scalability/User experience

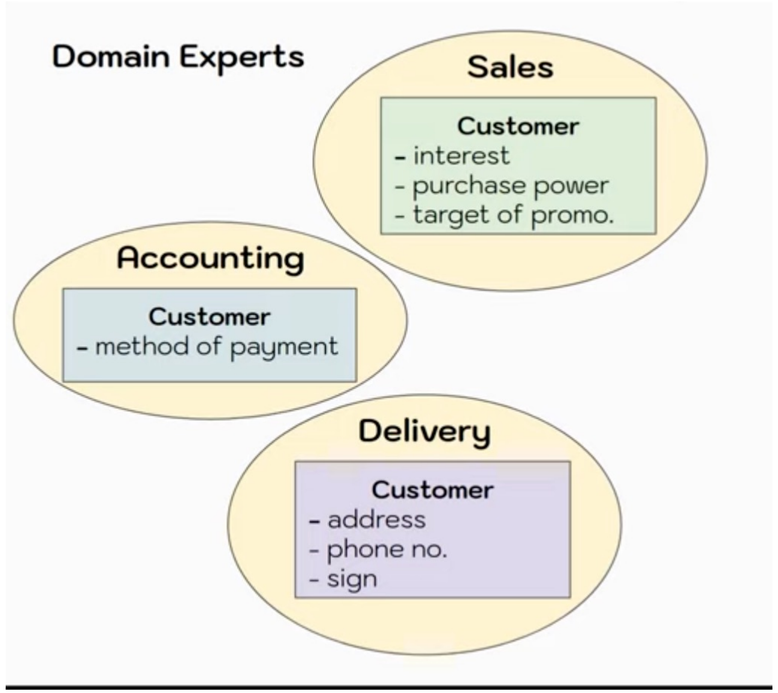


Domain-Driven Design
Business Logic Design of microservices

- To Solve Communication gap between developers and domain experts

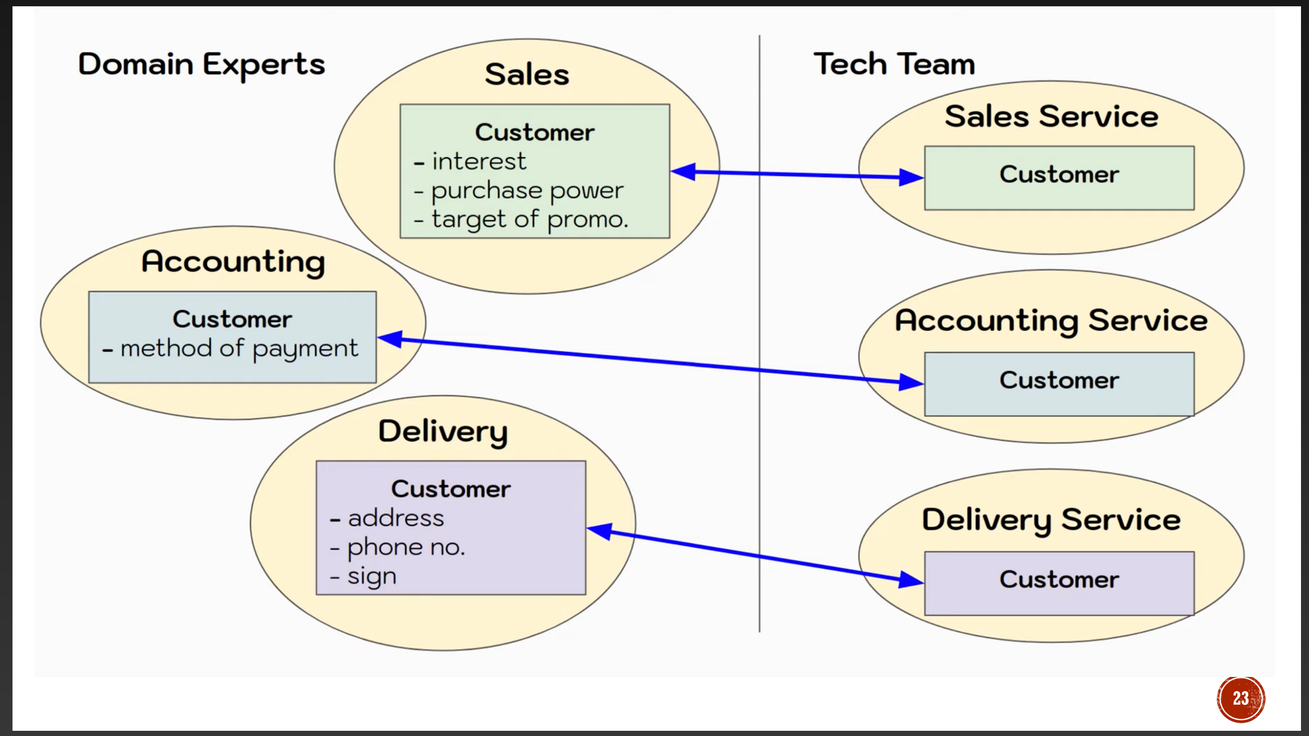


ภาพจากหนังสือ Domain Driven Design Distilled - Vaughn Vernon



Domain-Driven Design
Business Logic Design of microservices

- To Solve Communication gap between developers and domain experts

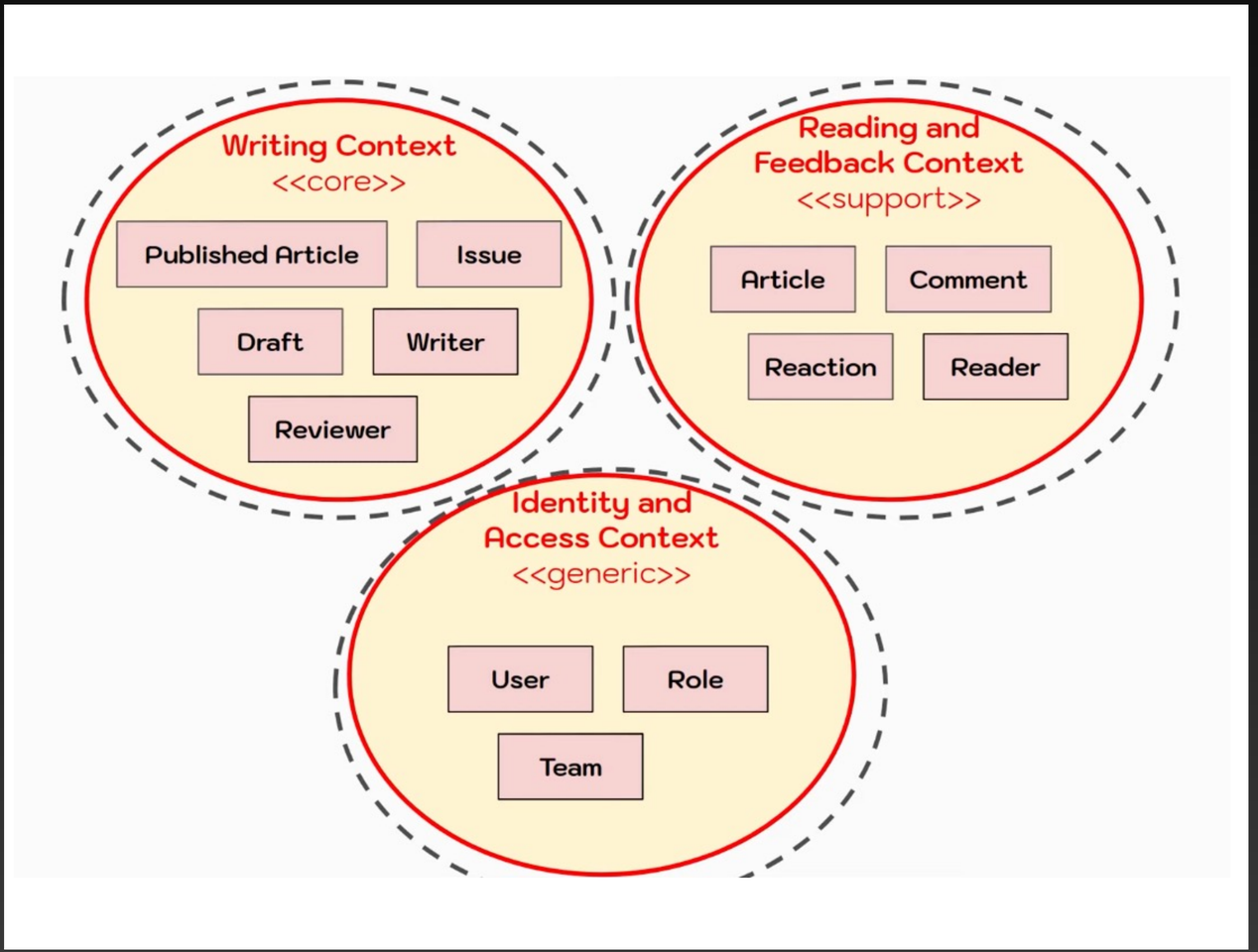


There is no correct Design, it depend on propose of the application.

