Lesson 9

MICROSERVICES

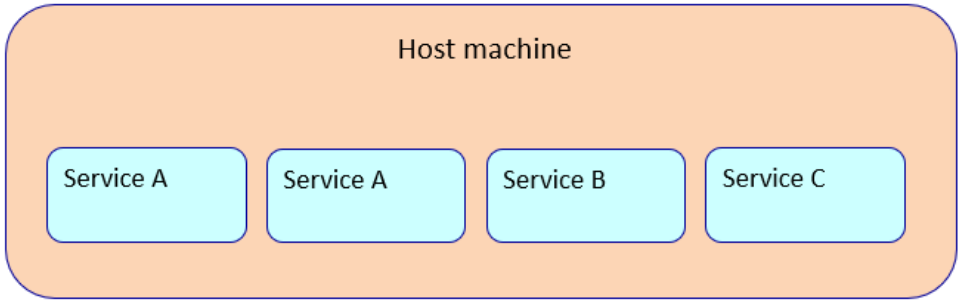
SERVICE DEPLOYMENT

Service deployment

- Service are written using different languages, frameworks, framework versions
- Run multiple service instances of a service for throughput and availability
- Building and deploying should be fast
- Instances need to be isolated
- Constrain the resources a service may consume (CPU, memory, etc.)
- Deployment should be reliable

Multiple service instances per host

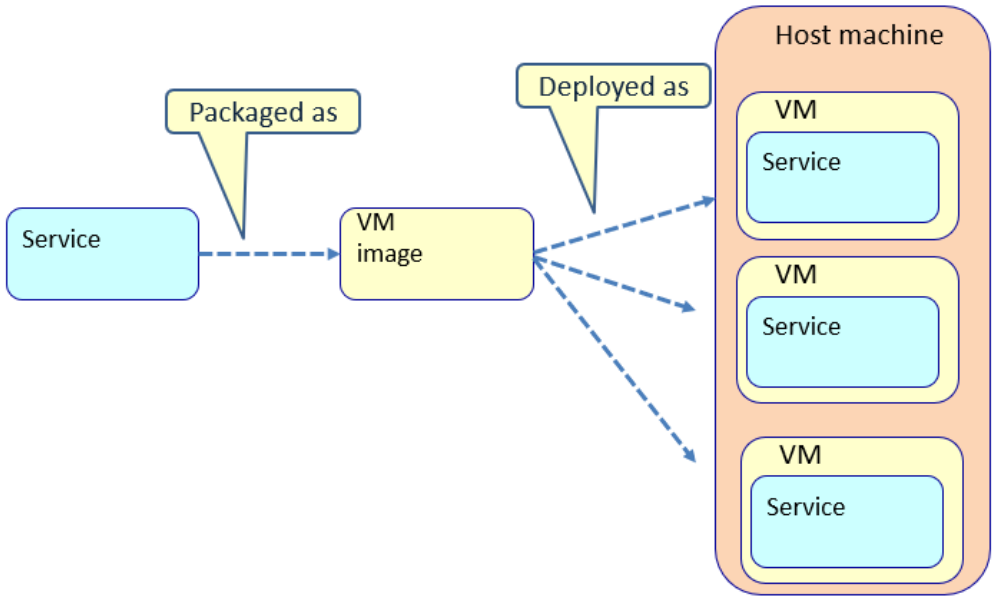
- Benefits
  - Efficient resource utilization
  - Fast deployment



- Drawbacks
  - Poor isolation
  - Poor visibility of resource utilization
  - Difficult to constrain resource utilization
  - Risk of dependency version conflicts
  - Poor encapsulation of implementation technology

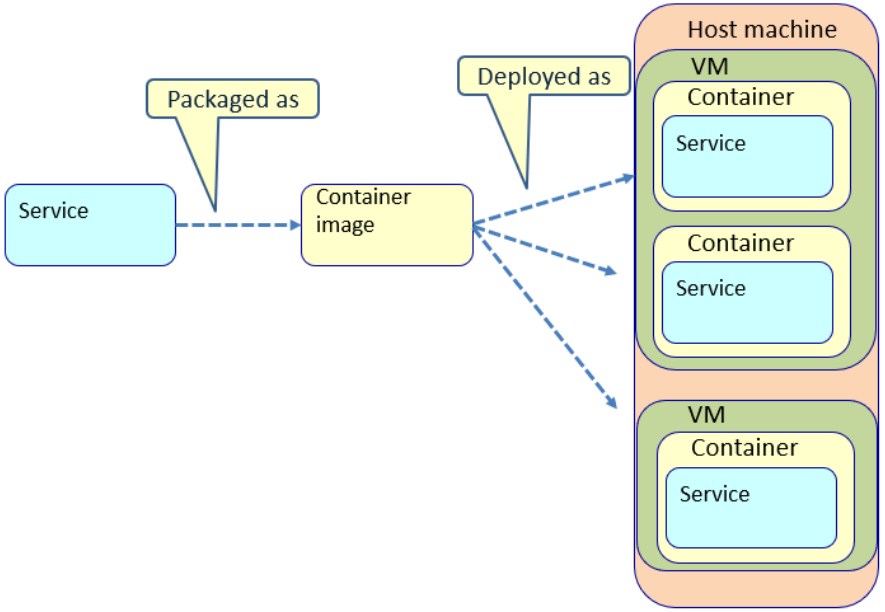
Service per VM

- Benefits
  - Great isolation
  - Great manageability
  - VM encapsulates implementation technology
  - Leverage cloud infrastructure for auto scaling/load balancing
- Drawbacks
  - Less efficient resource utilization
  - Slow deployment



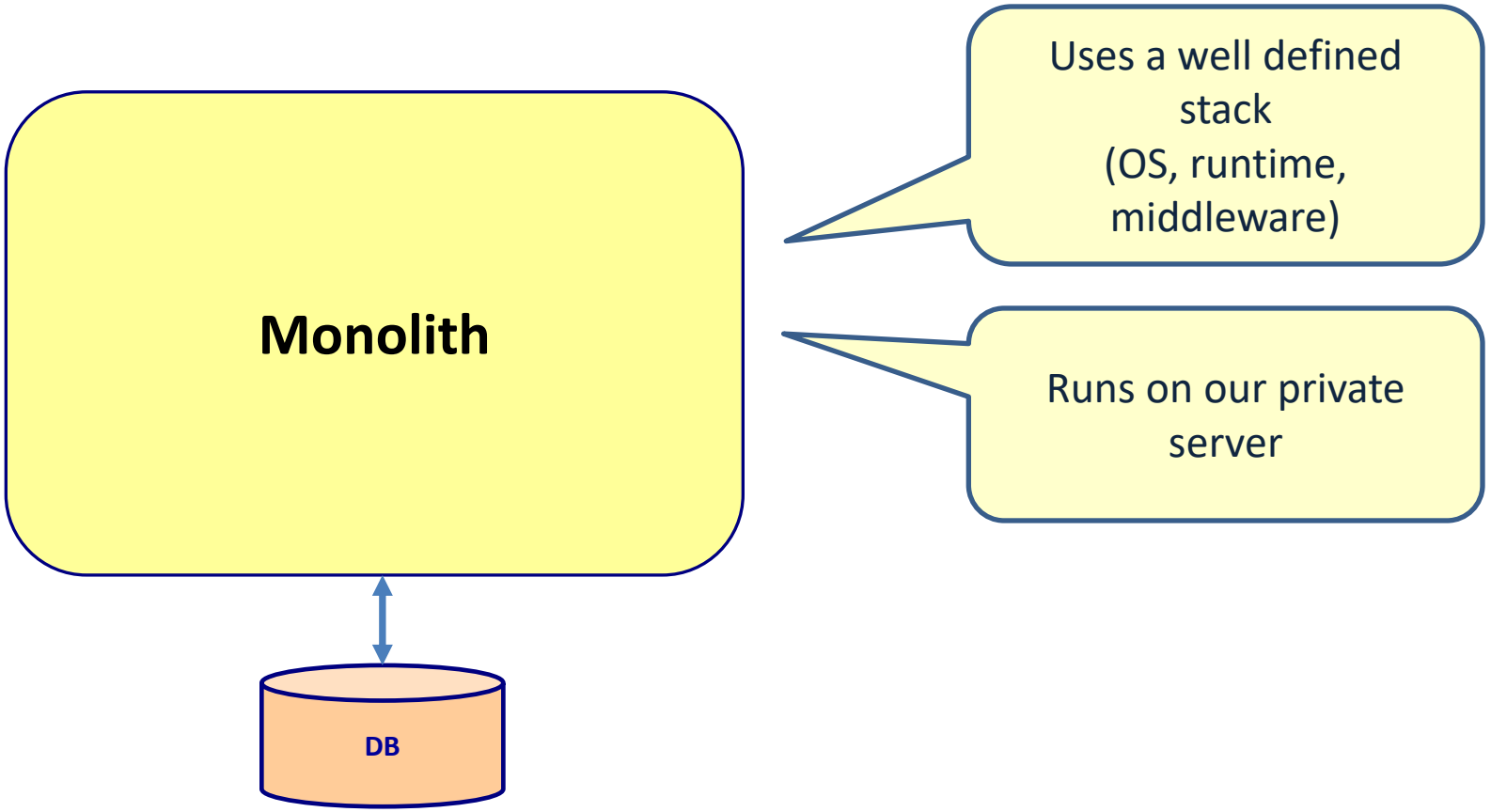
Service per container

- Benefits
  - Great isolation
  - Great manageability
  - Container encapsulates implementation technology
  - Efficient resource utilization
  - Fast deployment
- Drawbacks
  - Technology is not as mature as VM's
  - Containers are not as secure as VM's

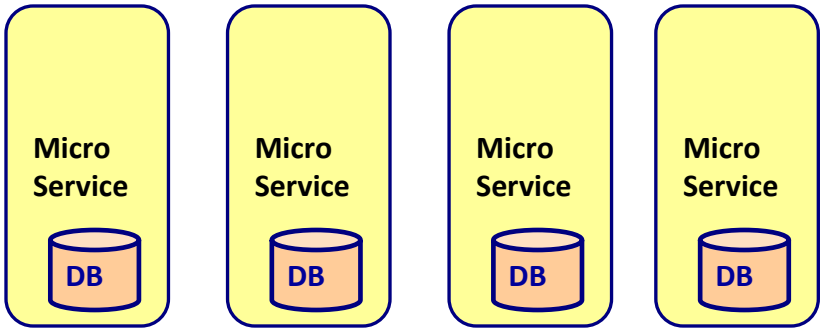


CONTAINERS

Monolith



Microservice



Microservices use different stacks. (OS, runtime, middleware)

Runs anywhere (private, cloud)

The challenge

User DB

postgresql + pgv8 + v8

Static website

nginx 1.5 + modsecurity + openssl + bootstrap 2

Queue

Redis + redis-sentinel

Analytics DB

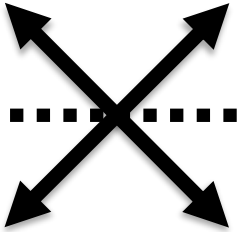
hadoop + hive + thrift + OpenJDK

Background workers

Web frontend

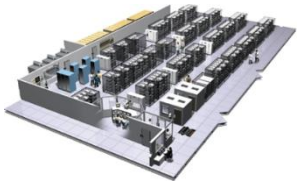
Ruby + Rails + sass + Unicorn

API endpoint



Public Cloud

Development VM



Production Cluster

QA server

Disaster recovery

Customer Data Center

Production Servers

Contributor's laptop

Matrix from hell

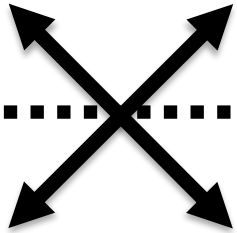
	Development VM	QA Server	Single Prod Server	Onsite Cluster	Public Cloud	Contributor's laptop	Customer Servers
Static website	?	?	?	?	?	?	?
Web frontend	?	?	?	?	?	?	?
Background workers	?	?	?	?	?	?	?
User DB	?	?	?	?	?	?	?
Analytics DB	?	?	?	?	?	?	?
Queue	?	?	?	?	?	?	?

