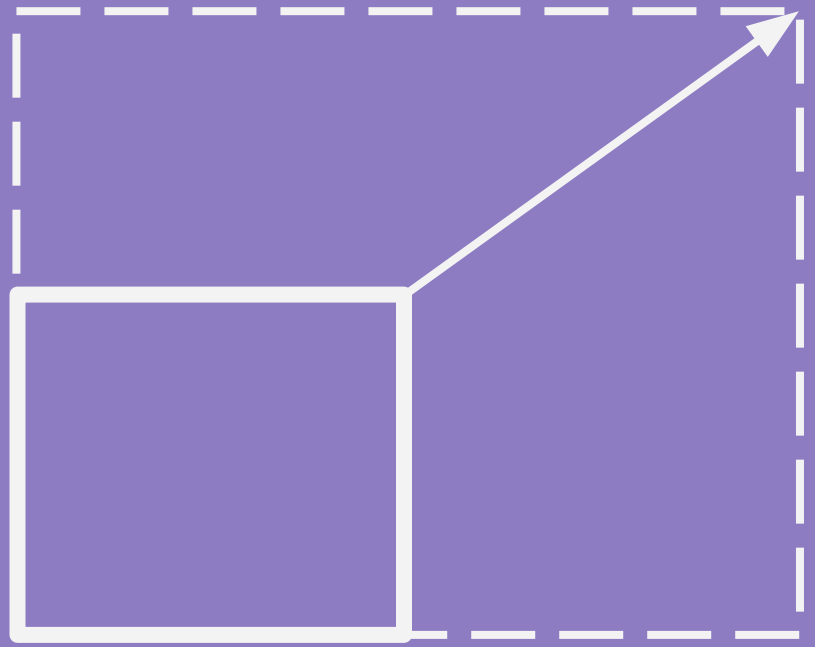
- Po **10.30 drzwi do budynku będą zamknięte** i bez identyfikatora nie będzie możliwości żeby dostać się do środka.
- Około 13.00 przerwa na pizzę.
- 15.30 koniec szkolenia.

Scalability in Event Driven Applications

Piotr Ceranek

1. Scalability
2. Architecture
3. Problem
4. CQRS
5. ES
6. Saga
7. Sources
8. Workshops

SCALABILITY



amazon

Azure

NETFLIX

Google

CLOUDFLARE®

amazon

Azure

NETFLIX

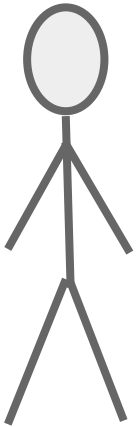
Google

CLOUDFLARE®

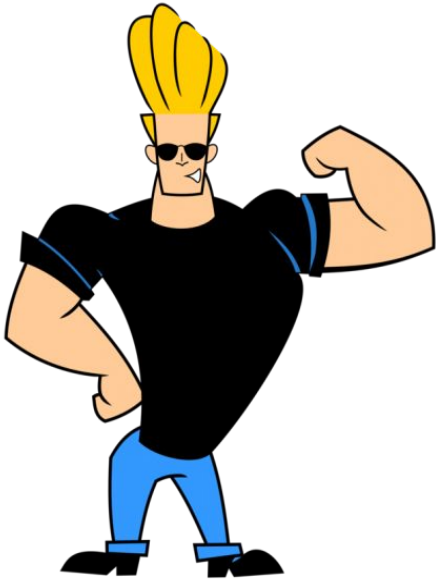
Application scalability is the ability of ...

Application scalability is the ability of a system, network, or process to handle a growing amount of work, or its potential to be enlarged to accommodate that growth.

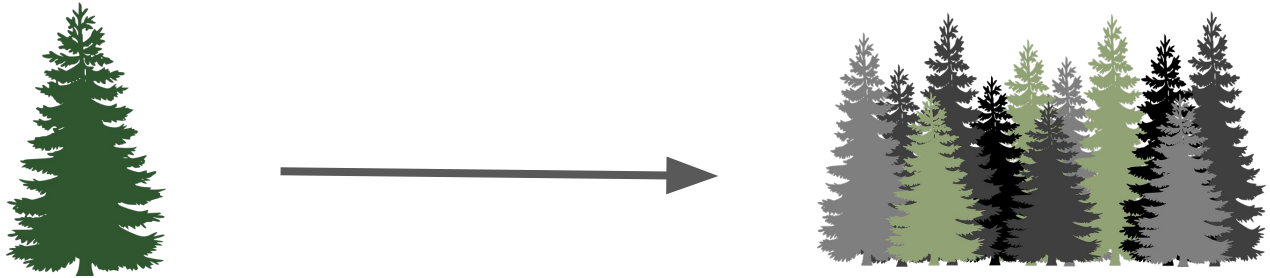
André B. Bondi



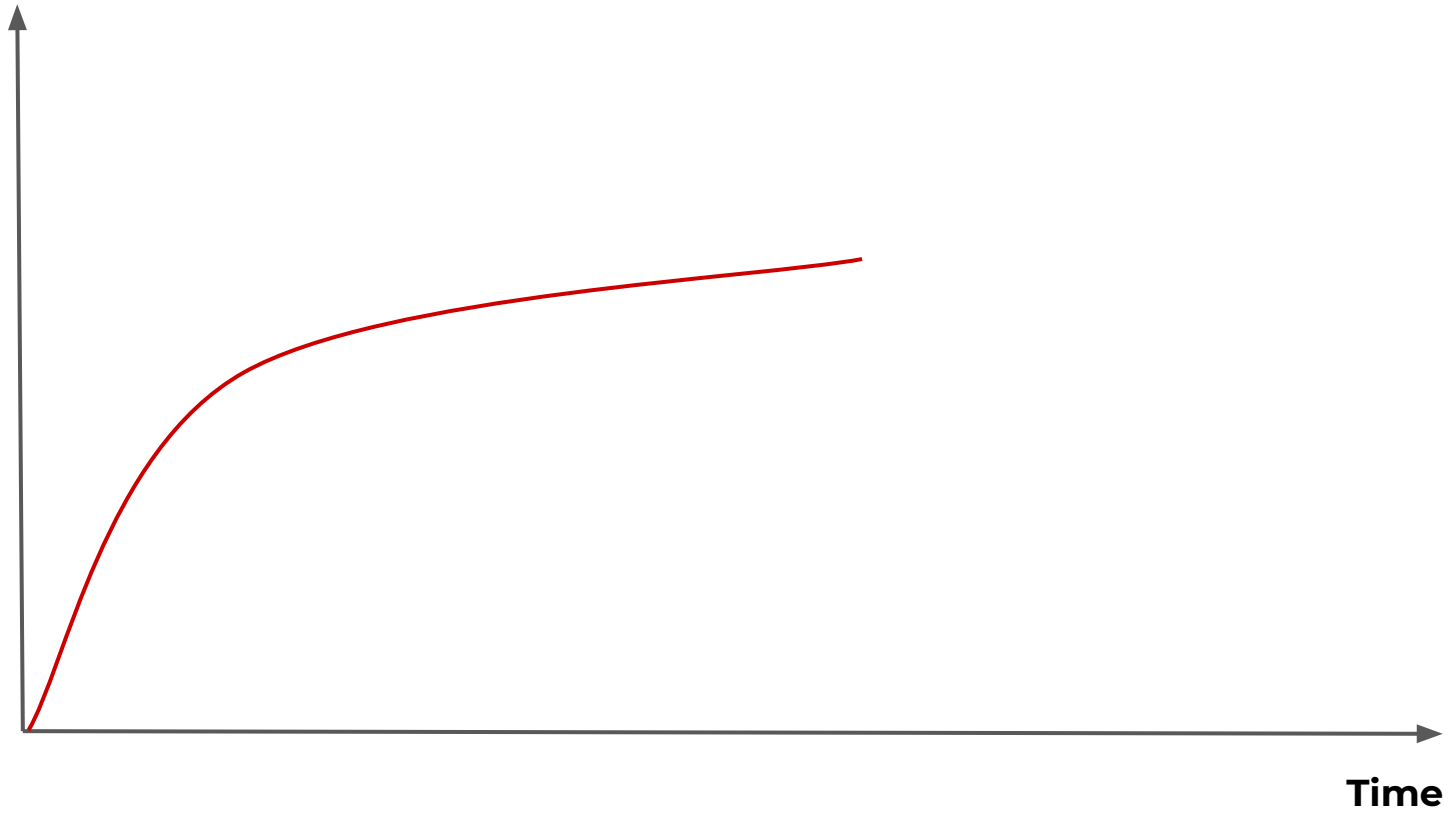
Scaling up consists in adding more computational power to the instance for example the RAM memory or the next generation processor.

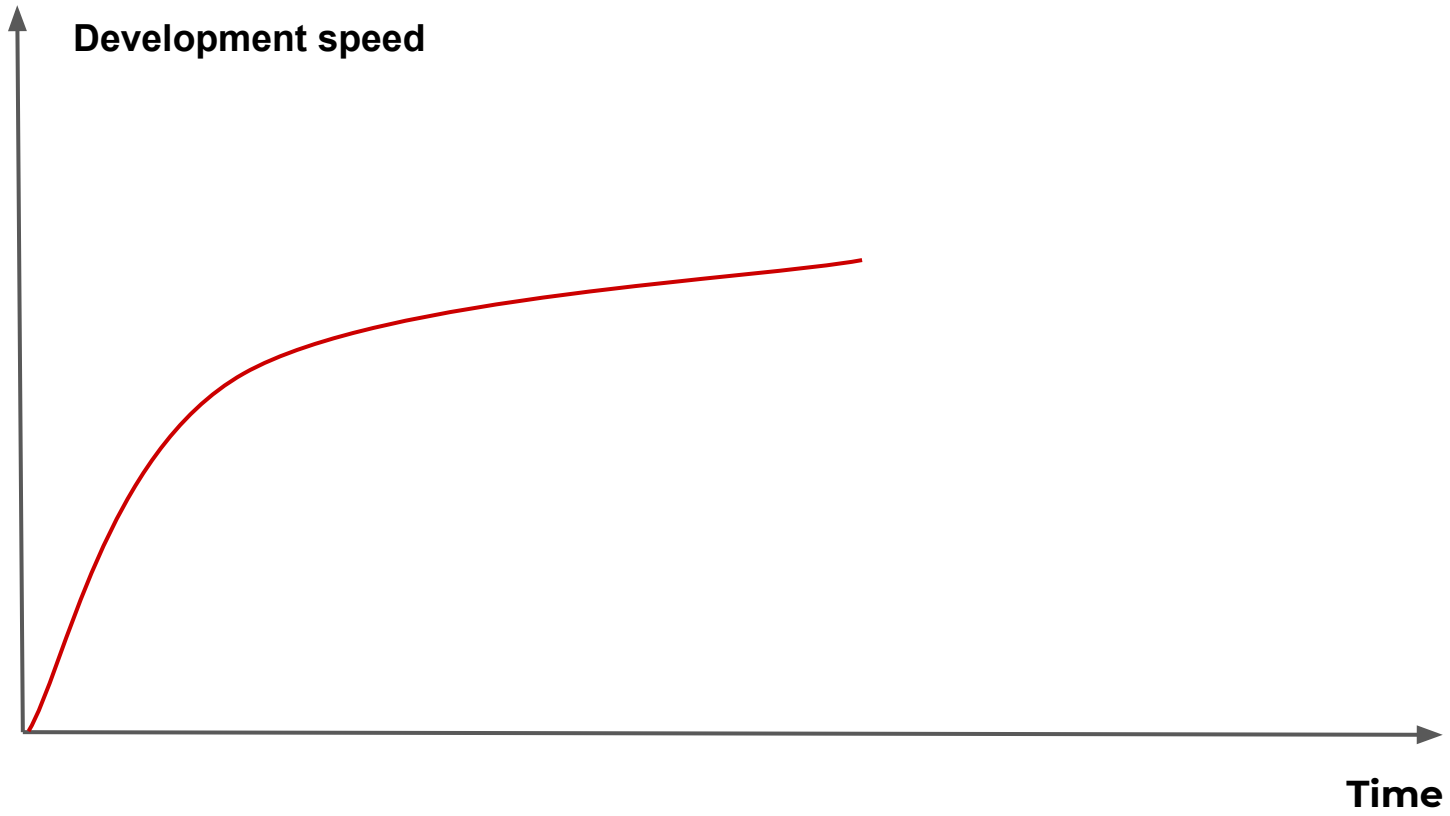


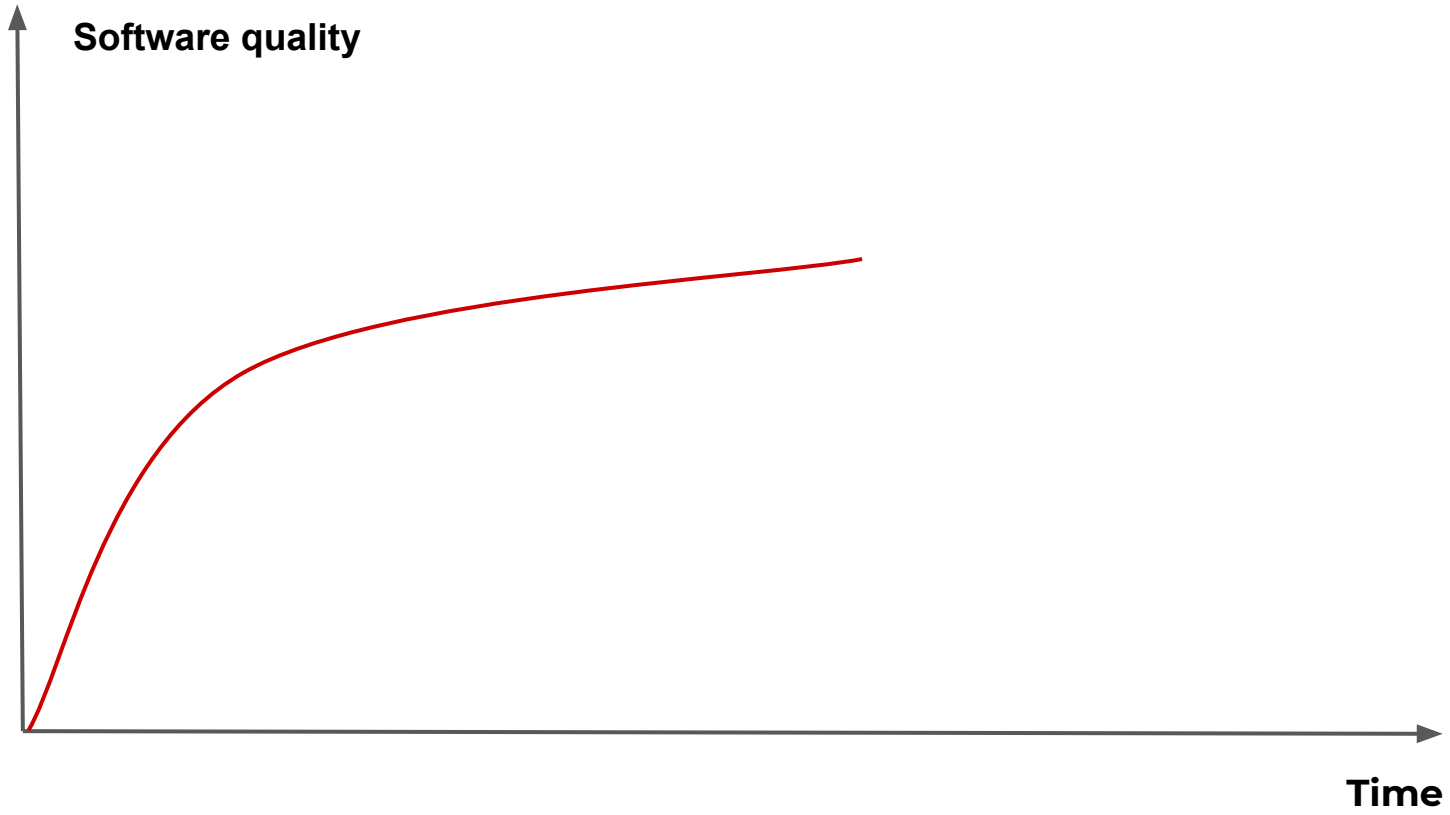
Scaling out consists in adding more nodes to a system. For example making distributed system from one node application.