Domain-Driven Design

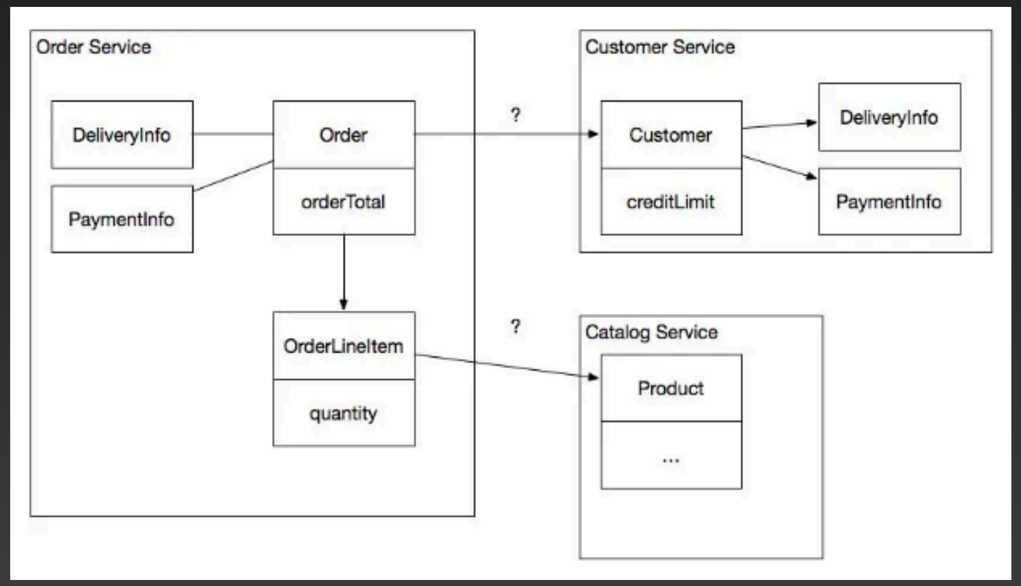
Solve Communication gap between developers and domain experts

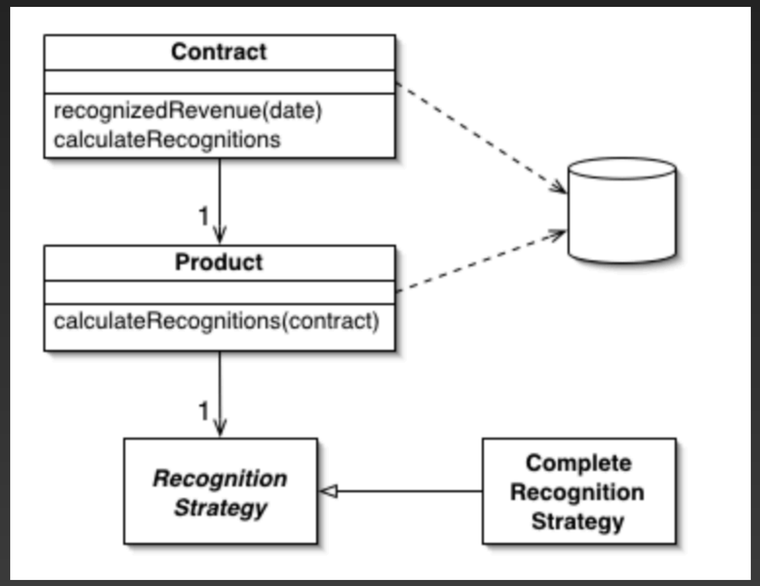
- Domain Model Pattern
- Subdomain
- Bounded Context
- Ubiquitous Language



Domain Model Pattern

- Domain Model is a conceptual model of the domain that incorporates both behavior and data
- Domain Model Pattern is a pattern used to implement complex business logic in DDD to Custer, Graph, UML, more understandable diagram.