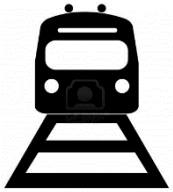
Cargo Transport Pre-1960

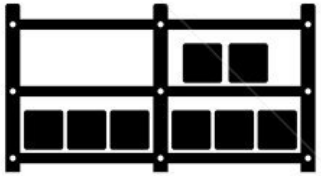






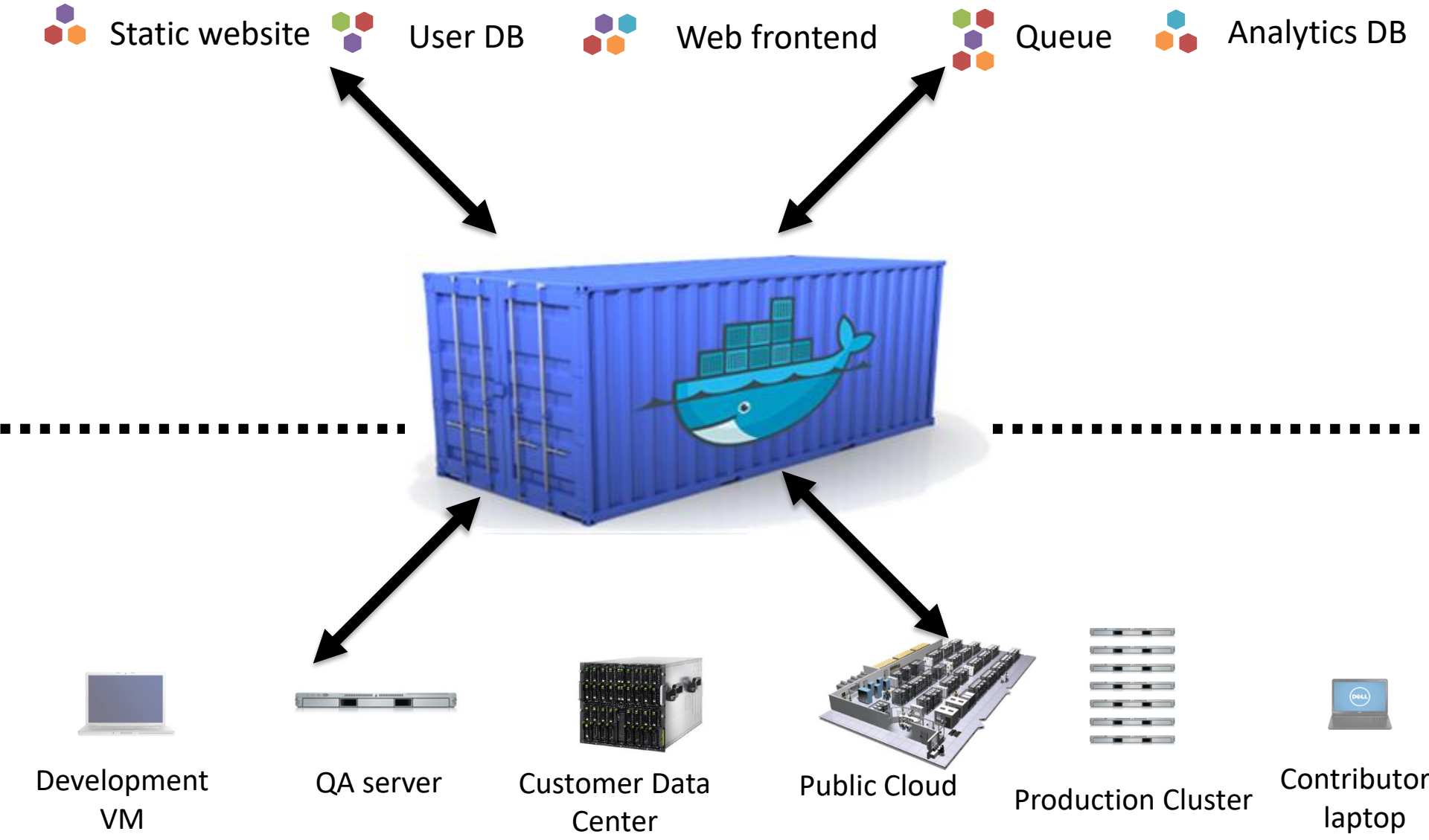



The solution

Docker is a shipping container for code

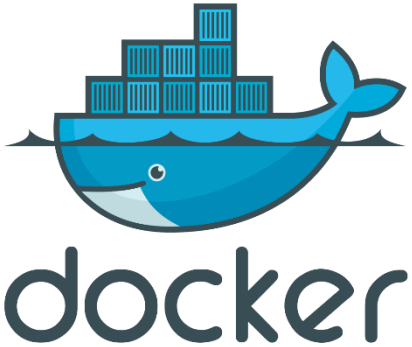


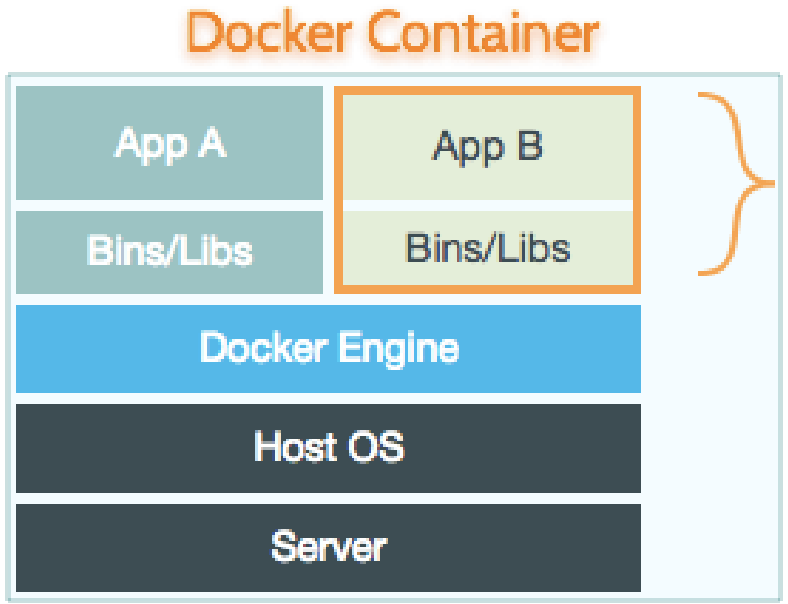
Separation of concern

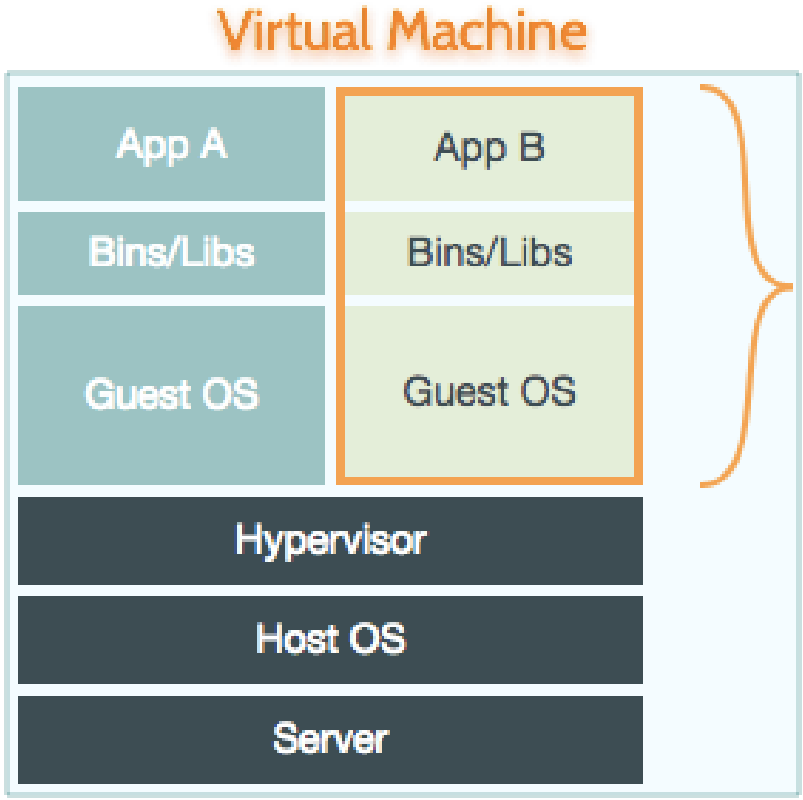
- Dan the Developer
  - Worries about what's "inside" the container
    - His code
    - His Libraries
    - His Package Manager
    - His Apps
    - His Data
- Oscar the Ops Guy
  - Worries about what's "outside" the container
    - Logging
    - Remote access
    - Monitoring
    - Network config
  - All containers start, stop, copy, attach, migrate, etc. the same way