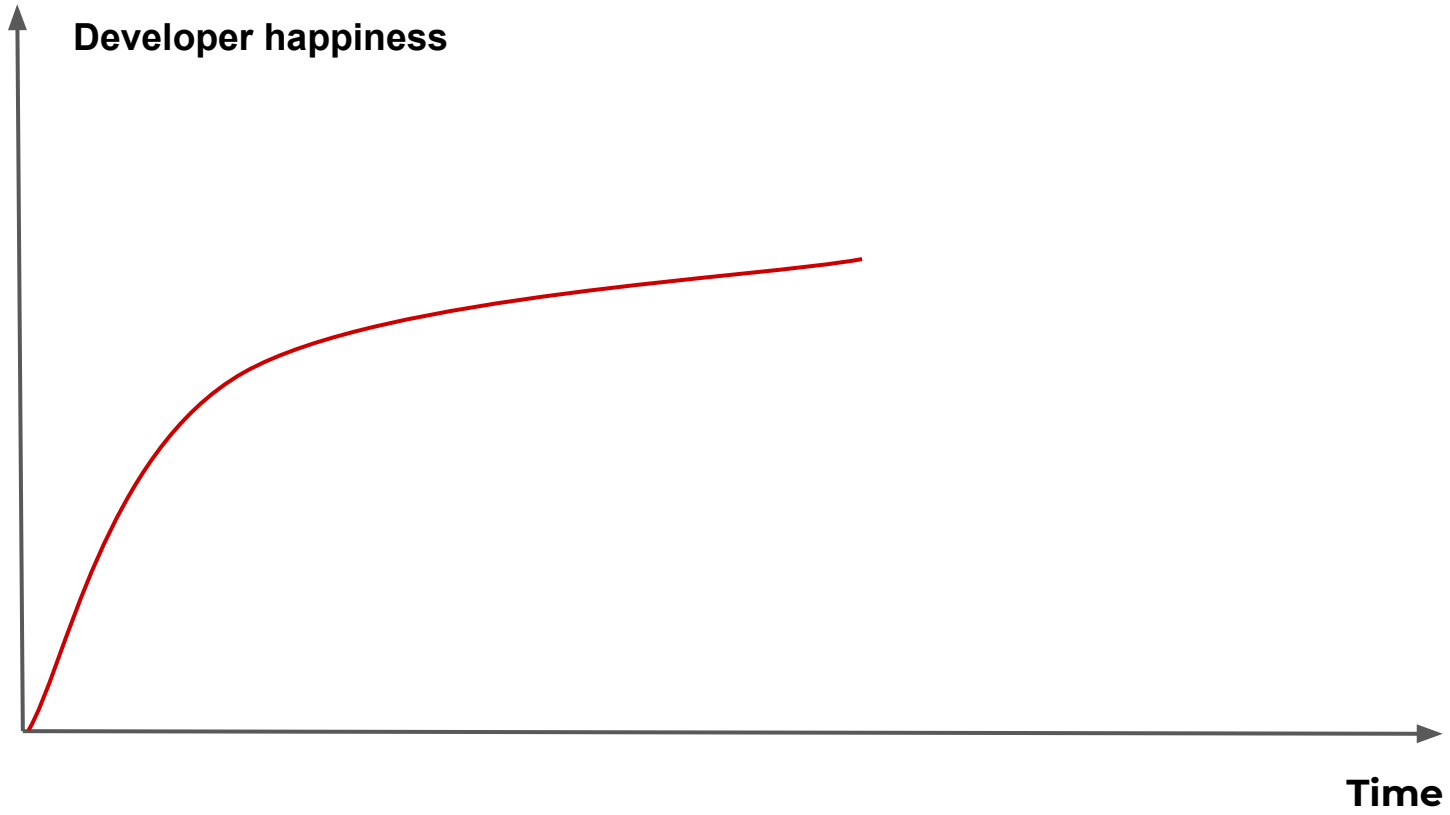


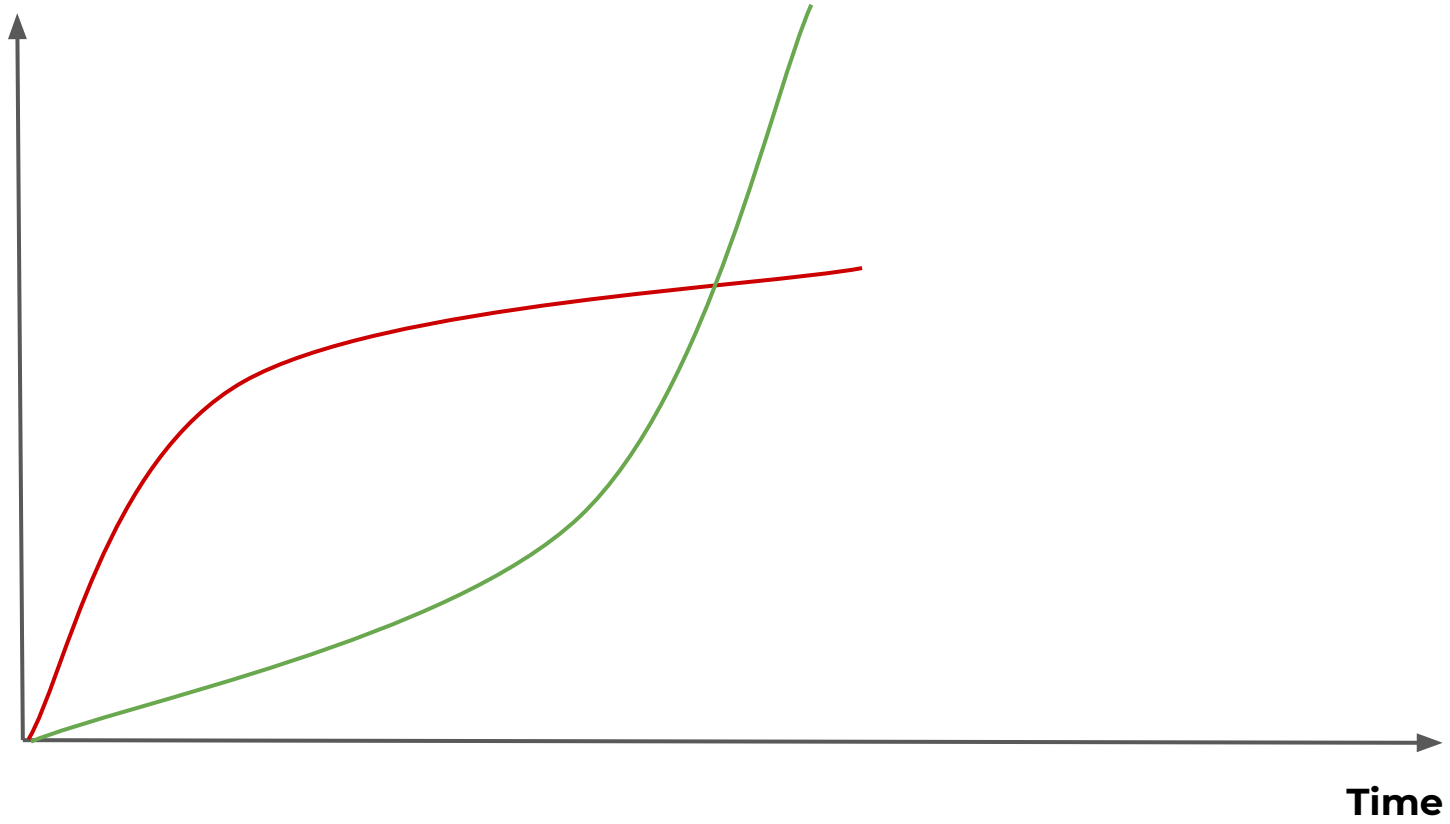

Our system is not efficient. What can we do about it?
Better CPU
More RAM
We should design better architecture


Time


Development speed
Time


Software quality
Time


Developer happiness
Time


Time

Architecture

It can both promise more than it can deliver and deliver more than it promises.

The goal of software architecture is to minimize the human resources required to build and maintain the required system.

The only way to go fast is to go well.

Robert C. Martin

Architecture represents the significant design decisions that shape a system, where significant is measured by **cost of change**.

Grady Booch

Architecture is the decision that you wish you could get right early in a project but that you are not necessarily more likely to get them right than any other.

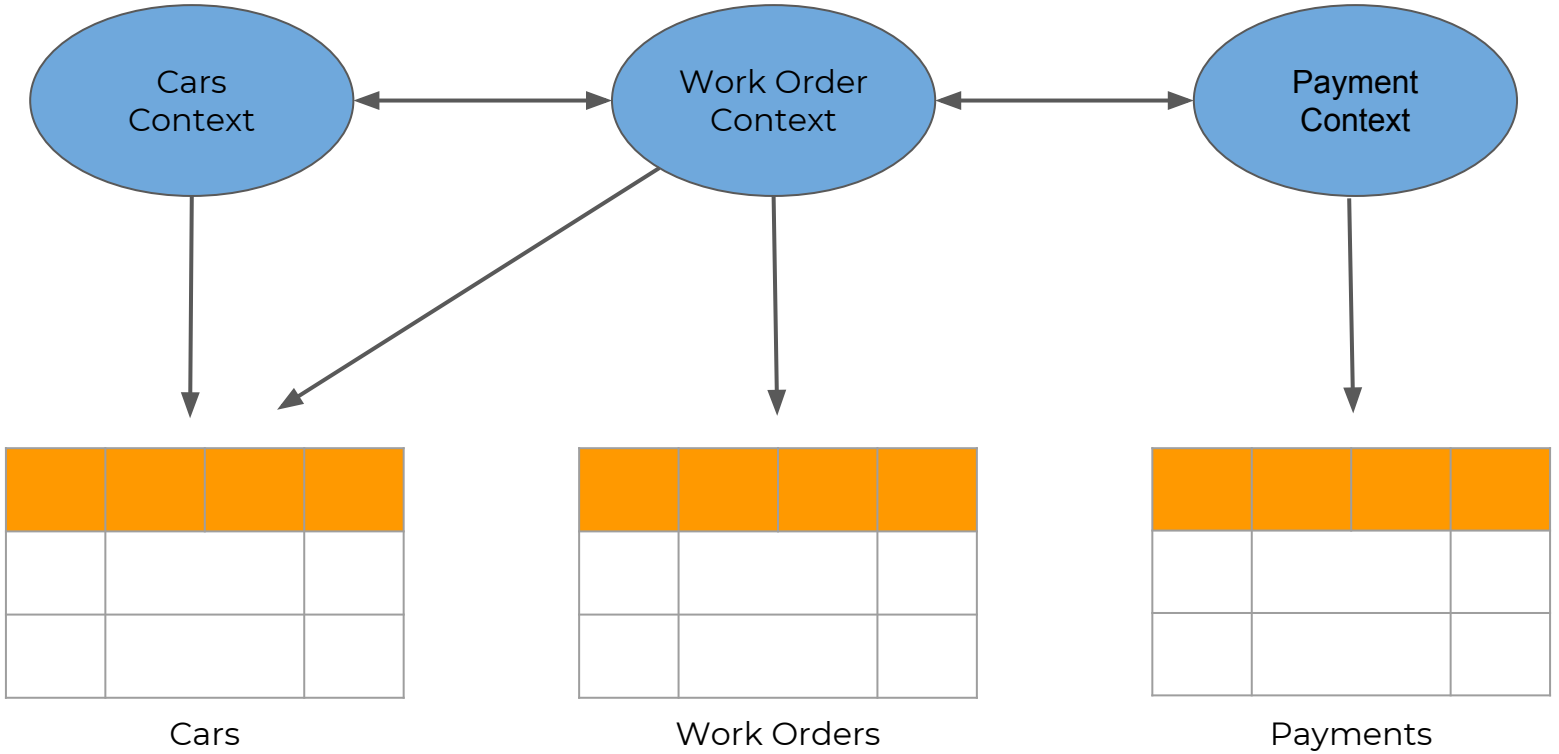
Ralph Jhonson

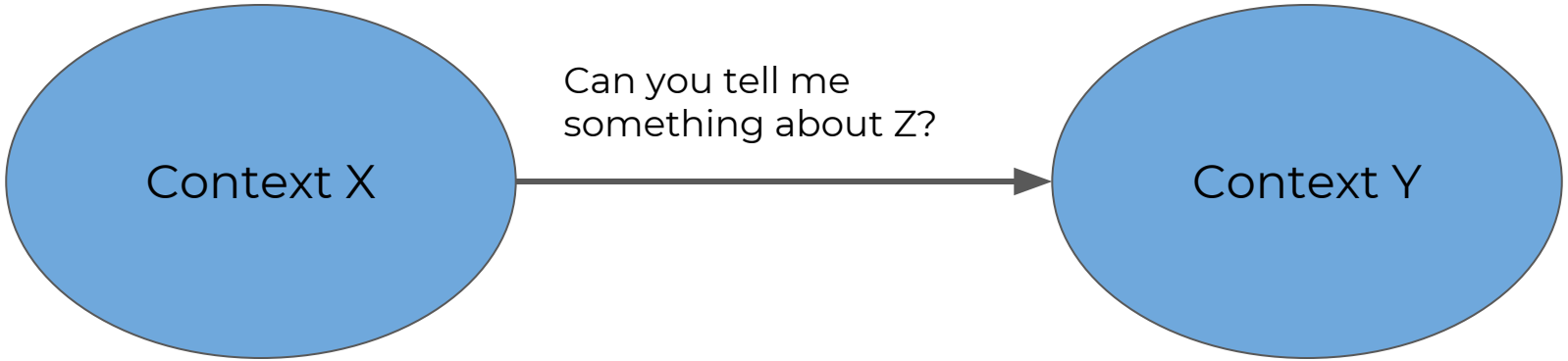
Layered Architecture
Microservices Architecture
Event-driven Architecture
Serverless Architecture
Clean Architecture
Hexagonal Architecture
Monolith Architecture
...

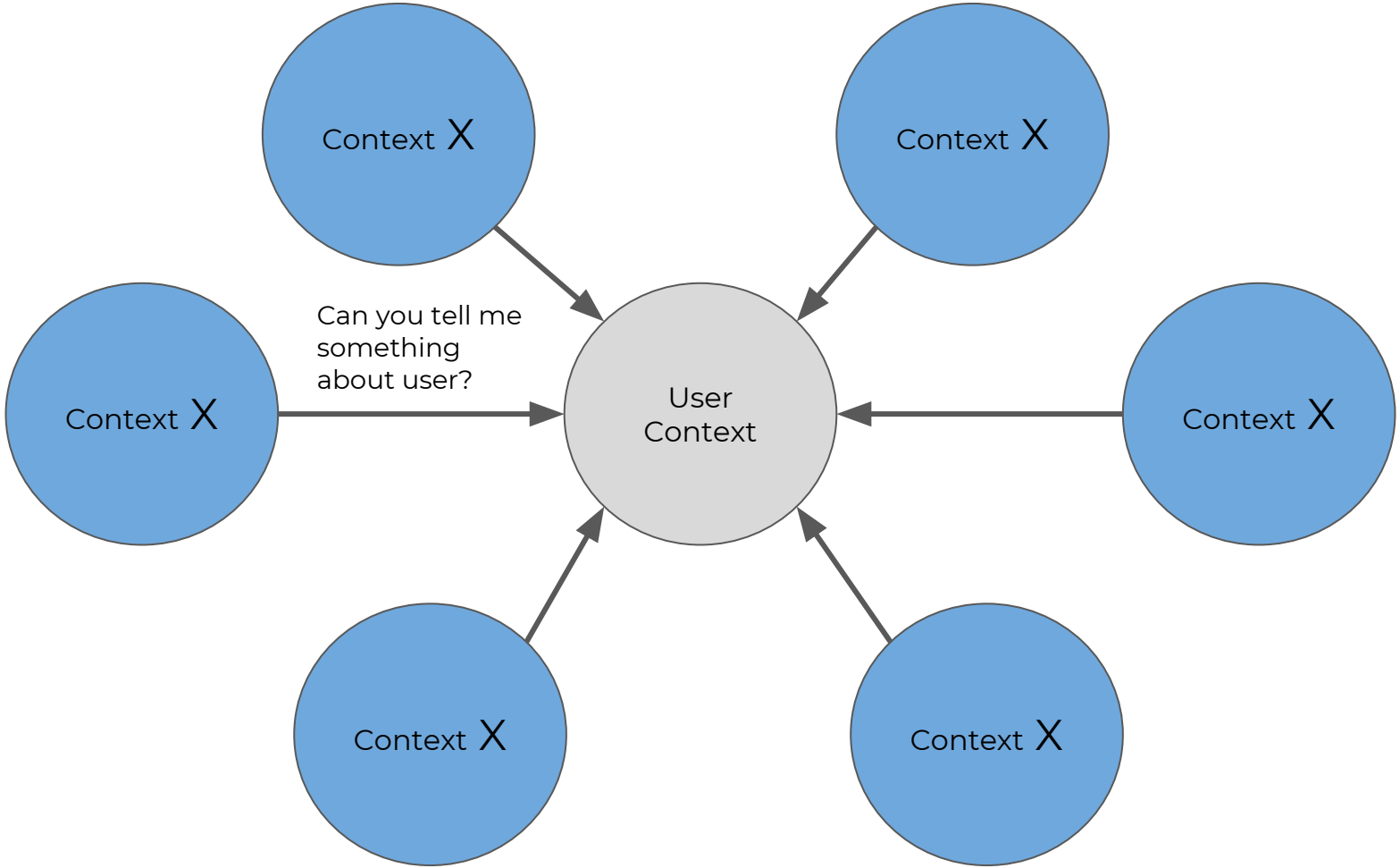
Problem

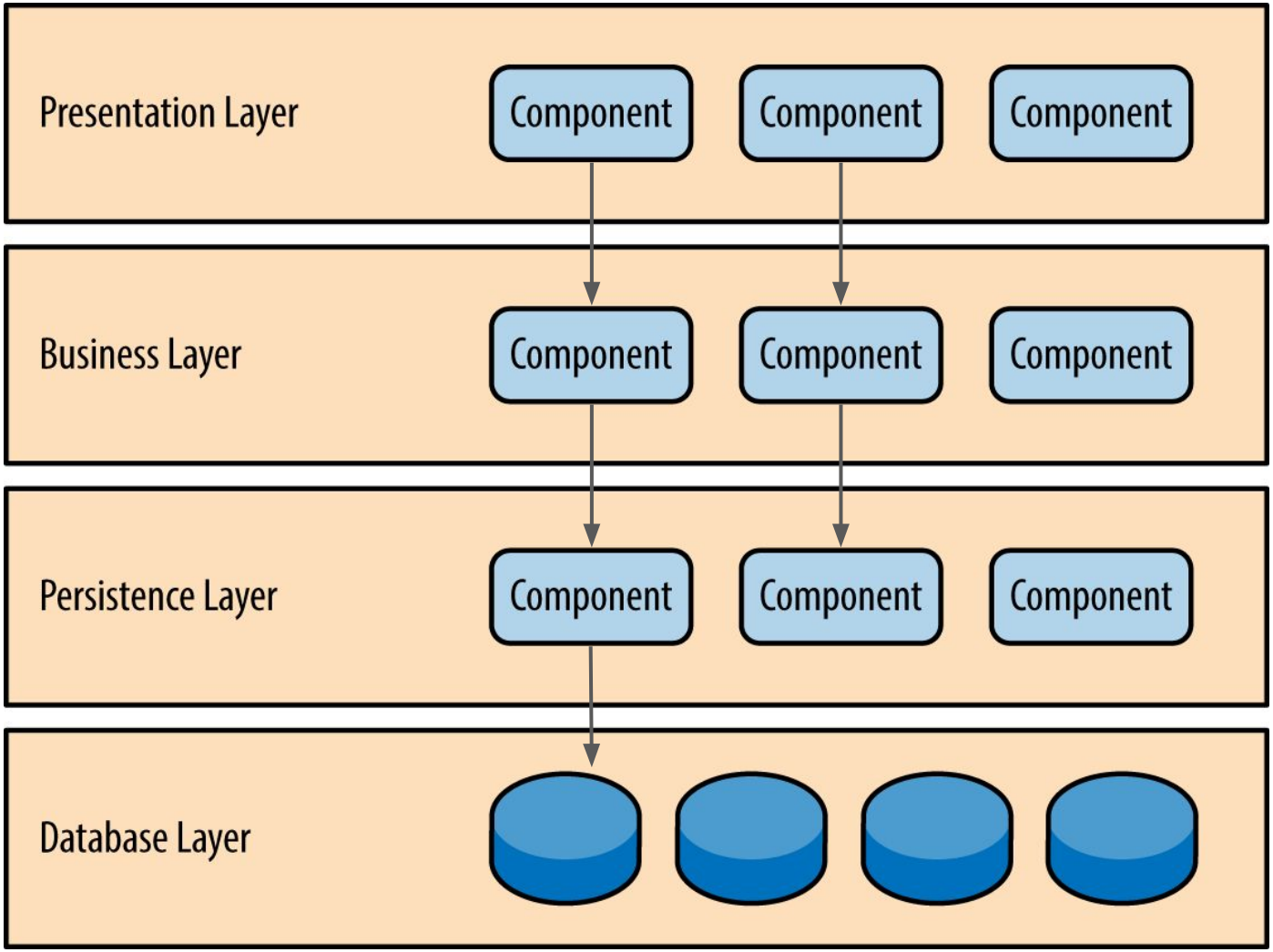
Advertisement Order

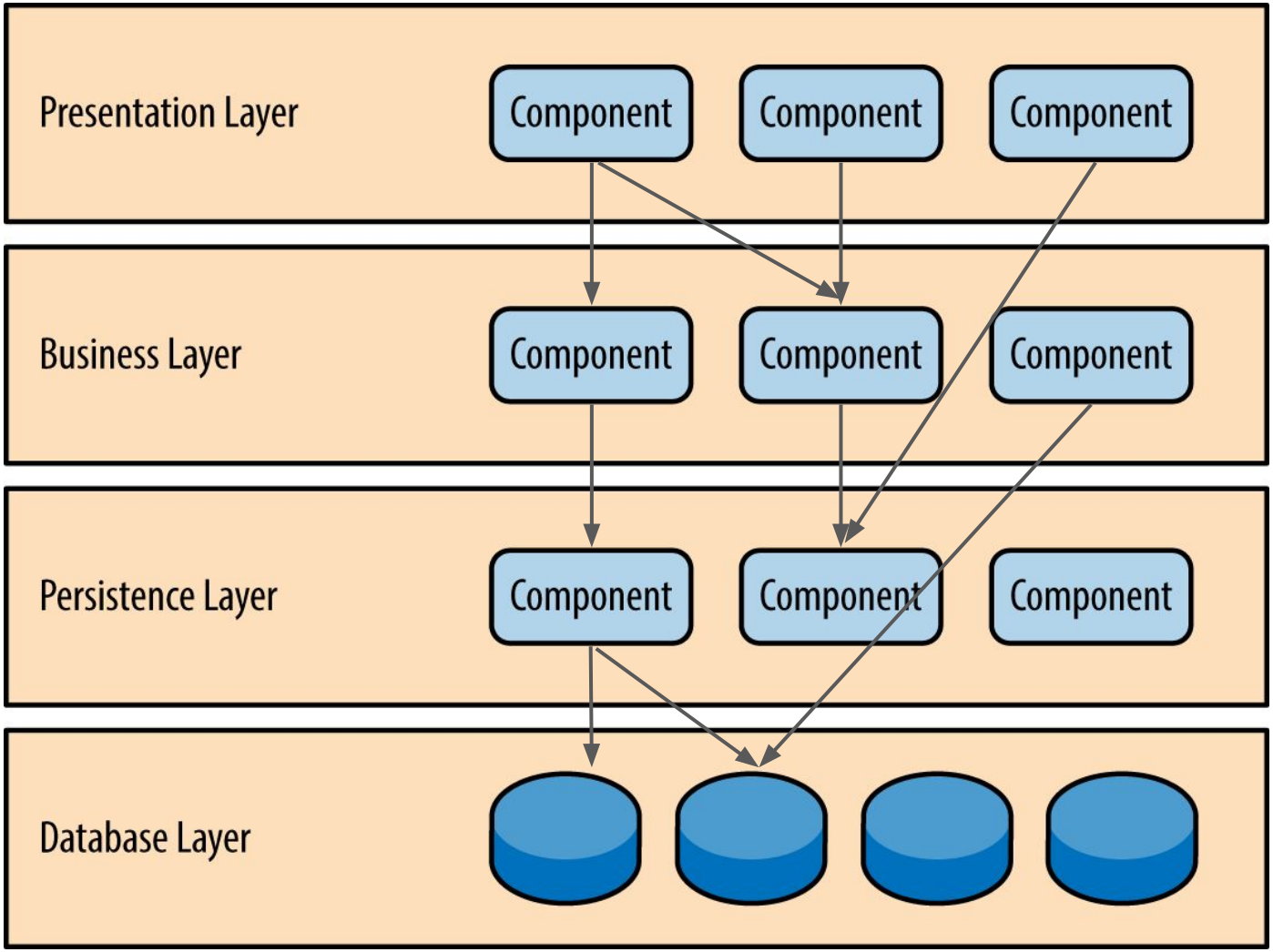
- Notify driver
- Mark car as booked
- Charge customer’s credit card