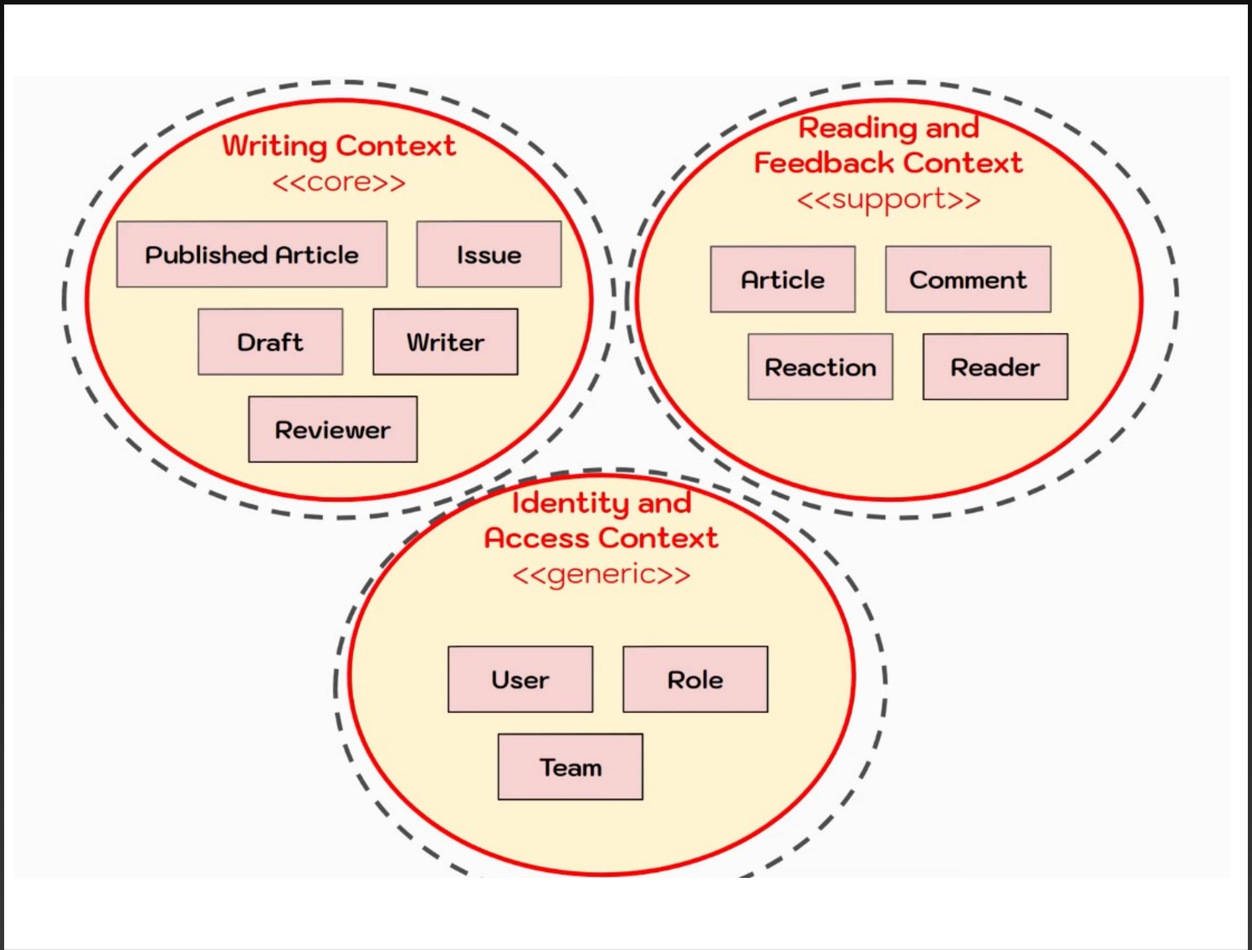




Subdomain

- Subdomain is a relative term used to describe a domain that is a child of another higher-level identified domain.
- In DDD, has 3 subdomains:
  - Core: advantage of the system, unique from other product with the same kind
  - Support: need to have for system to be runnable
  - Generic: common feature, should buy/rent as api instead of write a new one

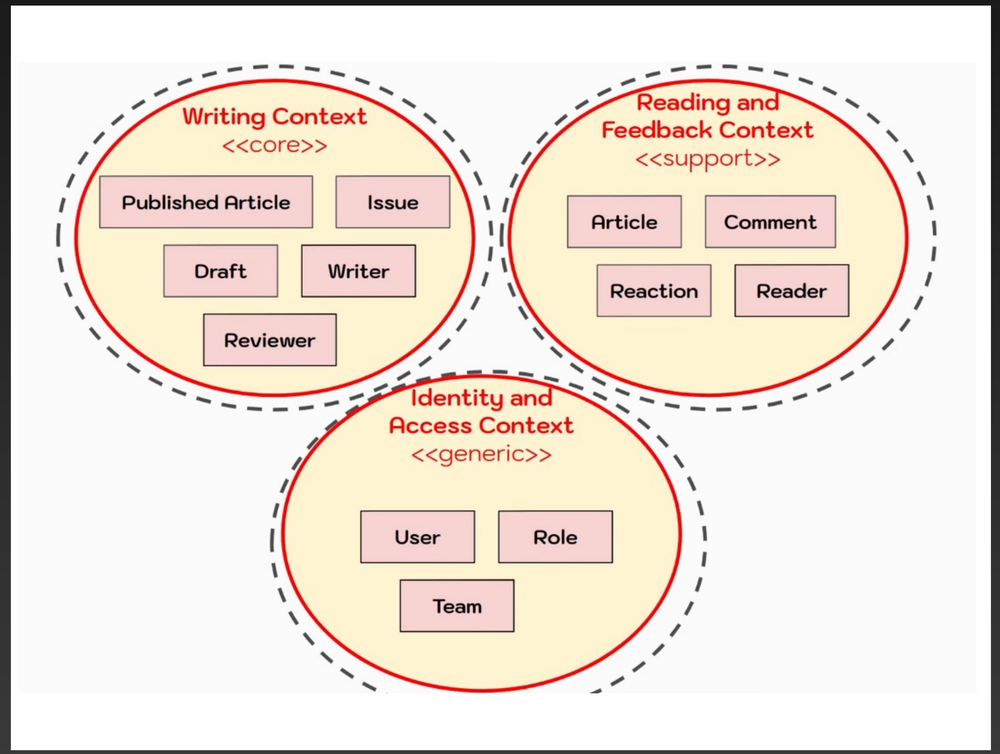
Subdomain
Example



Bounded Context

- Bounded Context is the boundary of a model that represents tose concepts, their relationships, and their rules in the same Subdomain

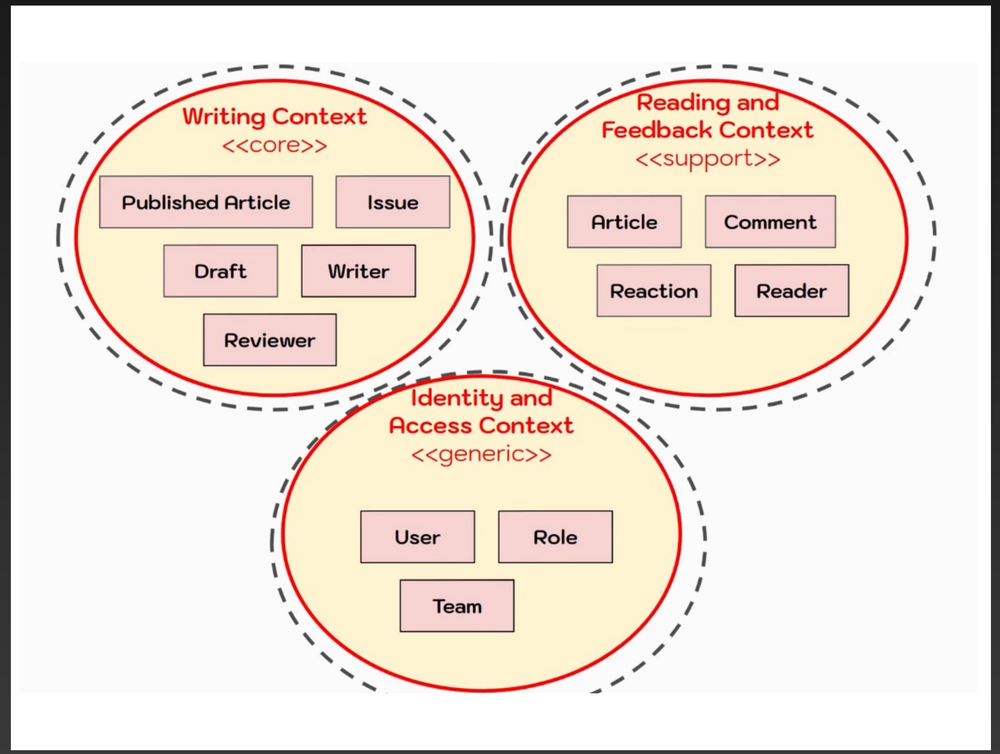
- Writing Context
- Reading and Feedback Context
- Identity and Access Context



Ubiquitous Language

- Ubiquitous Language is the practice of building up a common, rigorous language between developers and users.

