SPRINGONE2GX
WASHINGTON, DC

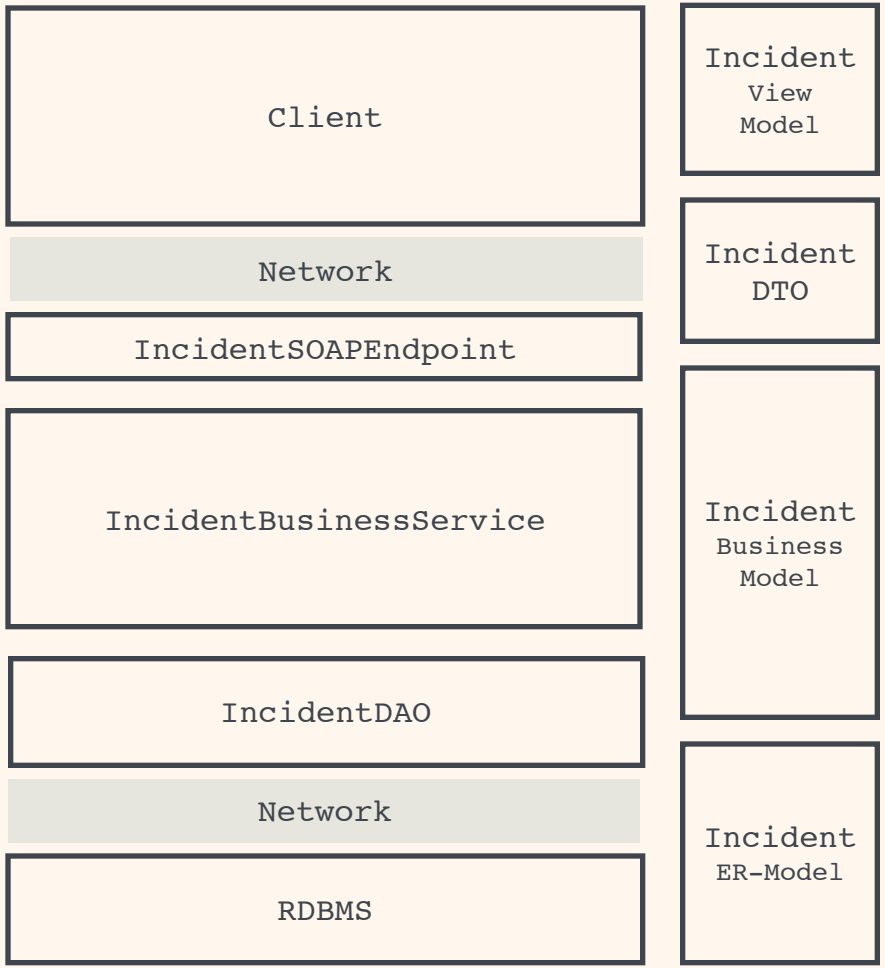
Building Microservices with Event Sourcing and CQRS

Michael Ploed - innoQ
@bitboss





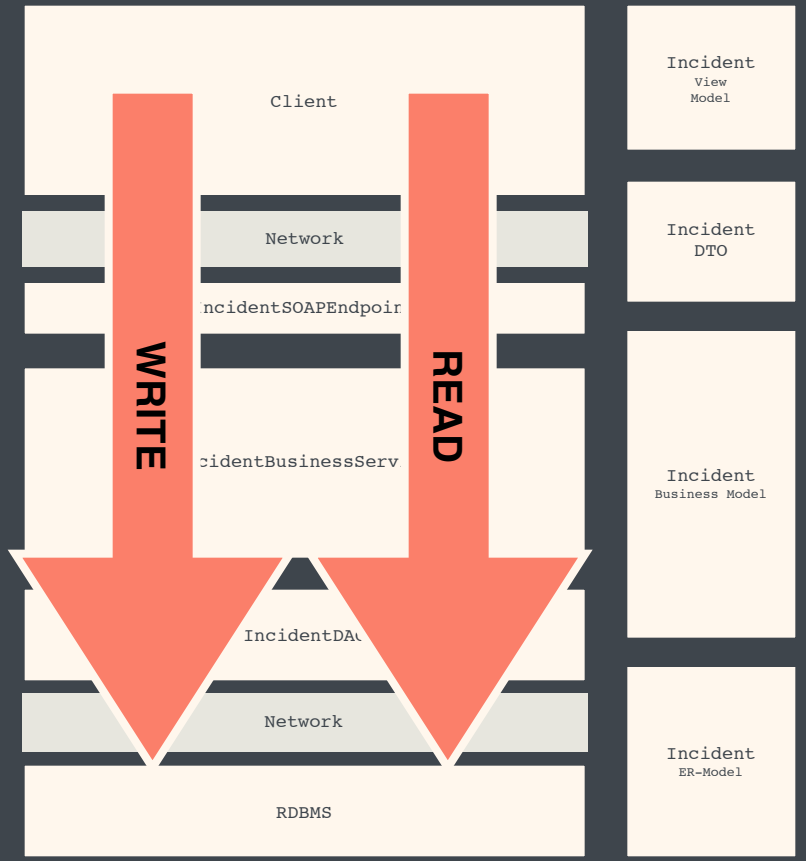
Let's review the classical old school N-Tier architecture



Characteristics

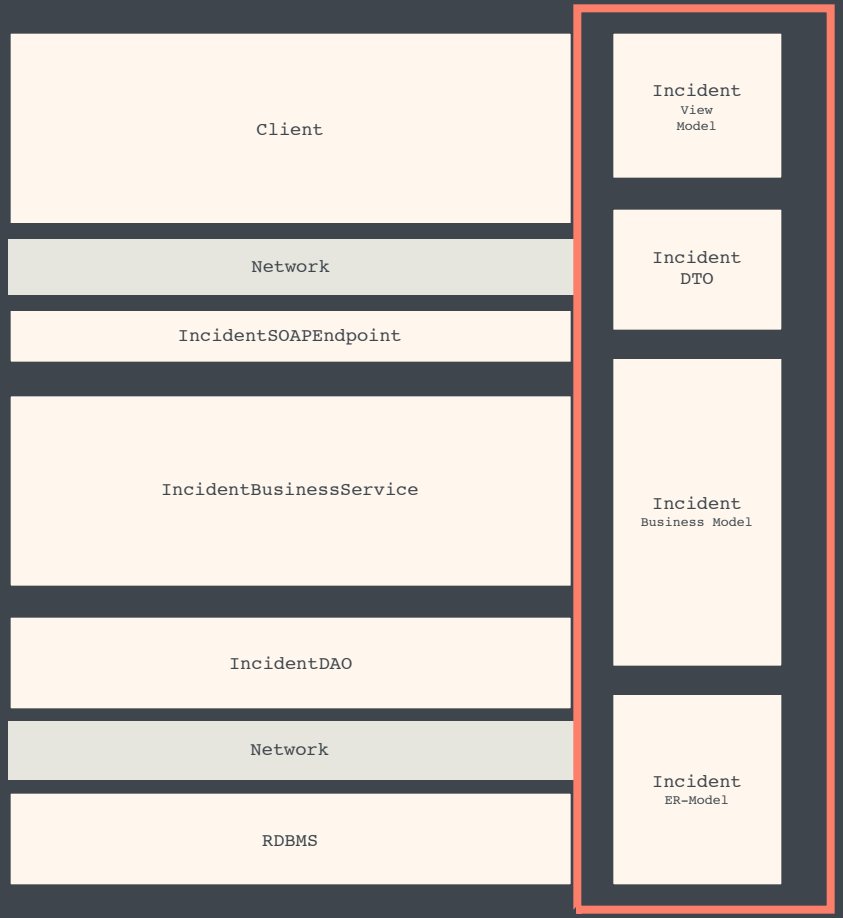
1

We read and write data through the same layers



2

We use the same model for read and write access

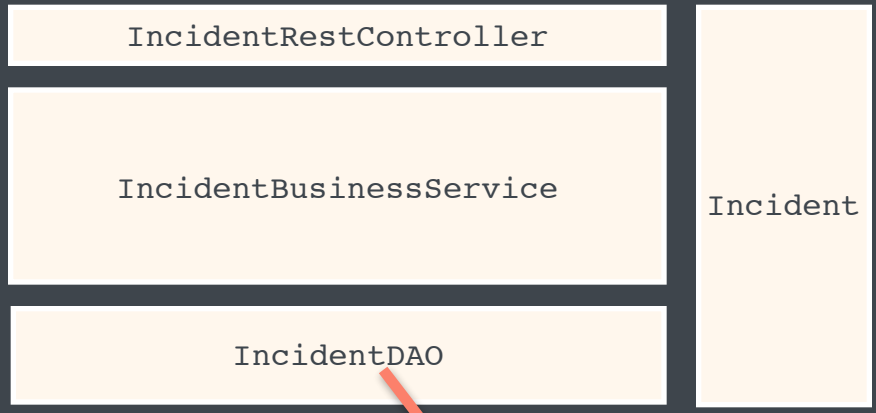


3

We use coarse grained deployment units that combine read and write code

4

We change datasets directly



ID	USER_ID	DATE	TEXT
1	23423	11.03.2014	**Mouse is broken**
2	67454	12.03.2014	EMail not working
3	93729	12.03.2014	Office license
…	…	…	…