Cars Context
Work Order Context
Payment Context
Cars
Work Orders
Payments


Context X
Can you tell me
something about Z?
Context Y

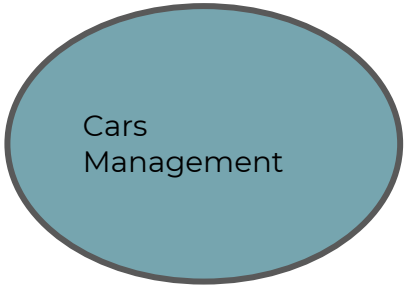

Context X
Context X
Context X
Can you tell me something about user?
User Context
Context X
Context X
Context X

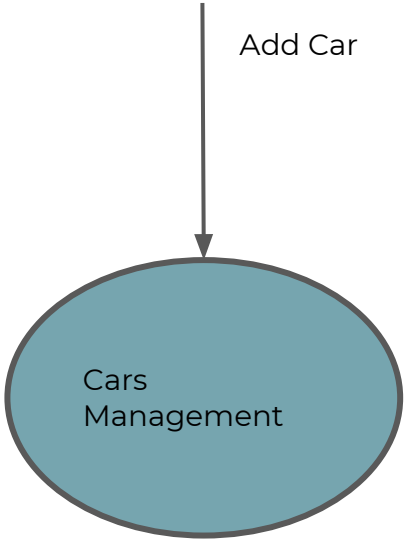

Presentation Layer
Component
Component
Component
Business Layer
Component
Component
Component
Persistence Layer
Component
Component
Component
Database Layer

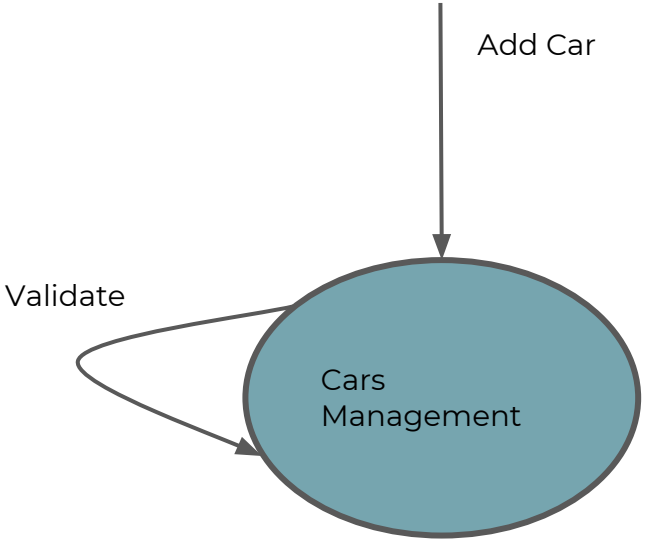

Presentation Layer
Component
Component
Component
Business Layer
Component
Component
Component
Persistence Layer
Component
Component
Component
Database Layer

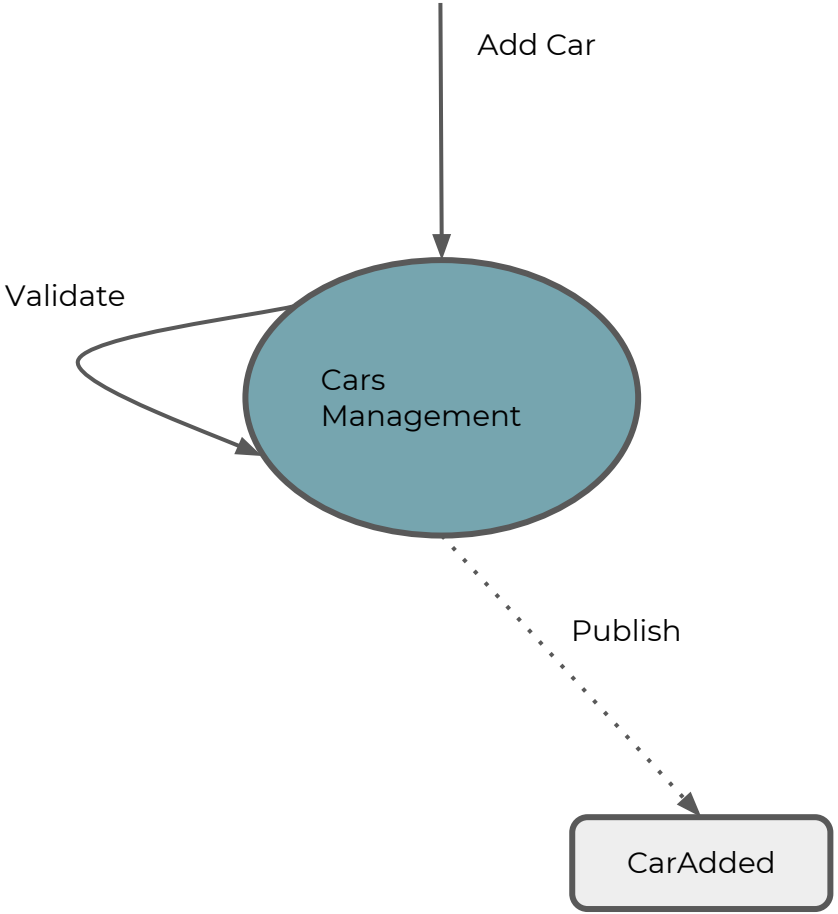
If we ask what context should know we usually end up with CRUD functionality. Instead of it we should rather ask **what context should do** and then we discover what it should know.

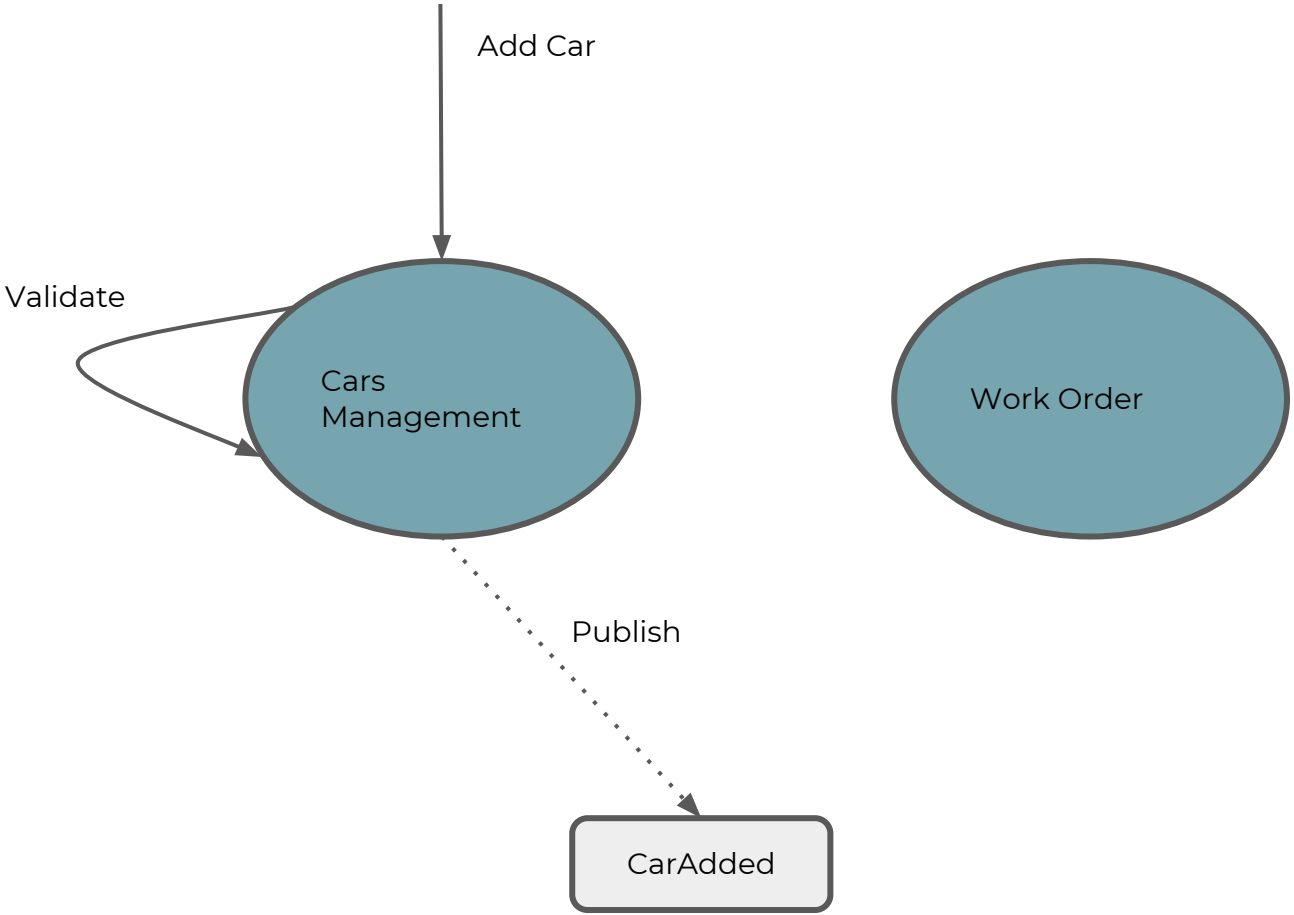
Let’s go back to the beginning. Use cases...