- Published Article
- Draft
- Issue
- ... etc.



Domain-Driven Design

building block: Entities vs Value Objects

Entities	Value Objects
Identity as interface	No identity
identifier equality	structural equality
Mutability	Immutability
Lifetime longer	Lifetime shortening
should have separate tables in database	should not have separate tables in database

Domain-Driven Design
building block: Aggregates

- Aggregates in DDD are:
  - the building blocks of microservices
  - cluster of objects that can be treated as a unit
  - graph consisting of a root entity and one or more other entities and value objects
- Typically business entities are Aggregates (e.g. customer, Account, Order, Product, ...)

Aggregate

Rules related to Aggregate

- Reference other aggregate roots via identity (primary key)
  - Domain model = collection of loosely connected aggregates

- Transaction = processing one command by one aggregate
  - Transaction scope = service = NoSQL database transaction

Aggregate

How to maintaining consistency with Aggregates?

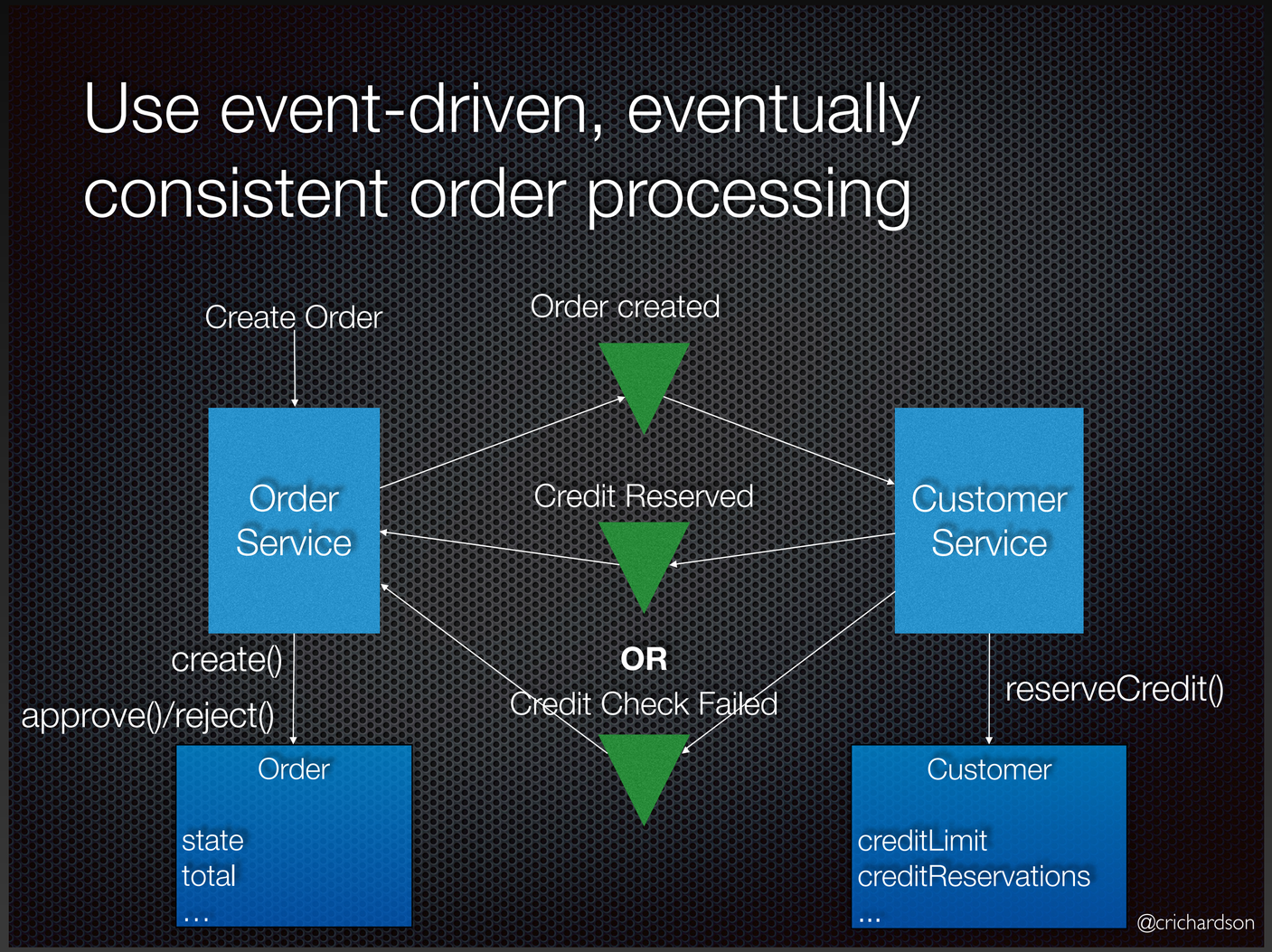
- Use Events
  - Event Driven: communicate, order processing
  - Event Sourcing: monitor, update status

Event-driven Architecture

communication structure between services

- For example,
  - has 2 services (Order and Customer)
  - order belongs to customer, customer has orders
- To connect them together -> use event-driven, event following
  - 1. order create()
  - 2. send `order created` to customer
  - 3. customer reserveCredit()
  - 4. customer send `credit reserved` (or `credit check failed`) to order
  - 5. order approve() or reject()

Event-driven Architecture
Example



Rollback Issue
Warning

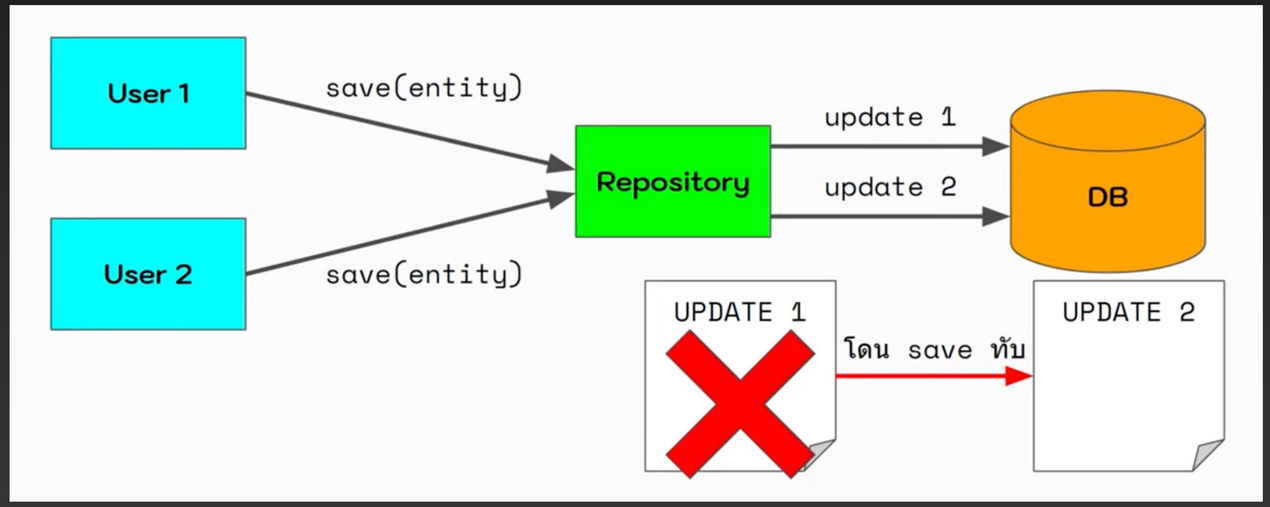
- Since Event-driven is not a Transition, it doesn’t has auto rollback

