CQRS

for strict consistency, CQRS is not good, this cause eventual consistency

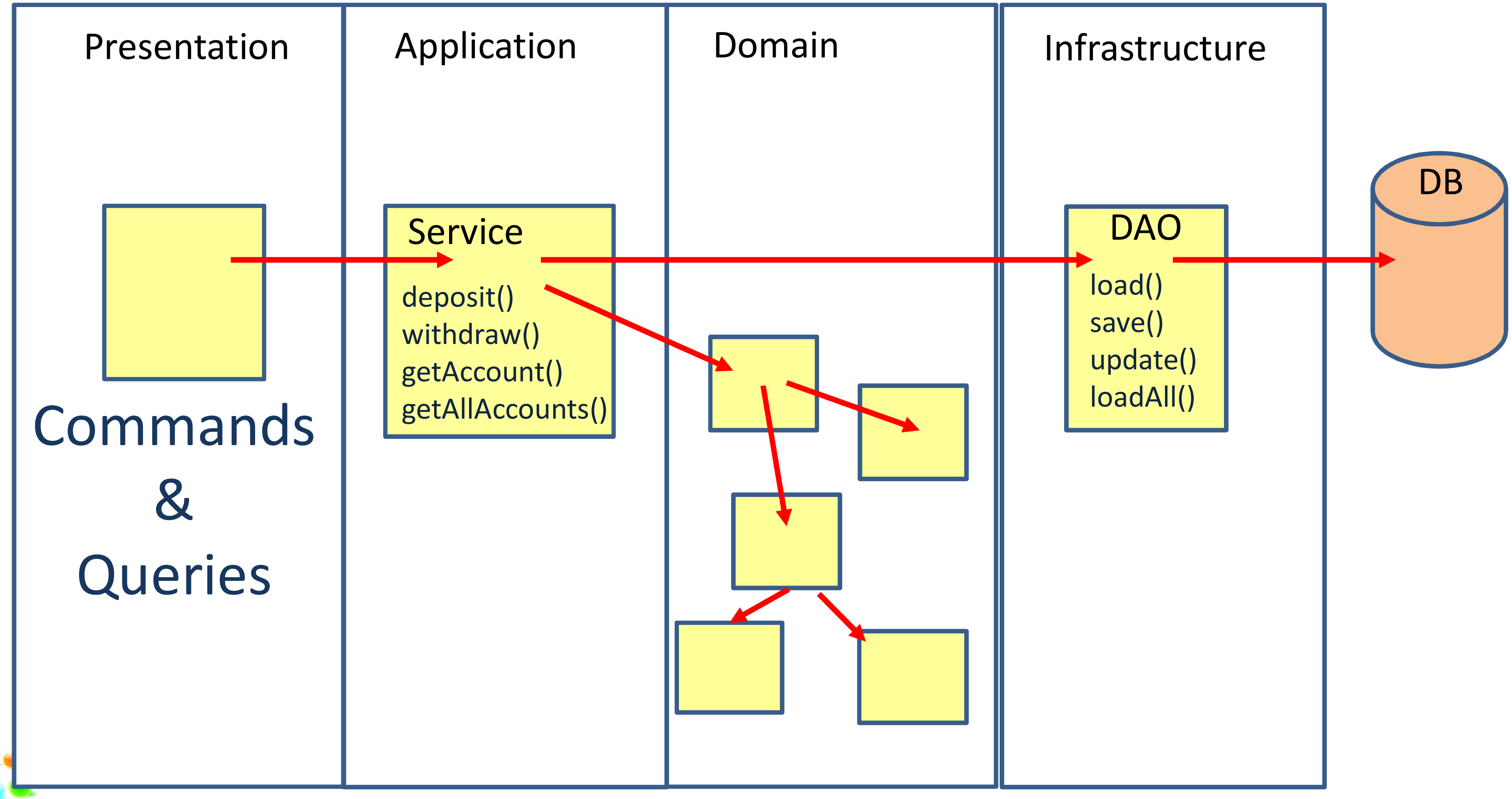
Command Query Responsibility Segregation (CQRS)

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

for strict consistency, CQRS is not good, this cause eventual consistency

Typical architecture

- One domain model that is used for commands and queries

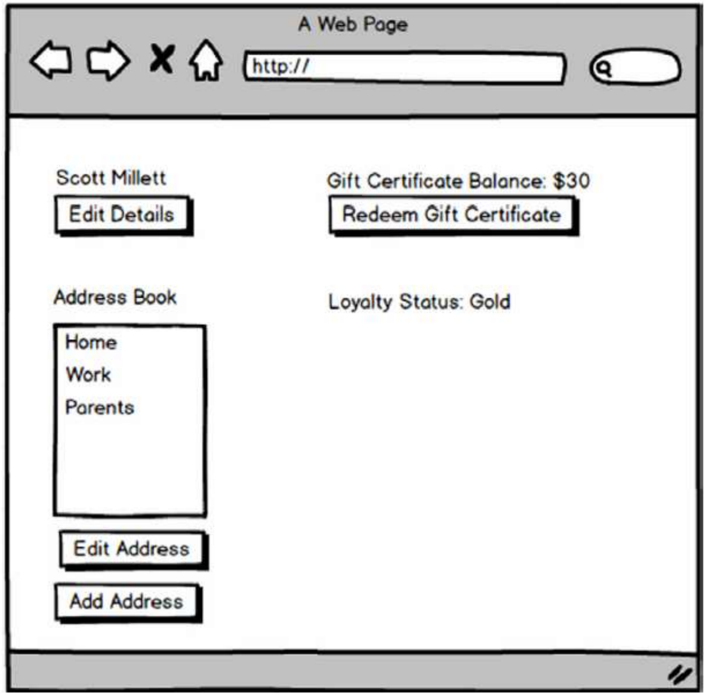


One model for both commands and queries

- 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

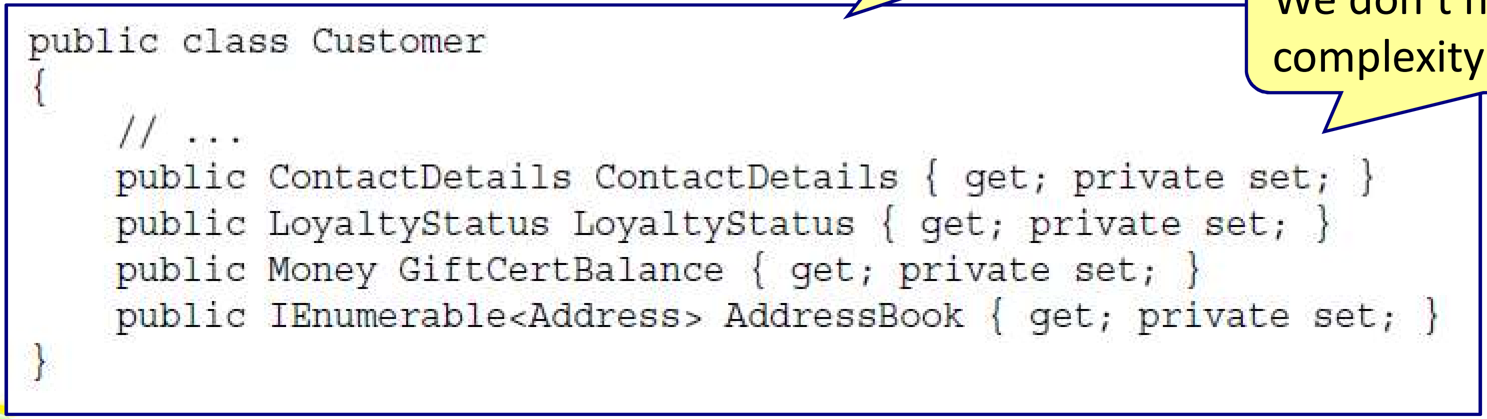
- Result: single model that is full of compromises