We've done that for ages already and it's proven

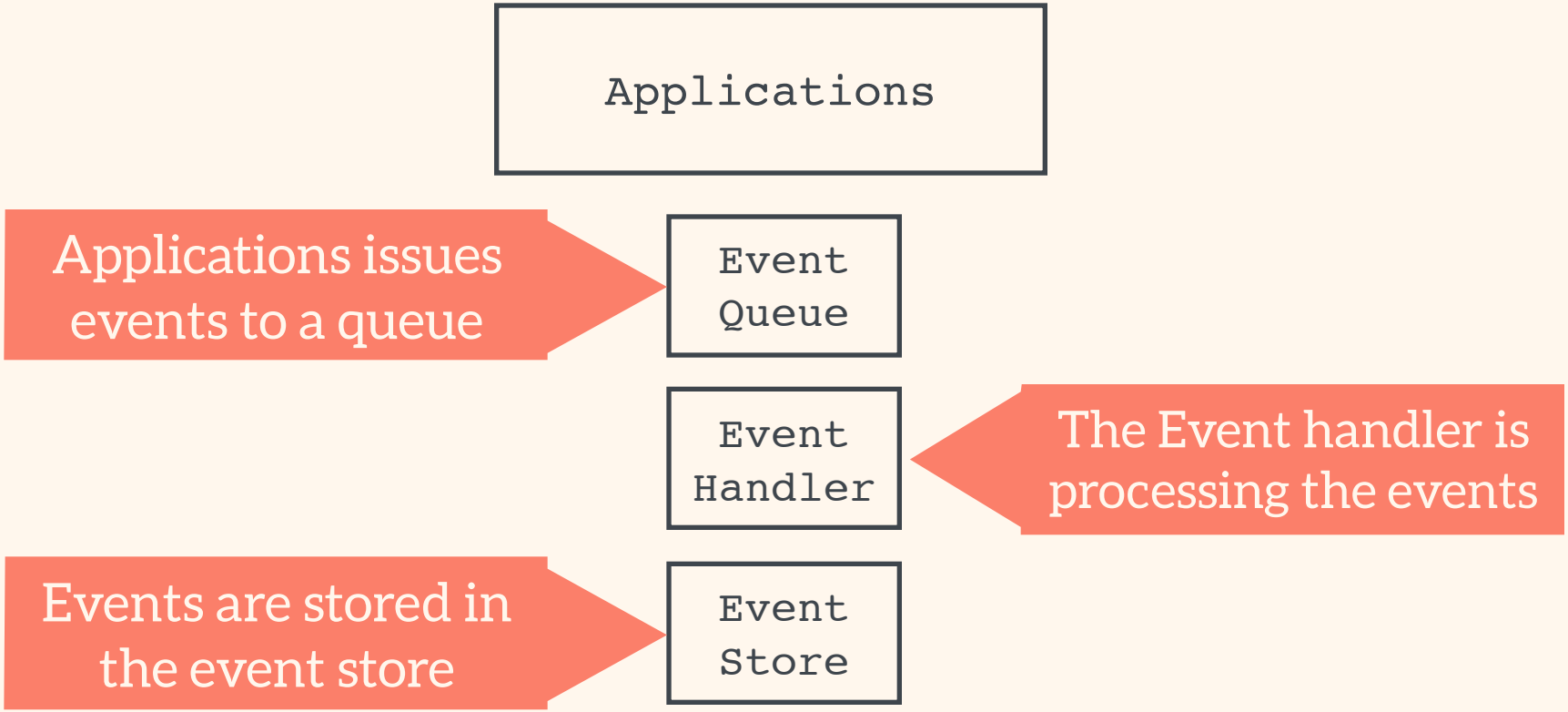
Many applications will run smooth and fine with this kind of approach

However there are drawbacks to this kind of architecture

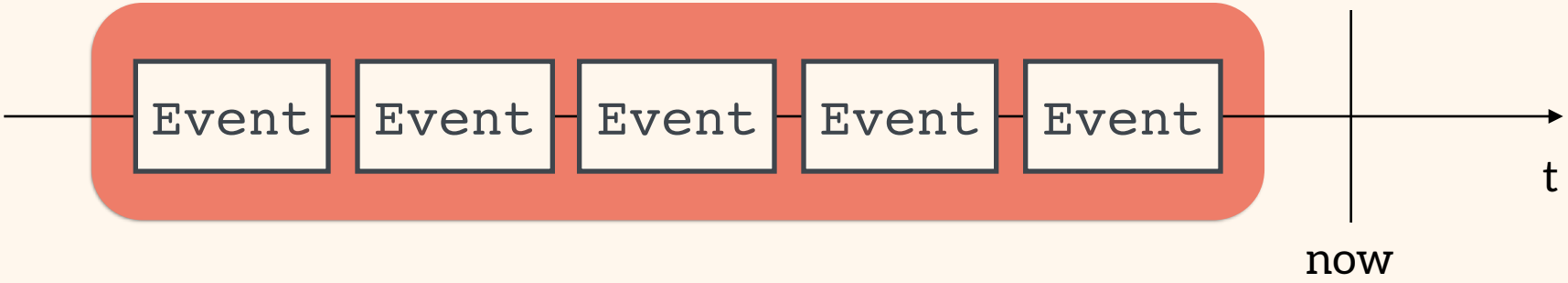
1. *The data model is a compromise*
2. *You can't scale read and write independently*
3. *No data history, no snapshots, no replay*
4. *Tendency to a monolithic approach*

Event Sourcing is an architectural pattern in which the state of the application is being determined by a sequence of events

Building Blocks



The sequence of events in the queue is called event stream



Event Stream Example

IncidentCreatedEvent

```
incidentNumber: 1
userNumber: 23423
timestamp: 11.03.2014 12:33:33
text: „Mouse broken“
status: „open“
```

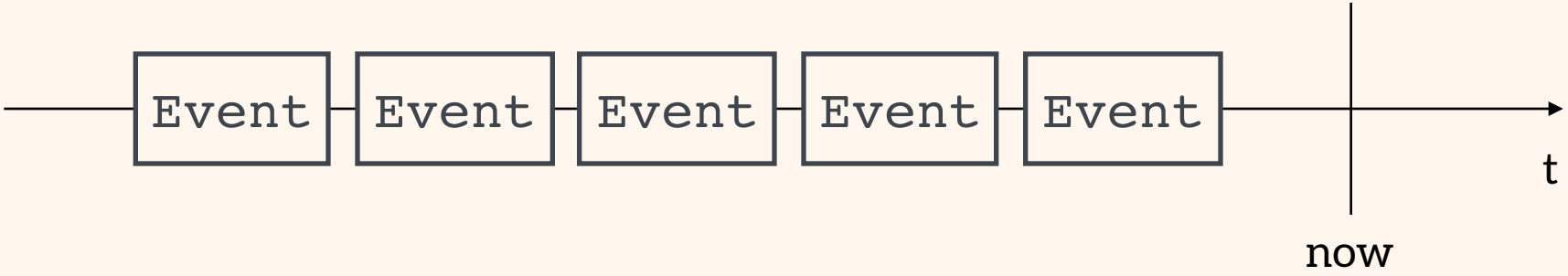
IncidentTextChangedEvent

```
incidentNumber: 1
text: „Left button of mouse broken“
```

IncidentClosedEvent

```
incidentNumber: 1
solution: „Mouse replaced“
status: „closed“
```

An event is something that happened in the past



The names of the events are part of the

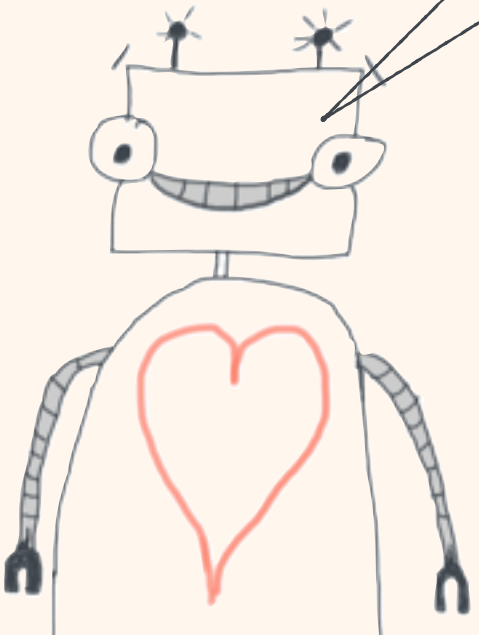
Ubiquitous Language

D D D

ShipmentDeliveredEvent

CustomerVerifiedEvent

CartCheckedOutEvent



CreateCustomerEvent

WillSaveItemEvent

DoStuffEvent

Code Example

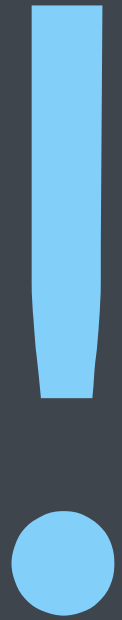
```
public class CustomerVerifiedEvent {
  private String eventId;
  private Date eventDate;
  private CustomerNumber customerNumber;
  private String comment;

  public CustomerVerifiedEvent(CustomerNumber custNum,
                               String comment) {
    this.customerNumber = cusNum;
    this.comment = comment;
    this.eventDate = new Date();
  }
}
```

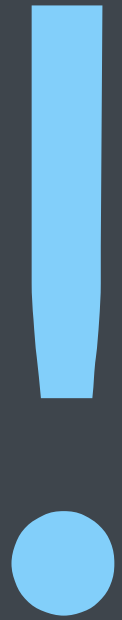
Scope your events based on

Aggregates

D D D



An Event is always immutable



There is no deletion of events

A delete is just another event

IncidentCreatedEvent

```
incidentNumber: 1
userNumber: 23423
timestamp: 11.03.2014 12:23:23
text: „Mouse is broken defekt“
status: „open“
```