- Solution: write code to support fail case by yourself

Dual Write Problem
Warning

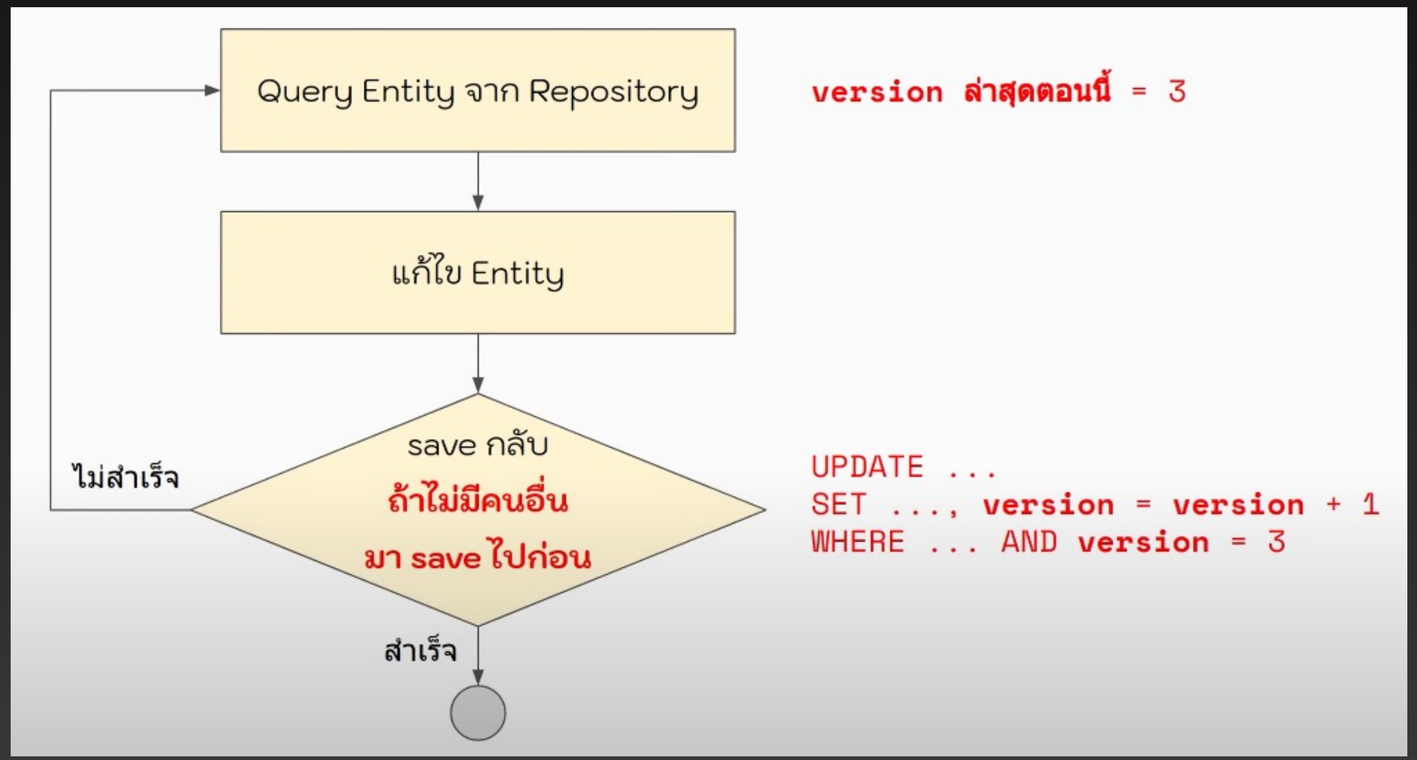
- Since a system may need to interact with many users at the same time, there is a possible that someone can commit changes in database before you did. So you or another users loss their changes (database inconsistency).

- Solution: use Optimistic Locking



Dual Write Problem
Warning

- Optimistic Locking = checking version before commit any change



Event Sourcing

Solves data consistency issues (transaction fail)

Submit Order

```
DomainEvent.publish(OrderSubmittedEvent(...))

order = all_orders.from_id(5555)

order.submit()

all_orders.save(order)  # update ข้อมูลใน db
```

สำเร็จ

ไม่สำเร็จ

Submit Order

```
order = all_orders.from_id(5555)

order.submit()

all_orders.save(order)  # update ข้อมูลใน db

DomainEvent.publish(OrderSubmittedEvent(...))
```

สำเร็จ

ไม่สำเร็จ

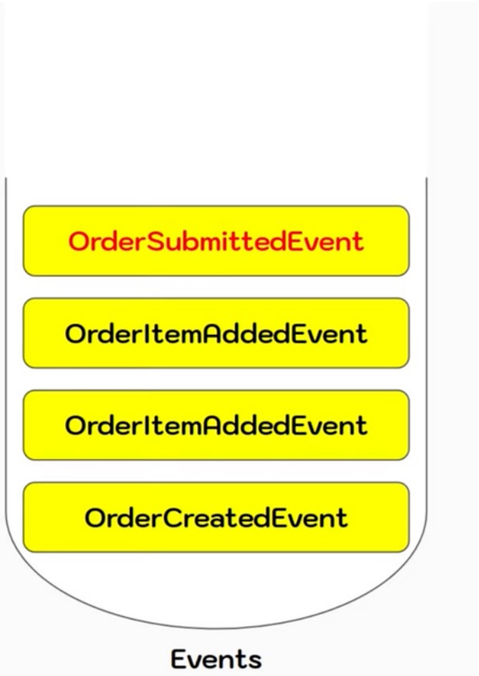
Event Sourcing

Solves data consistency issues (transaction fail)

- **Event Sourcing**

สถานะปัจจุบัน Order

```
{
  id: 1234,
  buyerId: 9999,
  items: [
    {productId: 1111, amount: 3},
    {productId: 2222, amount: 8}
  ],
  status: “submitted”
}
```