Example of complex aggregates



Complex aggregate because of UI needs

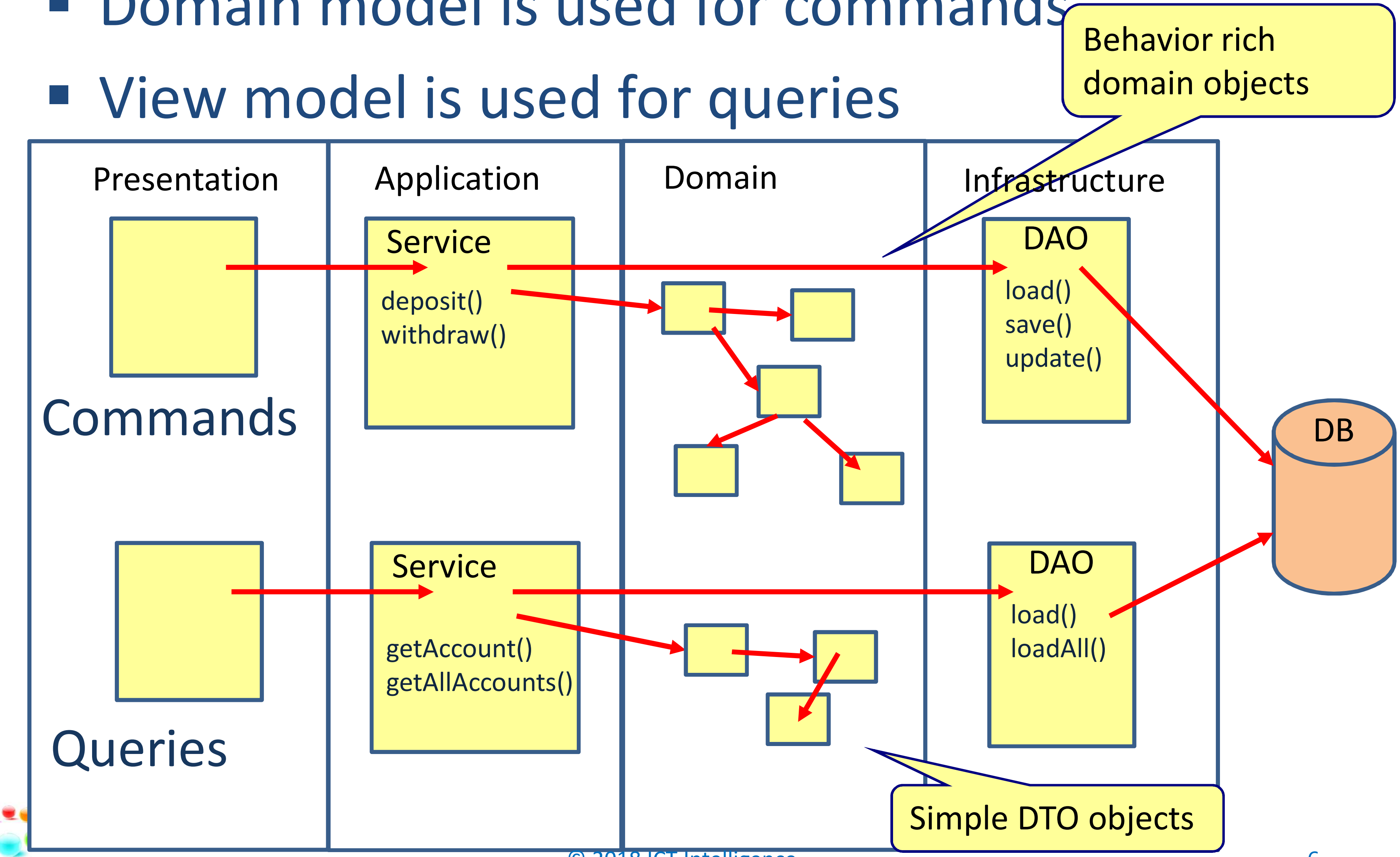
We don’t need this complexity for commands



```
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; }
}
```

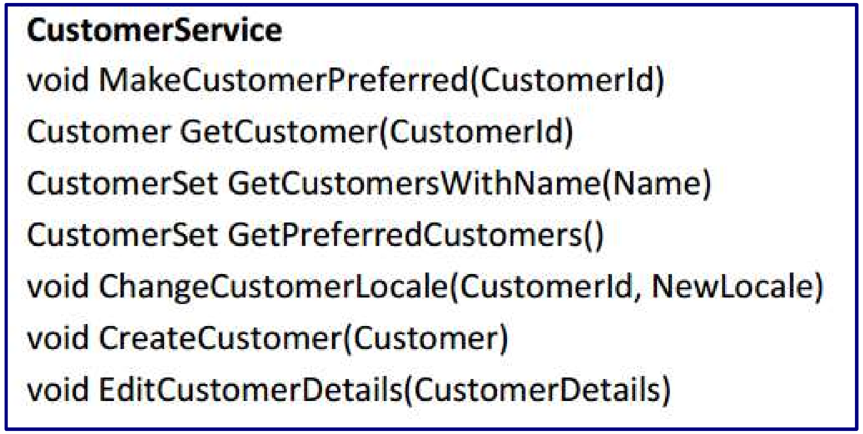
CQRS

- 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)

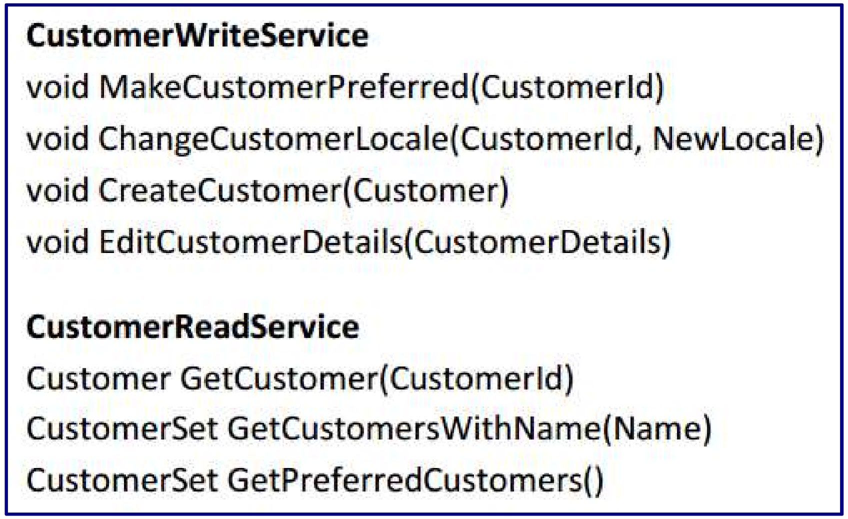
CustomerSet GetPreferredCustomers()

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)