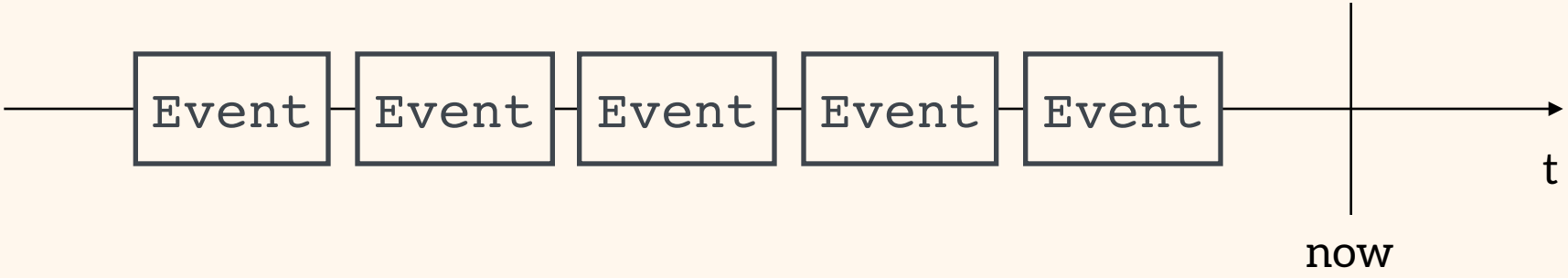
IncidentChangedEvent

```
incidentNumber: 1
text: „Maus ist Kaputt“
```

IncidentRemovedEvent

```
incidentNumber: 1
```

The event bus is usually implemented by a message broker



Let's reuse the ESB from the failed SOA project

NO

NO

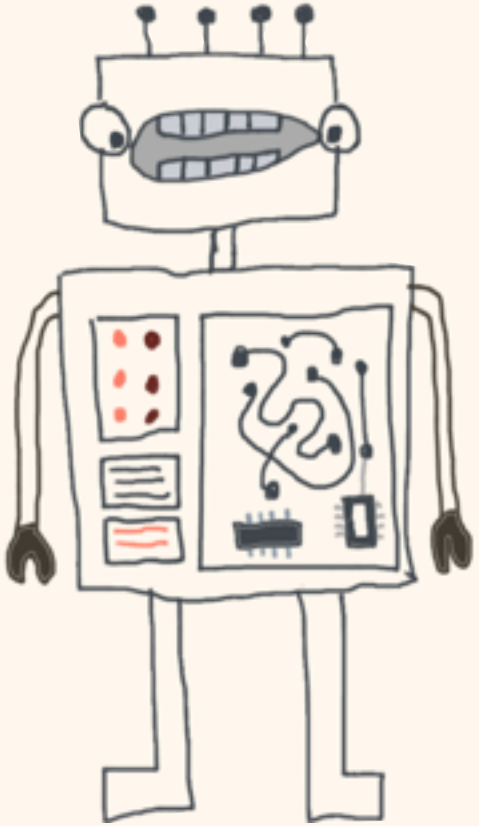
NO

Prefer dumb pipes with smart endpoints as a suitable message broker architecture

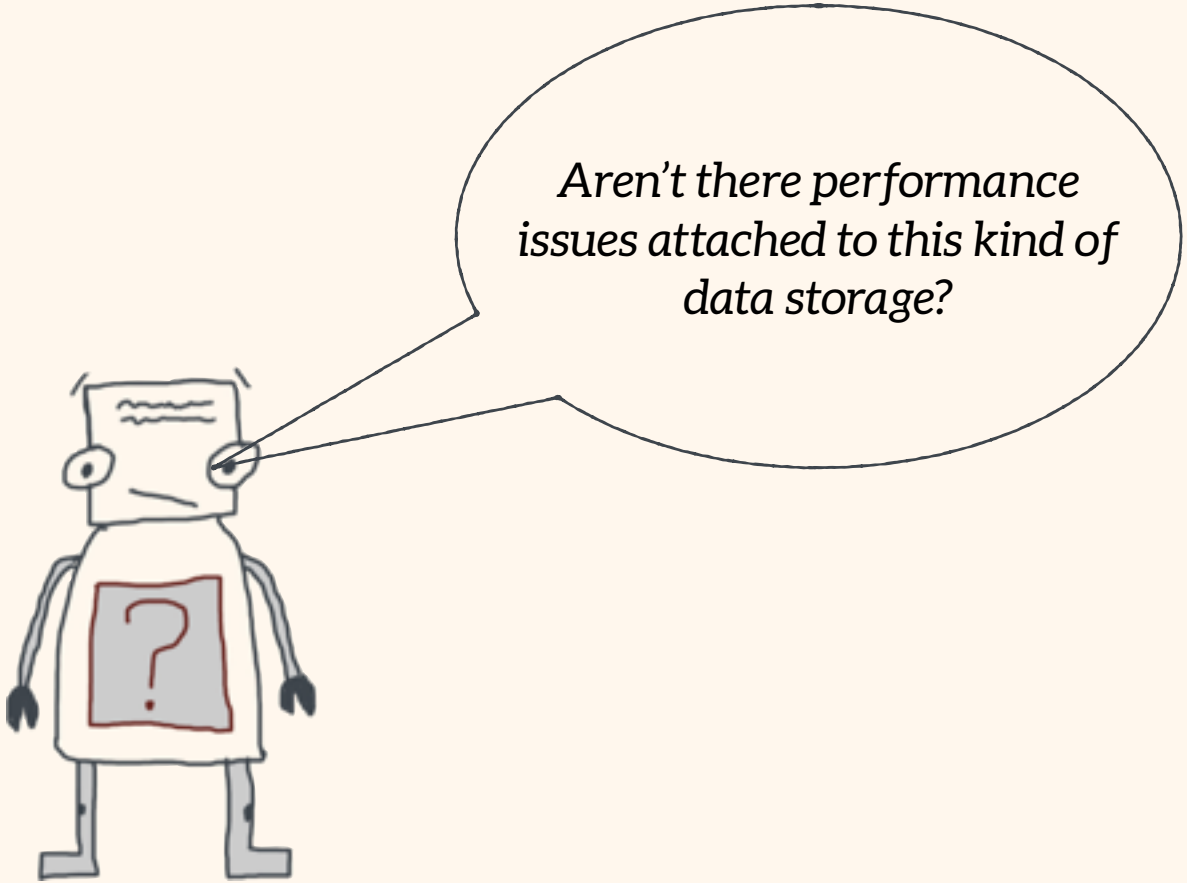
1. *Complete rebuild is possible*
2. *Temporal Queries*
3. *Event Replay*