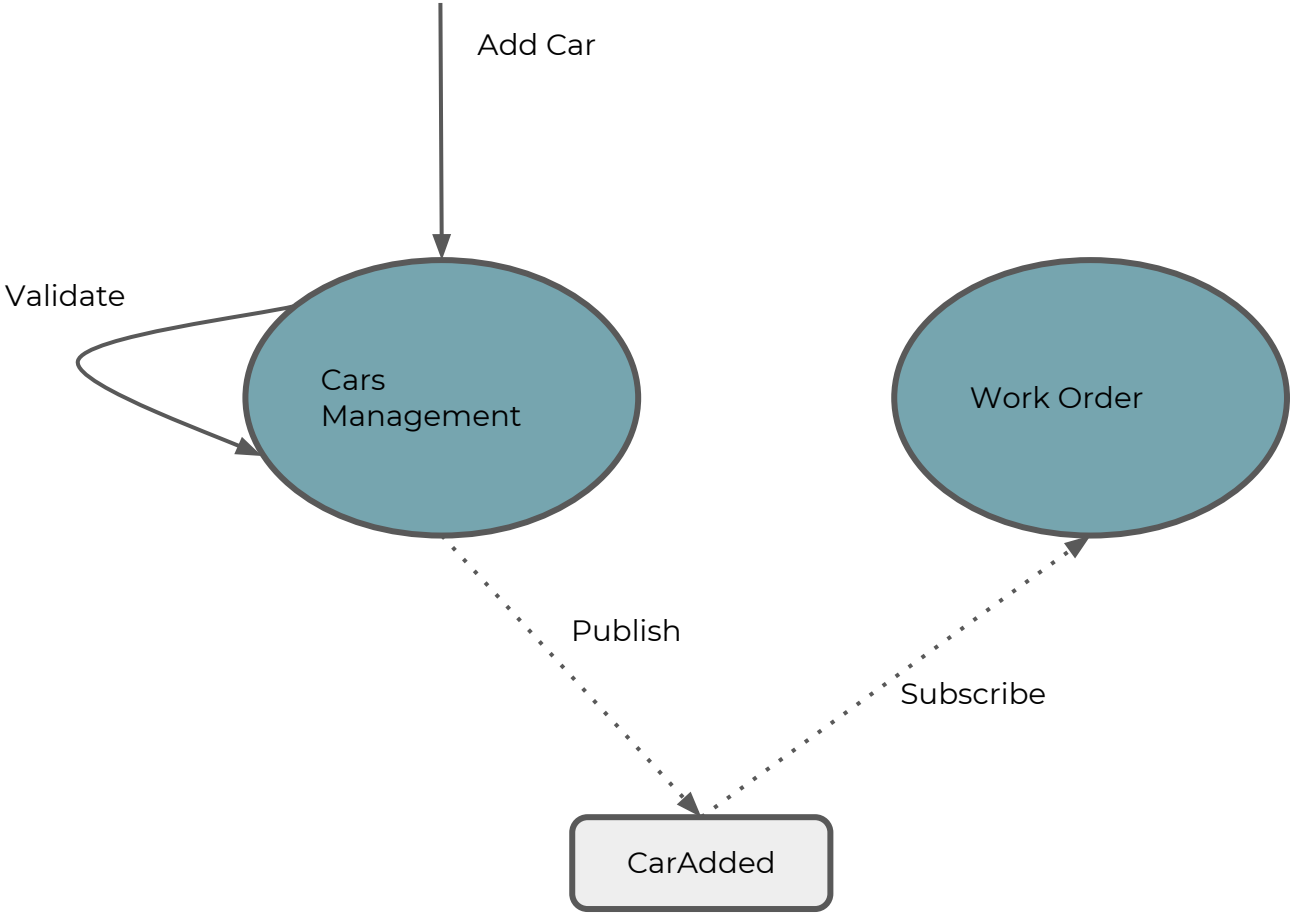

Cars
Management

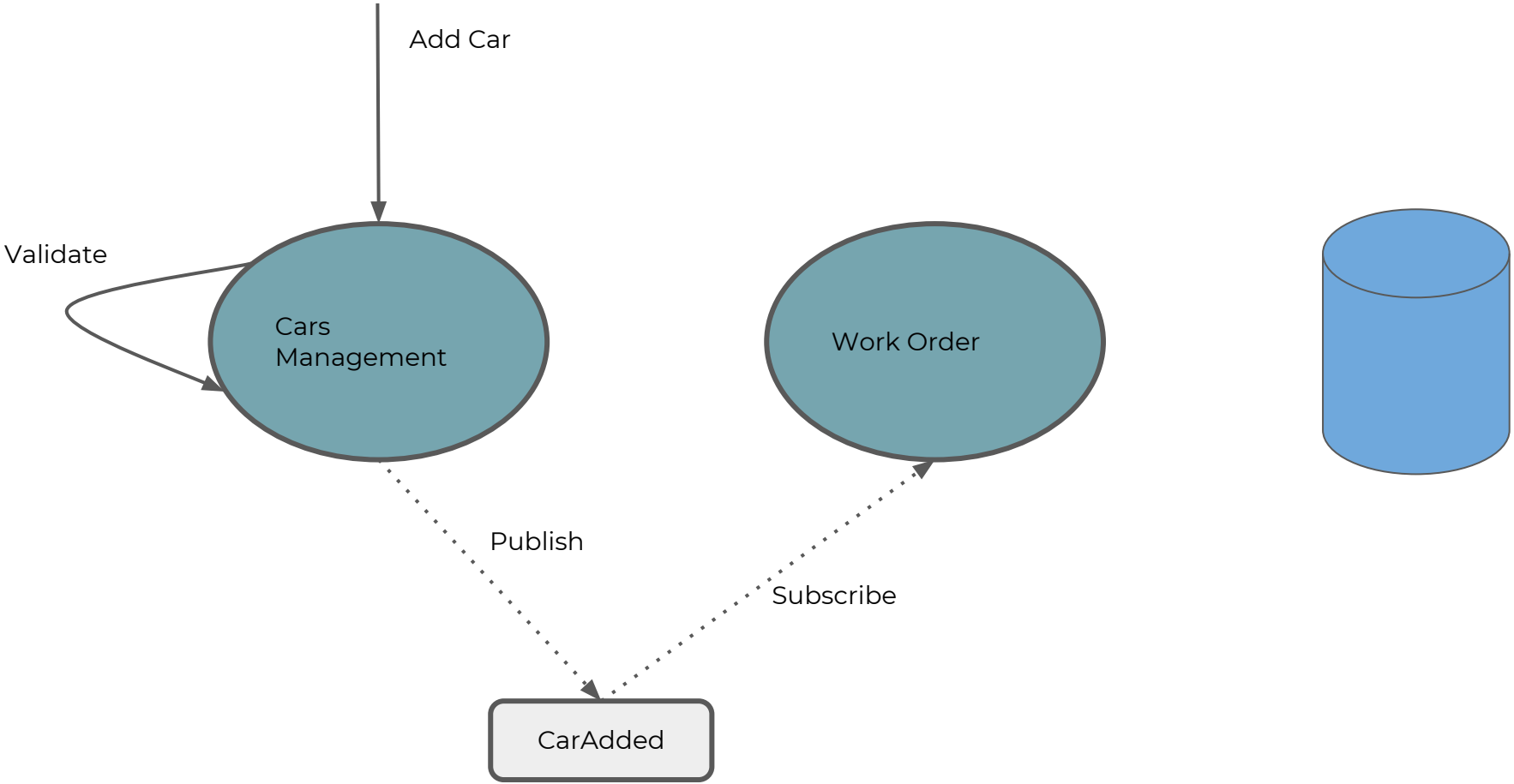

Add Car
Cars
Management

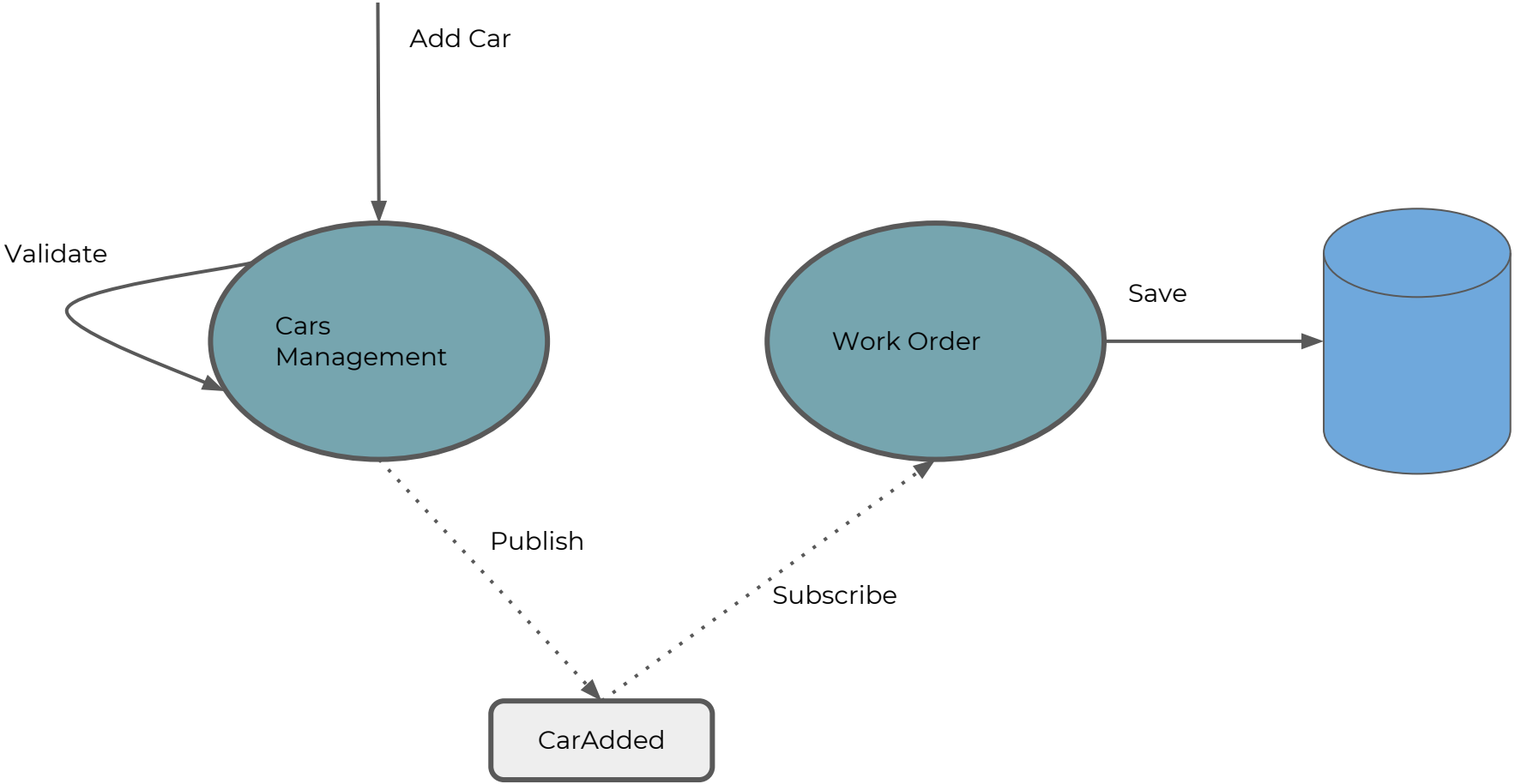

Add Car
Validate
Cars
Management


Add Car
Validate
Cars
Management
Publish
CarAdded

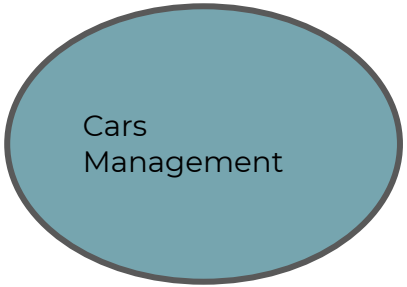

Add Car
Validate
Cars
Management
Work Order
Publish
CarAdded


Add Car
Validate
Cars
Management
Work Order
Publish
Subscribe
CarAdded

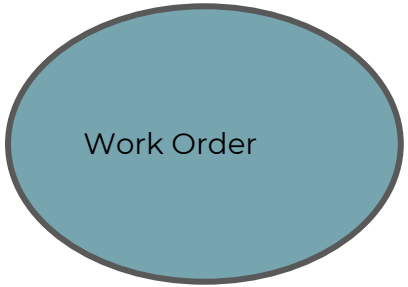

Add Car
Validate
Cars
Management
Work Order
Publish
Subscribe
CarAdded


Add Car
Validate
Cars
Management
Work Order
Save
Publish
Subscribe
CarAdded

Single Responsibility

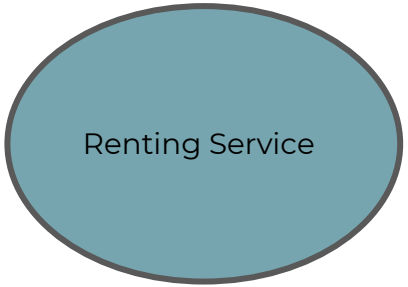


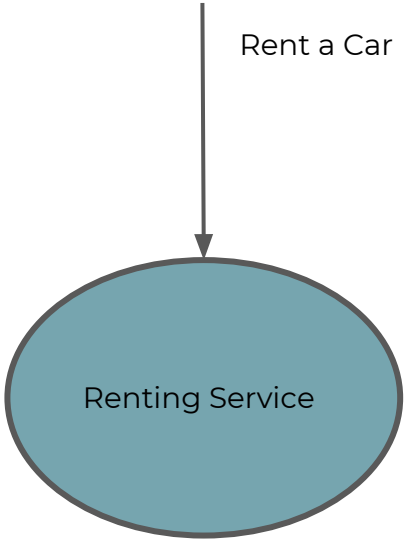
Validates Cars

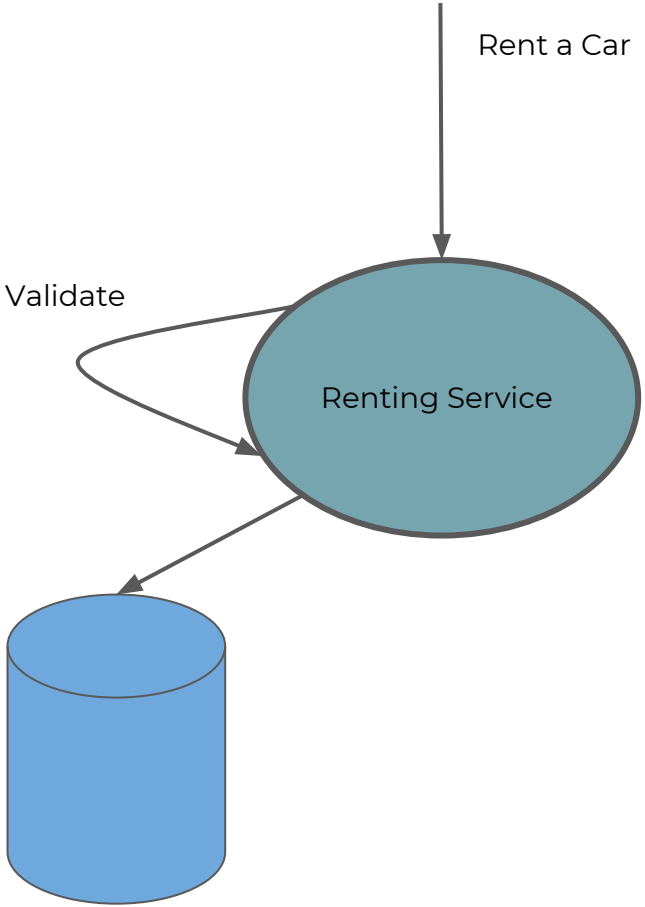


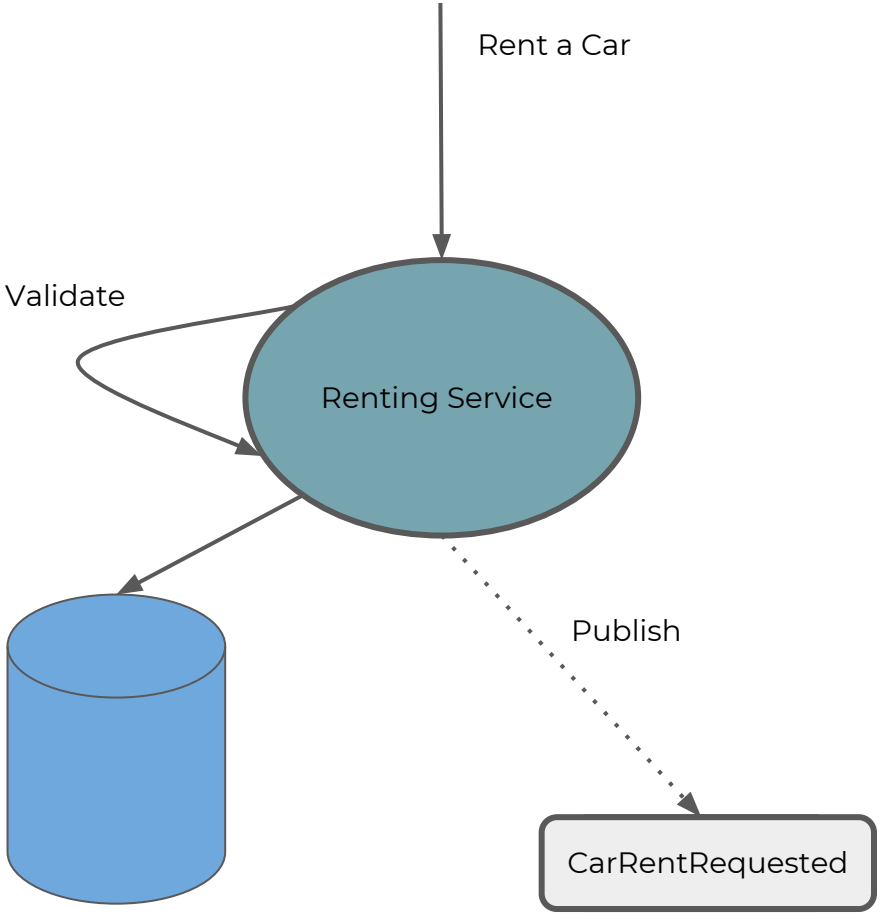
Books Cars

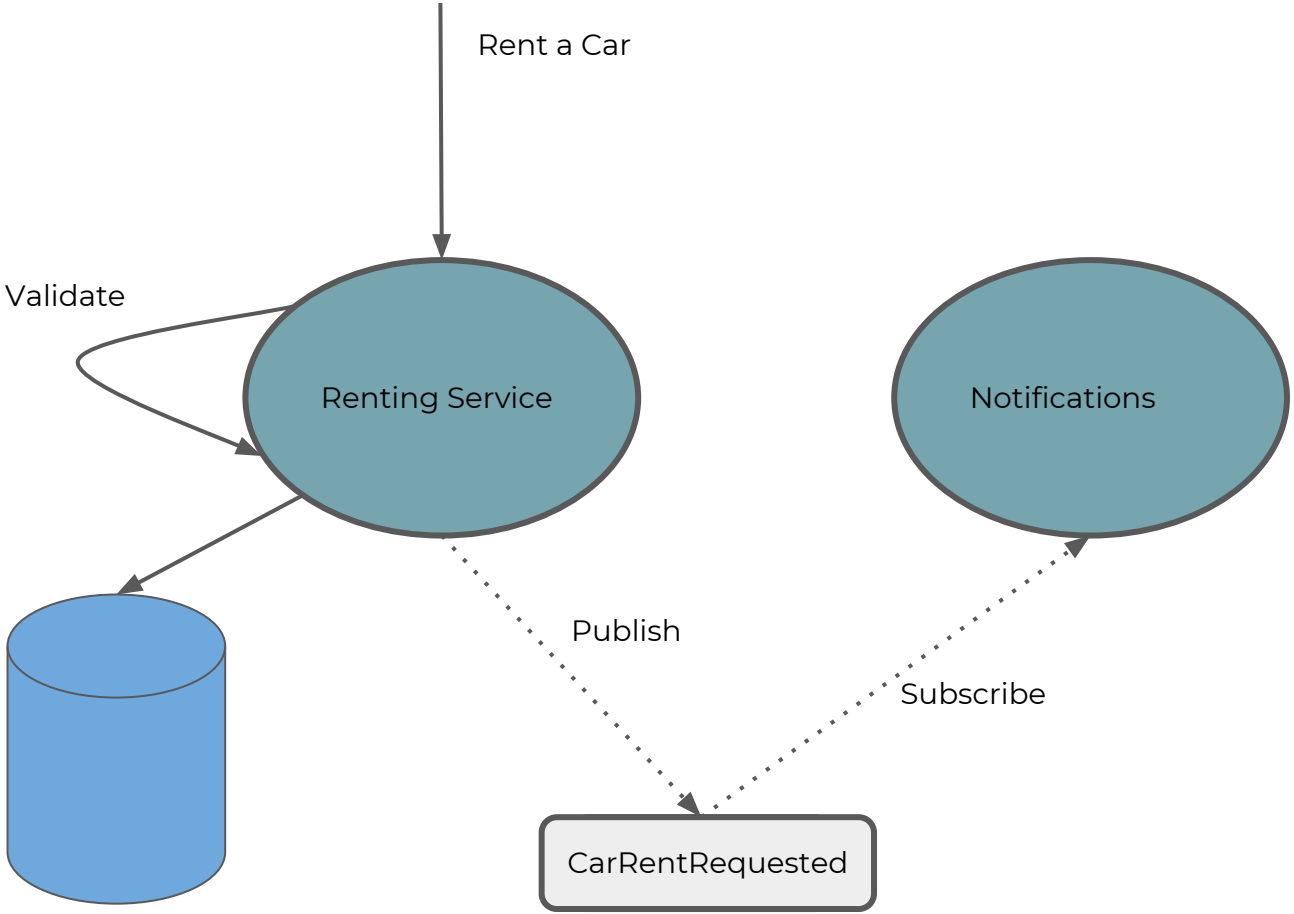
Application state changes are performed only by applying events.