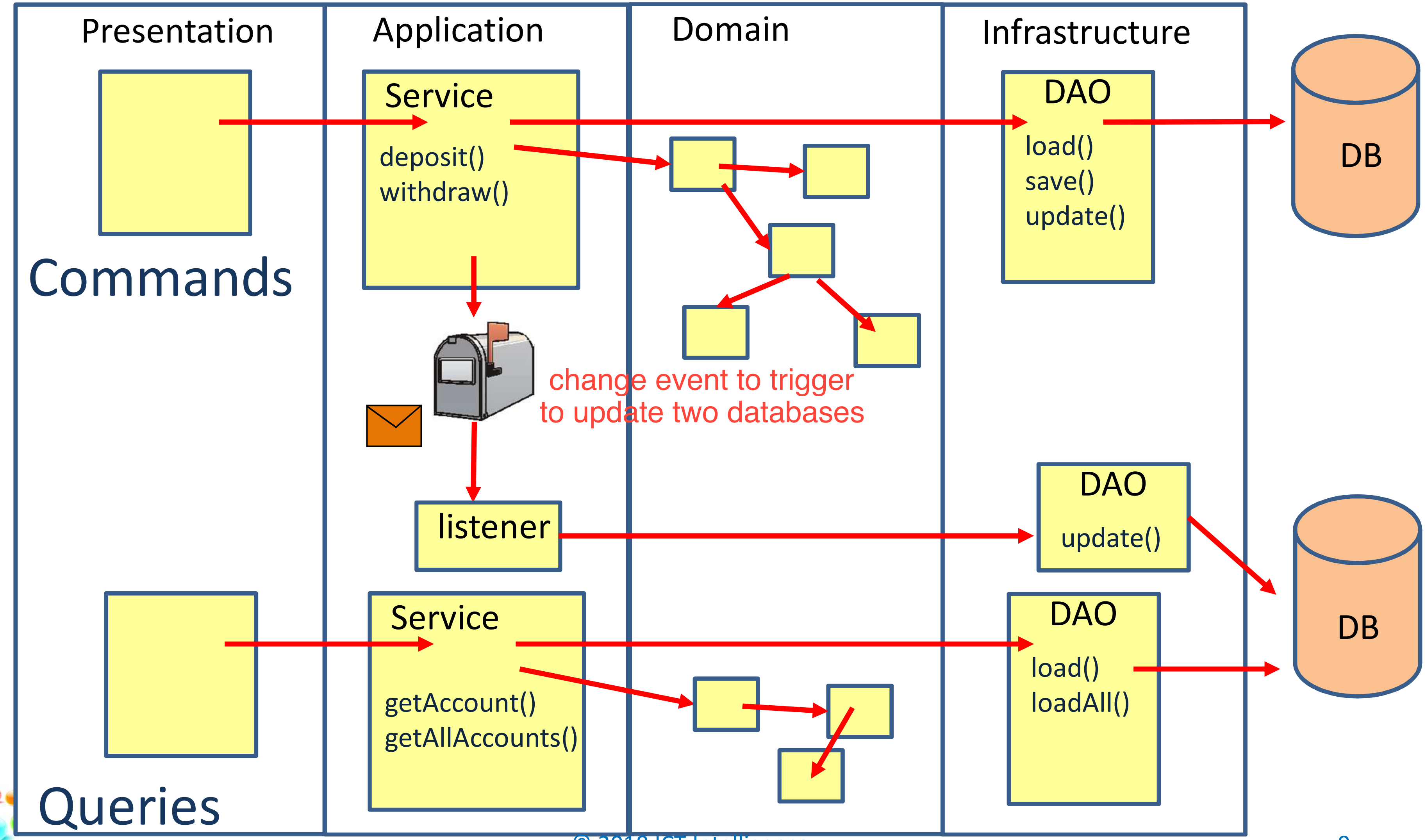
CustomerSet GetPreferredCustomers()

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 don’t happen very often. Scalability is often not important.
    - Query: queries happen very often, scalability is important

Eventual consistency

- Views will become eventual consistent



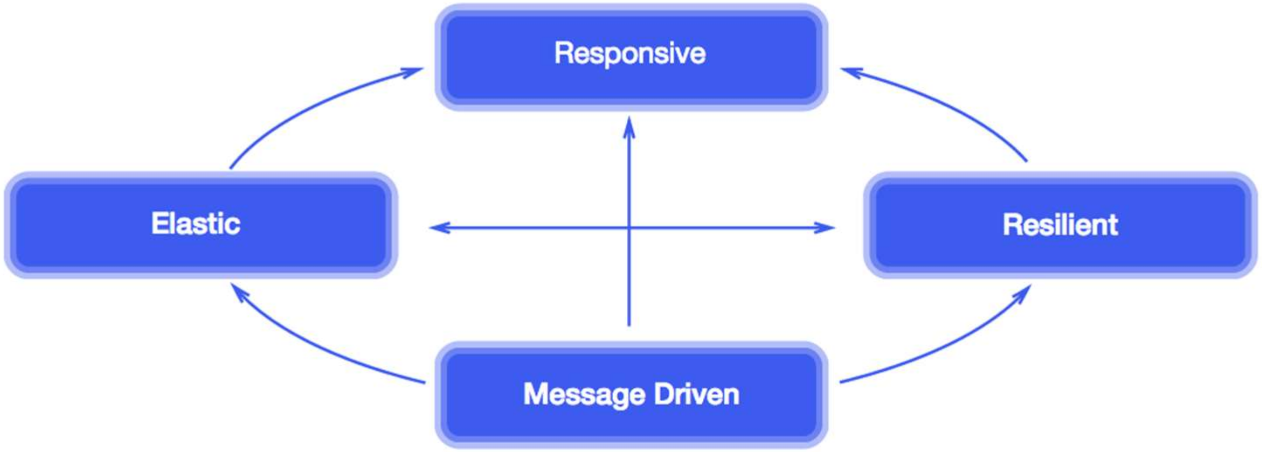
Main point

- Separation of commands and queries help us to make simpler domain models

- The more we are in tune with Laws of Nature, the more fulfillment and bliss we will experience.

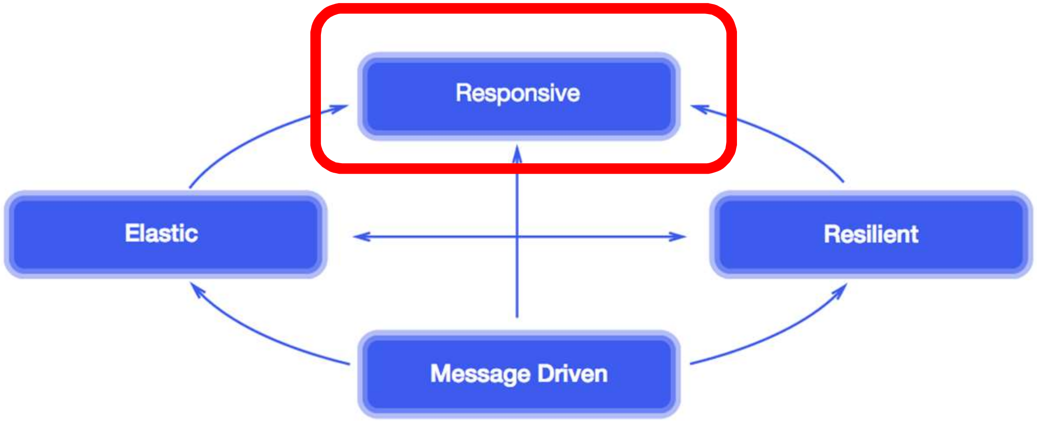
REACTIVE REST WITH SPRING WEBFLUX

Reactive applications



Responsive applications

- Reactive Streams
  - Non-blocking



- Implementations
  - JavaRx (Netflix)
  - Reactor (Pivotal)
    - Used by Spring: Spring webflux

Reactor

- Mono<T> : for handling 0 or 1 element
- Flux<T> : for handling N elements
- You can subscribe to a Mono or a Flux
  - Run some code when an object arrives in the Mono or Flux