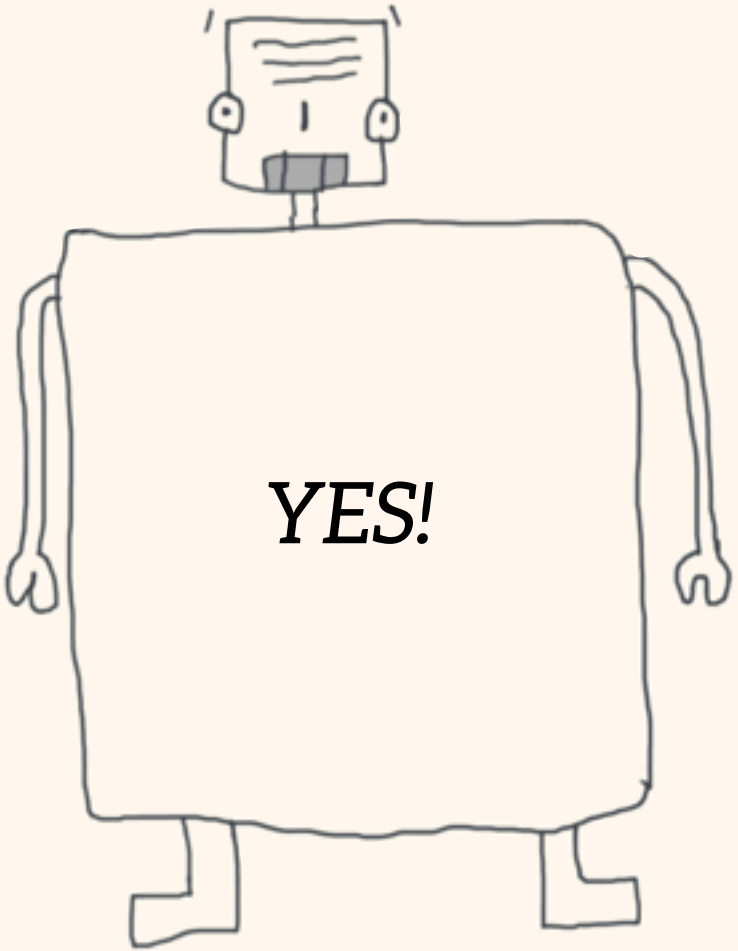
Well known examples

Version Control Systems
or
Database Transaction Logs

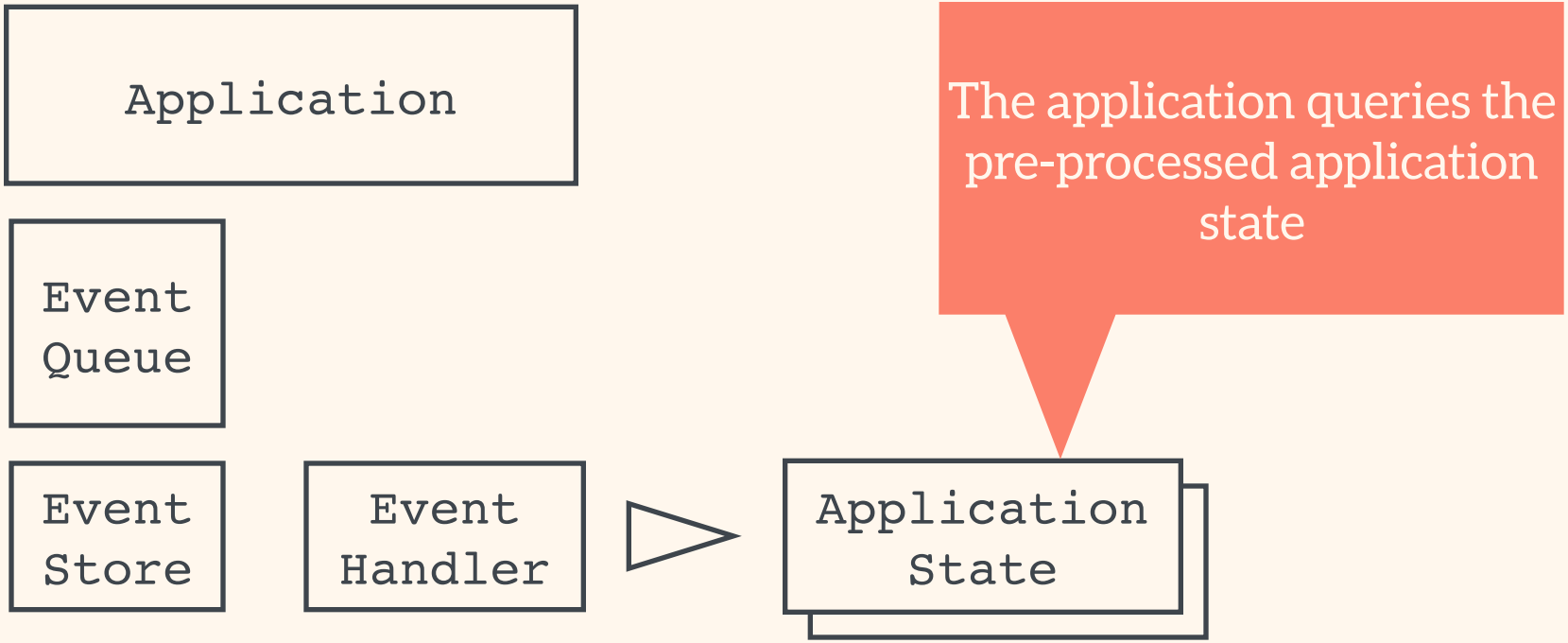


The Event Store has a very high business value


Aren't there performance issues attached to this kind of data storage?

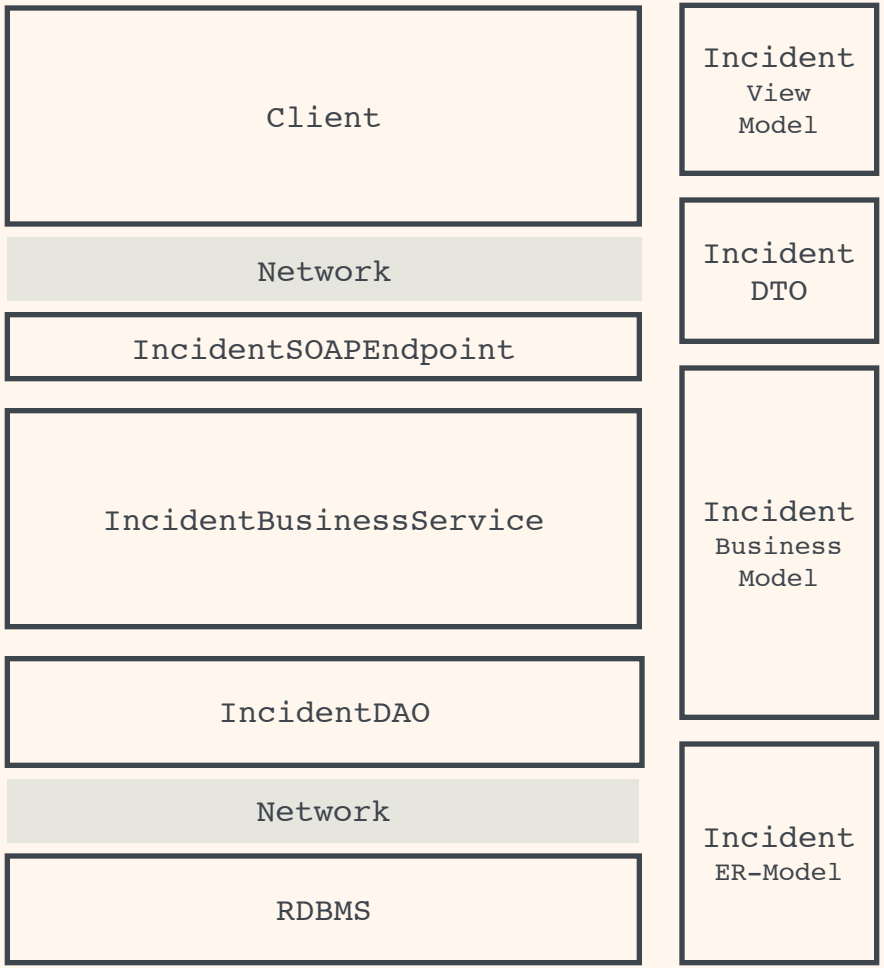

YES!

Think about application state

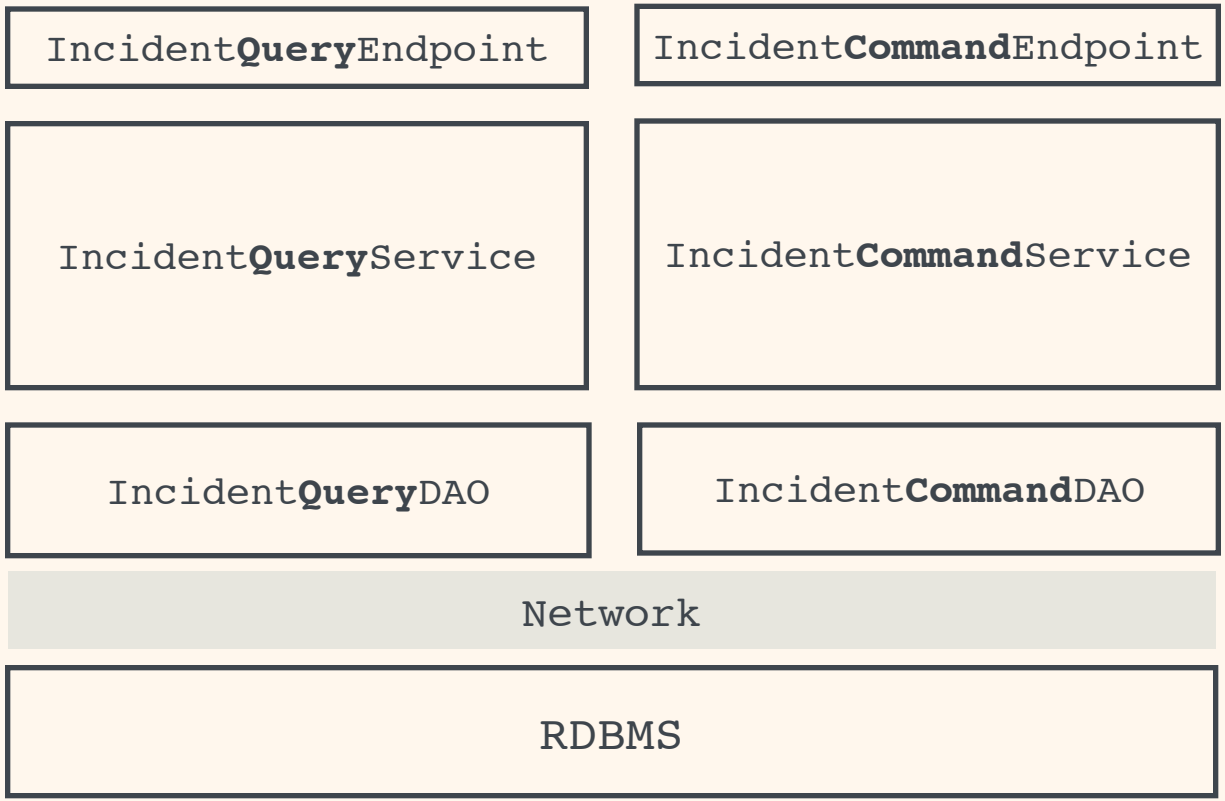


CQRS

Command
Query
Responsibility
Separation


Client
Network
IncidentSOAPEndpoint
IncidentBusinessService
IncidentDAO
Network
RDBMS
Incident View Model
Incident DTO
Incident Business Model
Incident ER-Model