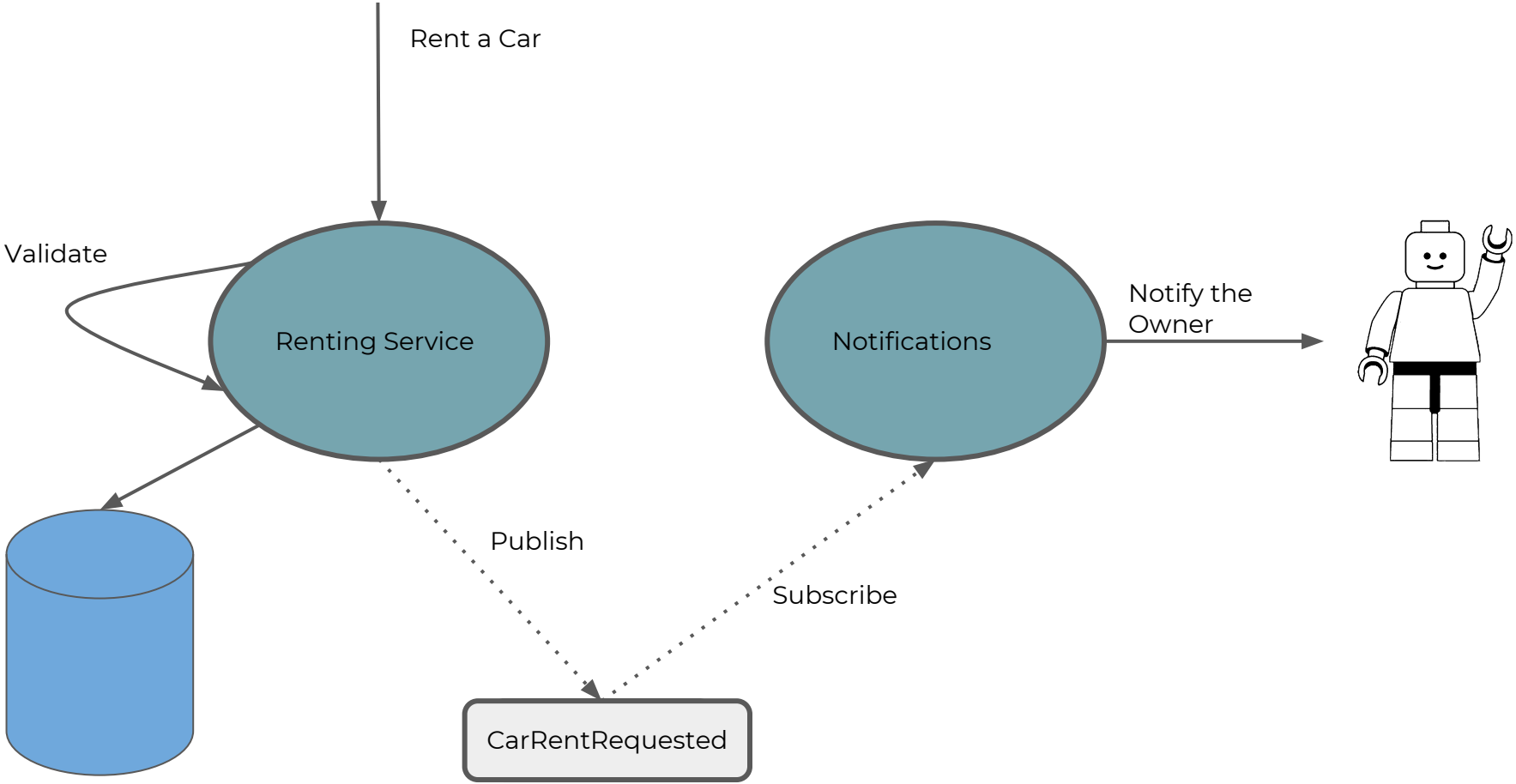

Renting Service

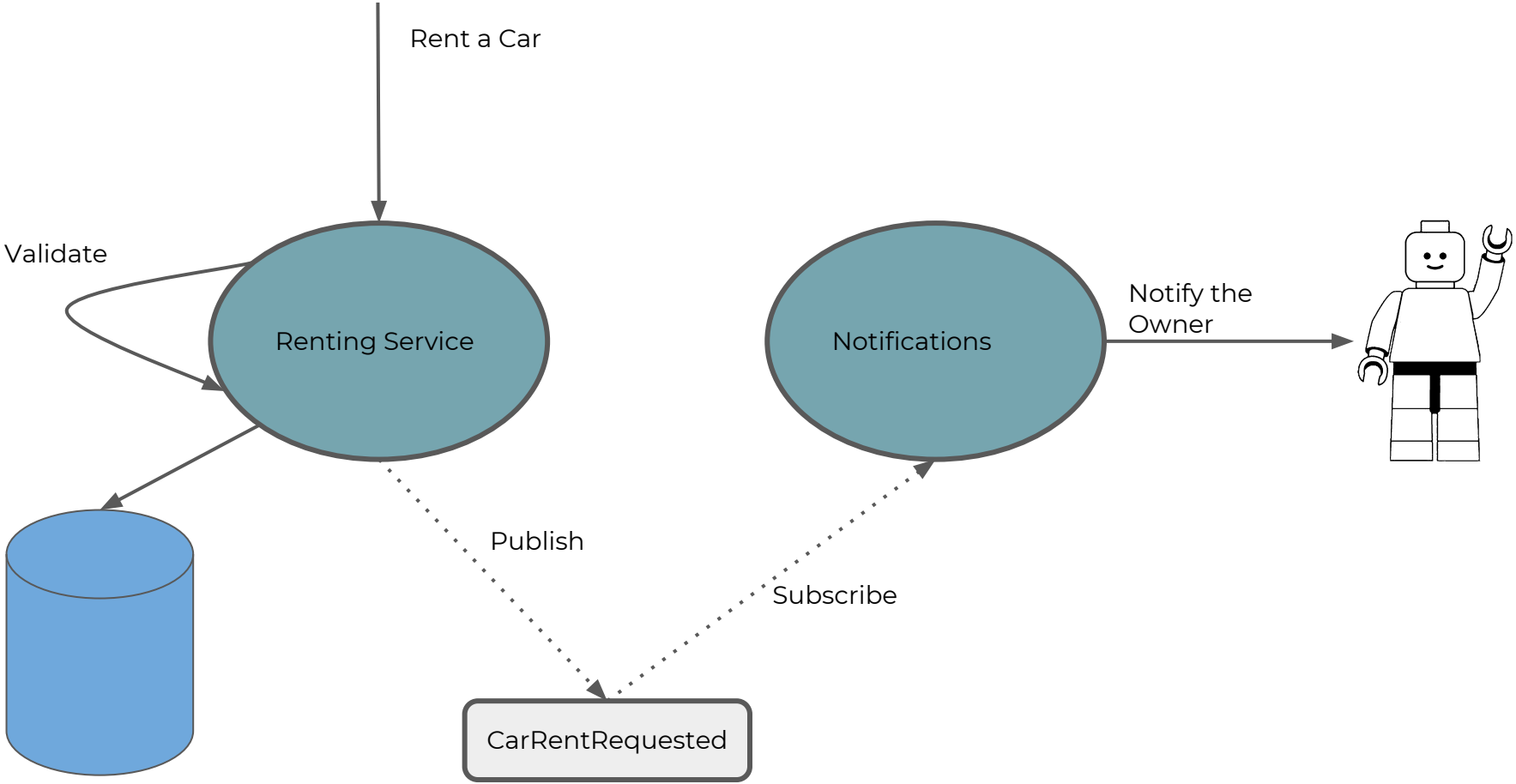

Rent a Car
Renting Service

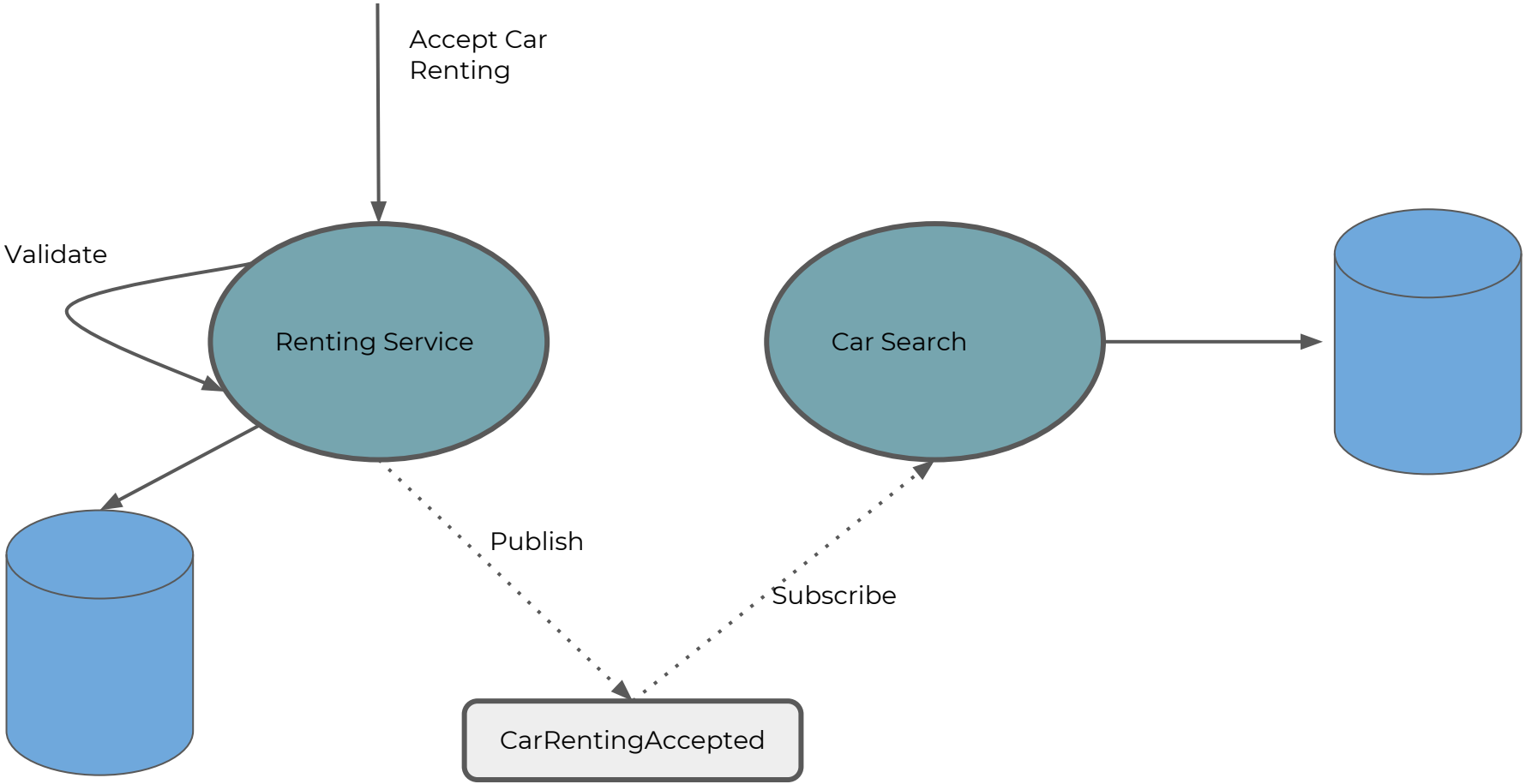

Rent a Car
Validate
Renting Service

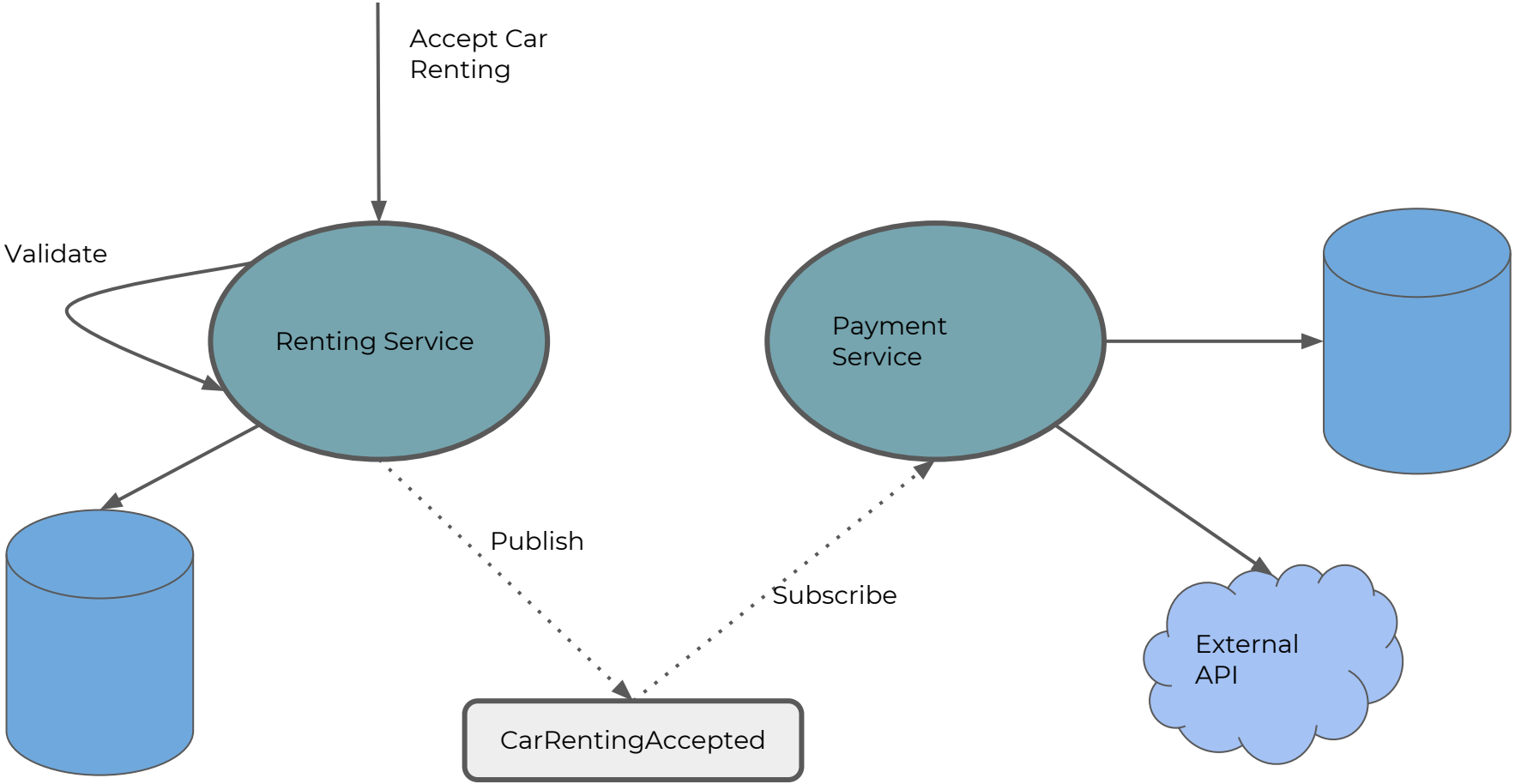

Rent a Car
Validate
Renting Service
Publish
CarRentRequested

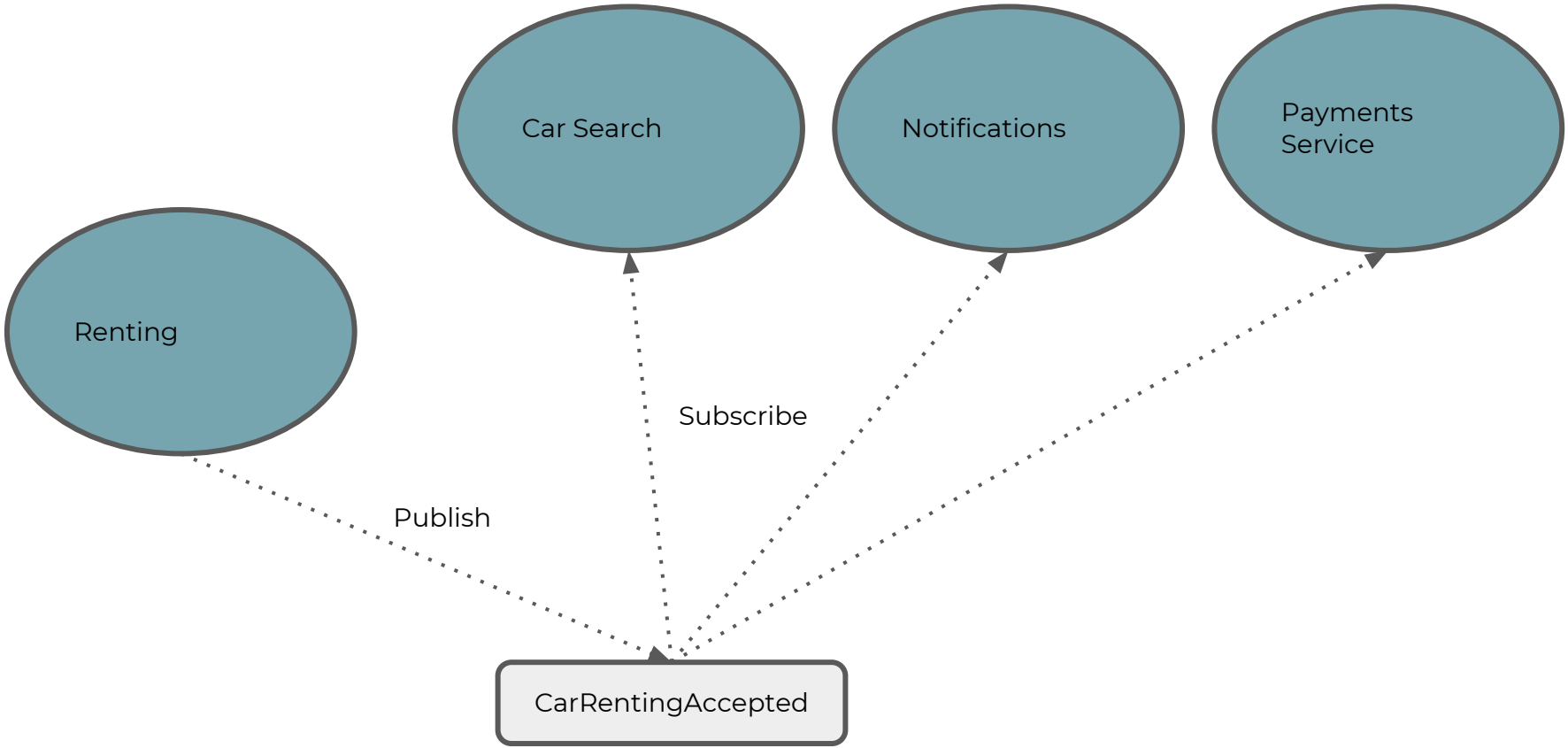

Rent a Car
Validate
Renting Service
Notifications
Publish
Subscribe
CarRentRequested


Rent a Car
Validate
Renting Service
Notifications
Notify the Owner
Publish
Subscribe
CarRentRequested

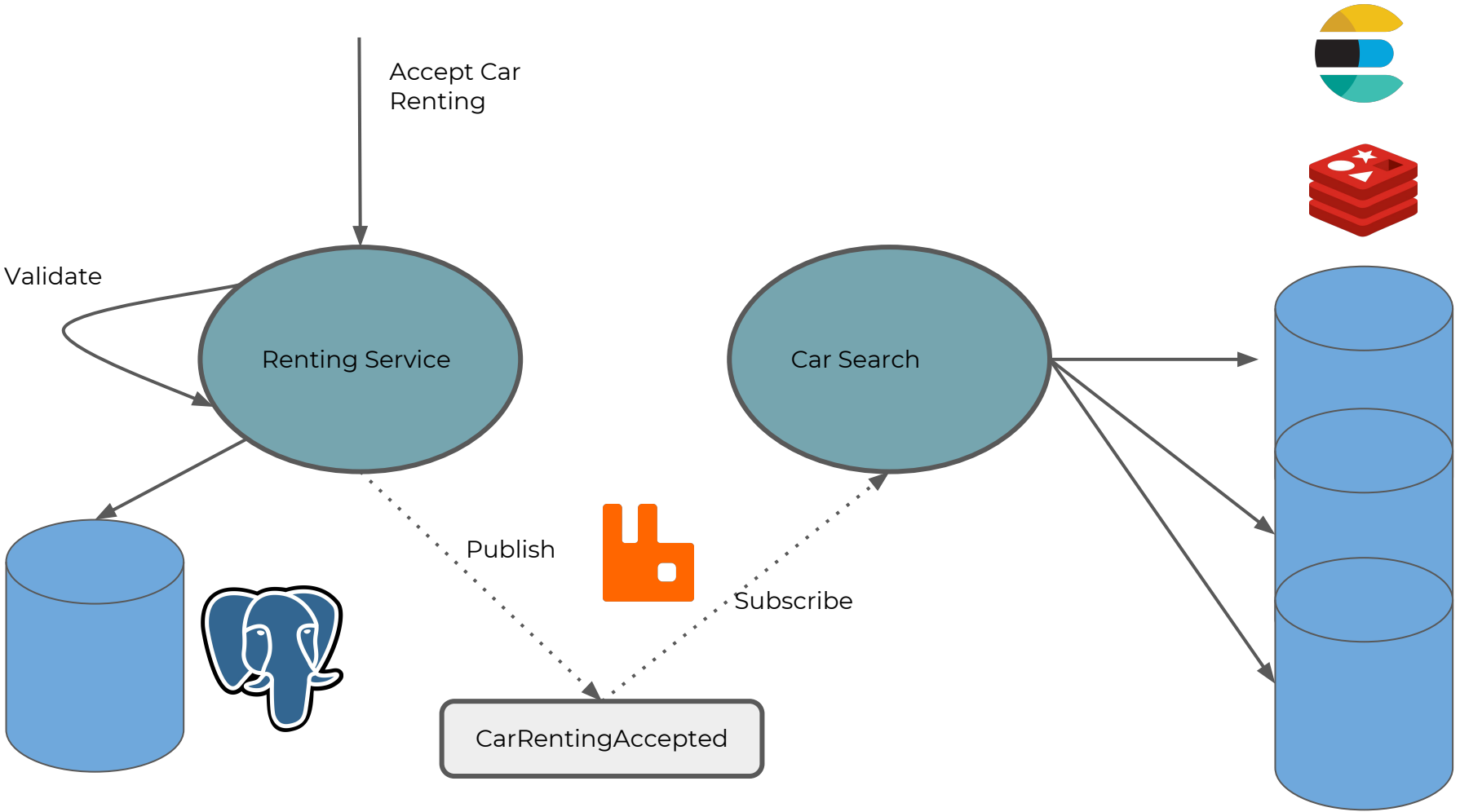

Rent a Car
Validate
Renting Service
Notifications
Notify the Owner
Publish
Subscribe
CarRentRequested


Accept Car Renting
Validate
Renting Service
Car Search
Publish
Subscribe
CarRentingAccepted


Accept Car Renting
Validate
Renting Service
Payment Service
Publish
Subscribe
External API
CarRentingAccepted


Car Search
Notifications
Payments Service
Renting
Subscribe
Publish
CarRentingAccepted

Small, decoupled, autonomous parts.


Accept Car Renting
Validate
Renting Service
Car Search
Publish
Subscribe
CarRentingAccepted

CQRS

CQS – Command Query Separation – concept by Bertrand Meyer from 1986. It stands that each method should be either command or query and they should be separated at code level.

Command – method that changes application state and returns nothing.

Query – method that returns result, but does not modify application state.

CQRS – Command Query Responsibility Segregation – implementing this concept is to segregate commands from queries and segregate it into classes which correspond to domains.

CQRS is not an architecture. It is pattern which can be implemented globally or partially.

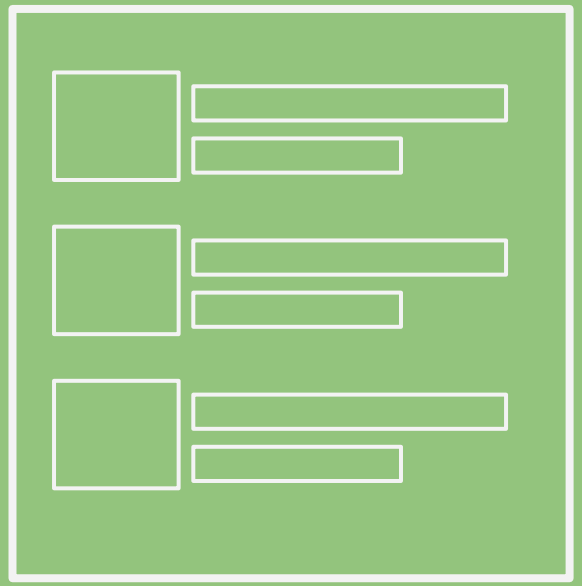
CQS is about methods and **CQRS** is about objects.