Mono

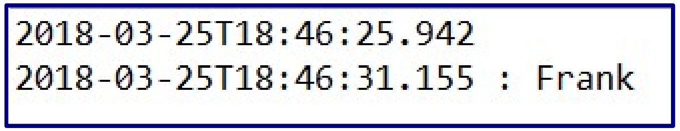
```
public class SpringReactiveClientApplication {

  public static void main(String[] args) throws InterruptedException {
    System.out.println(LocalDateTime.now());
    Mono<String> mono = Mono.just("Frank")
                            .delayElement(Duration.ofSeconds(5));

    mono.subscribe(s->printName(s));

    Thread.sleep(10000);
  }

  public static void printName(String name) {
    System.out.print(LocalDateTime.now()+" : ");
    System.out.println(name);
  }
}
```

Add the name to the mono after 5 seconds

Whenever the name arrives in the mono, print it out (Callback method)

Wait until the name has arrived in the mono

Callback method



```
2018-03-25T18:46:25.942
2018-03-25T18:46:31.155 : Frank
```

Flux

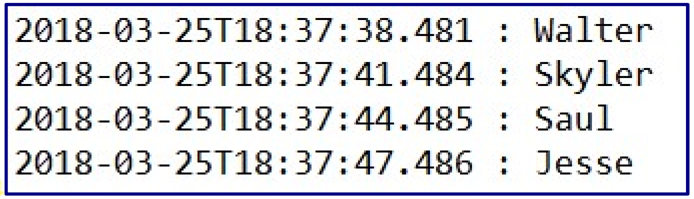
```
public class SReactiveApplication {

  public static void main(String[] args) throws InterruptedException {
    Flux<String> flux = Flux.just("Walter", "Skyler",  "Saul",  "Jesse")
                            .delayElements(Duration.ofSeconds(3));

    flux.subscribe(s->printName(s));
    Thread.sleep(15000);
  }



  public static void printName(String name) {
    System.out.print(LocalDateTime.now()+" : ");
    System.out.println(name);
  }
}
```

Add every 3 seconds a name to the flux

Whenever a name arrives in the flux, print it out (Callback method)

Wait until all names have arrived in the flux

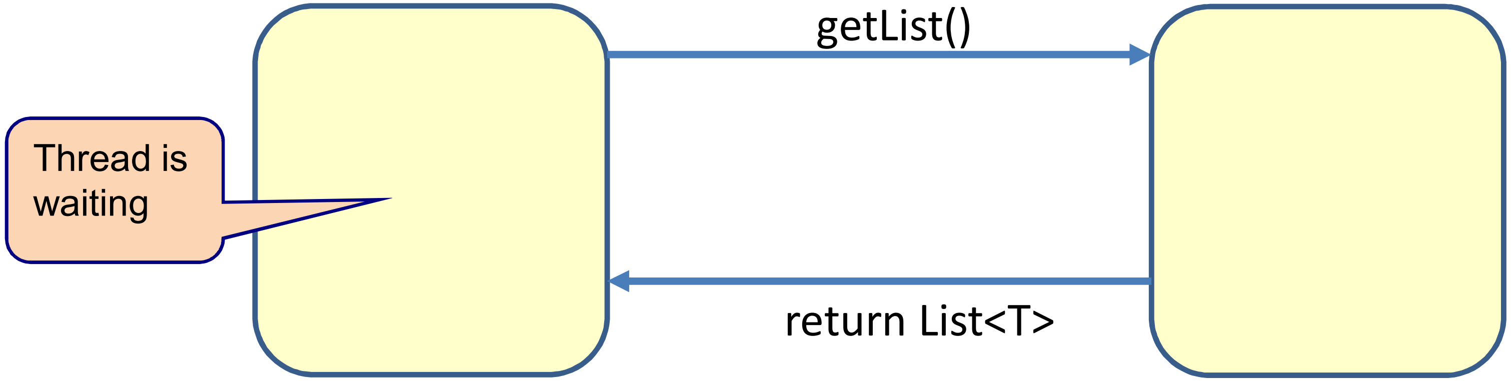
Callback method



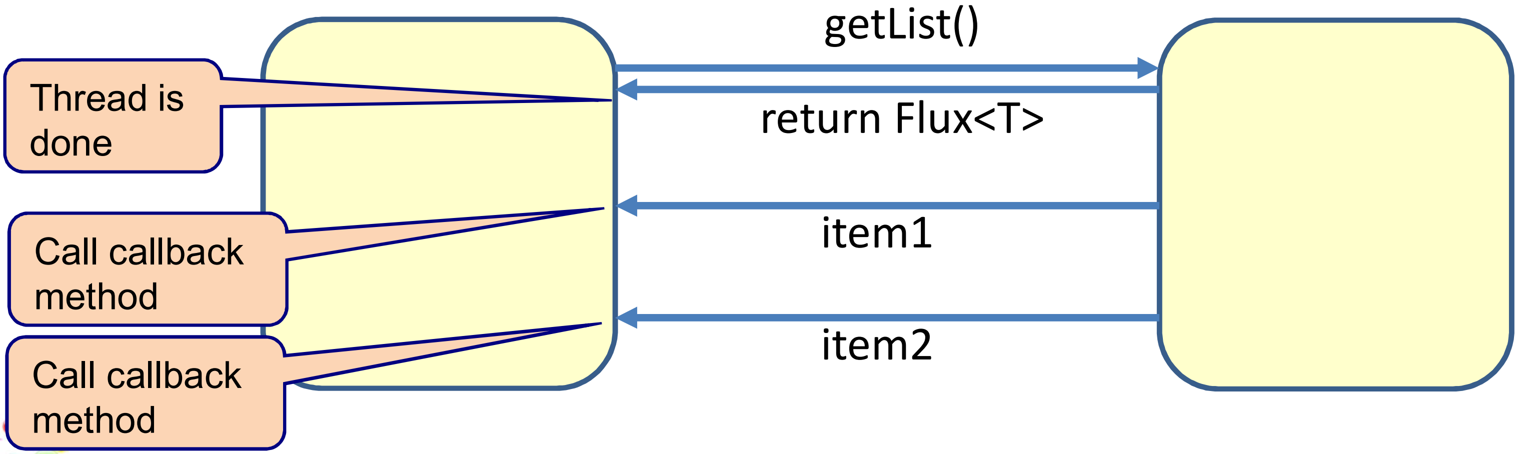
```
2018-03-25T18:37:38.481 : Walter
2018-03-25T18:37:41.484 : Skyler
2018-03-25T18:37:44.485 : Saul
2018-03-25T18:37:47.486 : Jesse
```

Imperative versus reactive

- Synchronous, blocking



- Asynchronous, non-blocking



Reactive systems

- Advantage
  - Performance
    - No need to wait till all results are available
  - Scaling
    - Less threads needed
- Disadvantage
  - The whole calling stack needs to be reactive
    - Client <->controller<->data access
  - Harder to debug