Containers

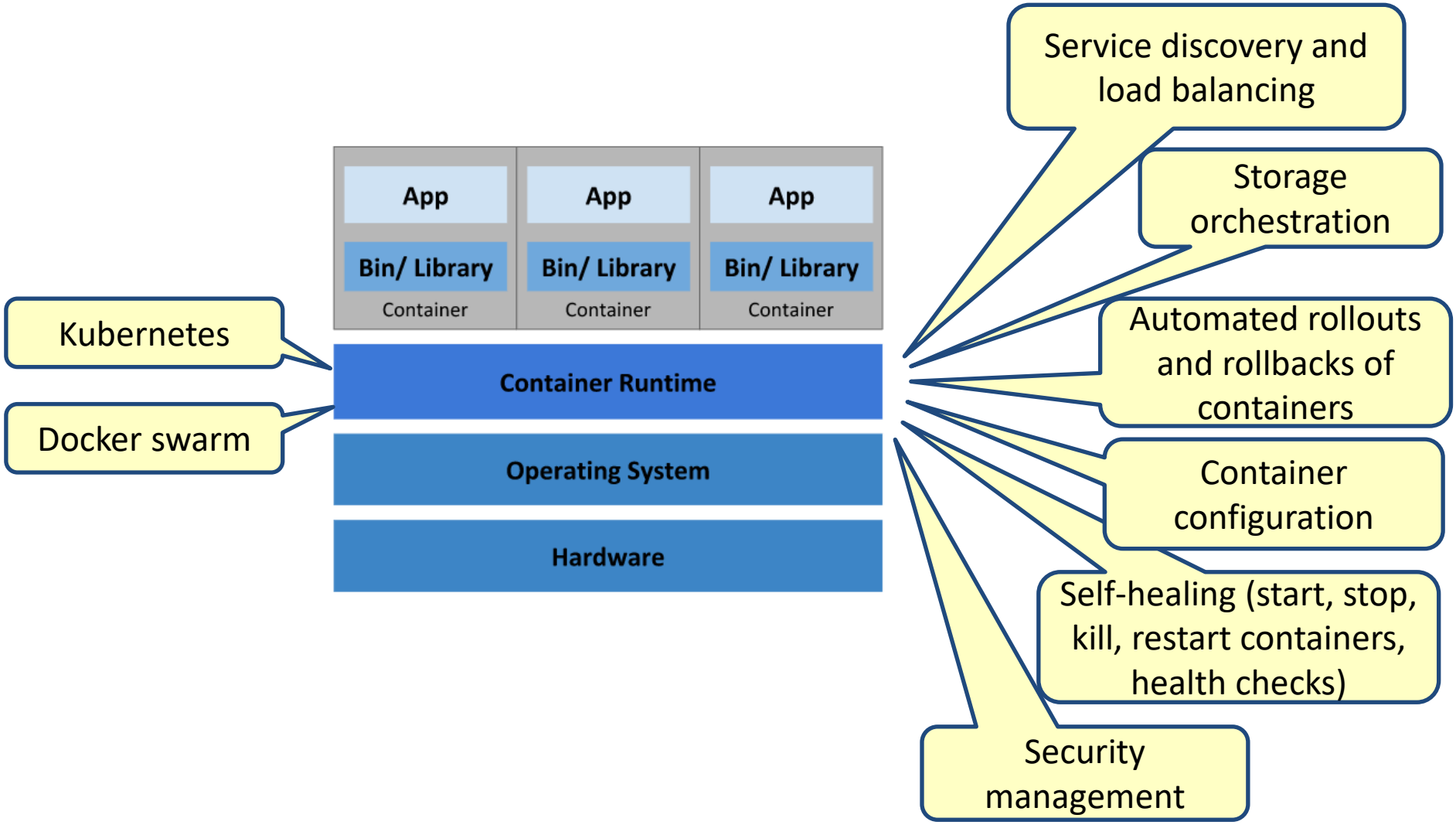
- Docker







Container management



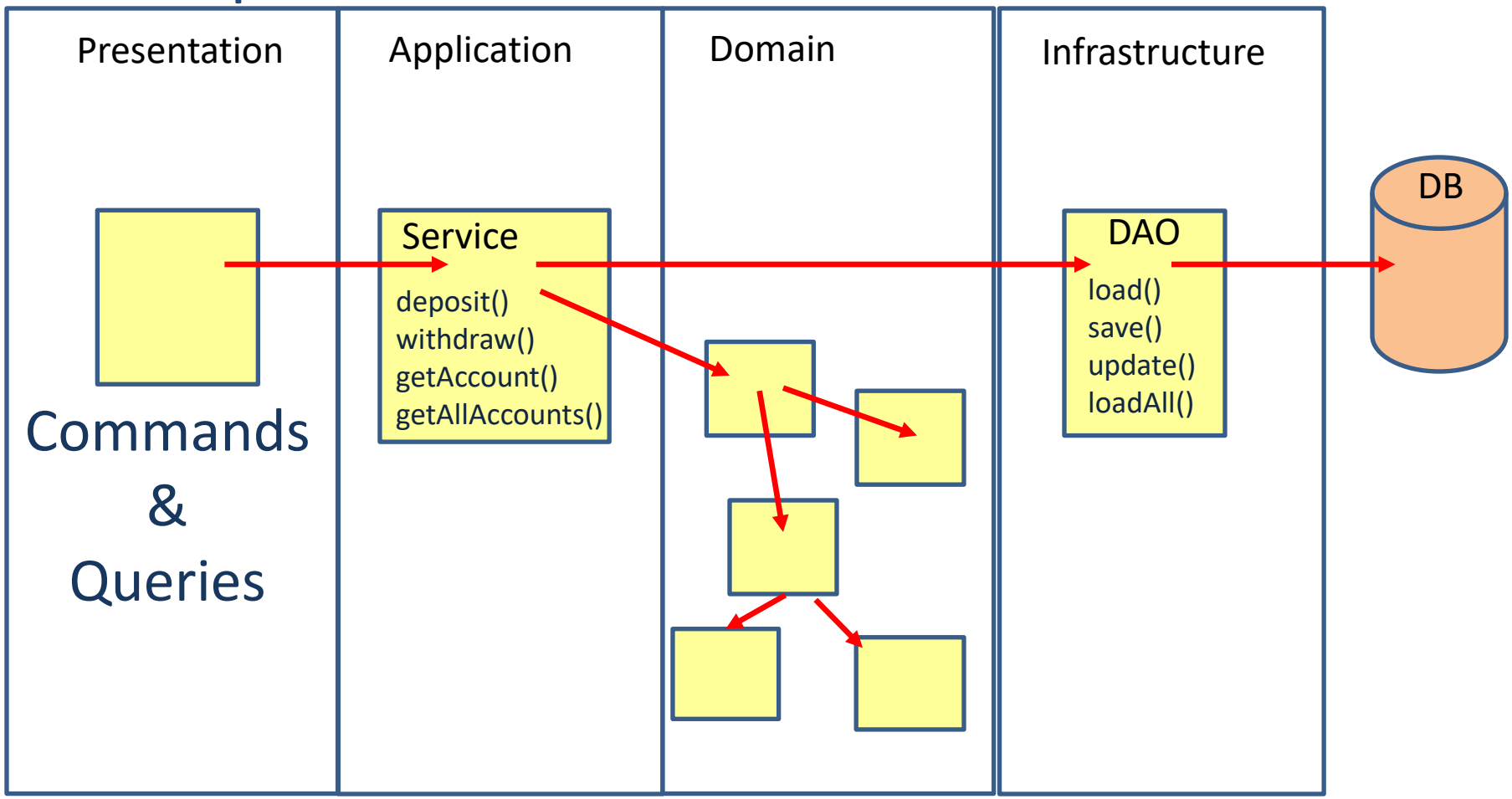
CQRS

Command Query Responsibility Segregation (CQRS)

- Separates the querying from command processing by providing two models instead of one.
  - One model is built to handle and process commands
  - One model is built for presentation needs (queries)

Typical architecture

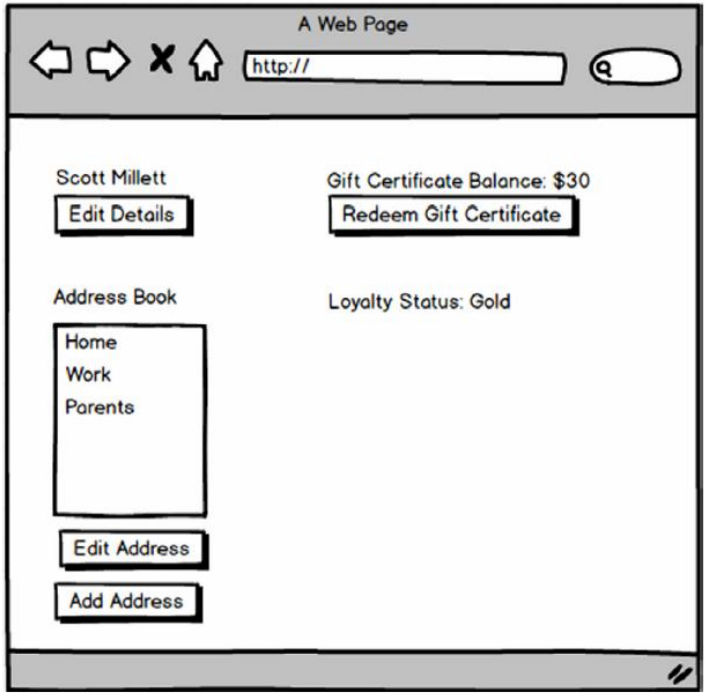
- One domain model that is used for commands and queries



One model for both commands and queries

- To support complex views and reporting
  - Required domain model becomes complex
  - Internal state needs to be exposed
  - Aggregates are merged for view requirements
  - Repositories often contain many extra methods to support presentation needs such as paging, querying, and free text searching
- Result: single model that is full of compromises