CQRS without Event Sourcing

Event Sourcing without CQRS

CODE

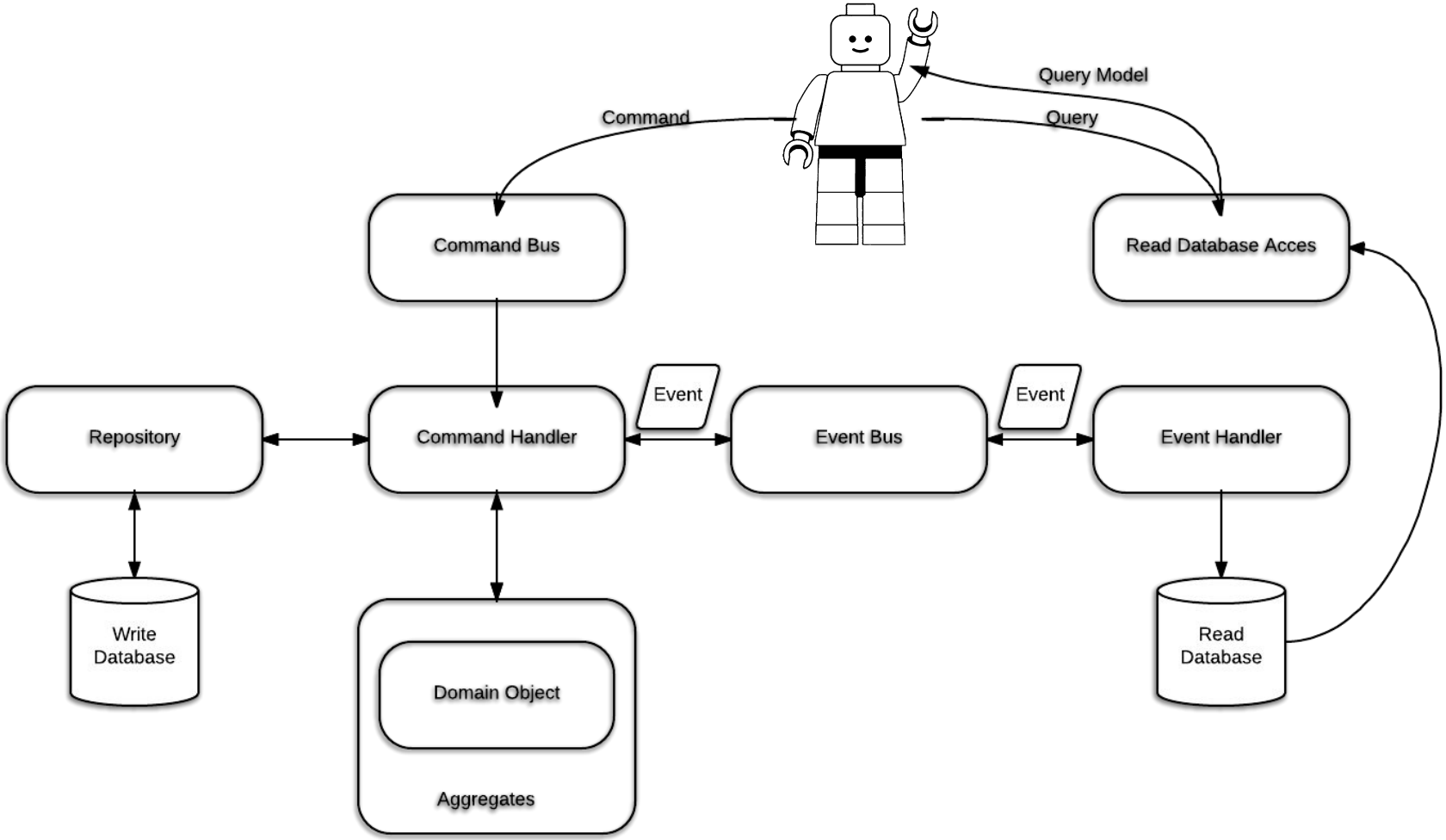
Event Sourcing

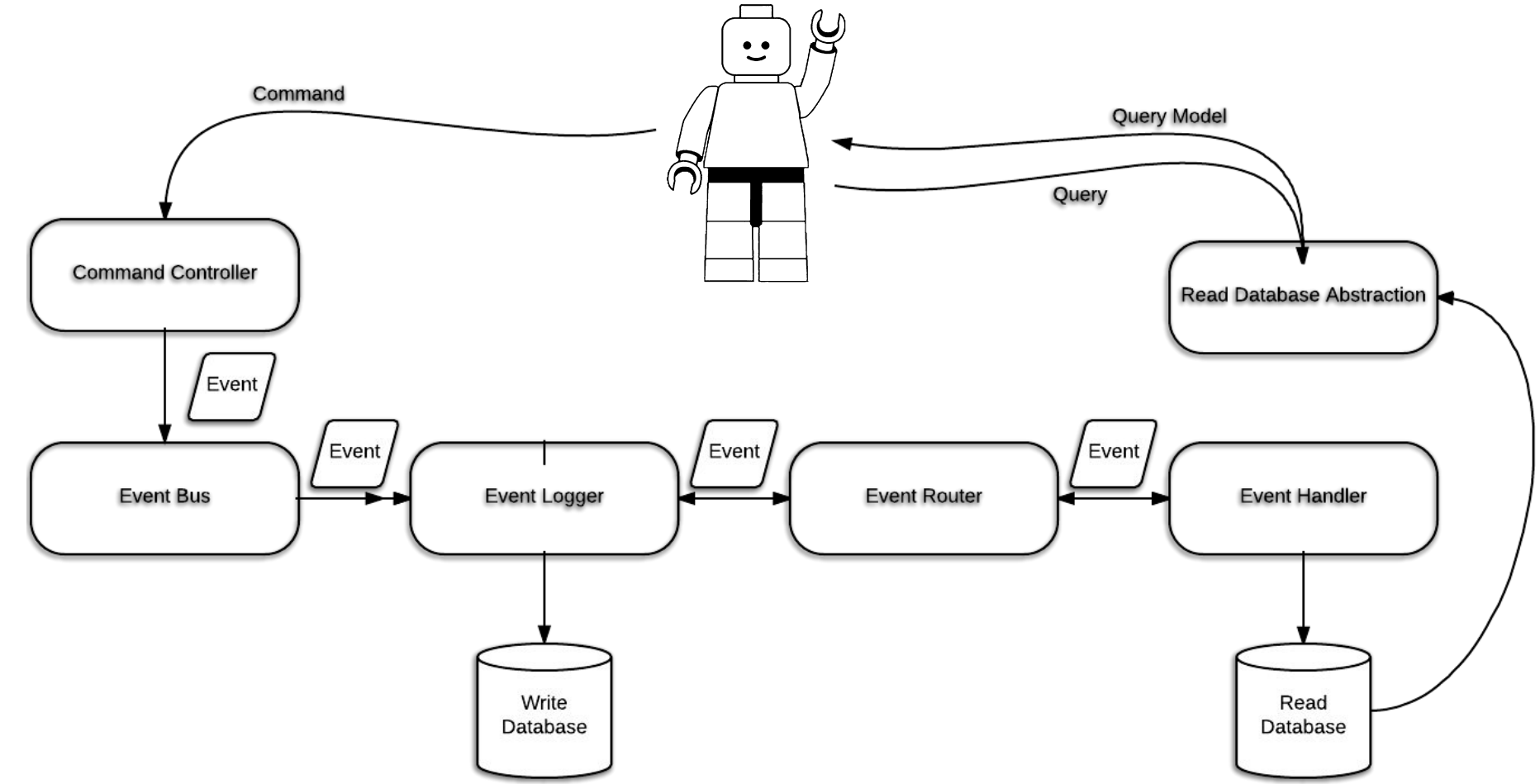


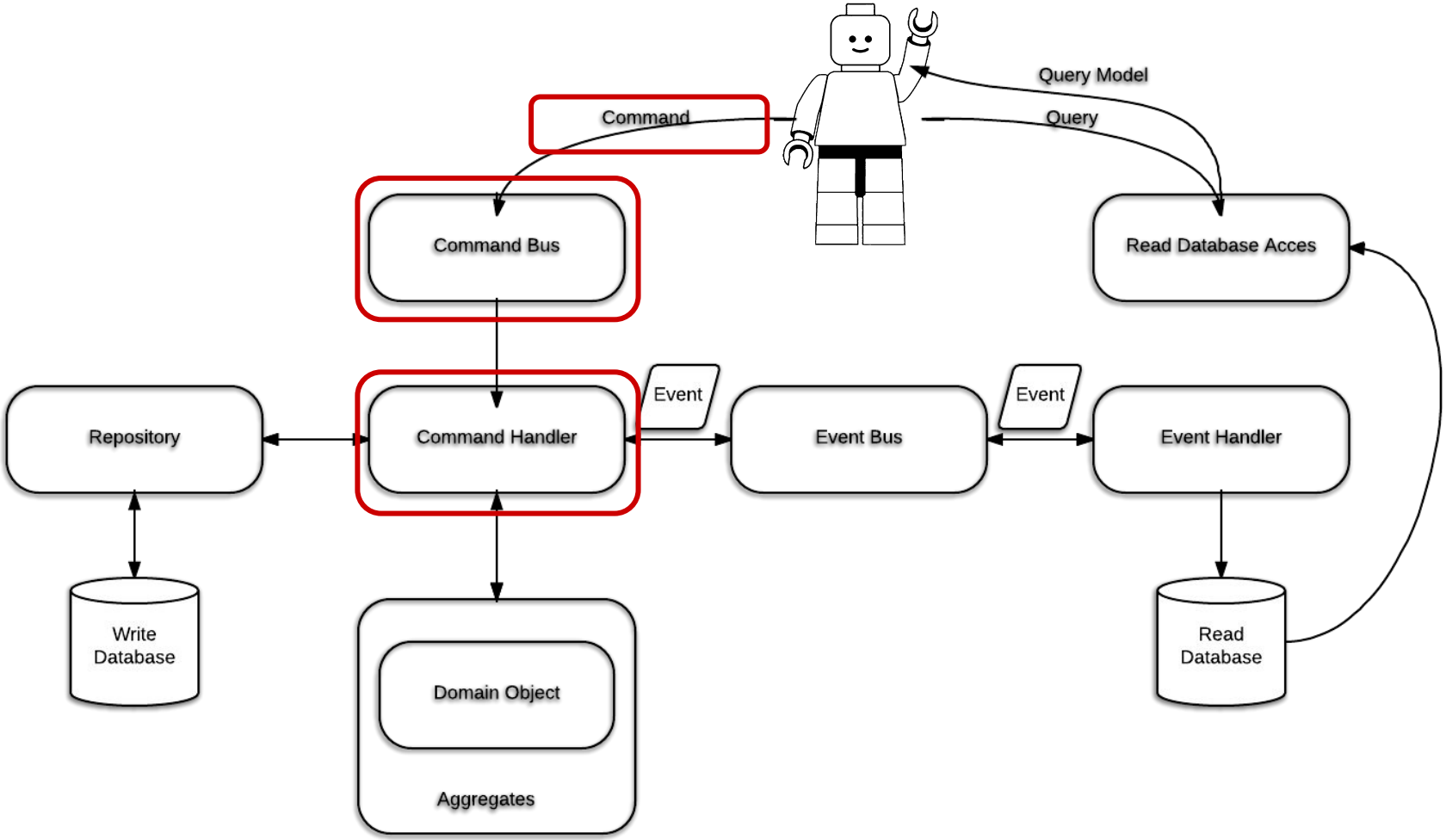
Event sourcing is (again) pattern (not architecture! There is event driven architecture and event sourcing implements that) that allow us to recover database image from **event store**.

Pros

1. Asymmetric scalability – application can be optimized towards read operations or write operations.
2. Good isolation of domains.
3. Easy deconstructing into microservices.
4. Audit database out of a box.
5. Application state recovery in any moment.


Query Model
Command
Query
Command Bus
Read Database Acces
Repository
Command Handler
Event
Event Bus
Event
Event Handler
Write Database
Domain Object
Aggregates
Read Database

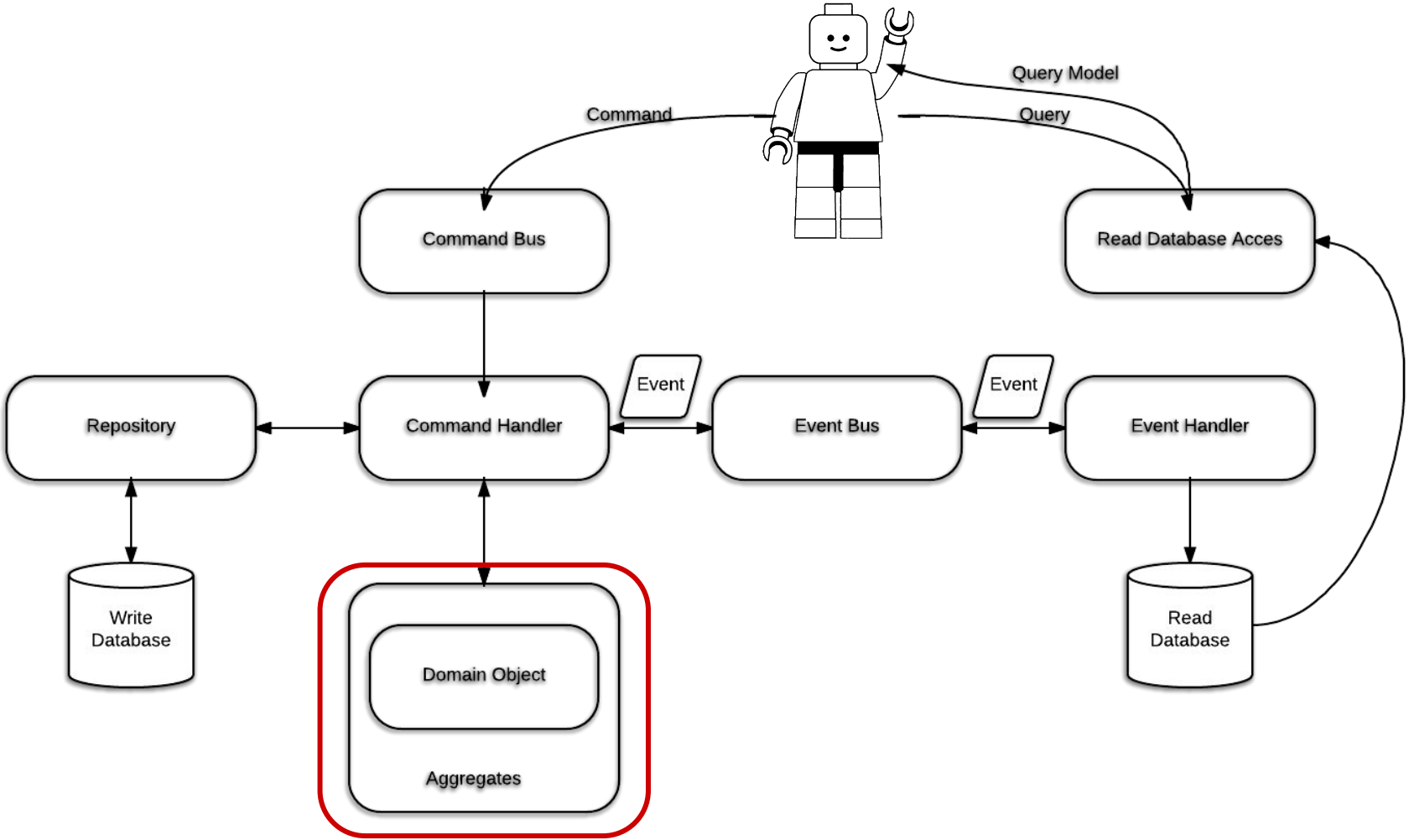

Command
Query Model
Query
Command Controller
Read Database Abstraction
Event
Event
Event
Event
Event Bus
Event Logger
Event Router
Event Handler
Write Database
Read Database


Command
Query Model
Query
Command Bus
Read Database Acces
Repository
Command Handler
Event
Event Bus
Event
Event Handler
Write Database
Domain Object
Aggregates
Read Database

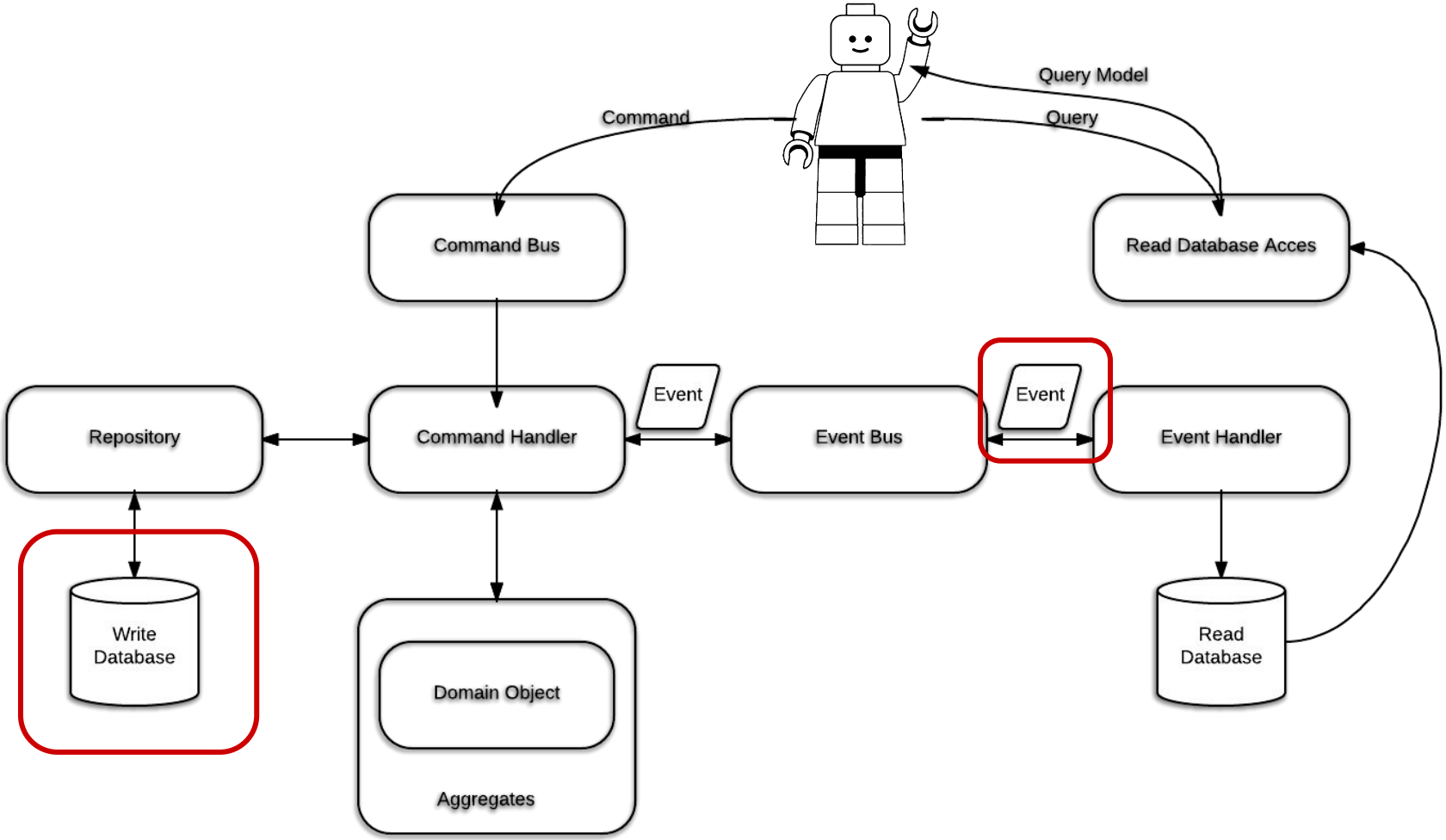
Command – object that represent user will. It can generate more than one event.

Command Bus – queues commands and routes command to command appropriate handler.

Command Handler – validates command and generates events which are passed to event bus.


Query Model
Command
Query
Command Bus
Read Database Acces
Repository
Command Handler
Event
Event Bus
Event
Event Handler
Write Database
Domain Object
Aggregates
Read Database

Domain objects – heart of our application. Business logic implementation. Aggregates groups domain object to treat them as identity (order and order position).


Query Model
Command
Query
Command Bus
Read Database Acces
Repository
Command Handler
Event
Event Bus
Event
Event Handler
Write Database
Domain Object
Aggregates
Read Database

Event – object which represents changes in a system:

- facts,
- **maps closely to business logic**, things that happened,
- immutable!,
- **reduces coupling**

(JSONSchema, Protobuf, ...).