Spring WebFlux

- Allows to build reactive web(REST) applications
- Uses Netty as embedded webserver

no tomcat

Spring webflux library

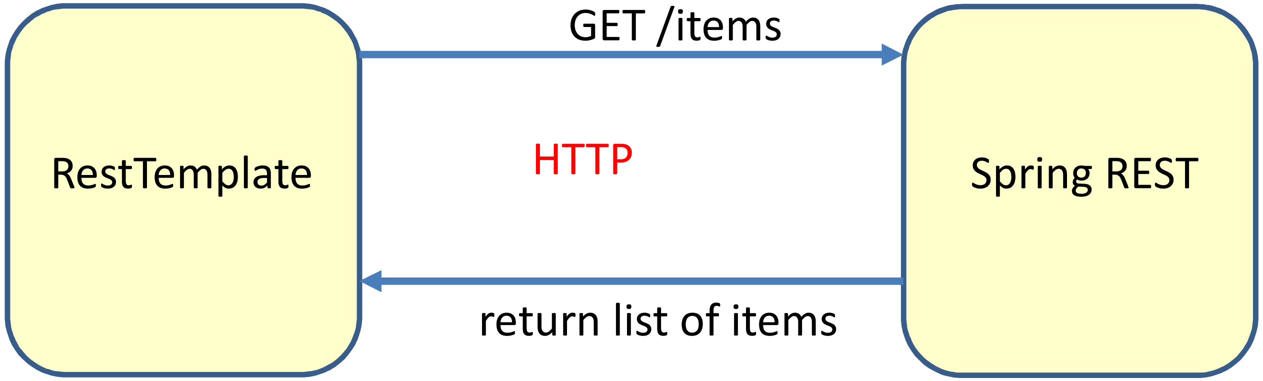
```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

inject netty instead of tomcat

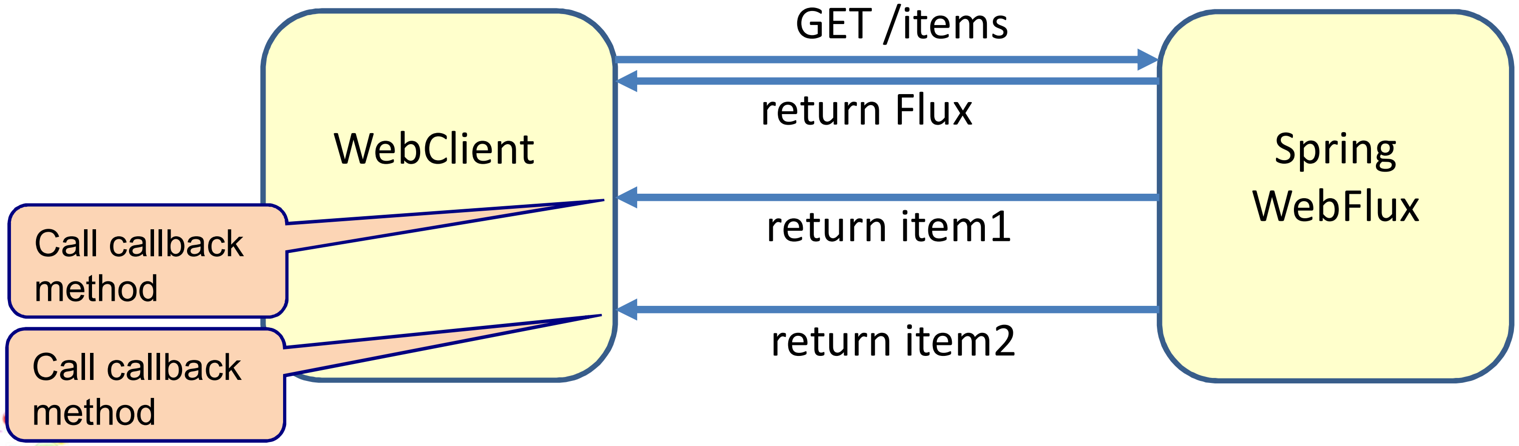
This will add the embedded Netty container which support reactive web

Reactive Web

- Synchronous, blocking



- Asynchronous, non-blocking



Reactive REST service

```
@RestController
public class CustomerController {

  @GetMapping(value="/customers", produces=MediaType.TEXT_EVENT_STREAM_VALUE)
  public Flux<Customer> getAllCustomers() {
    Flux<Customer> customerFlux = Flux.just(
      new Customer(new Long(1), "Walter", "White", 29),
      new Customer(new Long(2), "Skyler", "White", 24),
      new Customer(new Long(3), "Saul", "Goodman", 27),
      new Customer(new Long(4), "Jesse", "Pinkman", 24)
    ).delayElements(Duration.ofSeconds(3));
    return customerFlux;
  }
}
```

Generate a new Customer every 3 seconds

```
public class Customer {

  private long custId;
  private String firstname;
  private String lastname;
  private int age;
  ...
}
```

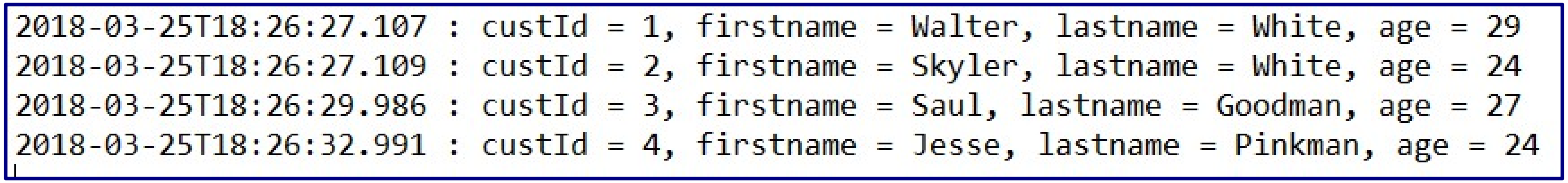
```
@SpringBootApplication
public class SpringReactiveApplication{

  public static void main(String[] args) {
    SpringApplication.run(SpringReactiveApplication.class, args);
  }
}
```

Reactive REST Client

```java
@SpringBootApplication
public class ClientApplication {

  public static void main(String[] args) throws InterruptedException{
    Flux<Customer> result = WebClient.create("http://localhost:8080/customers")
      .get()
      .retrieve()
      .bodyToFlux(Customer.class);
    result.subscribe(s->{
      System.out.print(LocalDateTime.now()+" : ");
      System.out.println(s);
    });