Example of complex aggregates



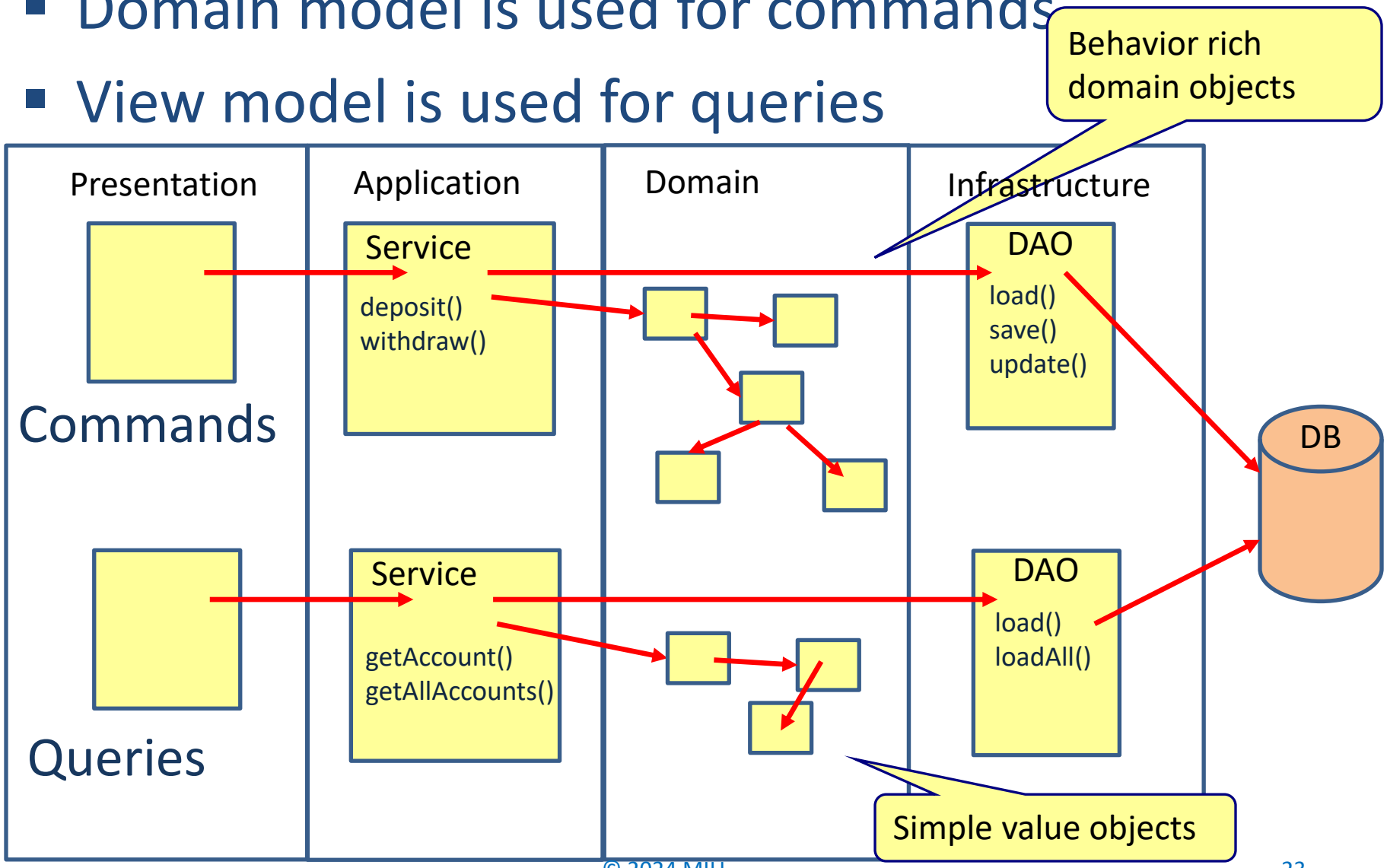
Complex aggregate because of UI needs

We don't need this complexity for commands

```
public class Customer
{
    // ...
    public ContactDetails ContactDetails { get; private set; }
    public LoyaltyStatus LoyaltyStatus { get; private set; }
    public Money GiftCertBalance { get; private set; }
    public IEnumerable<Address> AddressBook { get; private set; }
}
```

CQRS

- Domain model is used for commands
- View model is used for queries



2 services instead of one

Traditional service

CustomerService
void MakeCustomerPreferred(CustomerId)
Customer GetCustomer(CustomerId)
CustomerSet GetCustomersWithName(Name)
CustomerSet GetPreferredCustomers()
void ChangeCustomerLocale(CustomerId, NewLocale)
void CreateCustomer(Customer)
void EditCustomerDetails(CustomerDetails)

Service with CQRS

CustomerWriteService
void MakeCustomerPreferred(CustomerId)
void ChangeCustomerLocale(CustomerId, NewLocale)
void CreateCustomer(Customer)
void EditCustomerDetails(CustomerDetails)

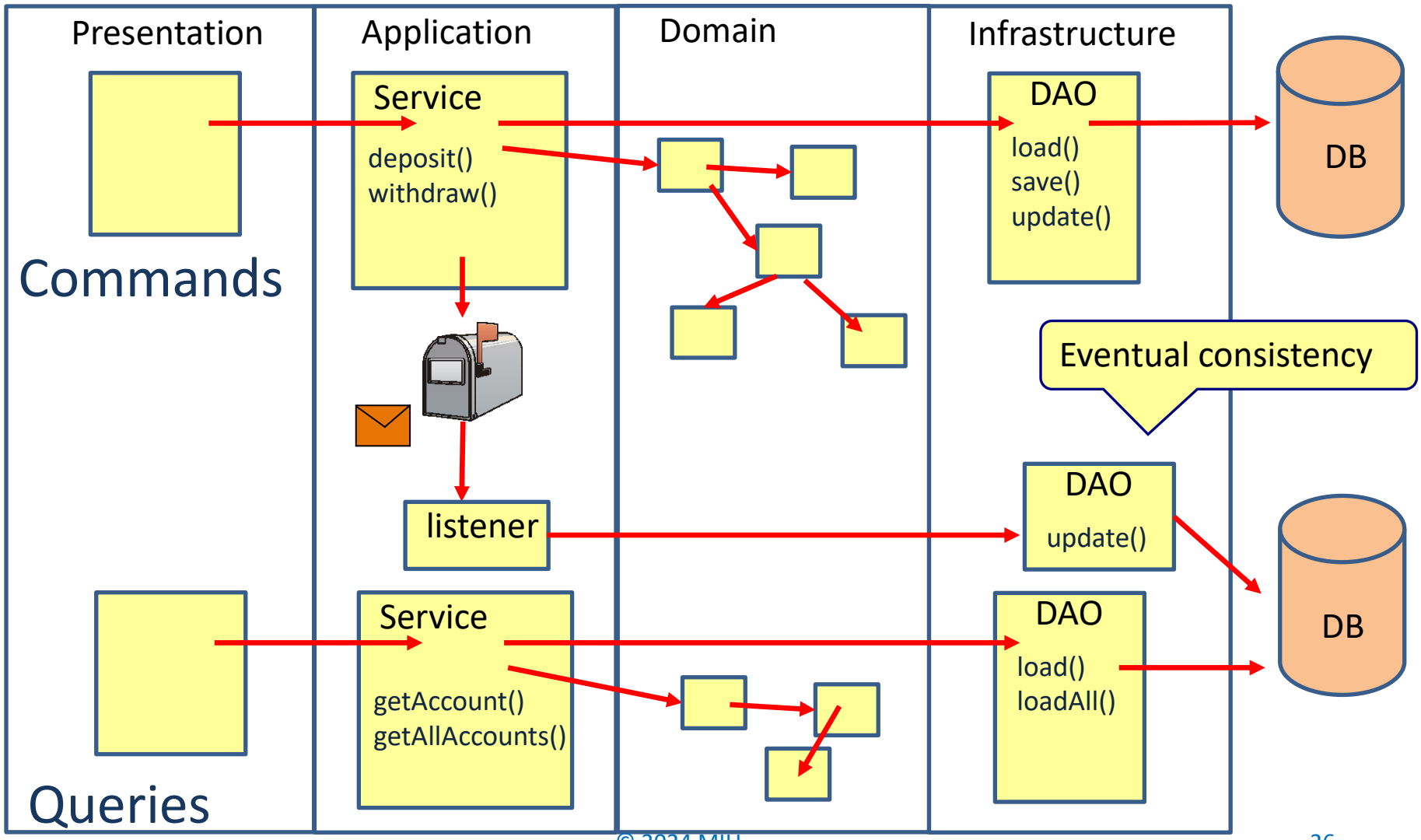
CustomerReadService
Customer GetCustomer(CustomerId)
CustomerSet GetCustomersWithName(Name)
CustomerSet GetPreferredCustomers()

Architectural properties

- Command and query side have different architectural properties
- Consistency
  - Command: needs consistency
  - Query: eventual consistency is mostly OK
- Data storage
  - Command: you want a normalized schema (3rd NF)
  - Query: denormalized (1st NF) is good for performance (no joins)
- Scalability
  - Command: commands happens not that often. Scalability is often not important.
  - Query: queries happen very often, scalability is important

Eventual consistency

- Views will become eventual consistent



CQRS advantages

- The query side is very simple
  - No transactions
- The query side can be optimized for performance
  - Fast noSQL database
  - Caching is easy
- Queries and commands can be scaled independently

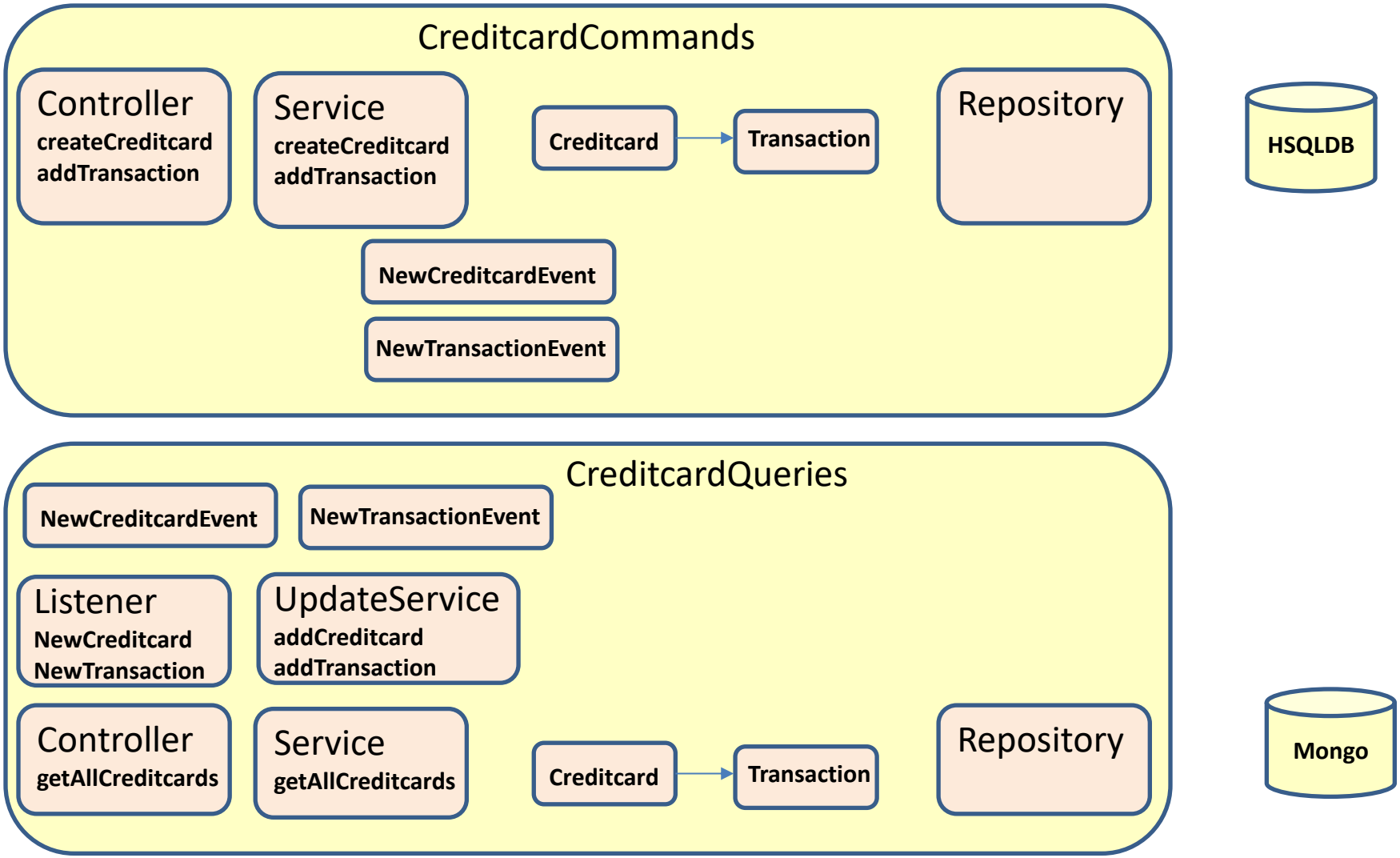
When to use CQRS?