Event

- Facts
- Multiple handlers
- Cannot be rejected
- Past tense

E.g. CarBooked

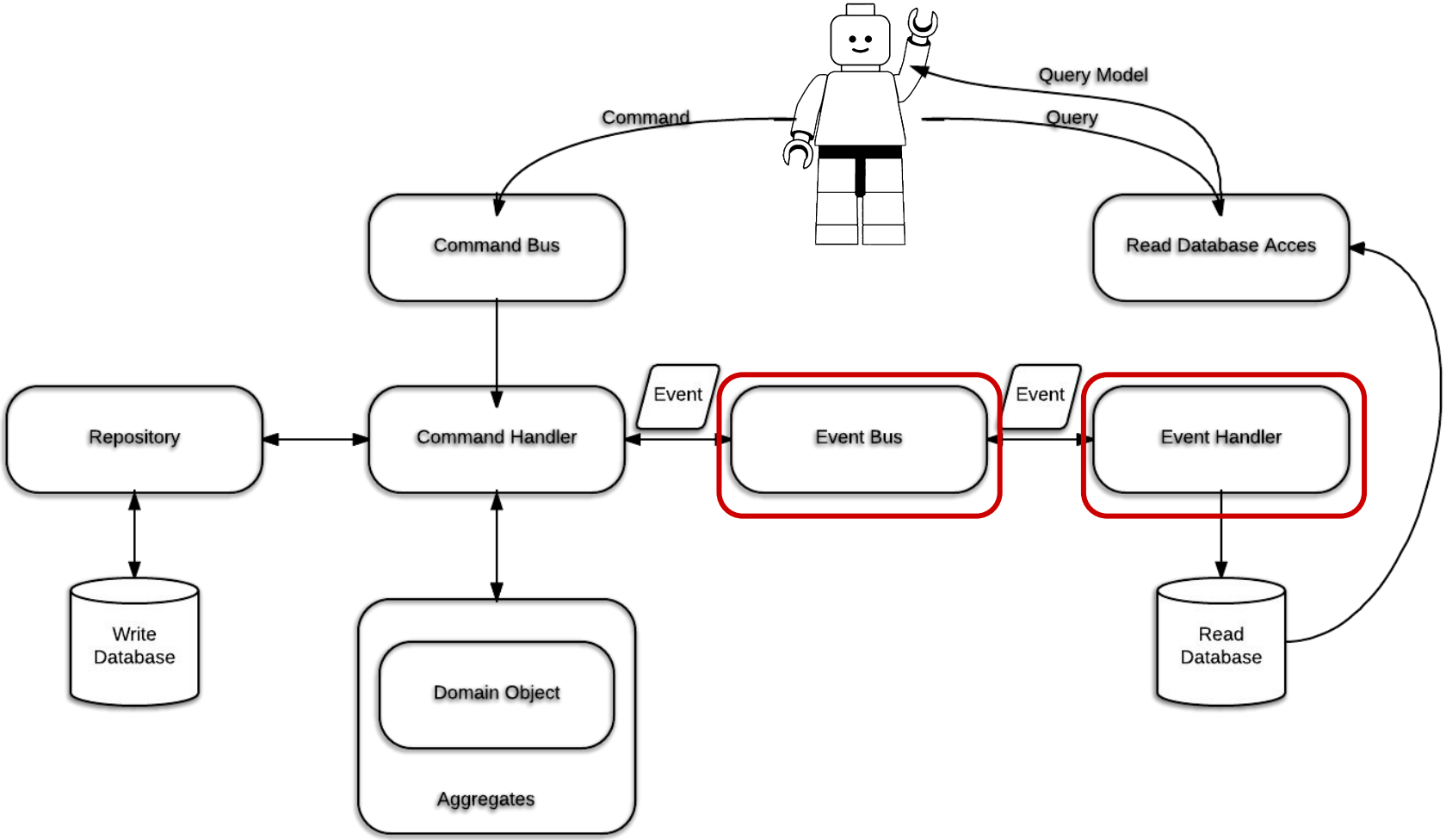
Command

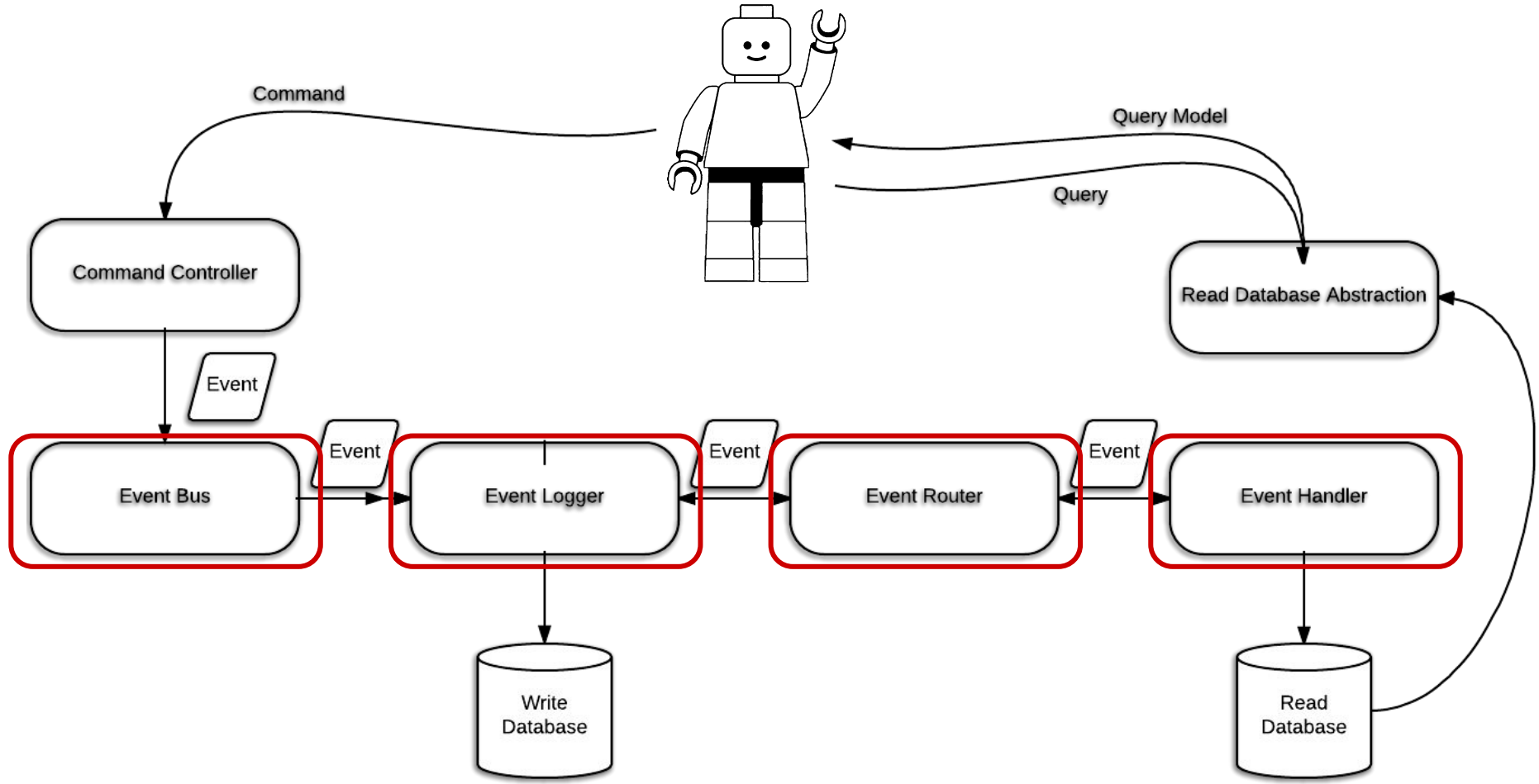
- Requests
- One handler
- Can be rejected
- Imperative

E.g. BookCar

Event Store – write database which stores every event (EventStore, Postgres, ...).

```
sealed class UserEvent : Event

data class UserRegistered(
    val id: UUID,
    val userName: String,
    val name: String,
    val currency: String,
    val ethereumAddress: String?,
    val password: String
) : UserEvent()

data class UserUpdated(
    val id: UUID,
    val userName: String?,
    val name: String?,
    val currency: String?,
    val ethereumAddress: String?
) : UserEvent()

data class UserDeleted(val id: UUID) : UserEvent()
```


Query Model
Command
Query
Command Bus
Read Database Acces
Repository
Command Handler
Event
Event Bus
Event
Event Handler
Write Database
Domain Object
Aggregates
Read Database


Command
Query Model
Query
Command Controller
Read Database Abstraction
Event
Event
Event
Event
Event Bus
Event Logger
Event Router
Event Handler
Write Database
Read Database

Event Bus – queues events and sends it to logger (RabbitMq, ZeroMq, Kafka, Emitter, Postgres, ...).

CODE

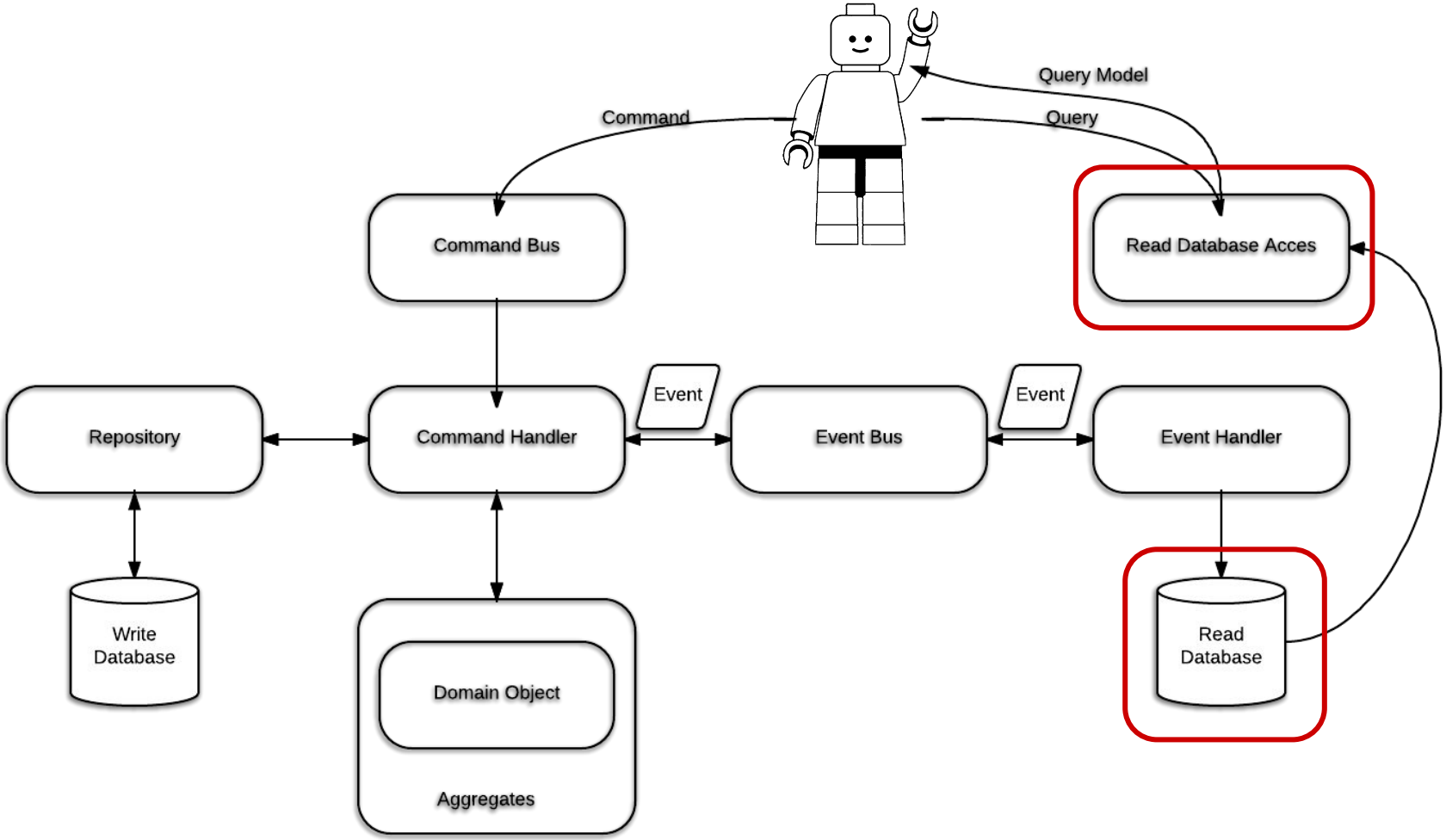
Event Logger – logs every event into event store and pass event to event router.

Event Router – routes events to appropriate event handler.

Event Handler – process event and write results to read database.

```
interface Event : Serializable

interface EventHandler<in T : Event> {
    fun canHandle(e: EventEnvelope<Event>): Boolean
    fun apply(e: T)
}

interface Logger<in T : Event> {
    fun logEvent(envelope: EventEnvelope<T>)
}

interface Router<in T : Event> {
    infix fun route(envelope: EventEnvelope<T>)
}

interface Persister<in T : Event> {
    infix fun persist(envelope: EventEnvelope<T>)
}
```

CODE


Command
Query Model
Query
Command Bus
Read Database Acces
Repository
Command Handler
Event
Event Bus
Event
Event Handler
Write Database
Domain Object
Aggregates
Read Database

Read Database Abstraction – layer that isolates access to data (Redis, SOLR, ElasticSearch).

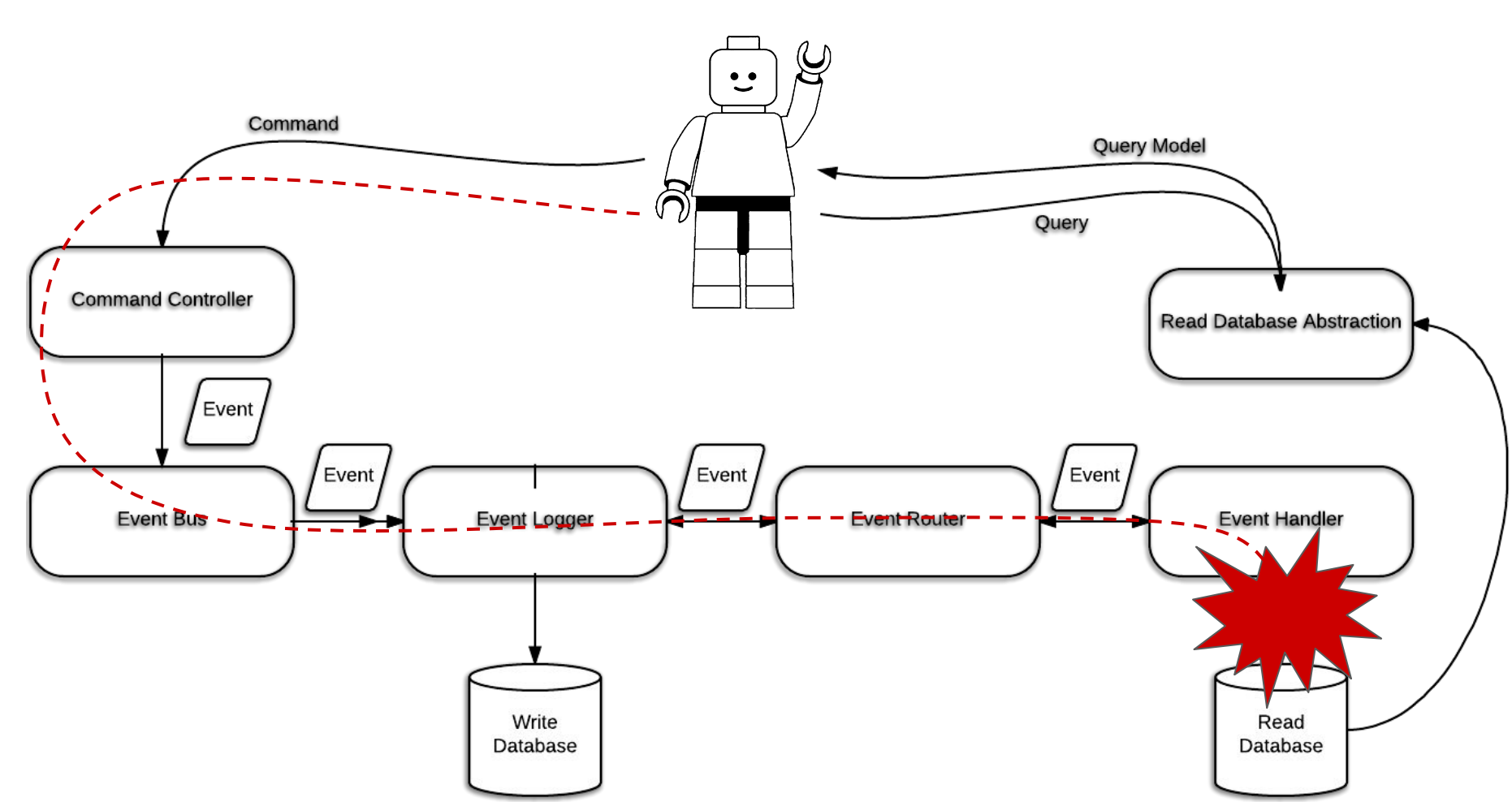
Problems

1. Managing events order
2. Handling long processes
3. Segregation events due to domain
4. High complexity

Saga

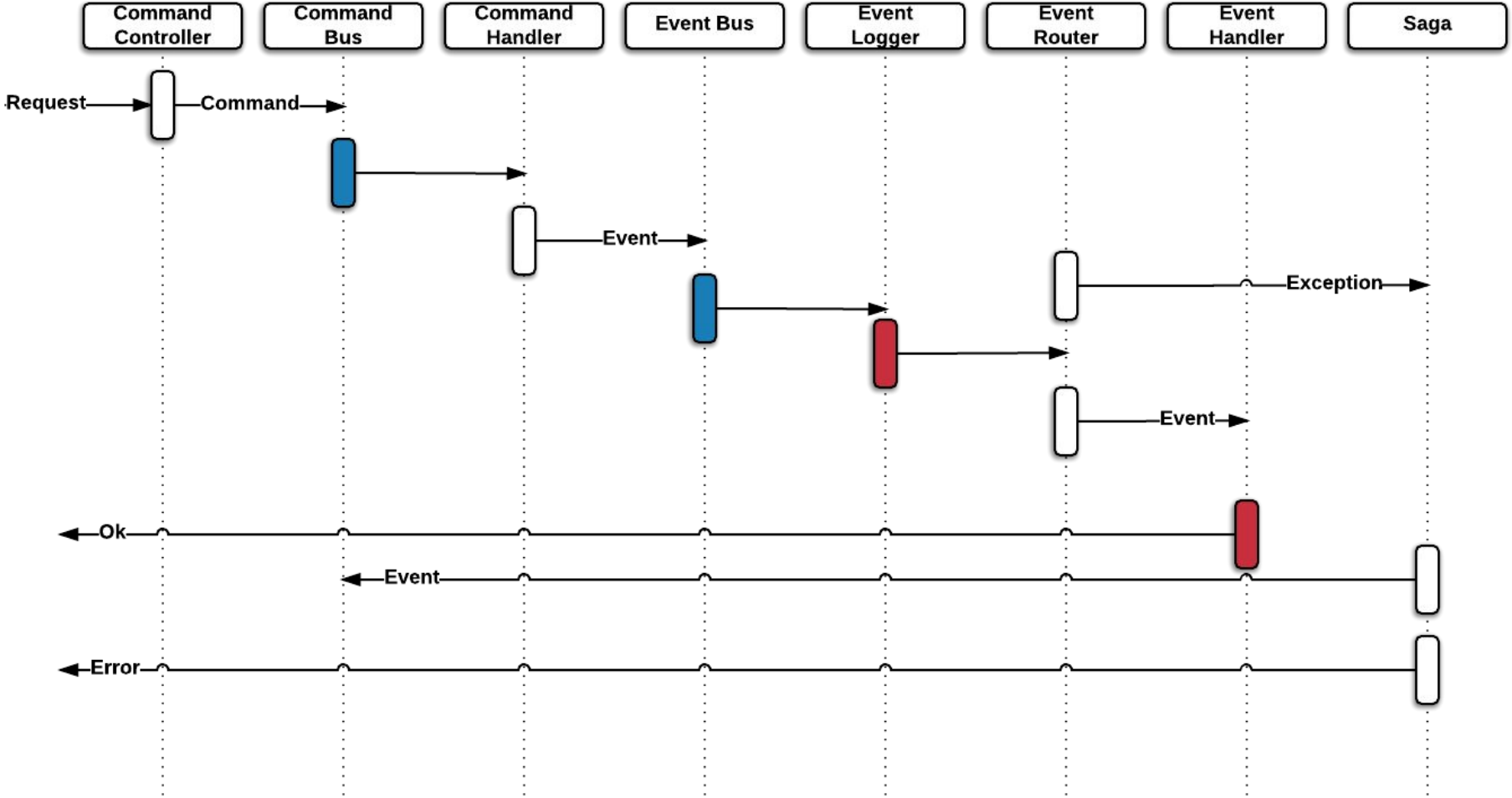
Disclaimer

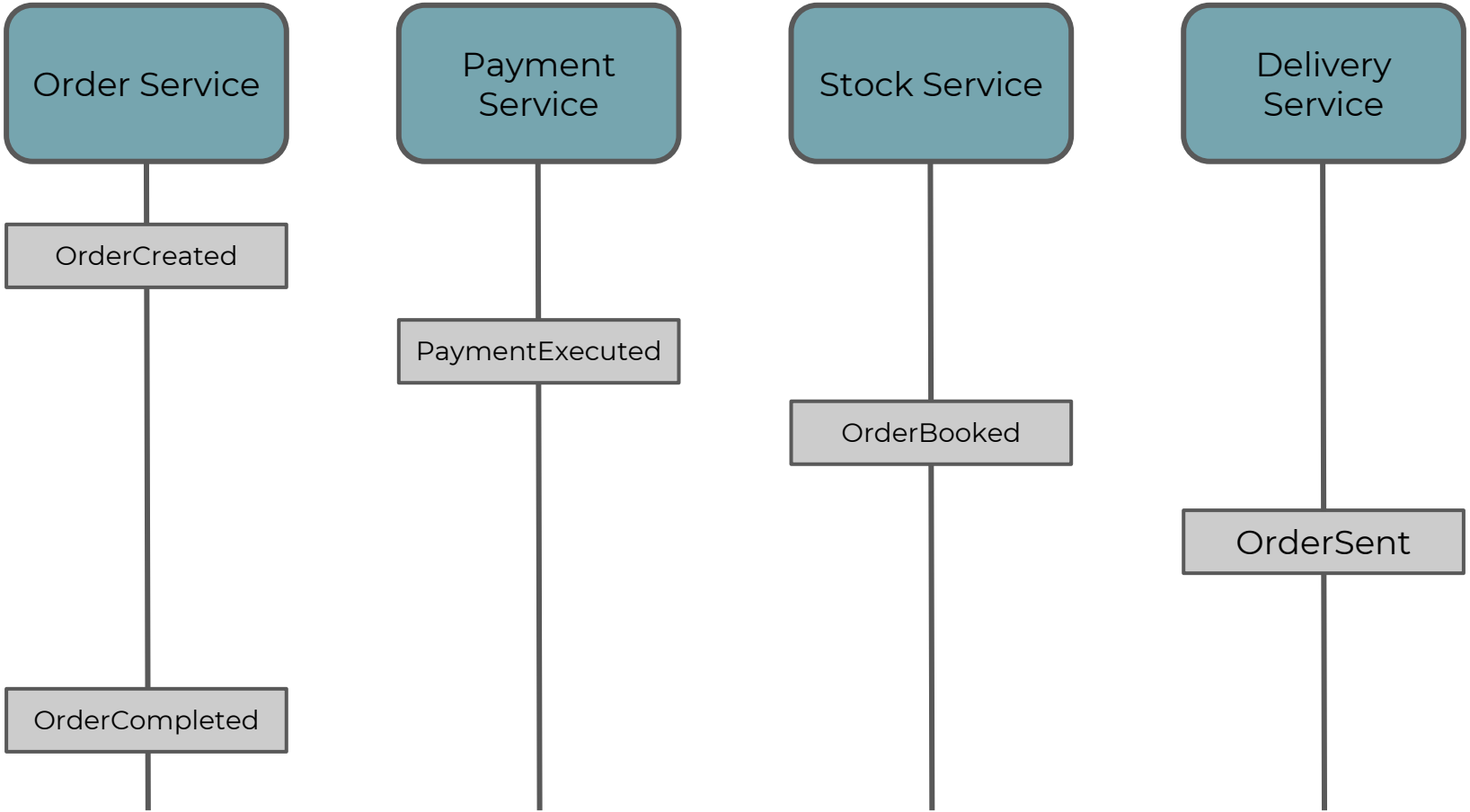
I usually write code using go or java so I’m talking about multithreaded languages.