Basically the idea behind CQRS is simple



Code Example

Classic Interface

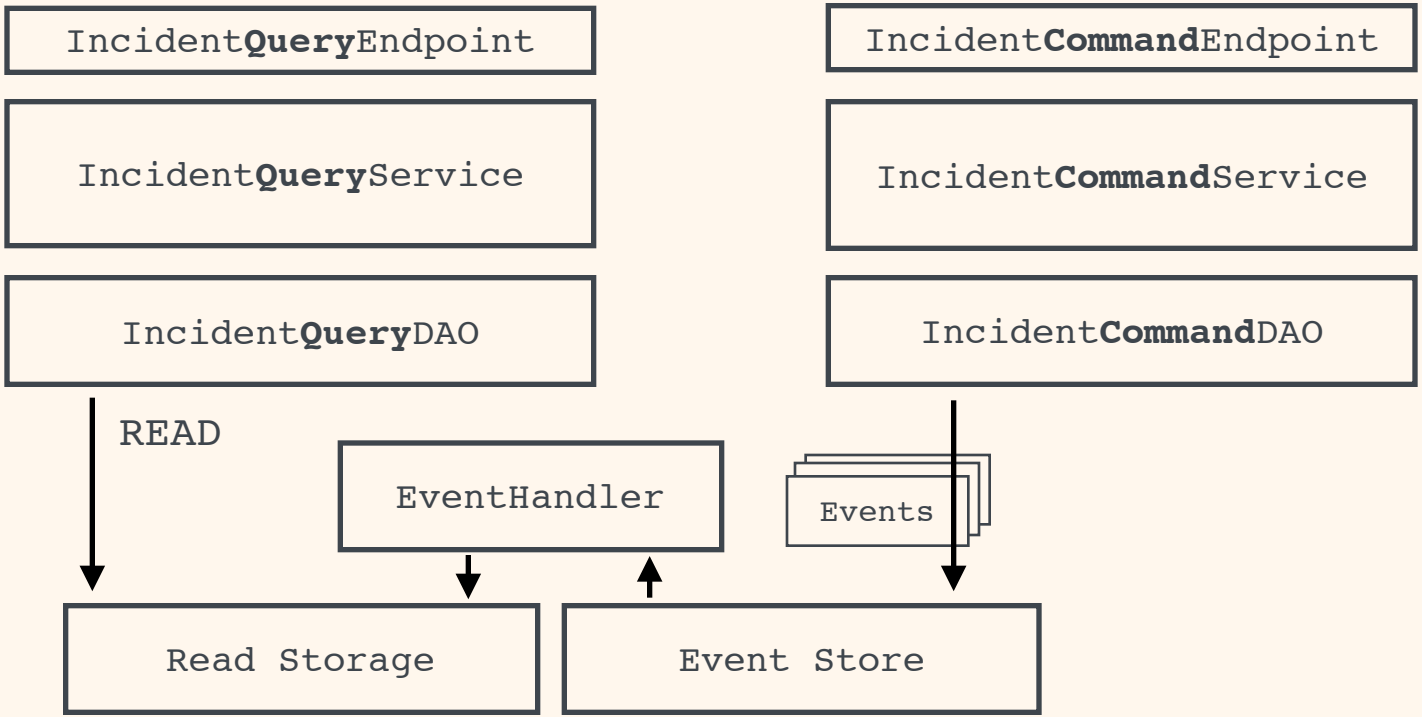
```
public interface IncidentManagementService {
      Incident saveIncident(Incident i);
      void updateIncident(Incident i);
      List<Incident> retrieveBySeverity(Severity s);
      Incident retriveById(Long id);
}
```

CQRS-ified Interfaces

```
public interface IncidentManagementQueryService {
      List<Incident> retrieveBySeverity(Severity s);
      Incident retriveById(Long id);
}
```

```
public interface IncidentManagementCommandService {
      Incident saveIncident(Incident i);
      void updateIncident(Incident i);
}
```

Event Sourcing & CQRS



1. *Individual scalability and deployment options*
2. *Technological freedom of choice for command, query and event handler code*
3. *Excellent Fit for Bounded Context (Domain Driven Design)*

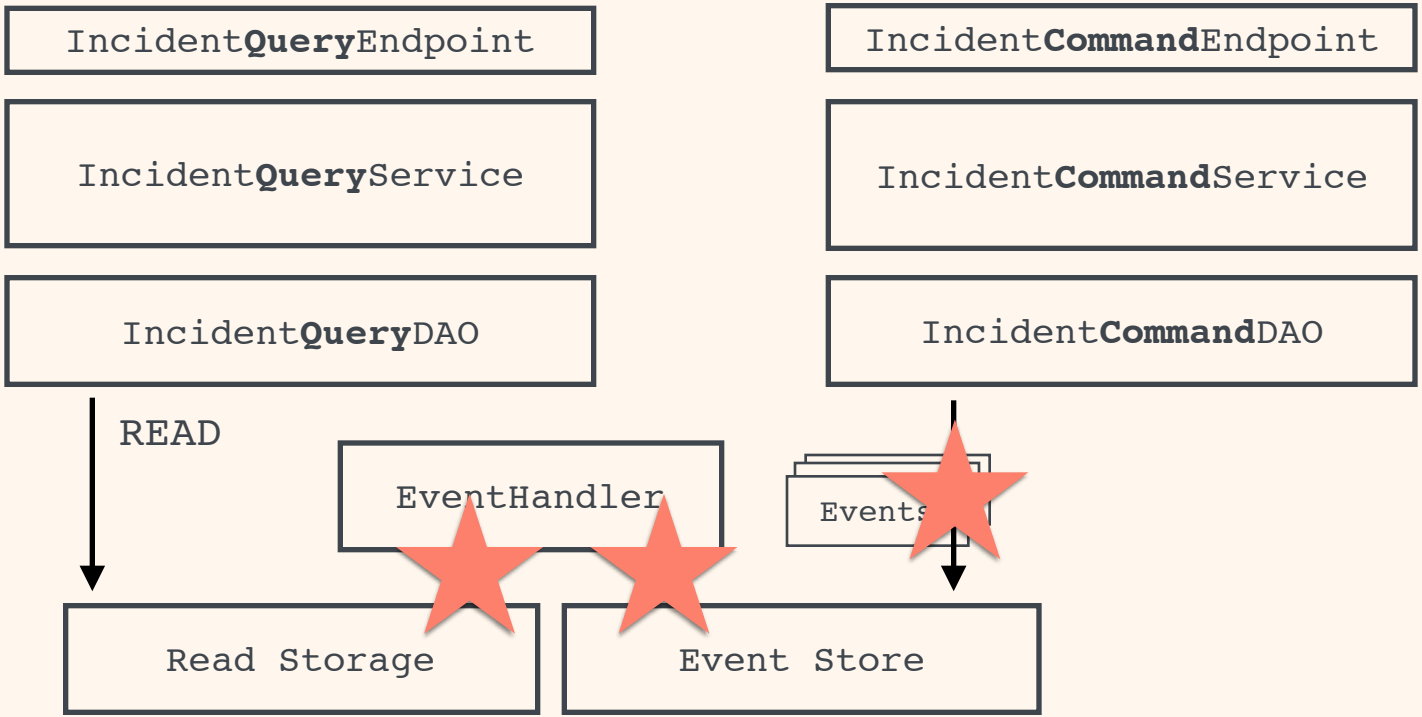
*Event Sourcing and CQRS are interesting architectural options. However there are **various challanges**, that have to be taken care of*