- When queries and commands have different scaling requirements
- When read performance is critical
- When your screens start to look very different then your tables
- When you apply event sourcing

When not to use CQRS

- Systems with simple CRUD functionality
- When you need strict consistency (instead of eventual consistency)

CQRS example



Main point

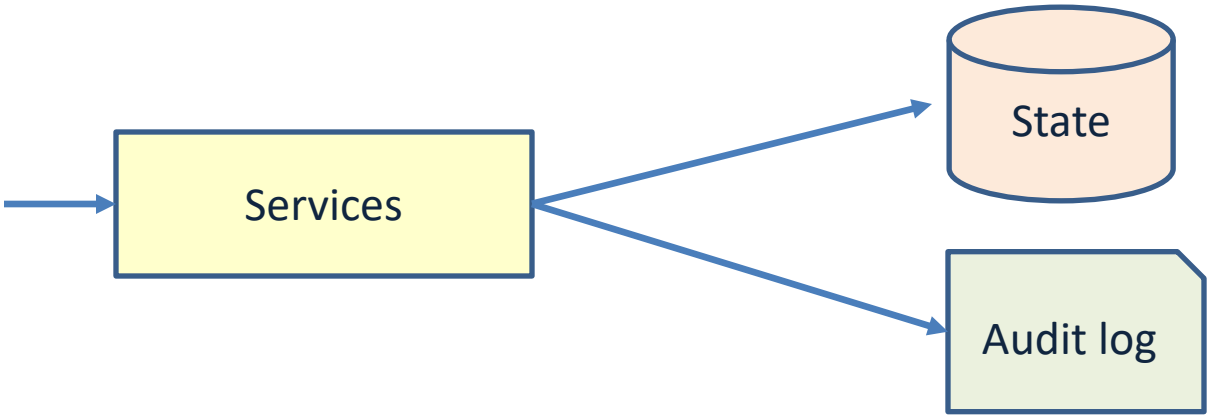
- In CQRS we separate the queries from the commands so that both can be designed, build, optimized and scaled separately.
- Every relative part of creation is connected at the level of the Unified Field.

EVENT SOURCING

State based persistance

- You capture where you are, but not how you got there
- You don’t know what happened in the past
- You cannot fix bad state because of bugs in the code

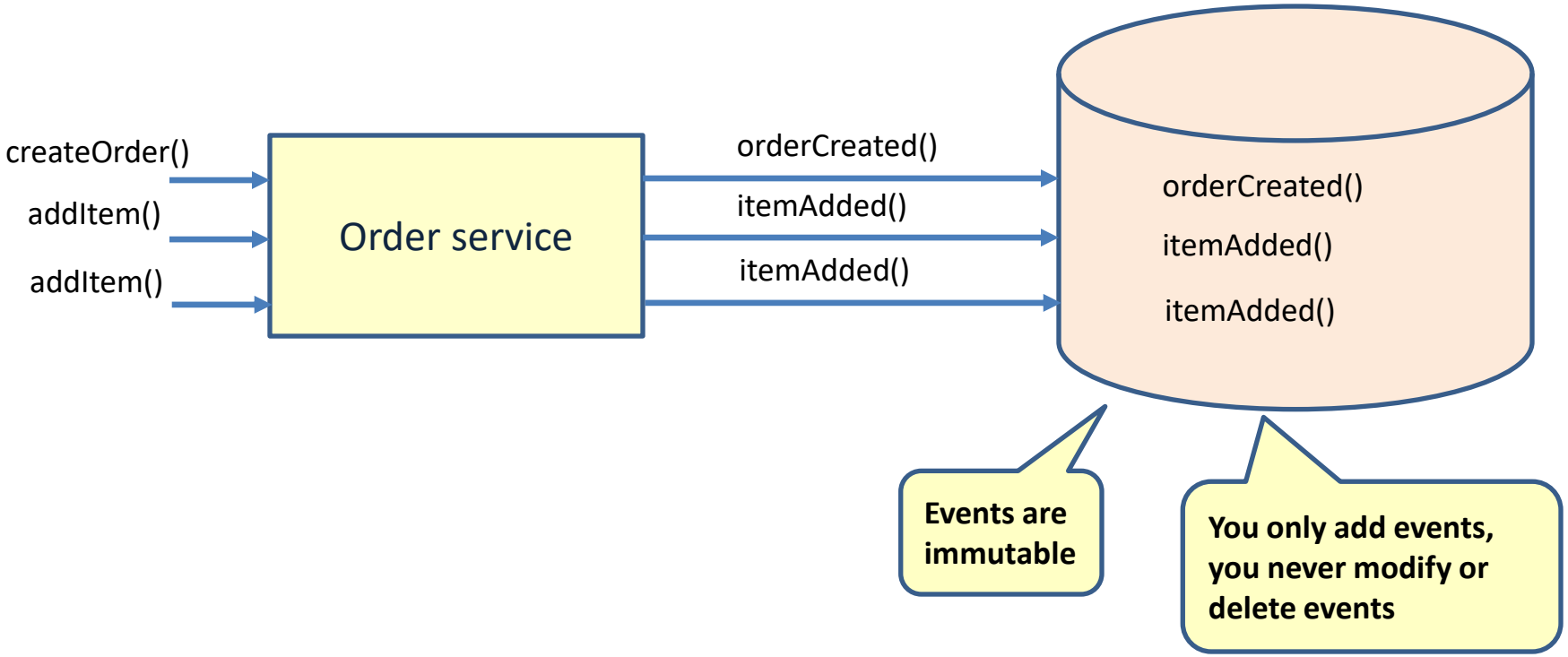
Audit log



- Now we know how we got there
- What if the audit log gets out of sync with the state due to a bug?
- Which is the source of truth? (Audit log)

Event sourcing

- Only capture how we got there



Event sourcing

- Bank account
  - Store all deposits and withdrawals
- Phone account
  - Store all phone calls
- Version control systems
  - Each commit or change is an event
- DBMS
  - Databases keep a transaction log of all inserts, deletes and updates

Storing state and storing events

- Store state

ID	status	data…
101	accepted	…

Store entity data

- Event sourcing

Entity ID	Entity type	Event ID	Event type	Event data…
101	Order	901	OrderCreated	…
101	Order	902	OrderApproved	…
101	Order	903	OrderShipped	…

Store state changing events

Store the state of a system

Structural representation

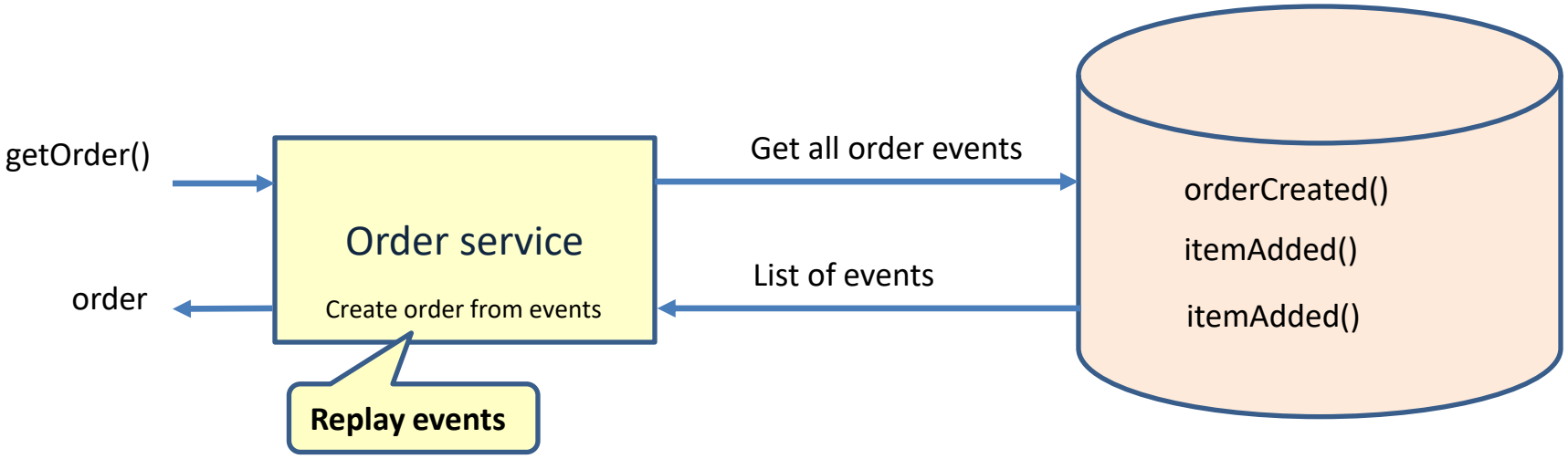
- List of ordered goods
- Payment information
- Shipping information