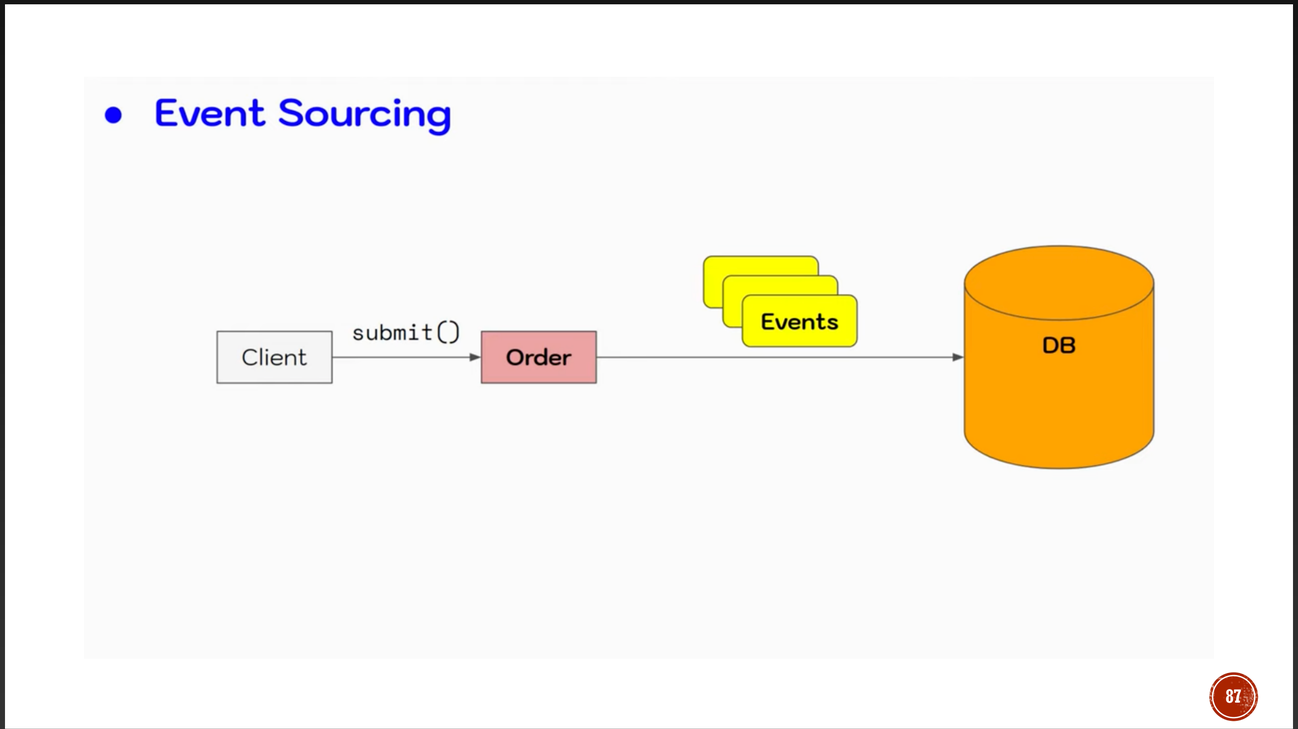
Event Sourcing

Solves data consistency issues (transaction fail)

- Replace 2PC that is not a viable option
  - App will not update database (2PC), but publish events to event store
- event store = database + msg broker
- (≈ event table, event log, msg topic channel, etc.)
- How: ‘The present is a fold over history’
  - add event to event table
  - when want the data, re-create state from the events in event store

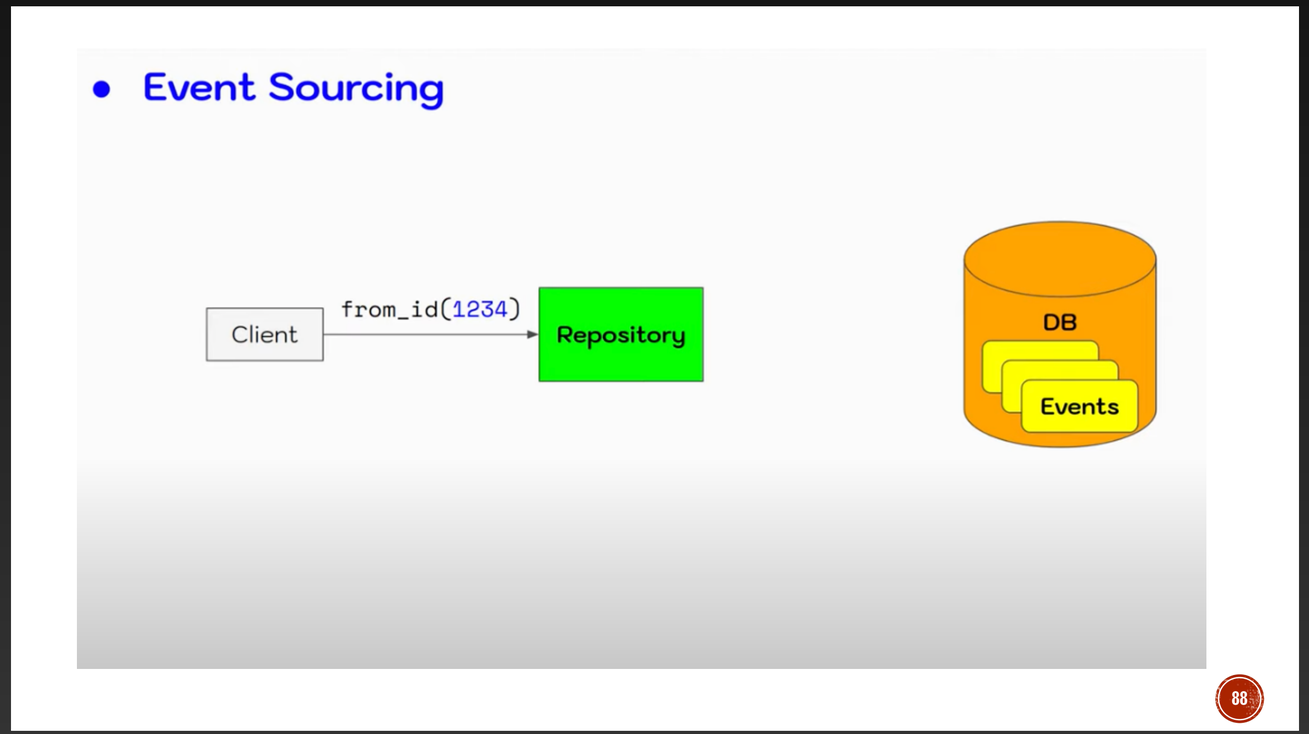
Event Sourcing

Solves data consistency issues (transaction fail)