    Thread.sleep(15000);
  }
}
```

Print the customer when it arrives

Wait for all Customers to arrive



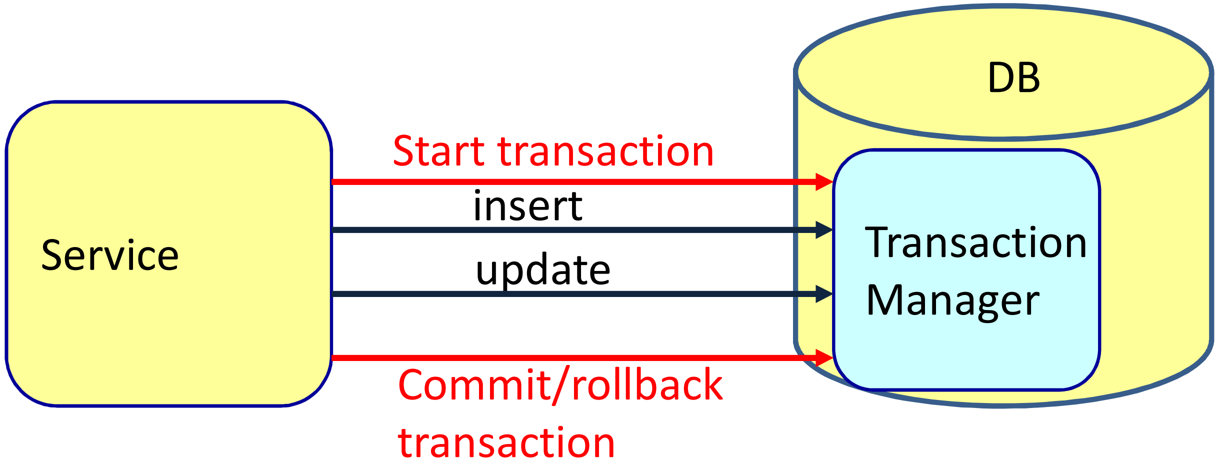
```
2018-03-25T18:26:27.107 : custId = 1, firstname = Walter, lastname = White, age = 29
2018-03-25T18:26:27.109 : custId = 2, firstname = Skyler, lastname = White, age = 24
2018-03-25T18:26:29.986 : custId = 3, firstname = Saul, lastname = Goodman, age = 27
2018-03-25T18:26:32.991 : custId = 4, firstname = Jesse, lastname = Pinkman, age = 24
```

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

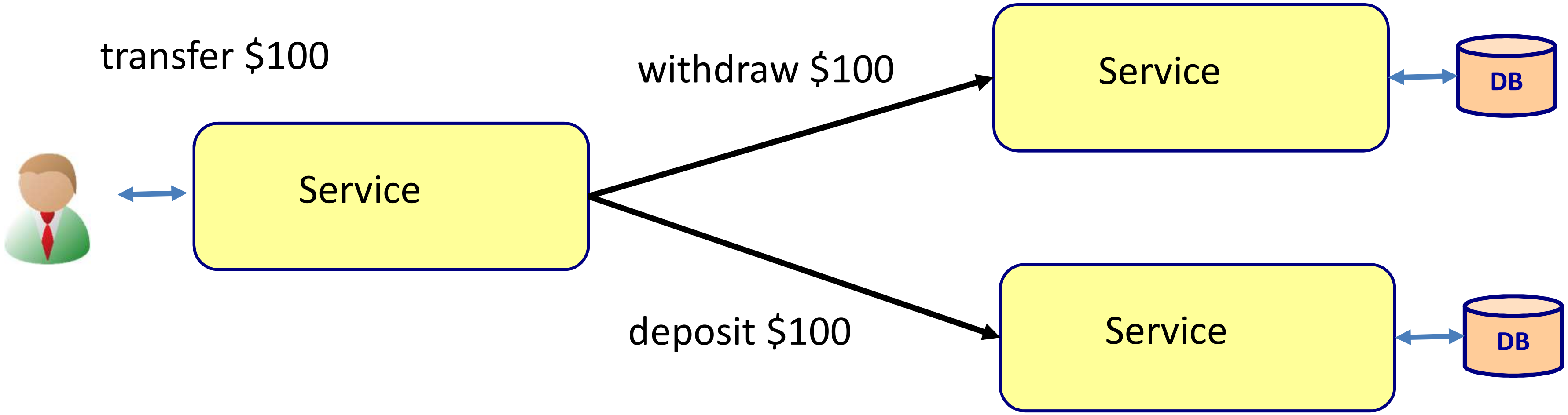
Local transaction



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

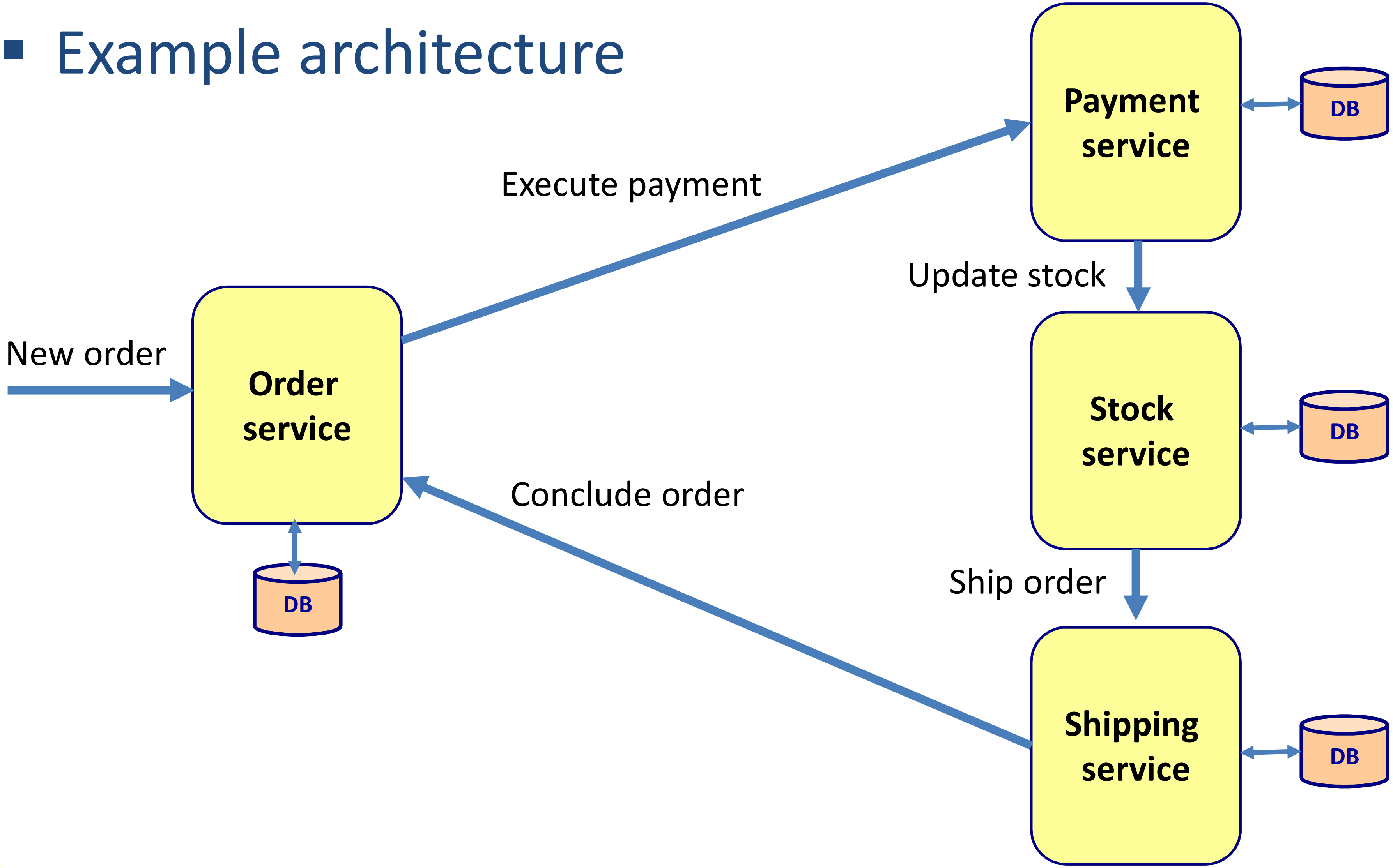
Transactions in a microservice architecture

- Distributed transactions

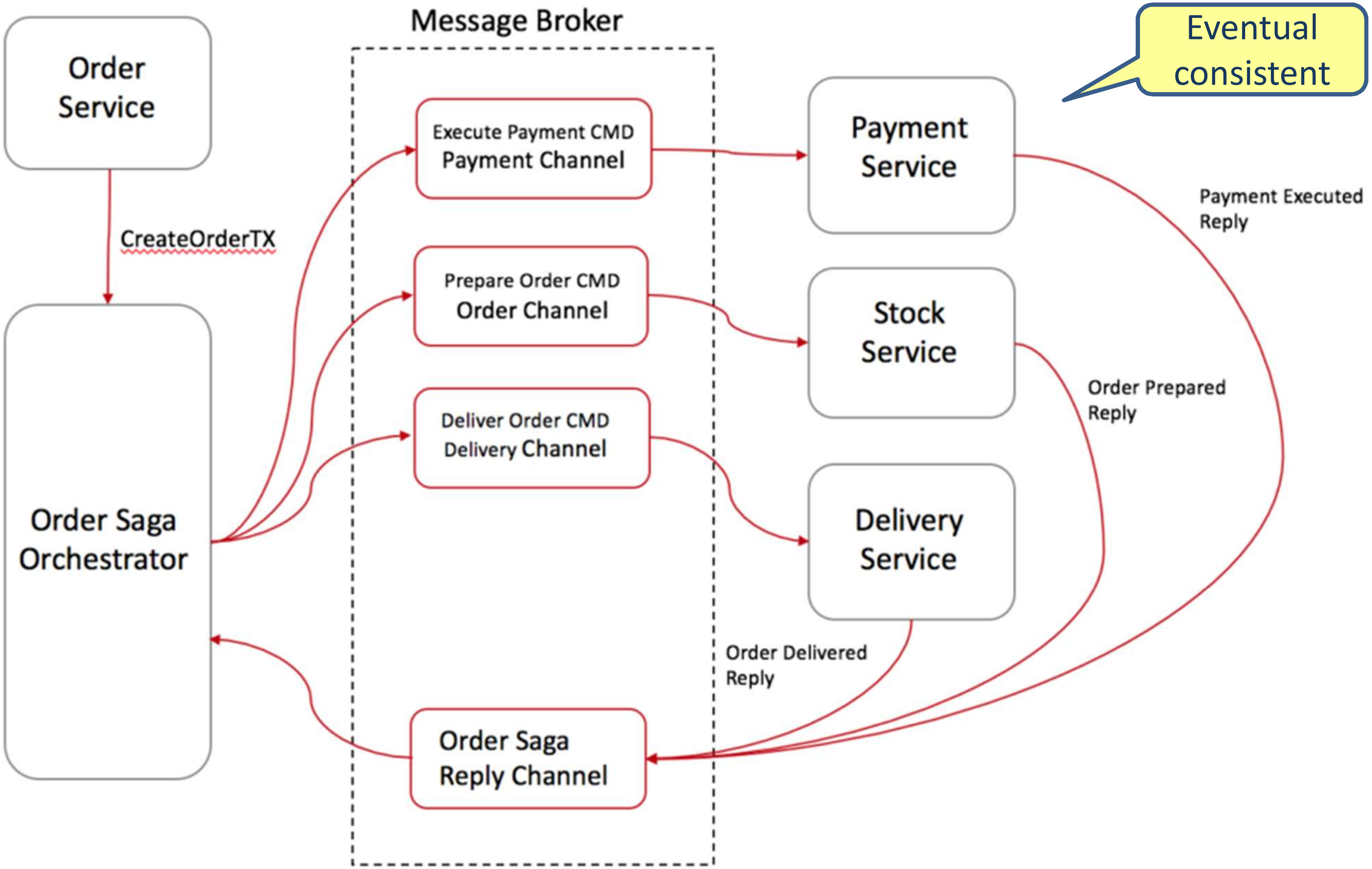


Saga pattern

- Example architecture



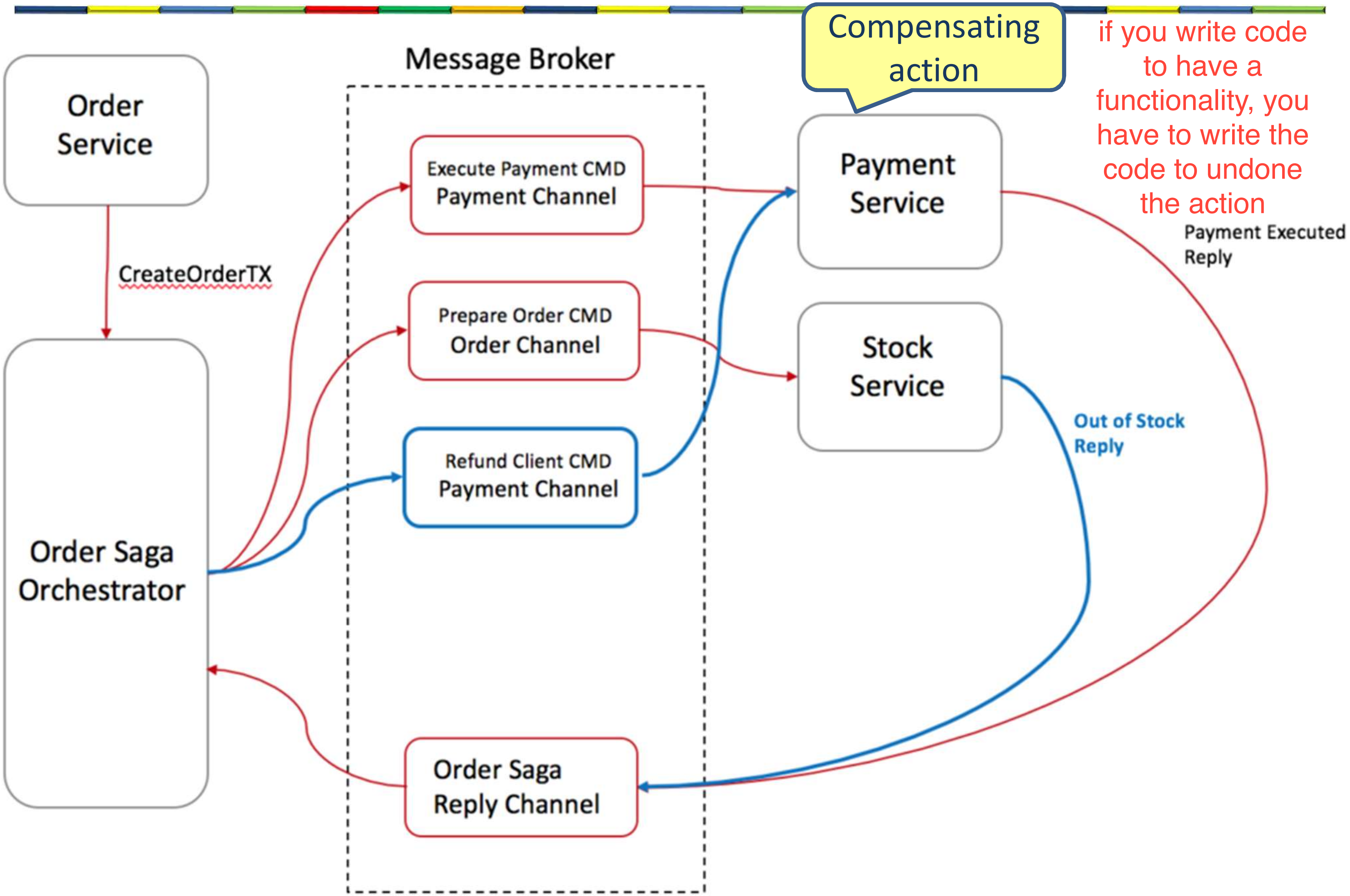