Store the events of a system

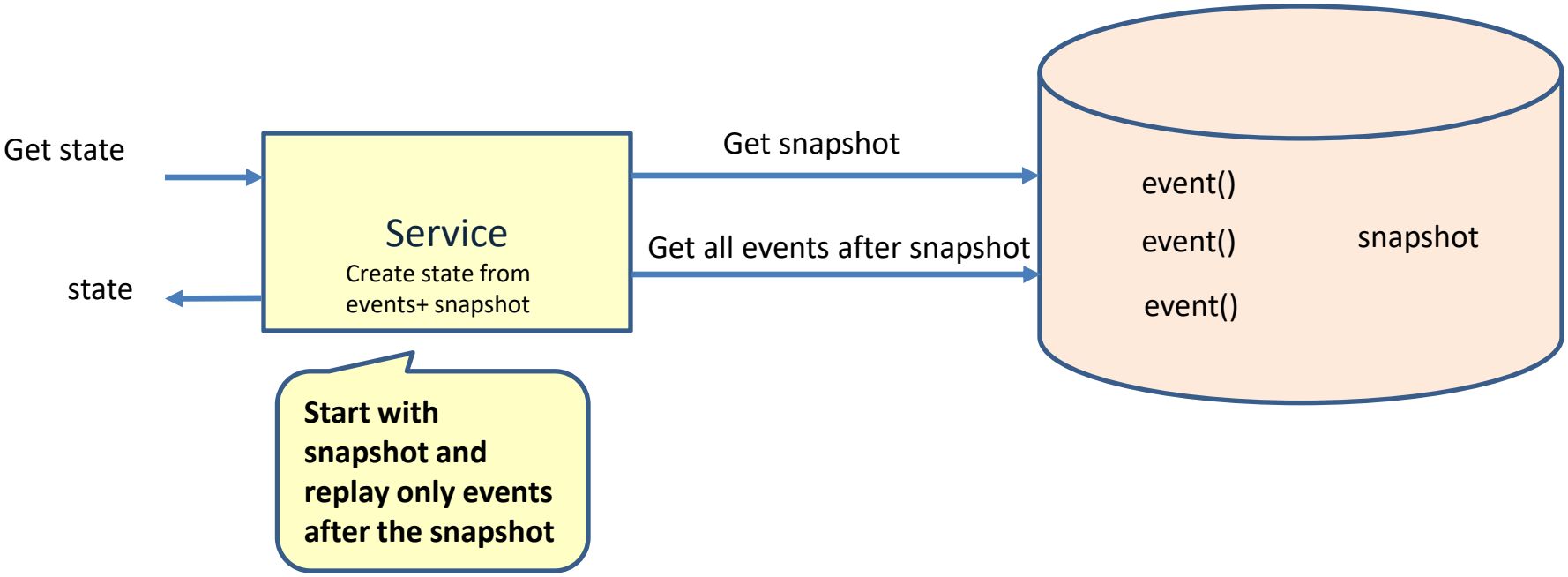
Event representation

- Add item #1
- Add item #2
- Add payment info
- Update item #2
- Remove item #1
- Add shipping info

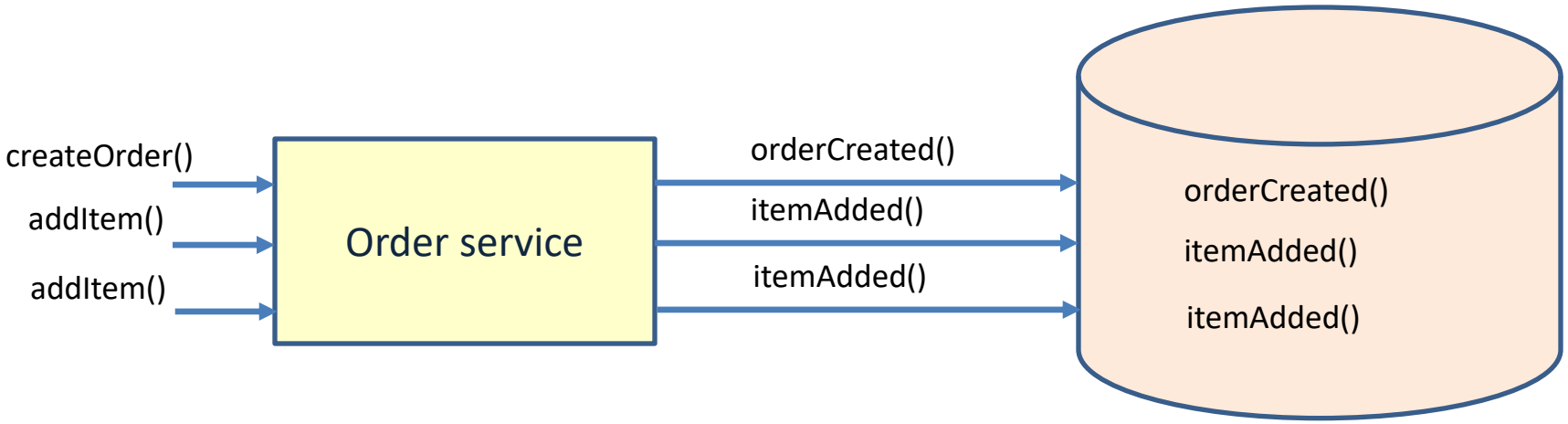
How do we get the state?



Snapshots

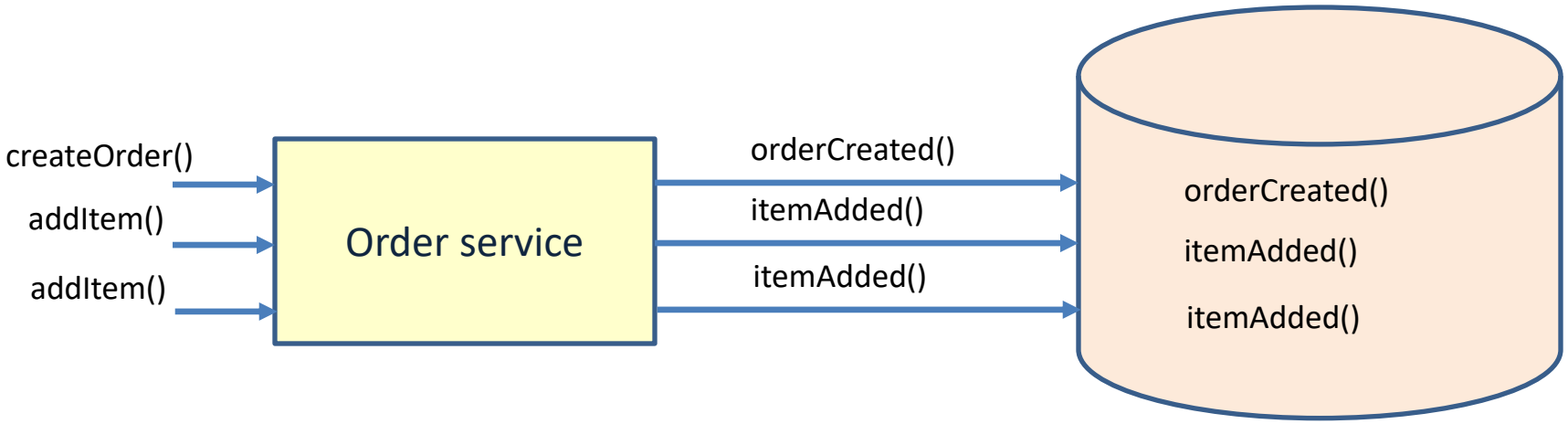

Get state
state
Service
Create state from
events+ snapshot
Get snapshot
Get all events after snapshot
event()
event()
event()
snapshot
Start with
snapshot and
replay only events
after the snapshot

Event sourcing advantages



- You don’t miss a thing
  - Business can analyze history of events
  - Bugs can be solved easier
- Creates a built-in audit log
- Allows for rewinding or undo changes
- Append is usually more efficient in databases
- No transactions needed

Event sourcing disadvantages



- You cannot easily see what your state is
  - You always have to replay the events first (lower read performance)
  - Always use CQRS when you apply event sourcing
- You need more storage
- What if events have to change?
  - We need event versioning
  - The logic needs to support all versions
- Queries over multiple event logs can be complex
  - Use CQRS

When to consider event sourcing?

- When you need an audit log
- When you want to derive additional business value from the event history
- When you need fast performance on the command side

Main point

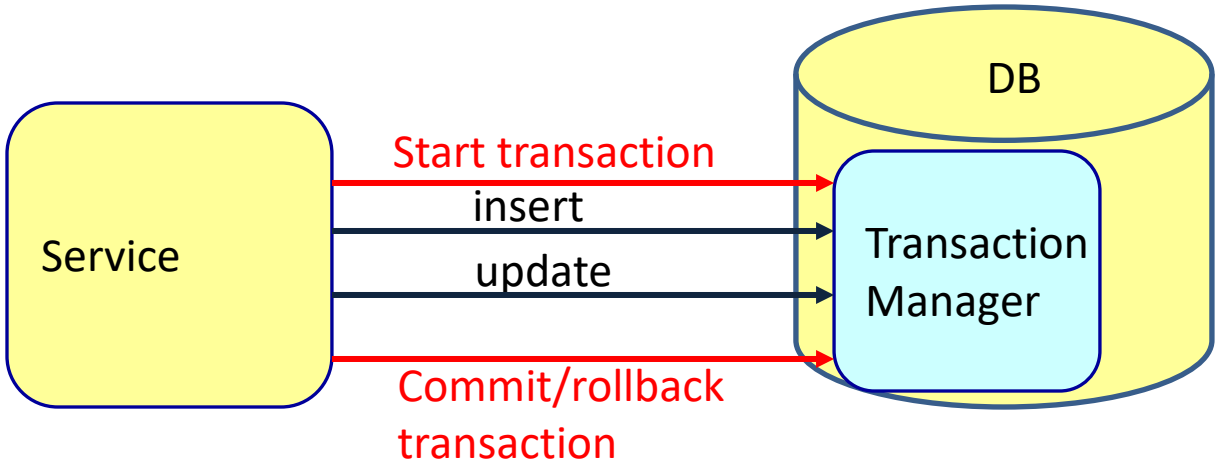
- In event sourcing we store the events instead of the current state.
- Pure consciousness is the field of all possibilities where one has access to all intelligence of creation.

TRANSACTIONS

Transactions

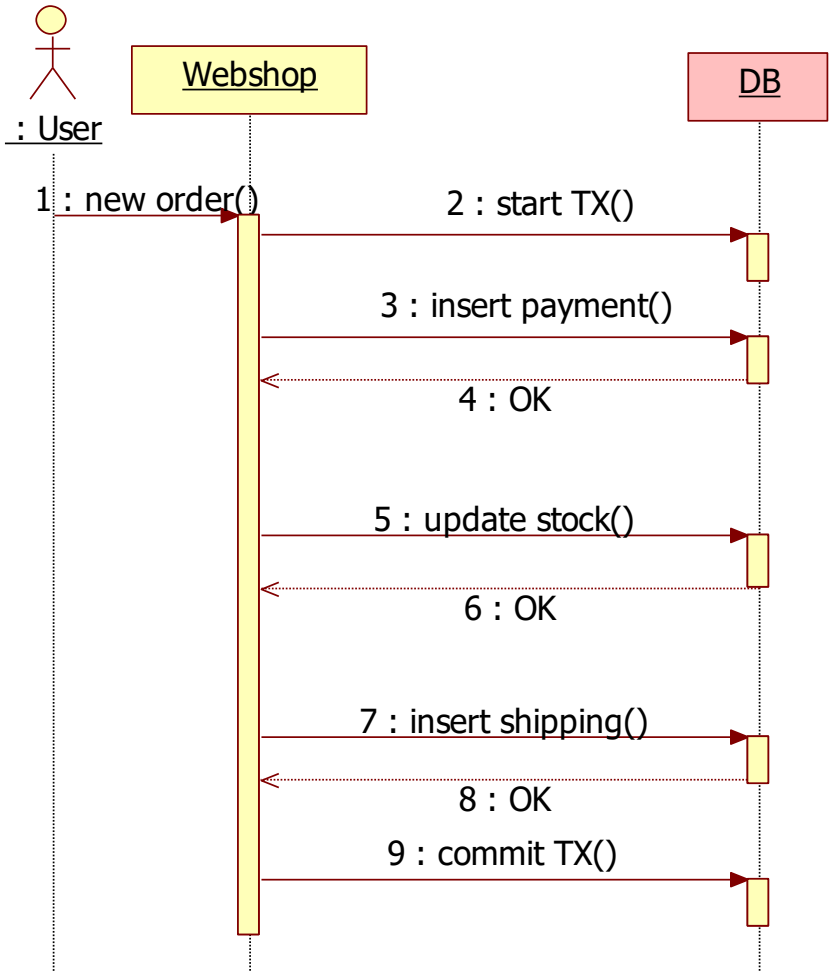
- A Transaction is a unit of work that is:
  - **ATOMIC:** The transaction is considered a single unit, either the entire transaction completes, or the entire transaction fails.
  - **CONSISTENT:** A transaction transforms the database from one consistent state to another consistent state
  - **ISOLATED:** Data inside a transaction can not be changed by another concurrent processes until the transaction has been committed
  - **DURABLE:** Once committed, the changes made by a transaction are persistent

