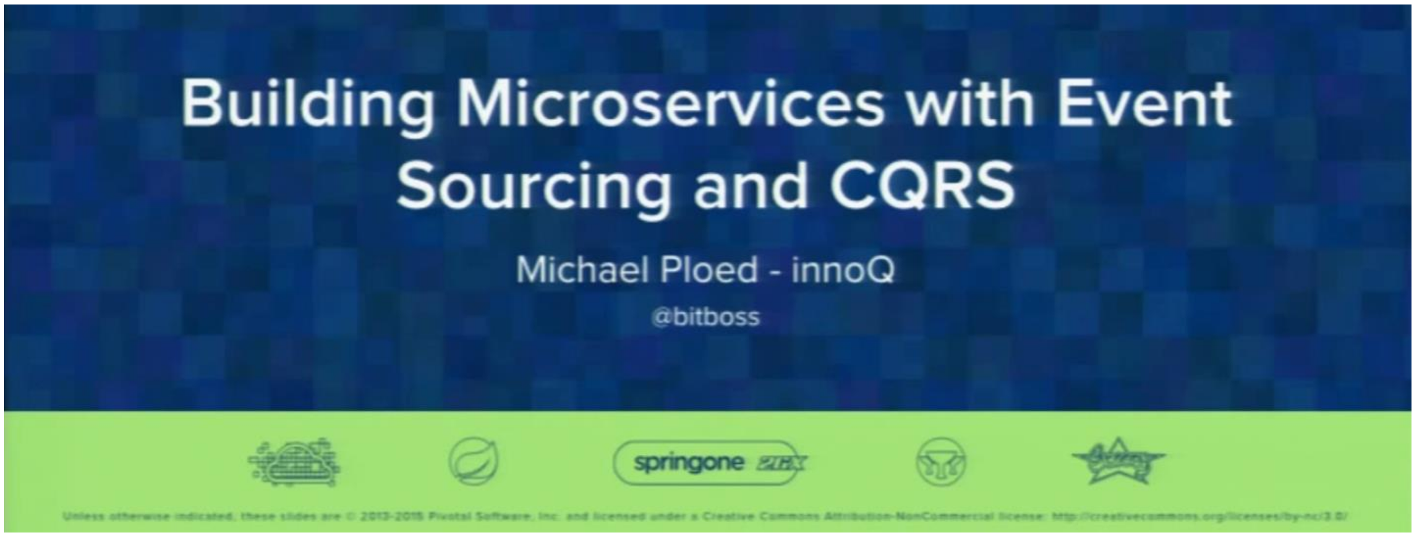
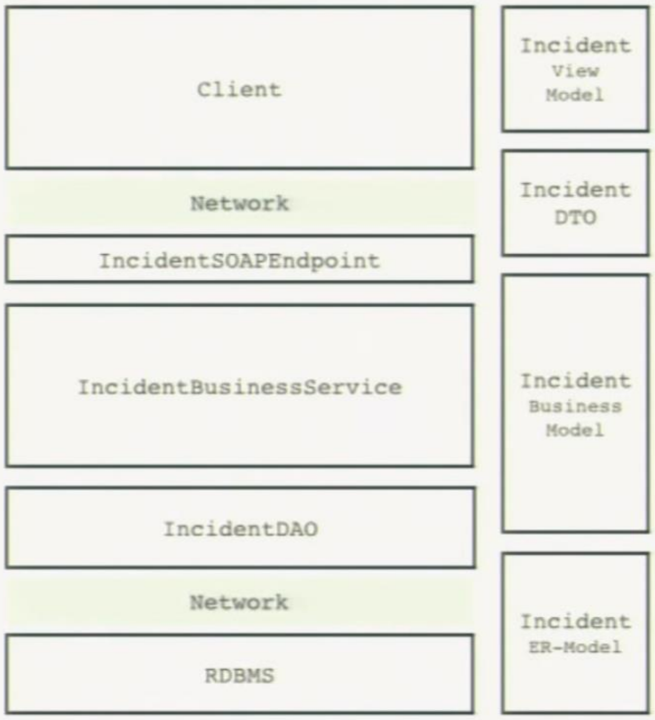