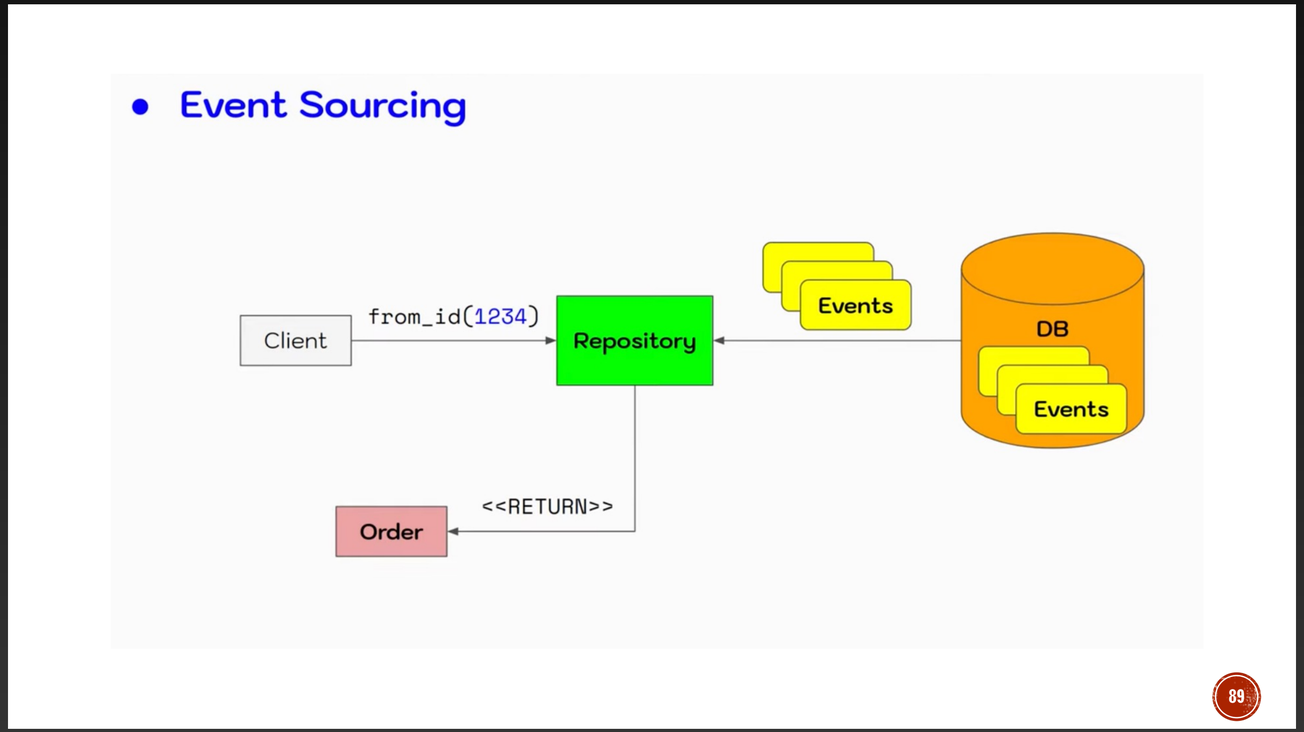
Event Sourcing

Solves data consistency issues (transaction fail)



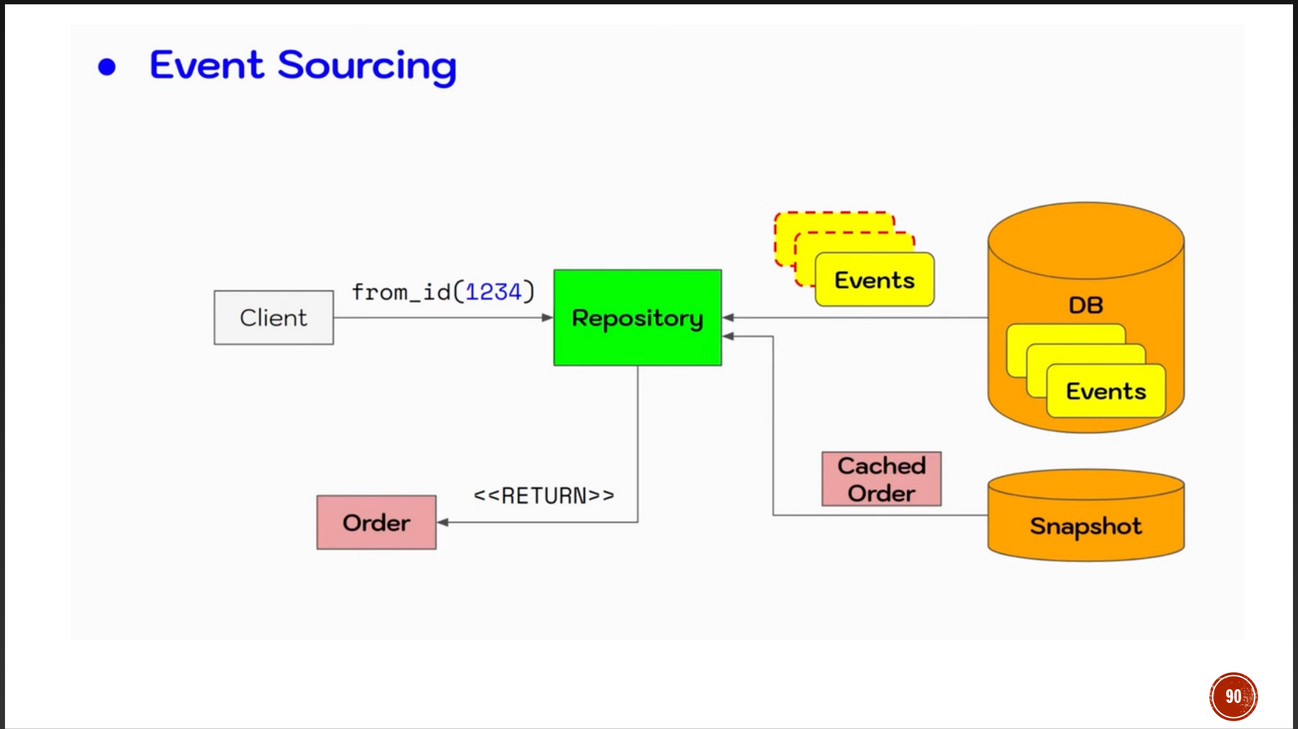
Event Sourcing

Solves data consistency issues (transaction fail)