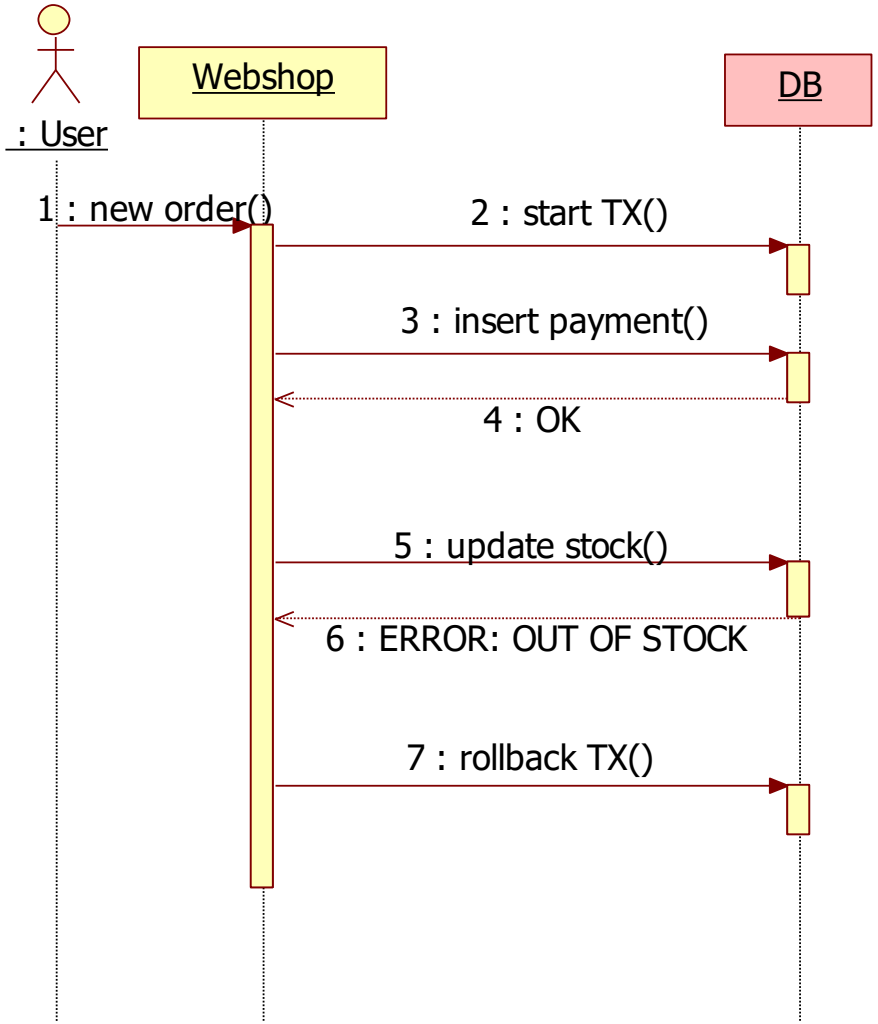
Local transaction



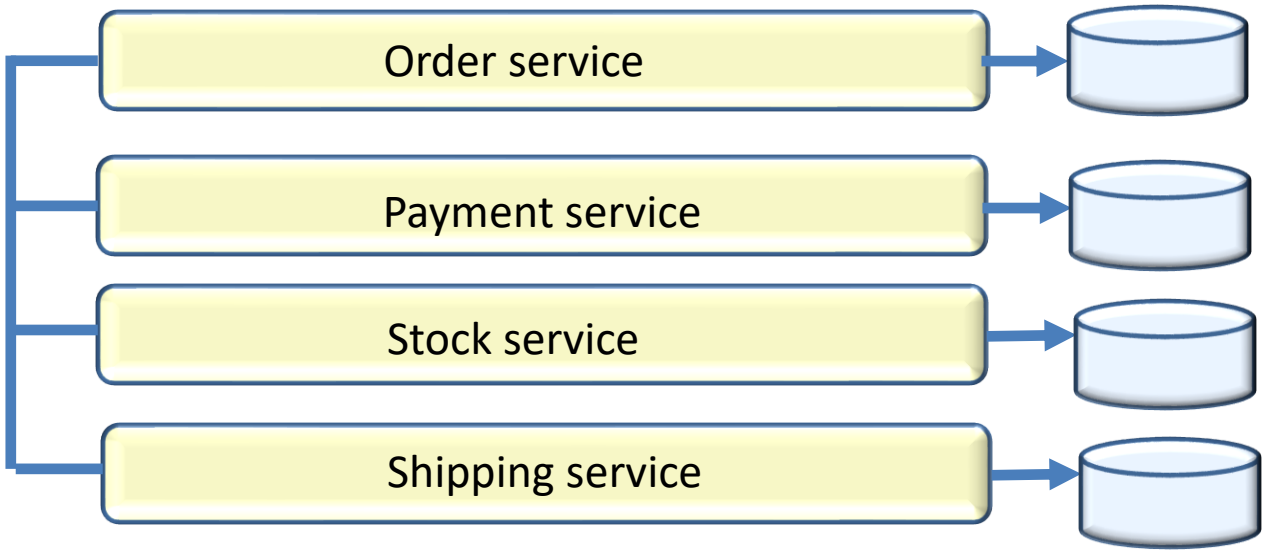
- The transaction is managed by the database
  - Simple
  - Fast
- Always try to keep transaction boundaries within a service