1. *Consistency*
2. *Validation*
3. *Parallel Updates*

YES

Systems based on CQRS and Event Sourcing are mostly eventually consistent

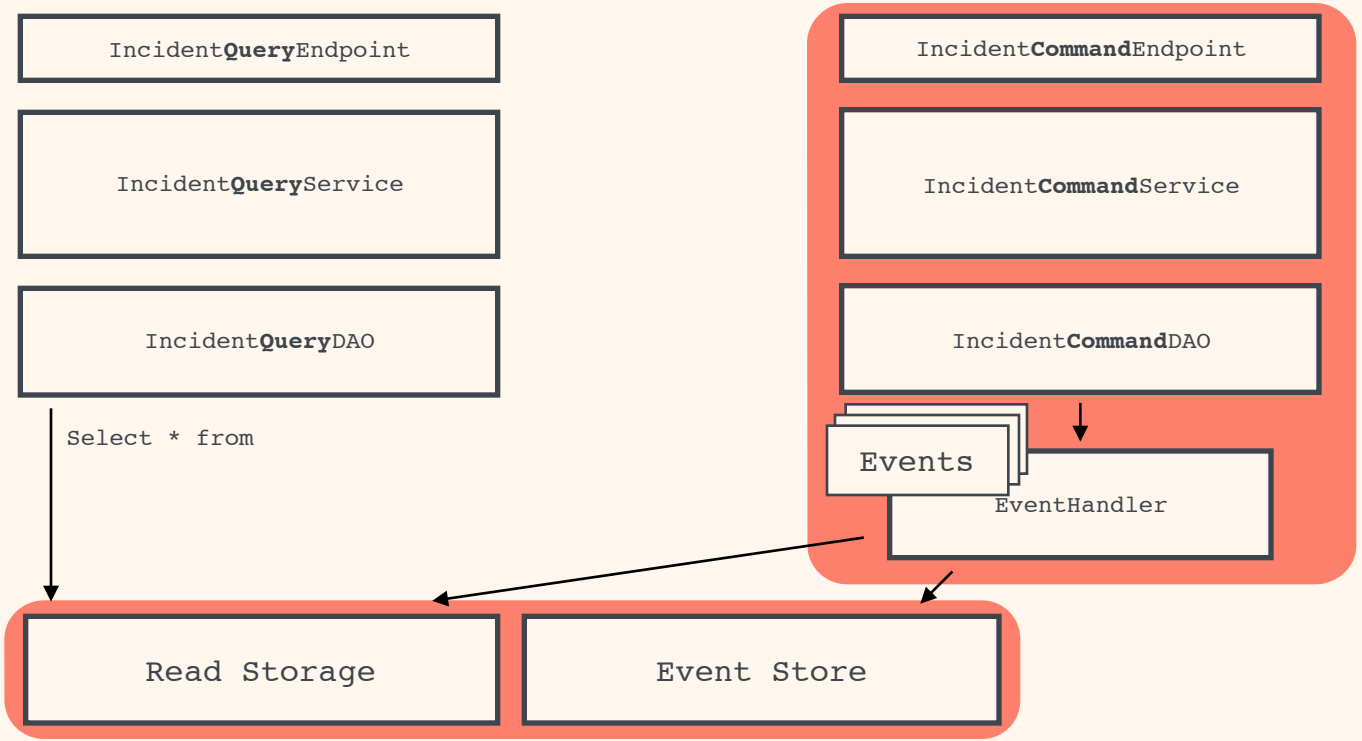
Eventual Consistency



BUT

You can build a fully consistent system which follows Event Sourcing principles

Full Consistency

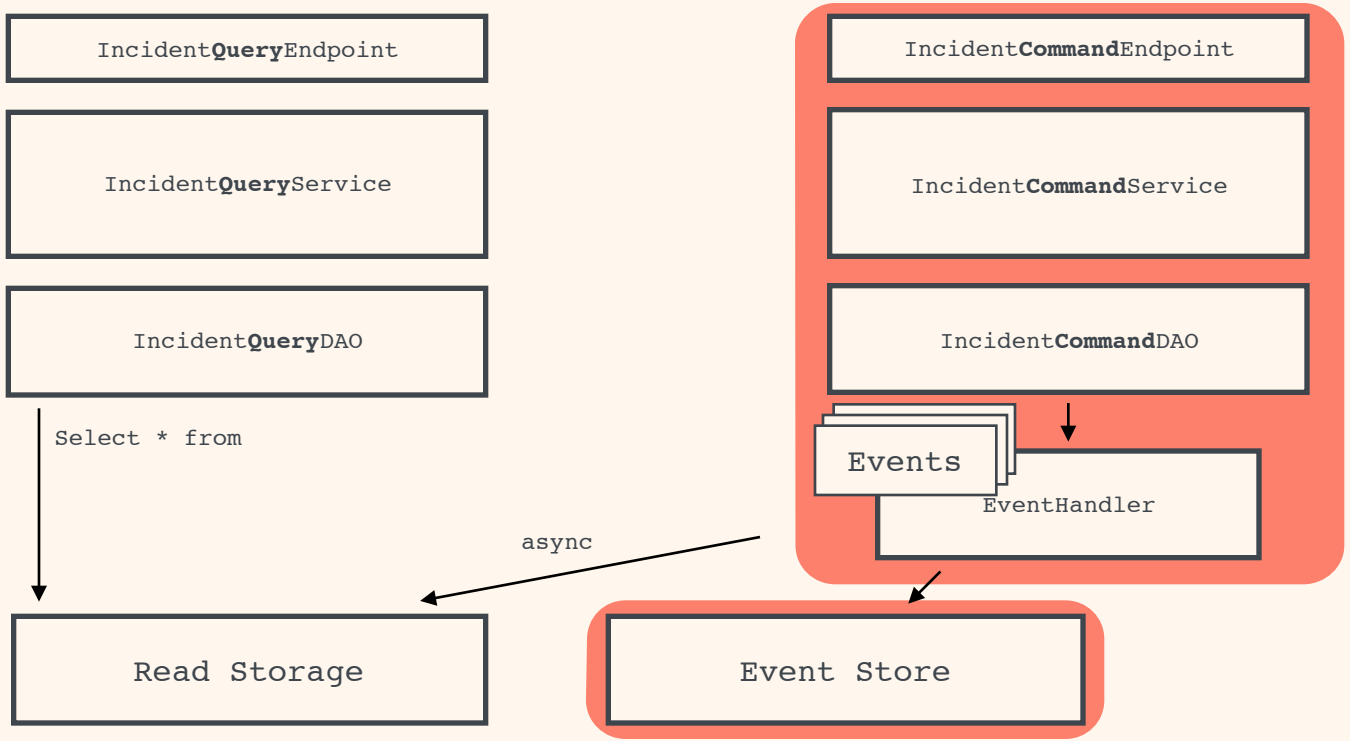


Your business domain drives the level of consistency not technology

Deeper Insight

D D D

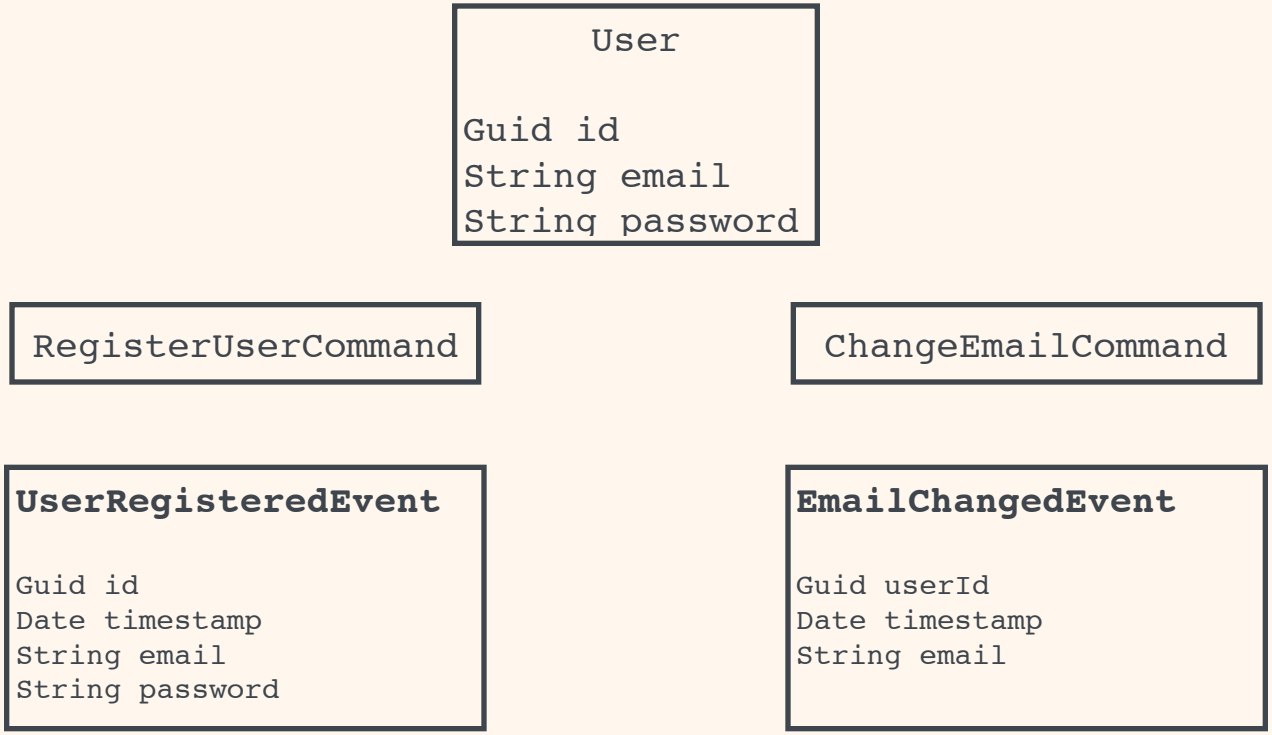
Increased (but still eventual) consistency



There is no standard solution

1. *Consistency*
2. *Validation*
3. *Parallel Updates*

Example Domain




We process 2 million+ registrations per day. A user can change her email address. However the emails address must be unique

How high is the probability that a validation fails

Which data is required for the validation

Where is the required data stored

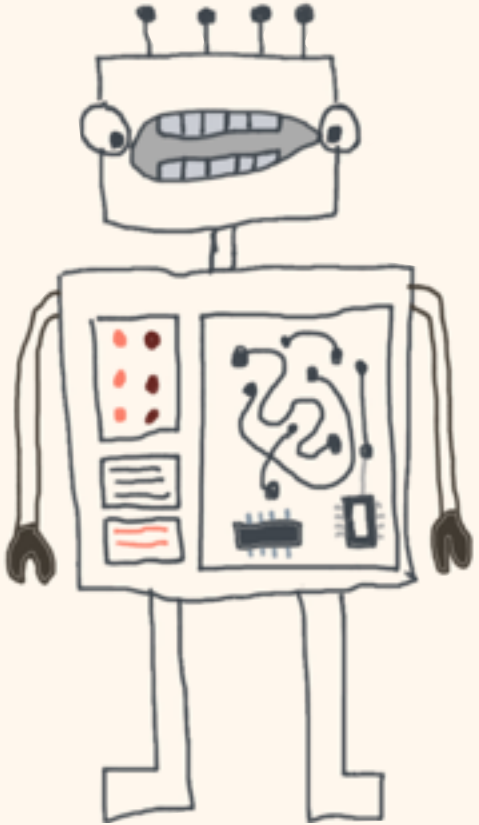
What is the business impact of a failed validation that is not recognized due to eventual consistency and how high is the probability of failure