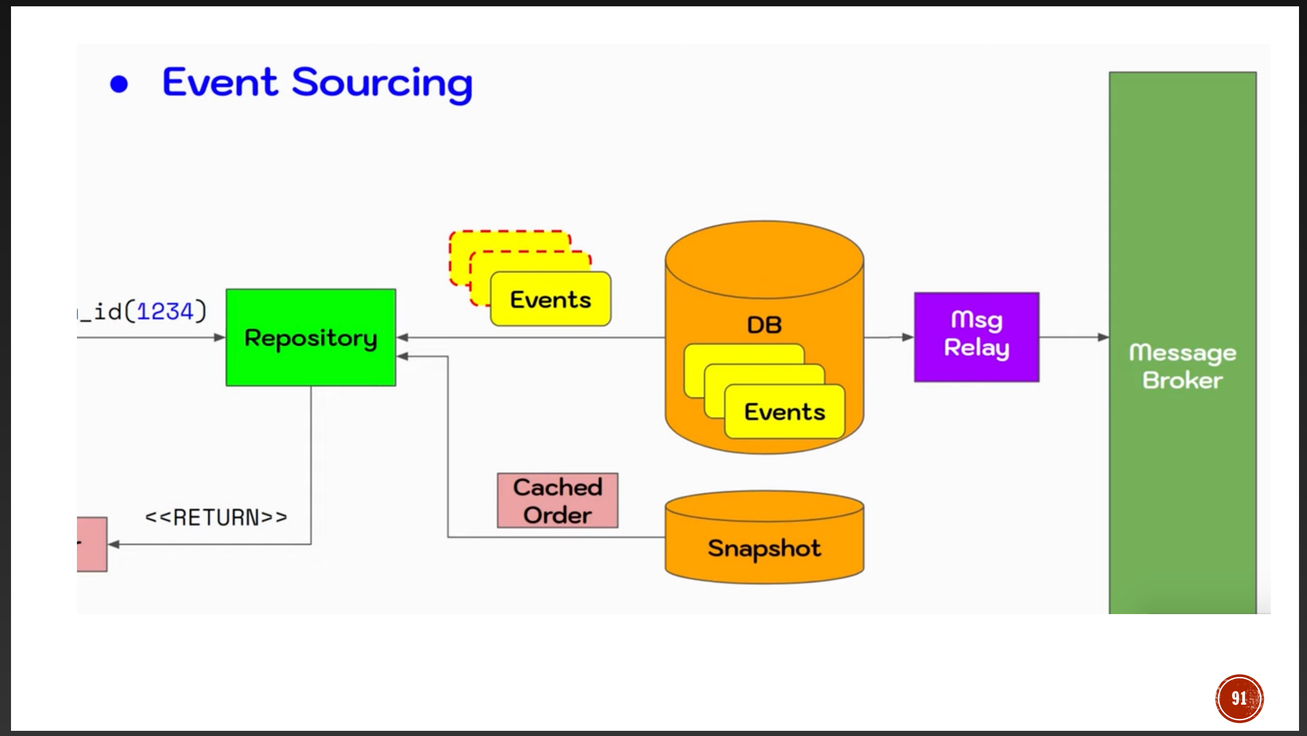
Event Sourcing

Solves data consistency issues (transaction fail)



Event Sourcing

Solves data consistency issues (transaction fail)



Event Sourcing
Advantages

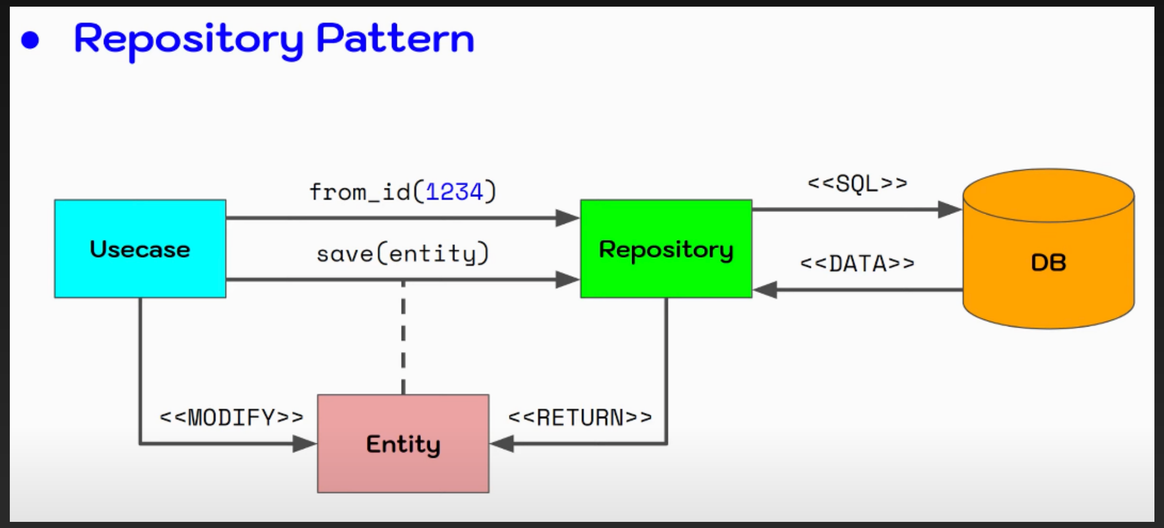
- Solves data consistency issues in a Microservice/NoSQL based architecture
- Reliable event publishing: publishes events needed by predictive analytics etc, user notifications,...
- Eliminates O/R mapping problem (mostly)
- Reifies state changes:
  - Built in, reliable audit log
  - temporal queries
- Preserved history: More easily implement future requirements

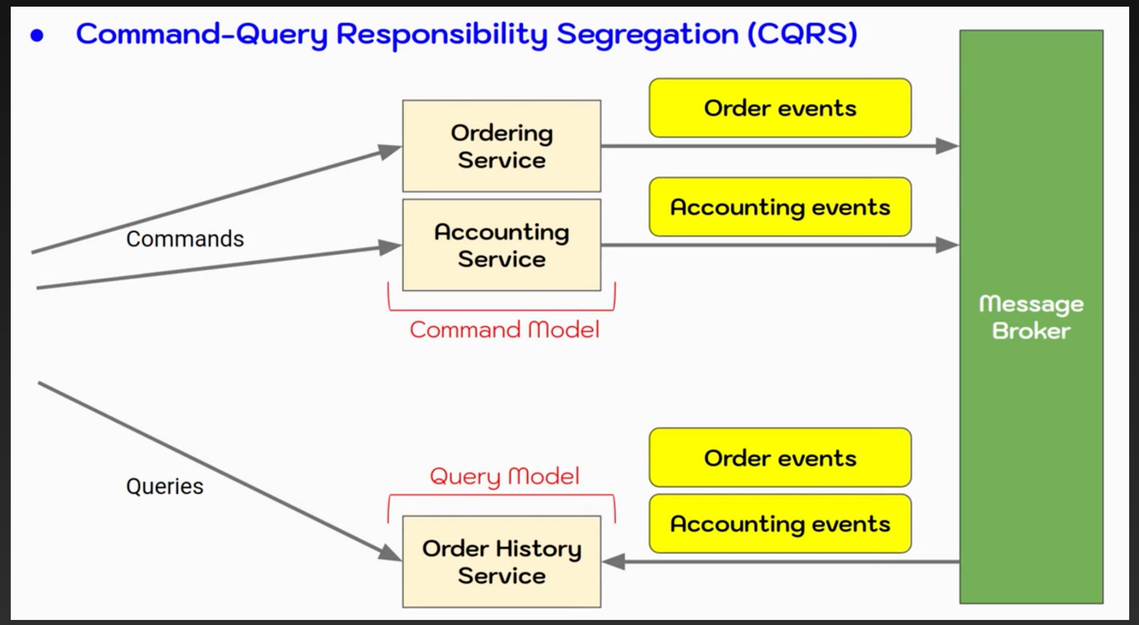
Event Sourcing
Drawbacks

- Requires application rewrite
- Weird and unfamiliar style of programming
- Events = a historical record of your bad design
- decisions Must detect and ignore duplicate event
  - Idempotent event handlers
  - Track most recent event and ignore older ones

CQRS

Another good way to implement queries





CQRS

Another good way to implement queries

- CQRS: Command Query Responsibility Segregation
- CQRS is an architectural pattern that splits the application into two parts.
  - the command side: handles commands (create, update, delete)
  - the query side: handles queries (read)
- The point is that the query side keeps the views synchronized with the aggregates by subscribing to events published by the command side.