Building Microservices with Event Sourcing and CQRS

Michael Ploed - innoQ
@bitboss

springone 2GX

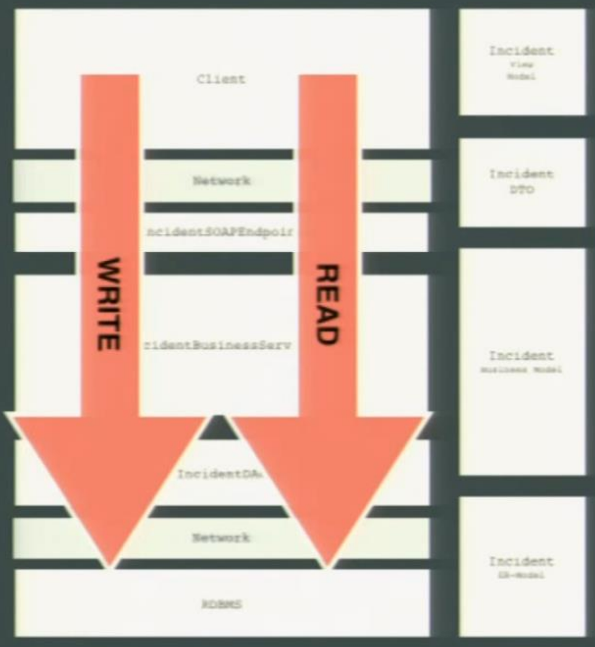
Let's review the classical old school N-Tier architecture