Your business domain drives the level of consistency

Deeper Insight

D D D

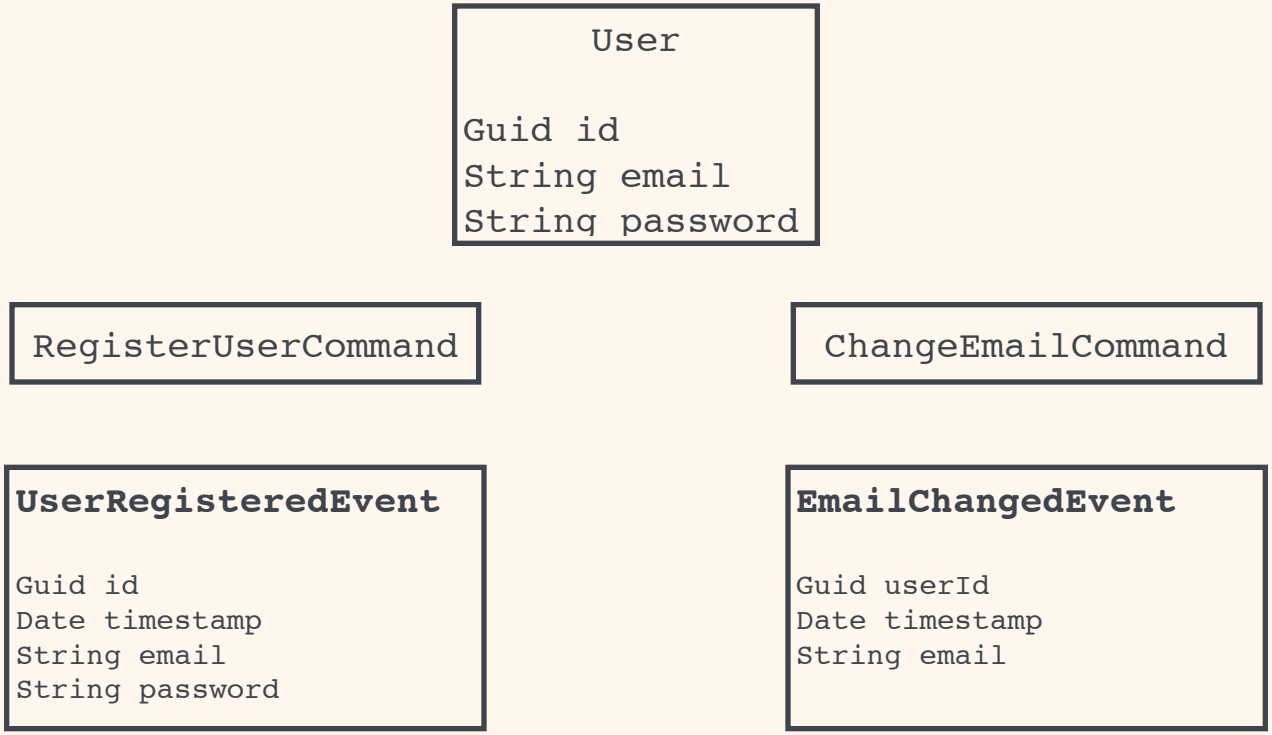
1. *Validate from Event Store*
2. *Validate from Read Store*
3. *Perform Validation in Event Handler*

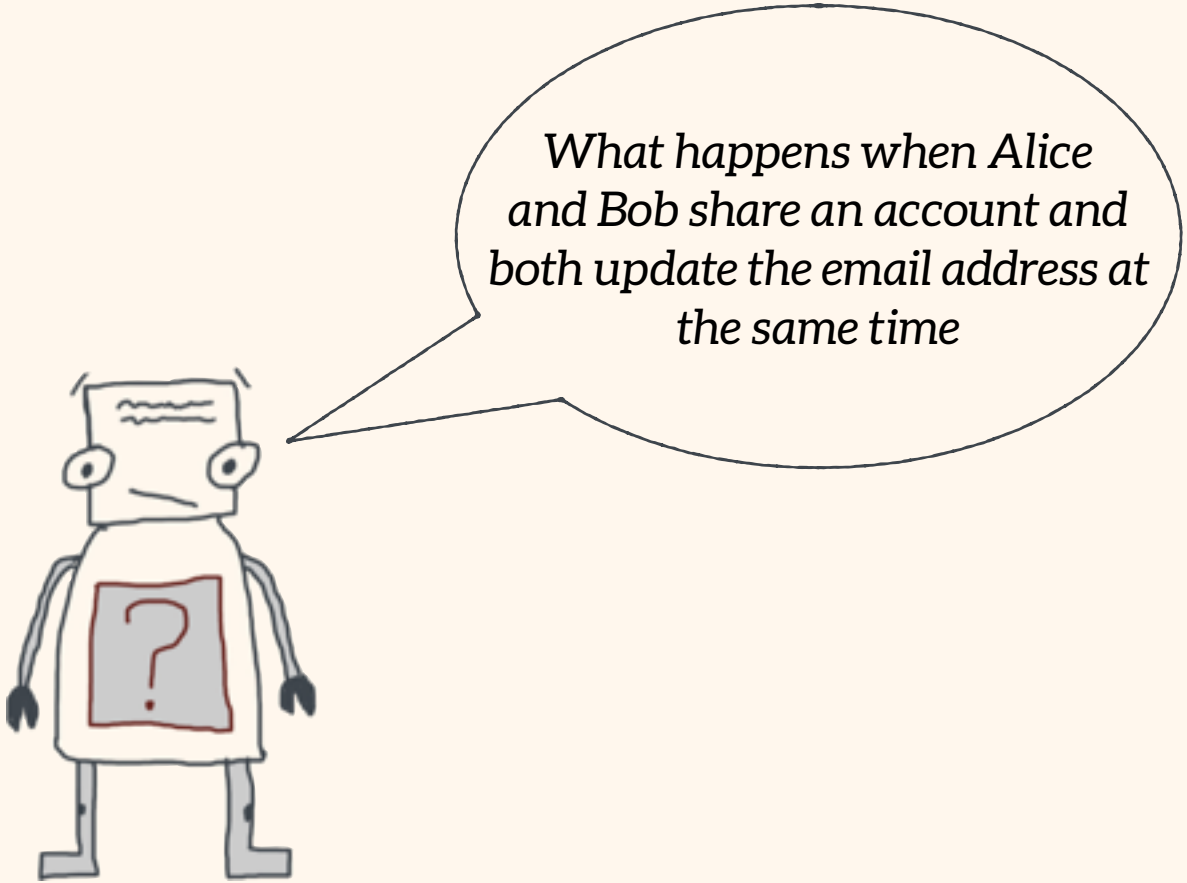


Never validate from the event store

1. *Consistency*
2. *Validation*
3. *Parallel Updates*

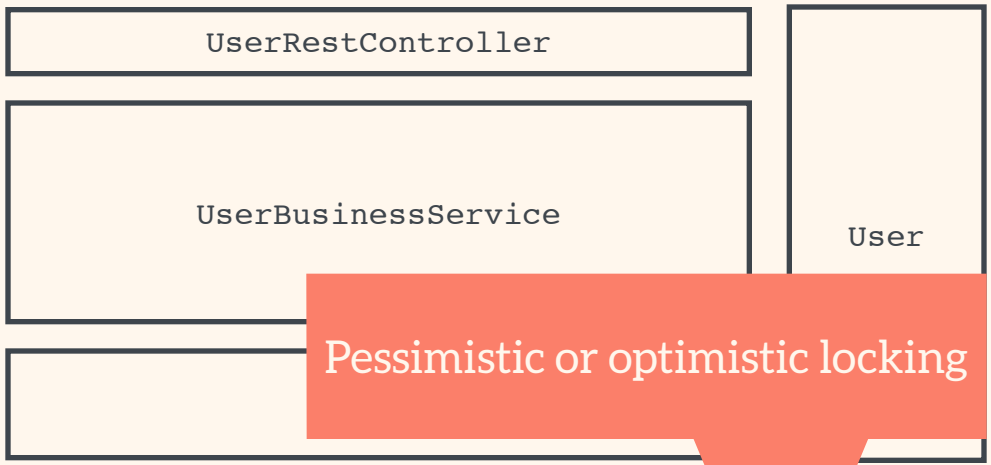
Example Domain




What happens when Alice and Bob share an account and both update the email address at the same time

What would we do in a „classic old school architecture“

Update



ID	EMAIL	PAS ORD
…	…	…
2341	alice_bob@xyz.com	lksjdaslkdjas
…	…	…

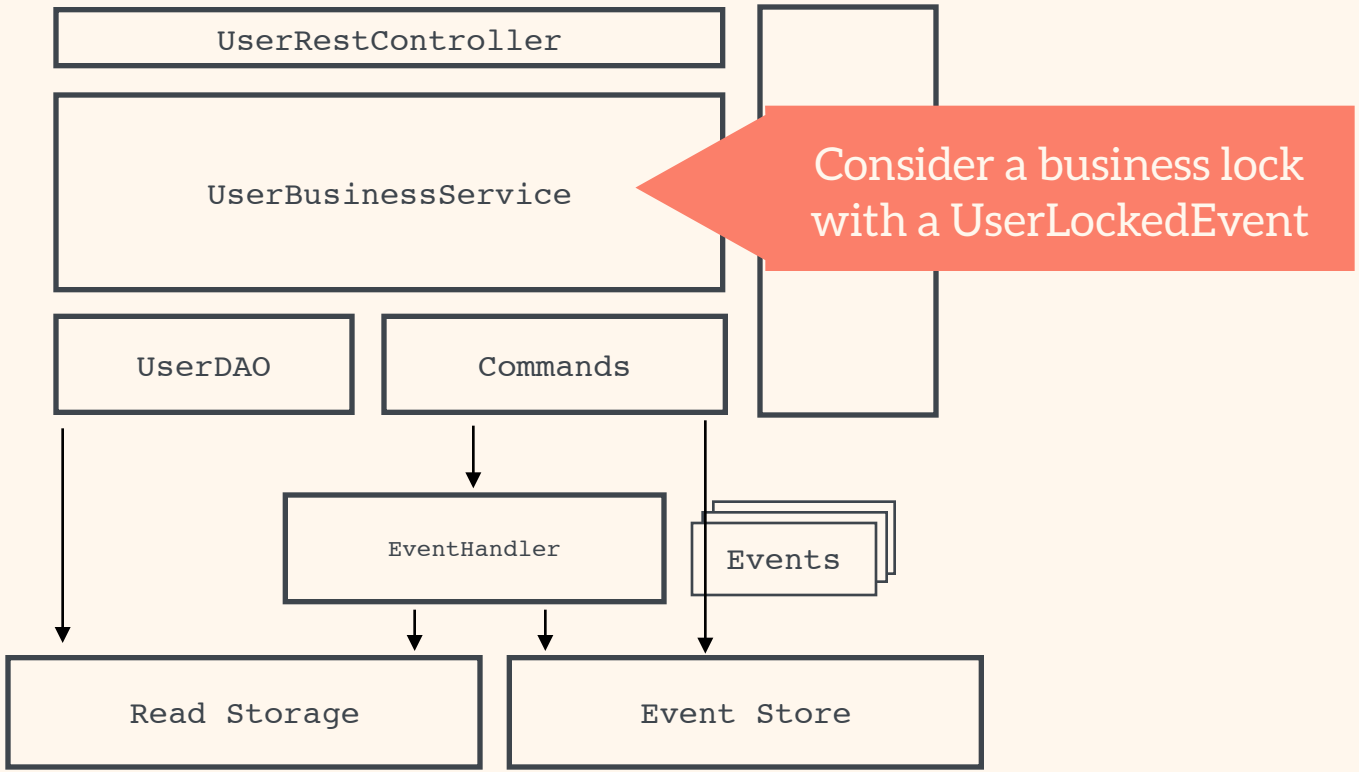
Your business domain drives the locking quality

Deeper Insight

D D D

Pessimistic locking on a data-level will hardly work in event sourcing architectures

Where to „pessimistically“ lock?