Saga with command/orchestration



Eventual consistent

eventual consistency

Saga with command/orchestration



if you write code to have a functionality, you have to write the code to undone the action

Saga with events/choreography

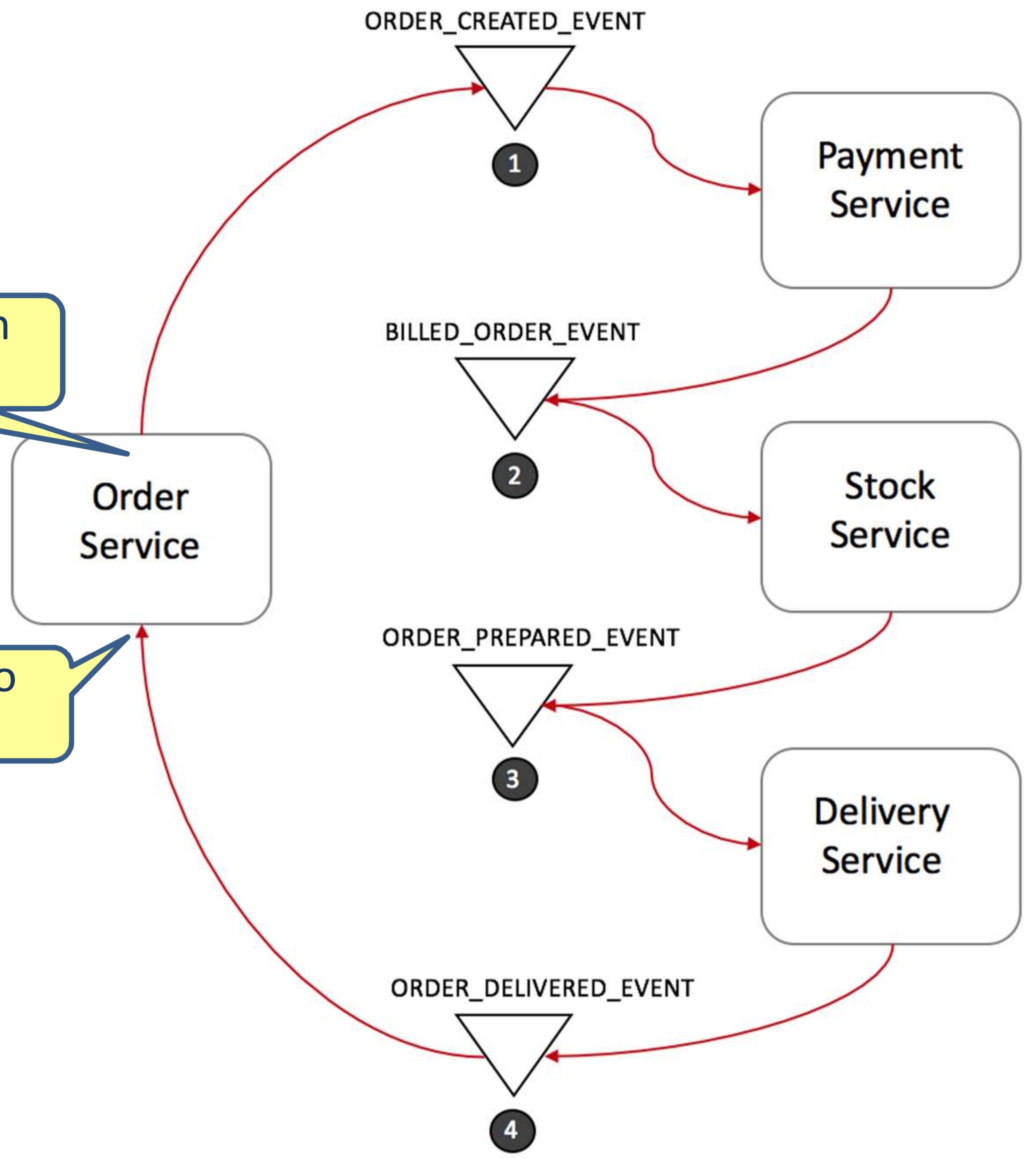
- Advantages
  - More control over transaction
  - Easier to monitor transaction
  - No cyclic dependencies

work better in simple cases

- Disadvantages
  - Need an extra orchestrator service
  - Orchestrator becomes too complex

Saga is way to manage data consistency across microservices in distributed transactio scenarios

Saga with events/choreography

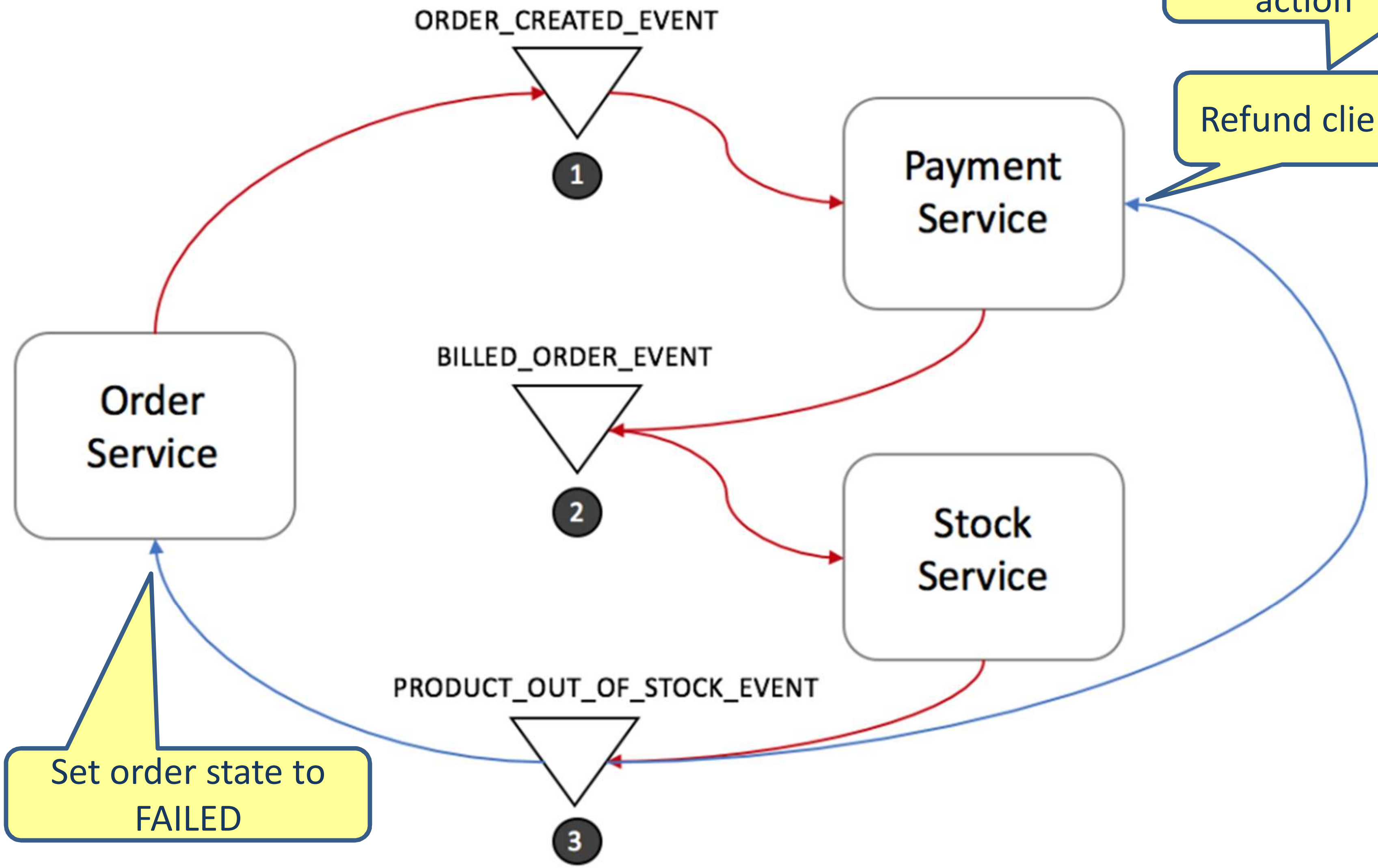


Create order with state PENDING

Set order state to DELIVERED

Eventual consistent

Saga with events/choreography



Compensating action

Refund client

Set order state to FAILED

Saga with events/choreography

- Advantages
  - Simple
  - Loosely coupled services

work better in complex cases

- Disadvantages
  - Difficult to track who listen to which events
  - Possibility of cyclic events

Main point

- Distributed transactions can only be well implemented with eventual consistency

- On the level of the unified field everything is orderly and consistent.

no use two phase commit in microservices

Connecting the parts of knowledge with the wholeness of knowledge

1. CQRS helps in optimizing microservices with regard to scalability, data storage and consistency .
2. Always try to implement transactions within on service, unless you have a good reason not to

3. **Transcendental consciousness** is a field of complete simplicity and all possibilities.
4. **Wholeness moving within itself:** In Unity Consciousness, one lives a life full of bliss, and maximum efficiency with least effort.