Characteristics

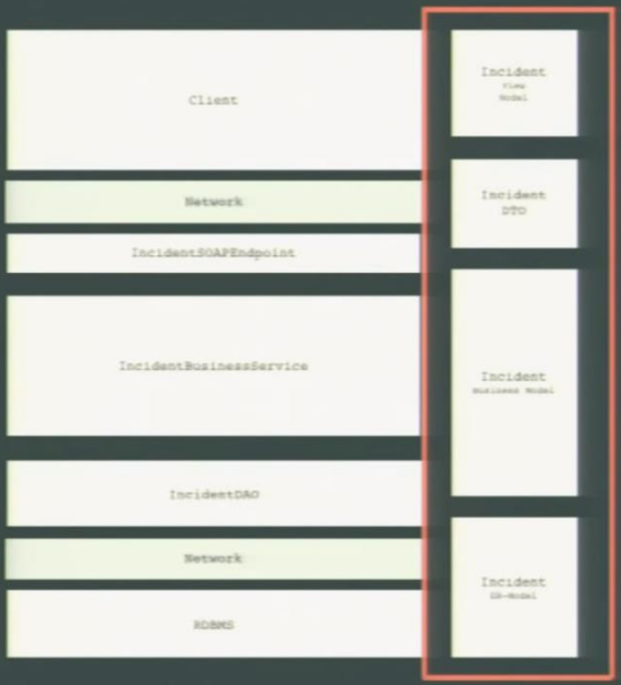
1

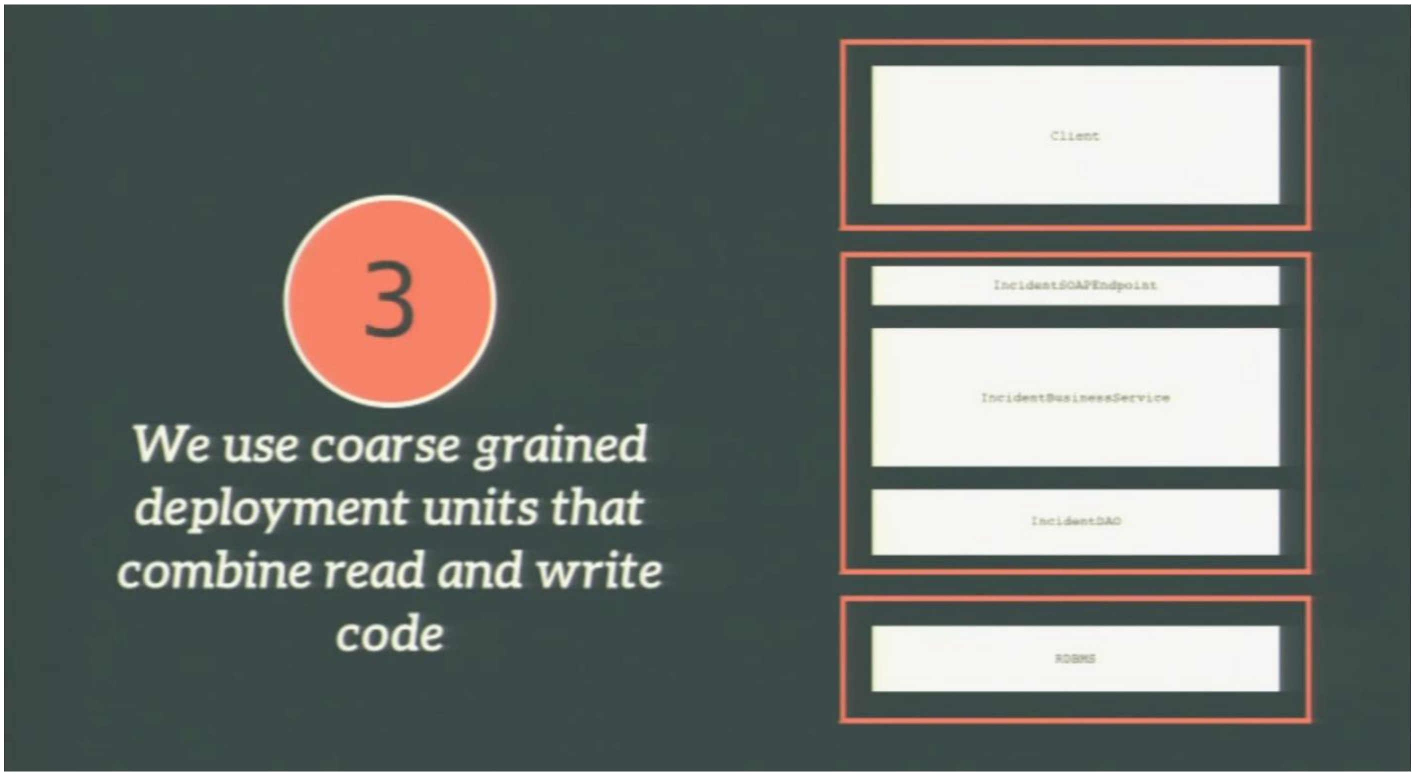
We read and write data through the same layers



2

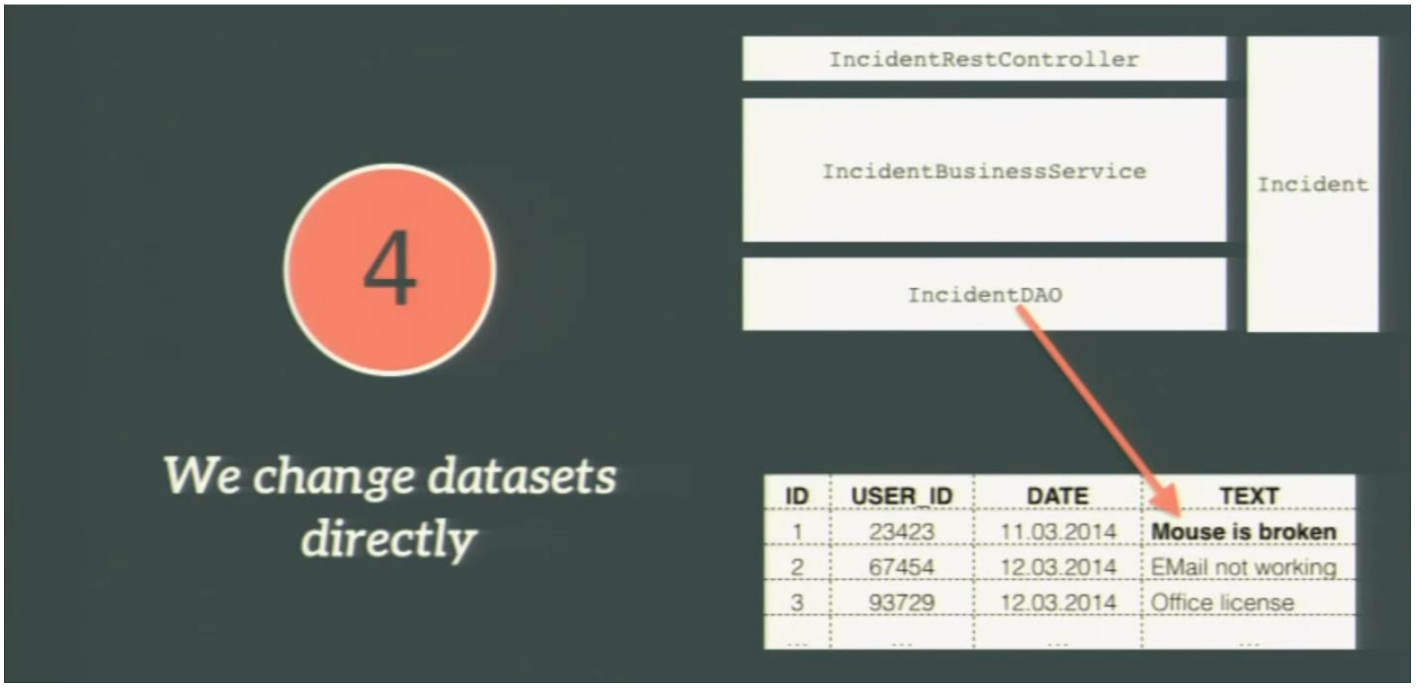
We use the same model for read and write access