Local transaction



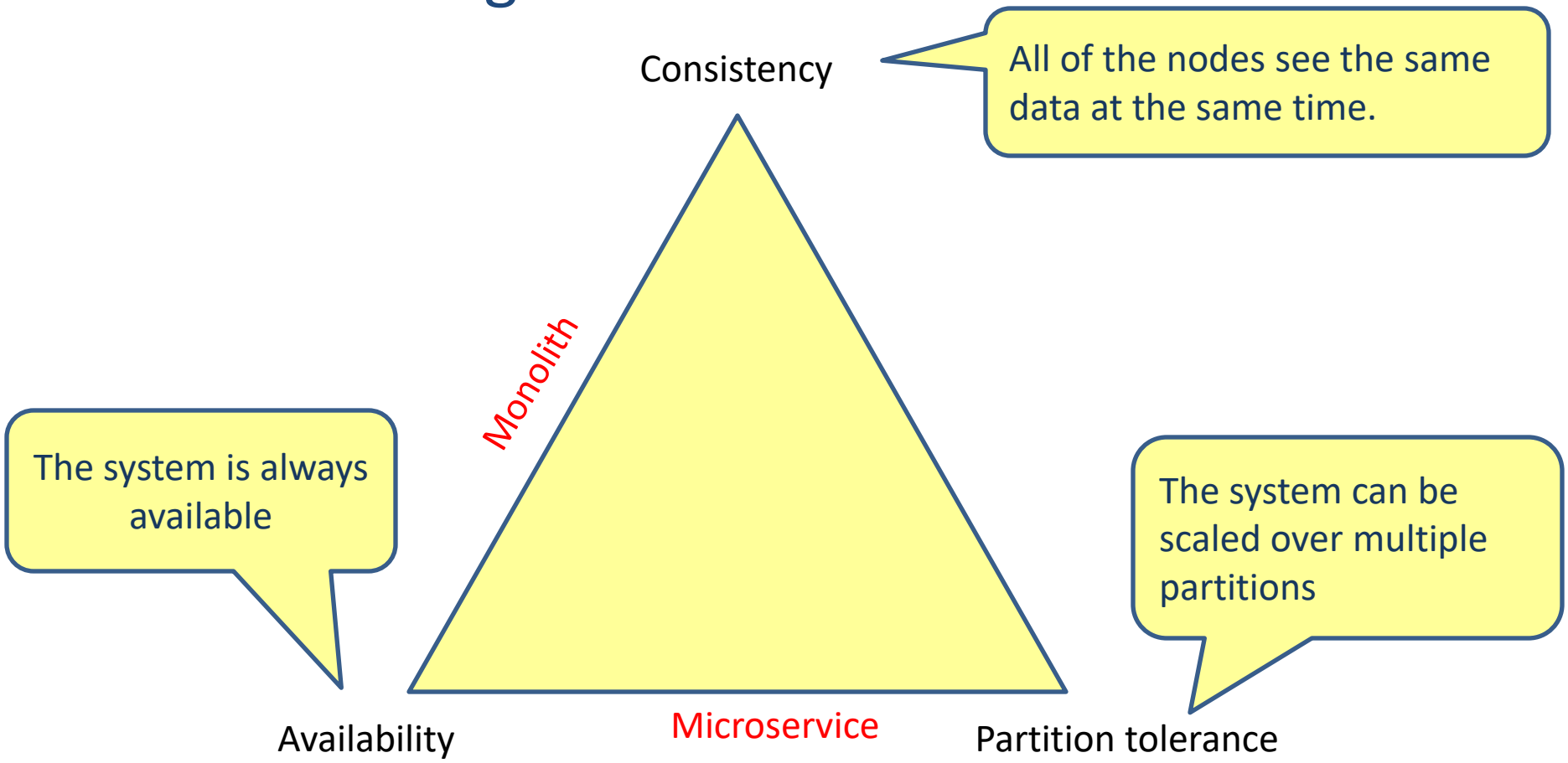


Distributed transactions

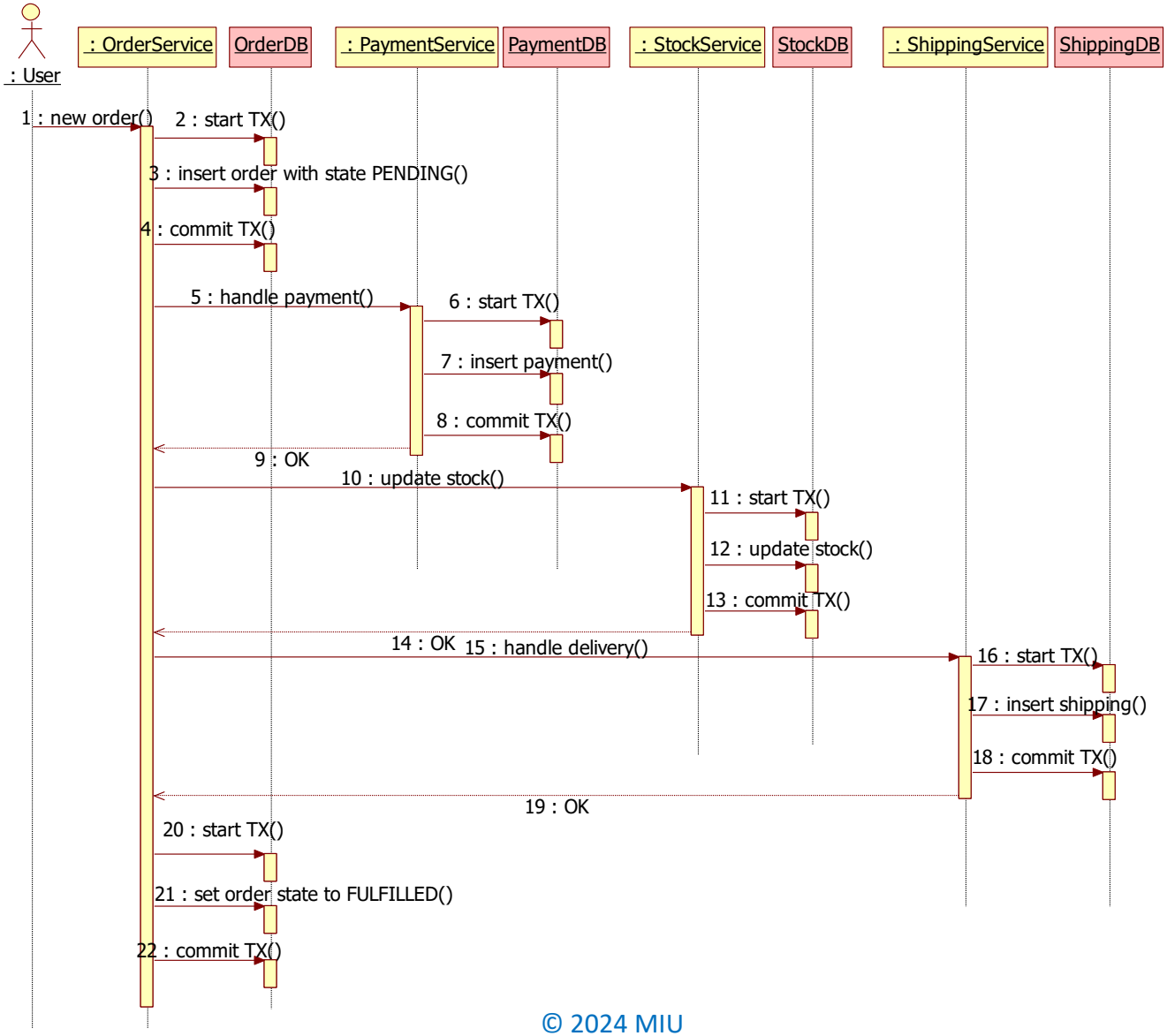


Brewer‘s CAP Theorem

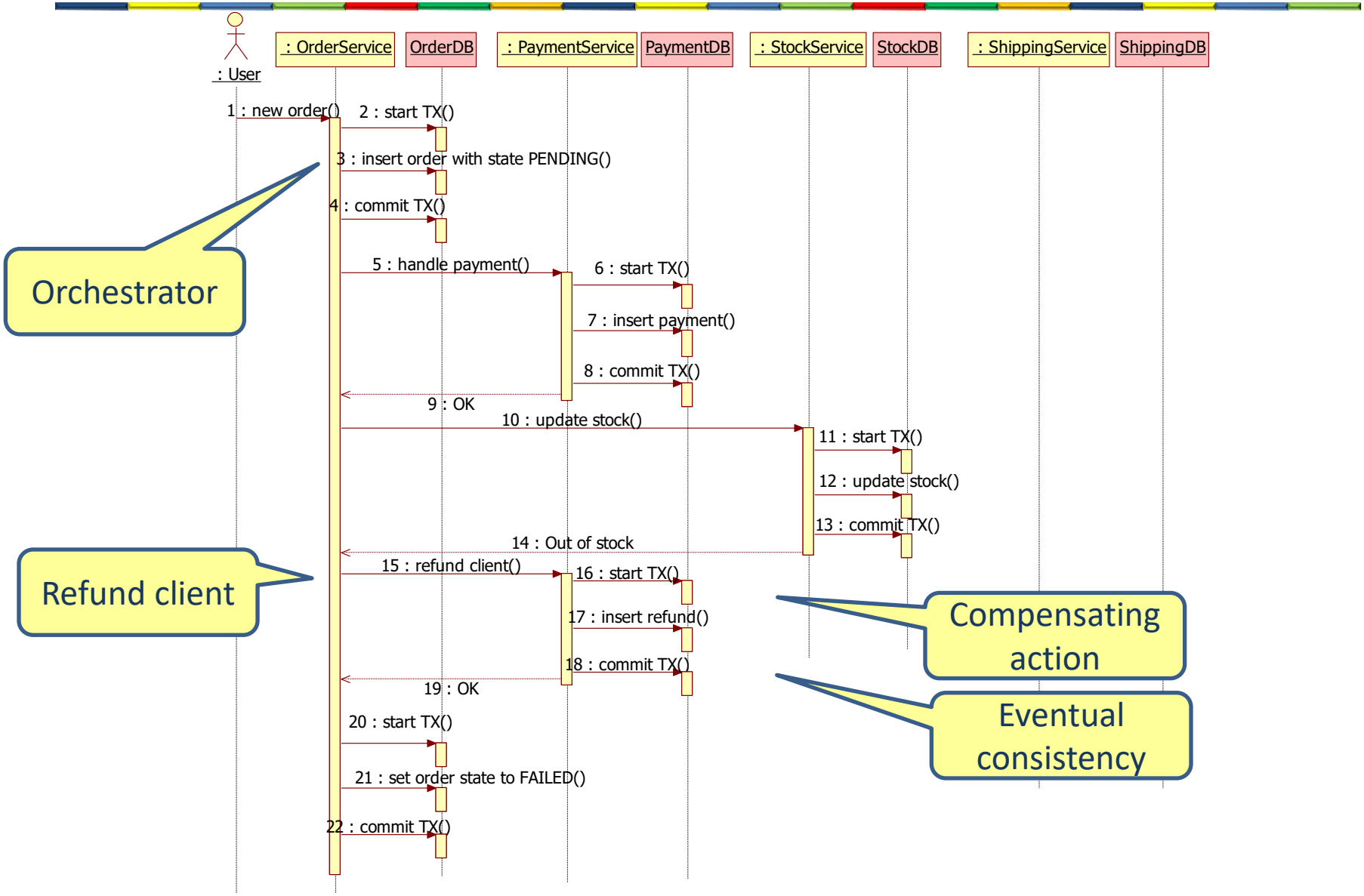
- A distributed system can support only two of the following characteristics



Distributed transaction (SAGA)

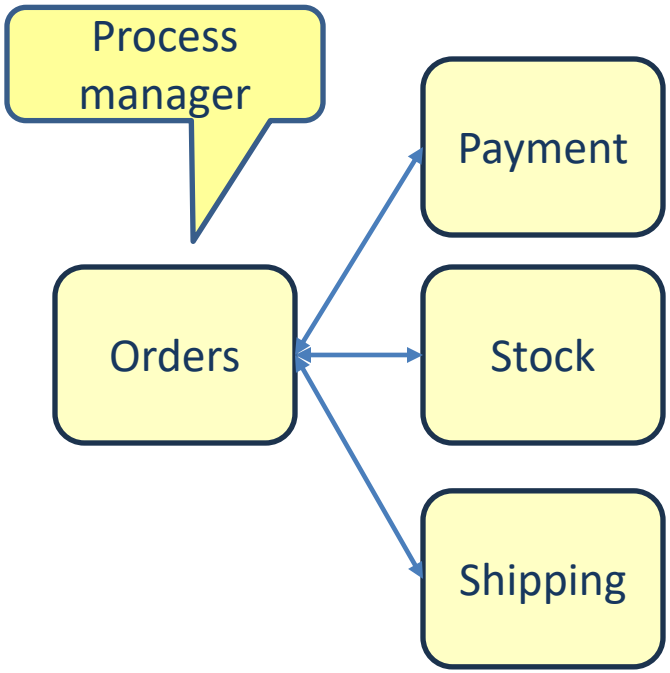


Distributed transaction (SAGA)

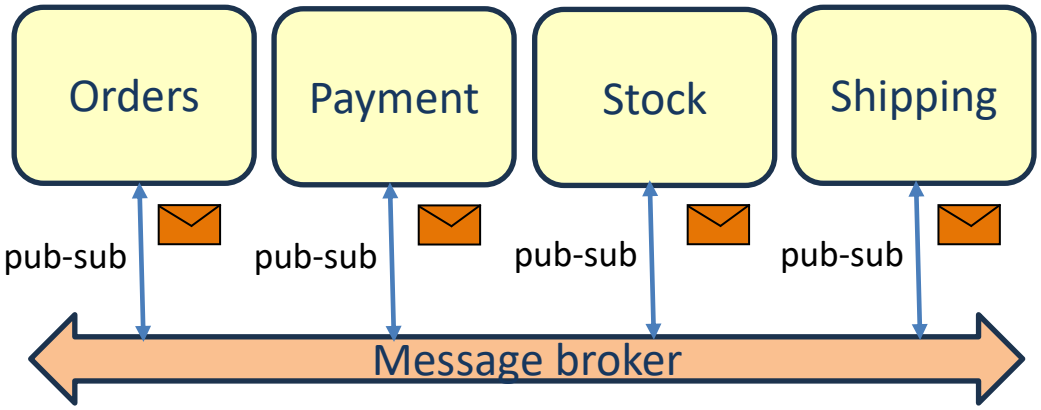


Orchestration & choreography

- Orchestration



- Choreography



Challenges of a microservice architecture

Challenge	Solution
Complex communication	Feign Registry API gateway
Performance	
Resilience	Registry replicas Load balancing between multiple service instances Circuit breaker
Security	
Transactions	Compensating transactions Eventual consistency
Keep data in sync	Publish-subscribe data change event
Keep interfaces in sync	Spring cloud contract
Keep configuration in sync	Config server
Monitor health of microservices	ELK + beats
Follow/monitor business processes	Zipkin ELK