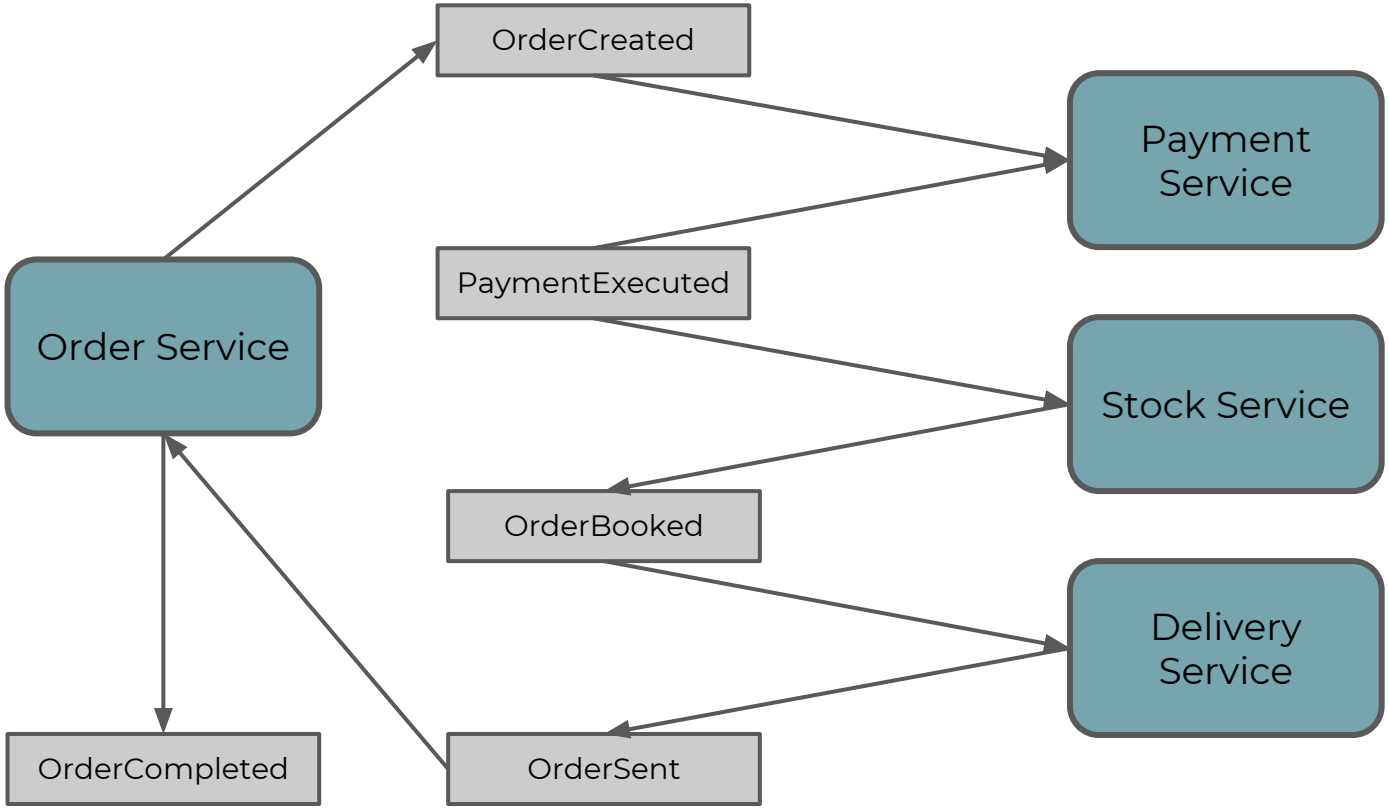

Command
Query Model
Query
Command Controller
Read Database Abstraction
Event
Event
Event
Event
Event Bus
Event Logger
Event Router
Event Handler
Write Database
Read Database

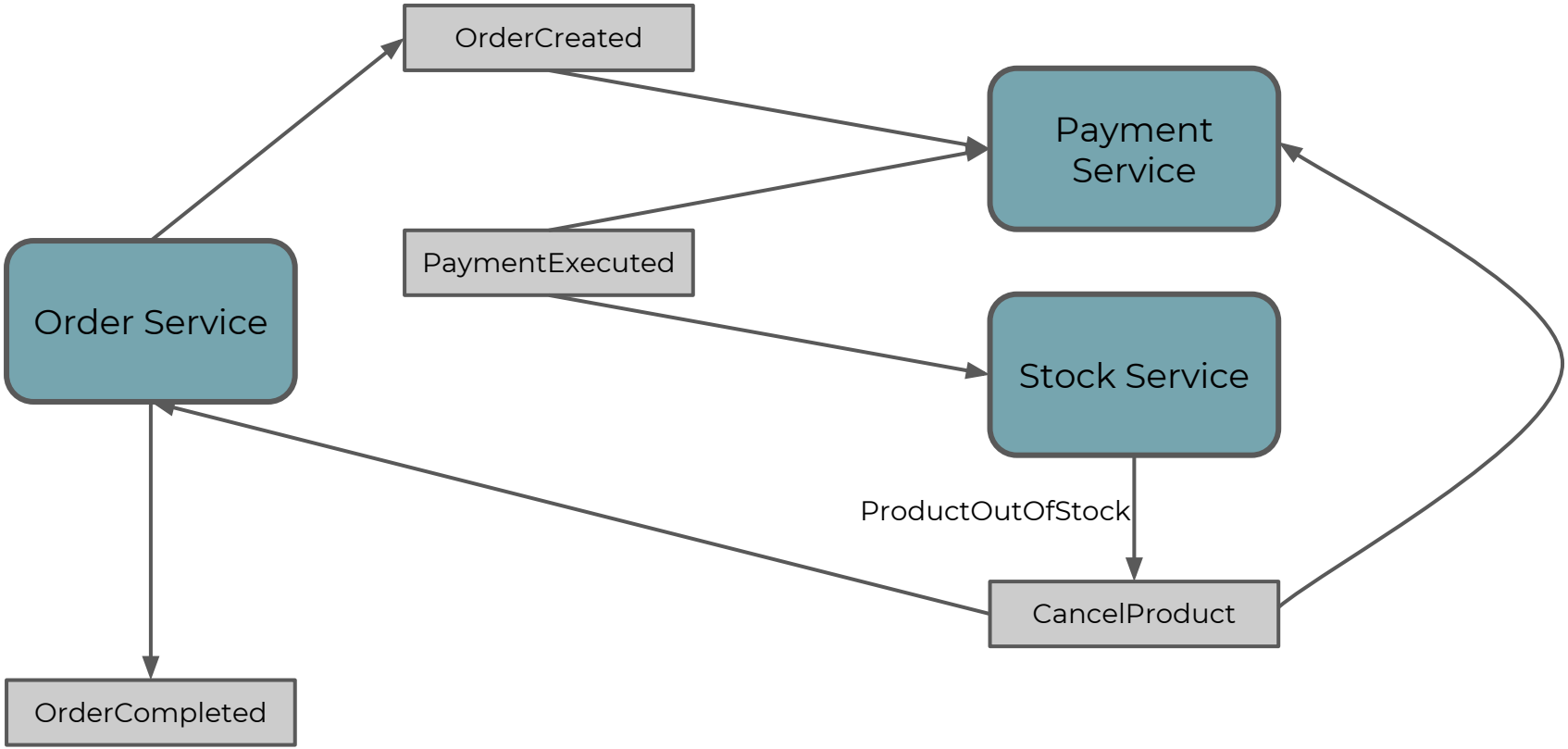
Events/Choreography – no central coordination. Each service produces and listen to other service's events.

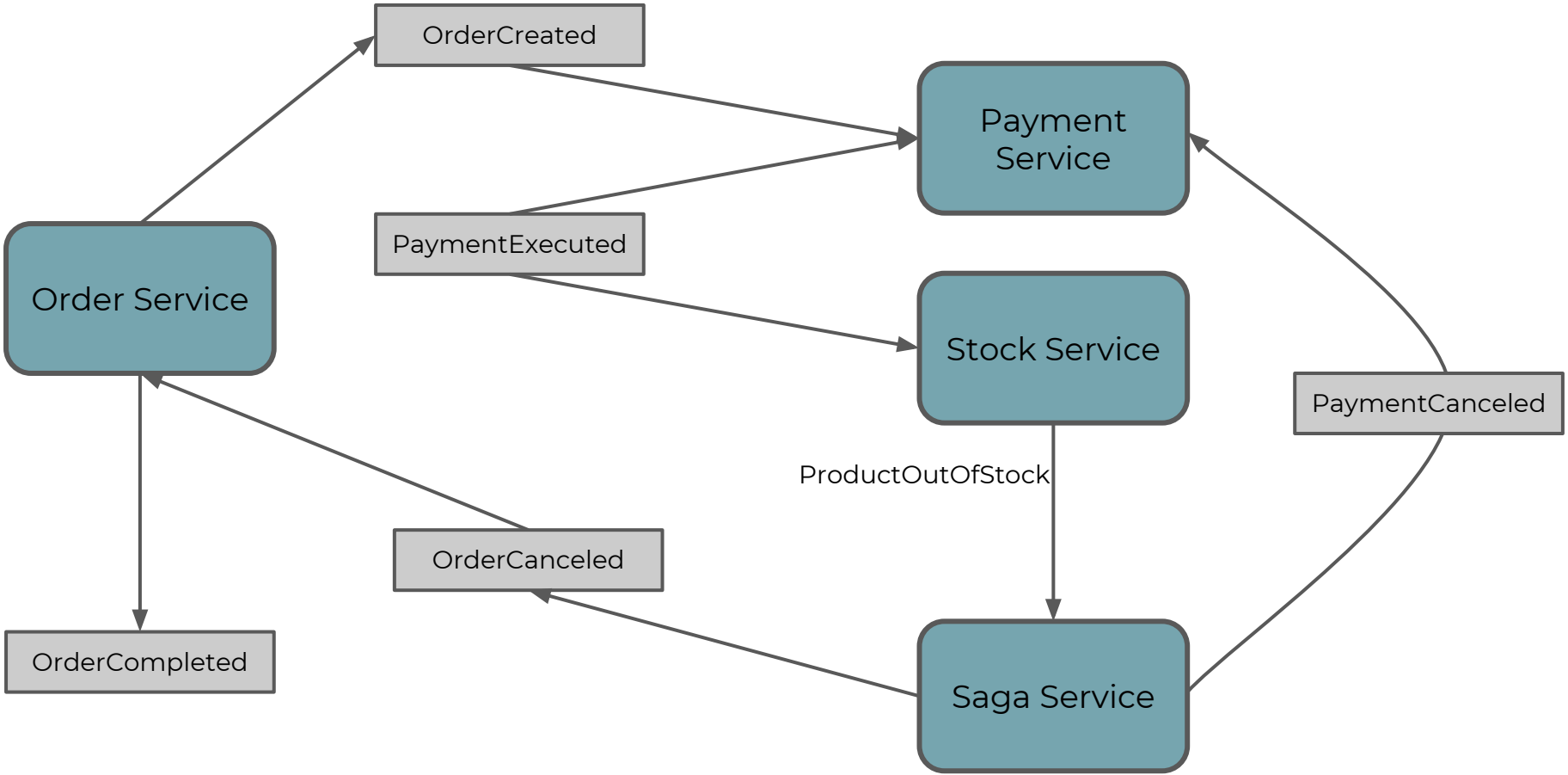
Command/Orchestration – with coordinator service responsible for business decision making.


Command Controller
Command Bus
Command Handler
Event Bus
Event Logger
Event Router
Event Handler
Saga
Request
Command
Event
Exception
Event
Ok
Event
Error


Order Service
Payment Service
Stock Service
Delivery Service
OrderCreated
PaymentExecuted
OrderBooked
OrderSent
OrderCompleted


OrderCreated
Payment Service
PaymentExecuted
Order Service
Stock Service
OrderBooked
Delivery Service
OrderCompleted
OrderSent


OrderCreated
Payment Service
PaymentExecuted
Order Service
Stock Service
ProductOutOfStock
CancelProduct
OrderCompleted


OrderCreated
Payment Service
PaymentExecuted
Order Service
Stock Service
PaymentCanceled
ProductOutOfStock
OrderCanceled
OrderCompleted
Saga Service

CODE

Summary

Event-driven architecture – uses events.

CQRS – Pattern to segregate responsibilities into objects.

Event Sourcing – logging events to event store.

CQRS and Event Sourcing are not full-blown architectures. They are patterns which well apply into **several problems** and as **parts of the system**. It is not good solution for everything. Mixing architecture is good!

Benefits

- Loose coupling
- Isolated parts
- Easy to change
- Small components
- Scalable
- Natural audits
- Better division of work in team
- False updates problem removed

Disadvantages

- More initial work (be careful if it’s worth it)
- Eventual consistency (Saga)
- Versionings of events is hard
- Discovering events
- Hard to start
- Different unfamiliar style of programming

Materials

Event Store

The open-source, functional database with Complex Event Processing in JavaScript.

Out of a box:

- Events
- Event Streams
- Event Projections
- Http api, clients in different languages
- Open Source
- Available as cluster

Materials

That I used and I recommend

https://www.youtube.com/watch?v=JHGkaShoyNs

https://martinfowler.com/bliki/CQRS.html

https://martinfowler.com/eaaDev/EventSourcing.html

https://martinfowler.com/bliki/BoundedContext.html

https://martinfowler.com/bliki/CommandQuerySeparation.html

http://highscalability.com/

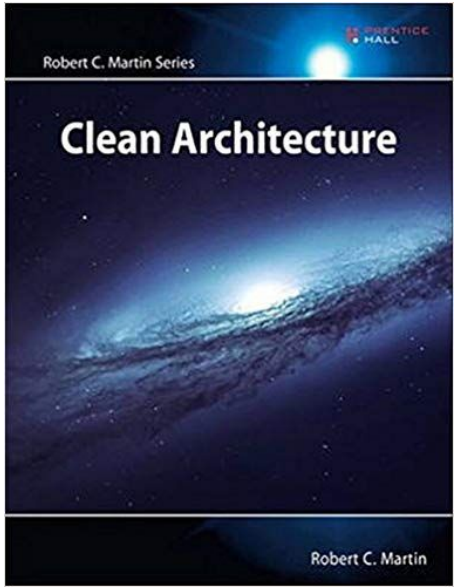
https://eventuate.io/whyeventsourcing.html

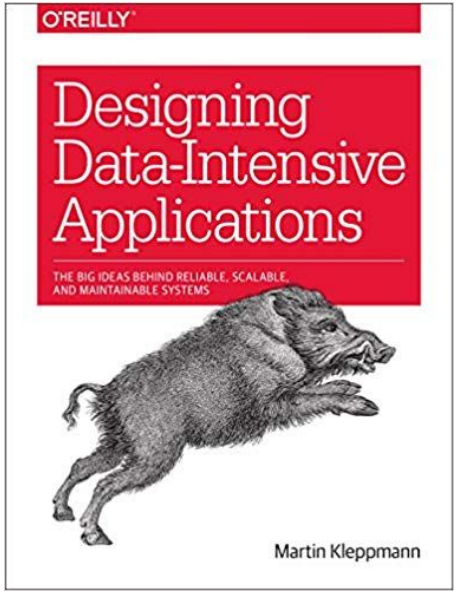
https://www.youtube.com/watch?v=LDW0QWie21s

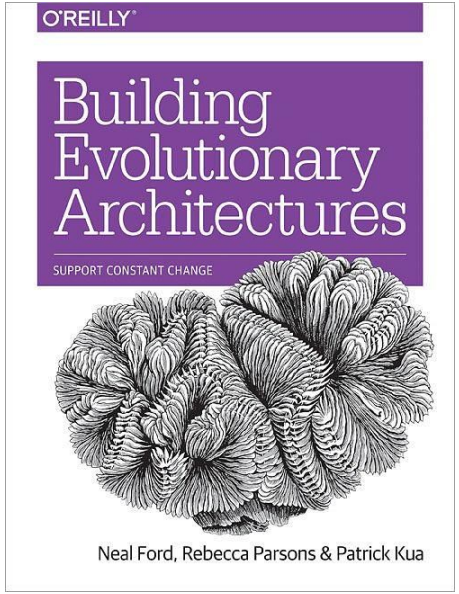
https://www.youtube.com/watch?v=8JKjvY4etTY

https://bulldogjob.pl/articles/122-cqrs-i-event-sourcing-czyli-latwa-droga-do-skalowalnosci-naszych-systemow_

https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf


Robert C. Martin Series
Clean Architecture
Robert C. Martin


O'REILLY
Designing Data-Intensive Applications
THE BIG IDEAS BEHIND RELIABLE, SCALABLE, AND MAINTAINABLE SYSTEMS
Martin Kleppmann


O'REILLY
Building Evolutionary Architectures
SUPPORT CONSTANT CHANGE
Neal Ford, Rebecca Parsons & Patrick Kua

All Things Distributed

Werner Vogels' weblog on building scalable and robust distributed systems.

Ciao Milano! – An AWS Region is coming to Italy!

By Werner Vogels on 13 November 2018 12:00 AM PST | Comments (0)

Contact Info

Werner Vogels

CTO - Amazon.com

My name is Martin Fowler: I'm an author, speaker, and loud-mouth on the design of enterprise software. This site is dedicated to improving the profession of software development, with a focus on skills and techniques that will last a developer for most of their career. I'm the editor of the site and the most prolific writer. It was originally just my personal site, but over the last few years many colleagues have written excellent material that I've been happy to host here. I work for ThoughtWorks, a really rather good software delivery and consulting company. To find your way around this site, go to the intro guide.

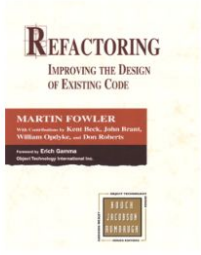
News and Updates

My **atom feed** (RSS) announces any updates to this site, as well as various news about my activities and other things I think you may be interested in. I also make regular announcements via my **twitter** feed, which I copy to my facebook page.

photostream 118

Sat 27 Oct 2018 10:22 EDT

Refactoring



Refactoring has become a core skill for software developers, it is the foundation behind evolutionary architecture and modern agile software development. I wrote the original book on refactoring in 2000, and it continues to be an interest of mine.

I've recently posted several essays on refactoring here:

photo: Manuel Gomez Dardenne

Workshops

Workshops

Create little event sourcing based web app (you can use a little helper from github.com/piotrowy/espeo-training-java). The application should contain at least:

- One long process.
- Three domains.
- Each domain should contain at least one event.

*Implement saga solution.

Exercise

Implement car rental service. Each user can have two roles: lender, borrower. The system should be able to:

1. Registrate new borrower or lender.
2. List and search all available cars.
3. List and search all rented cars.
4. Perform borrow and lend operation.
5. Send notification.

Thank you!

Piotr Ceranek