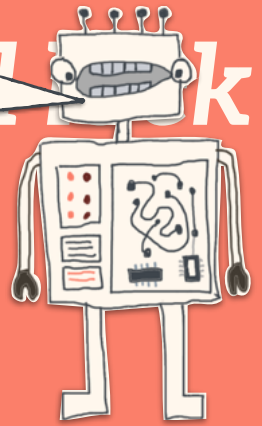


?

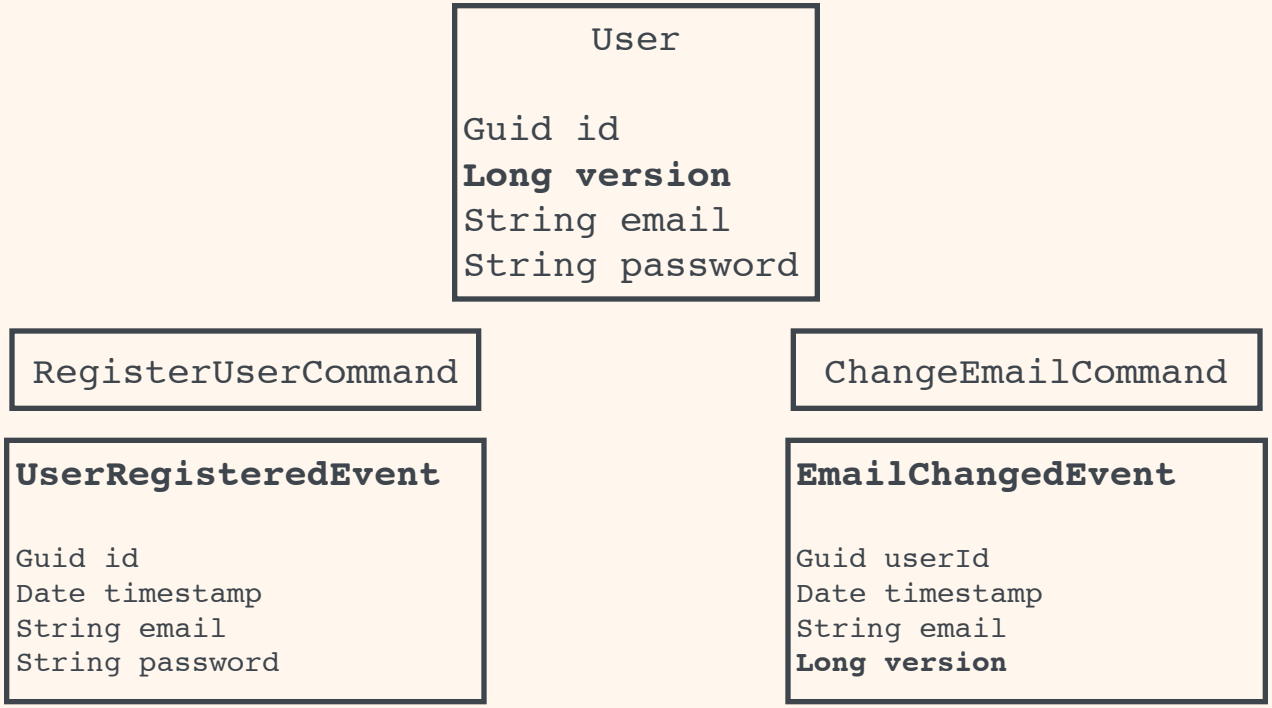
Do you

REALLY

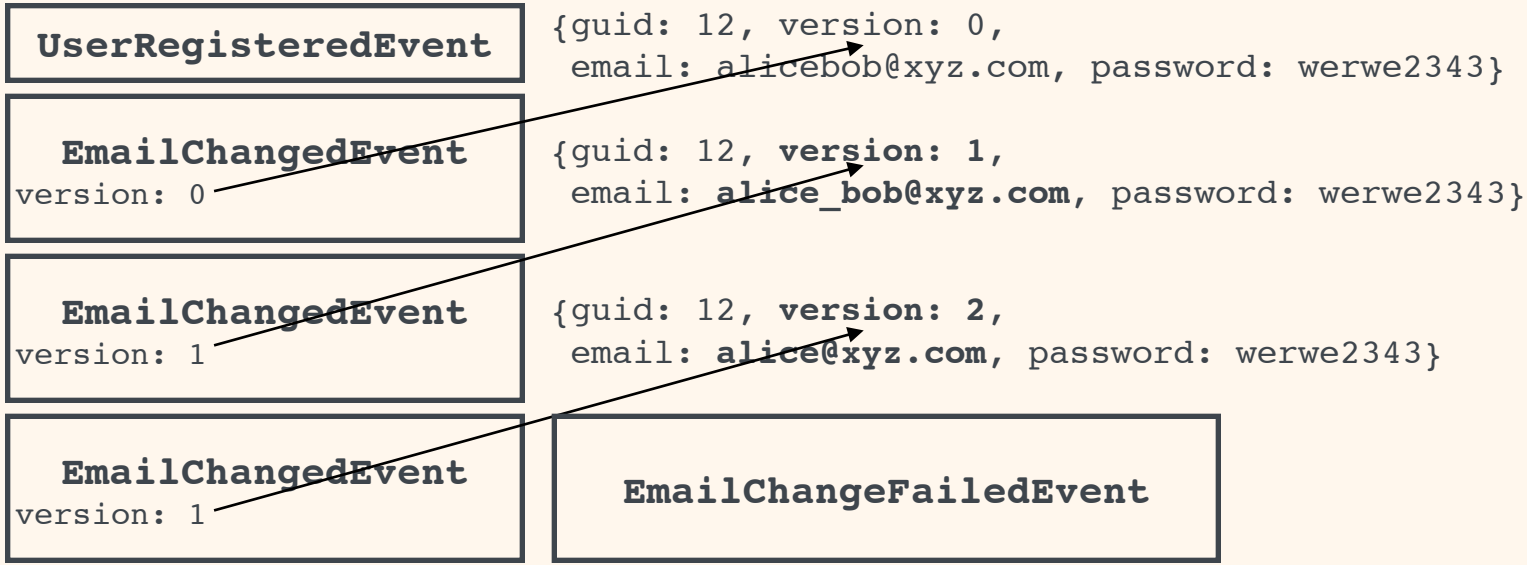
Most „classic architecture“ applications are already running fine with optimistic locks

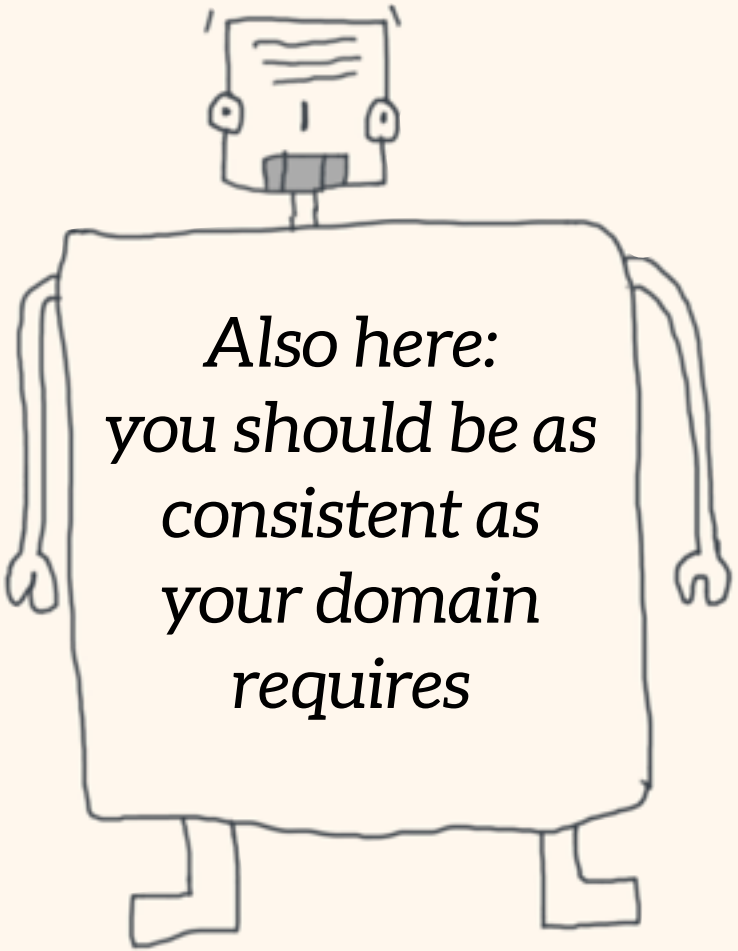


Introduce a version field for the domain entity



Each writing event increases the version




Also here:
you should be as
consistent as
your domain
requires

SPRINGONE2GX
WASHINGTON, DC

Thank You!

Michael Ploed - innoQ

Twitter: @bitboss

Slides: http://slideshare.net/mploed