3

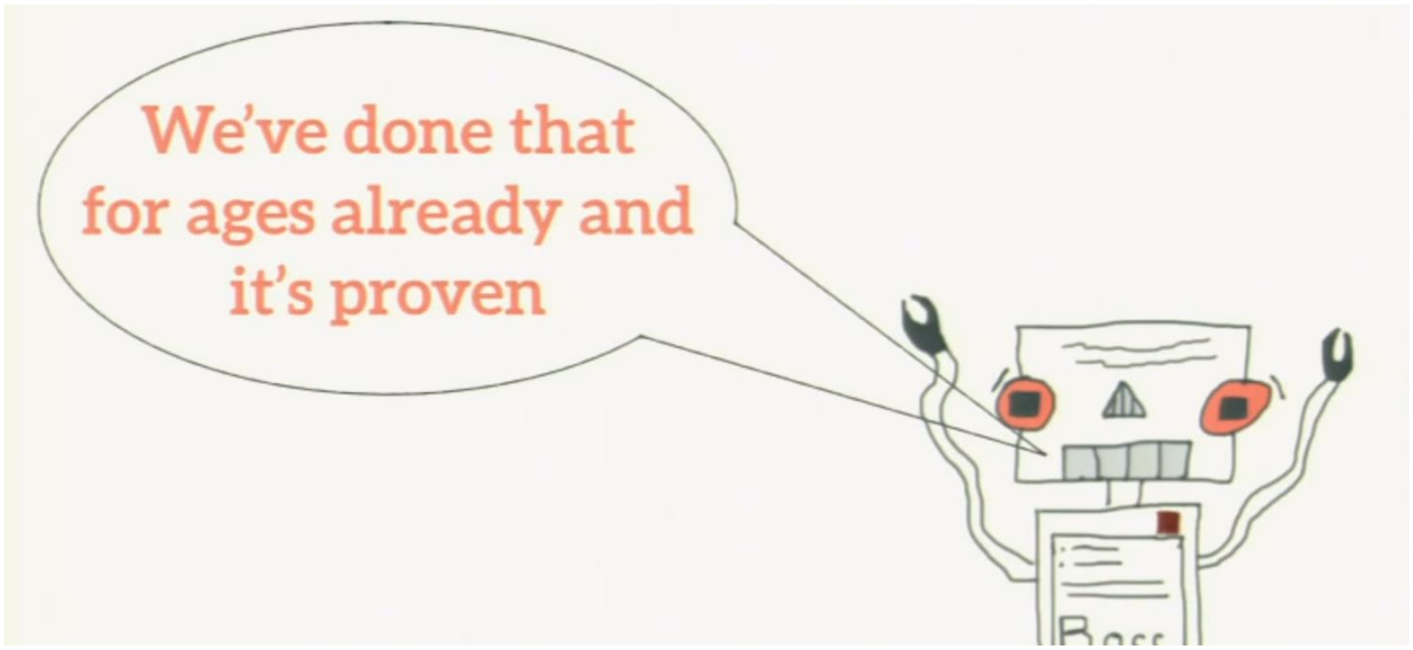
We use coarse grained deployment units that combine read and write code



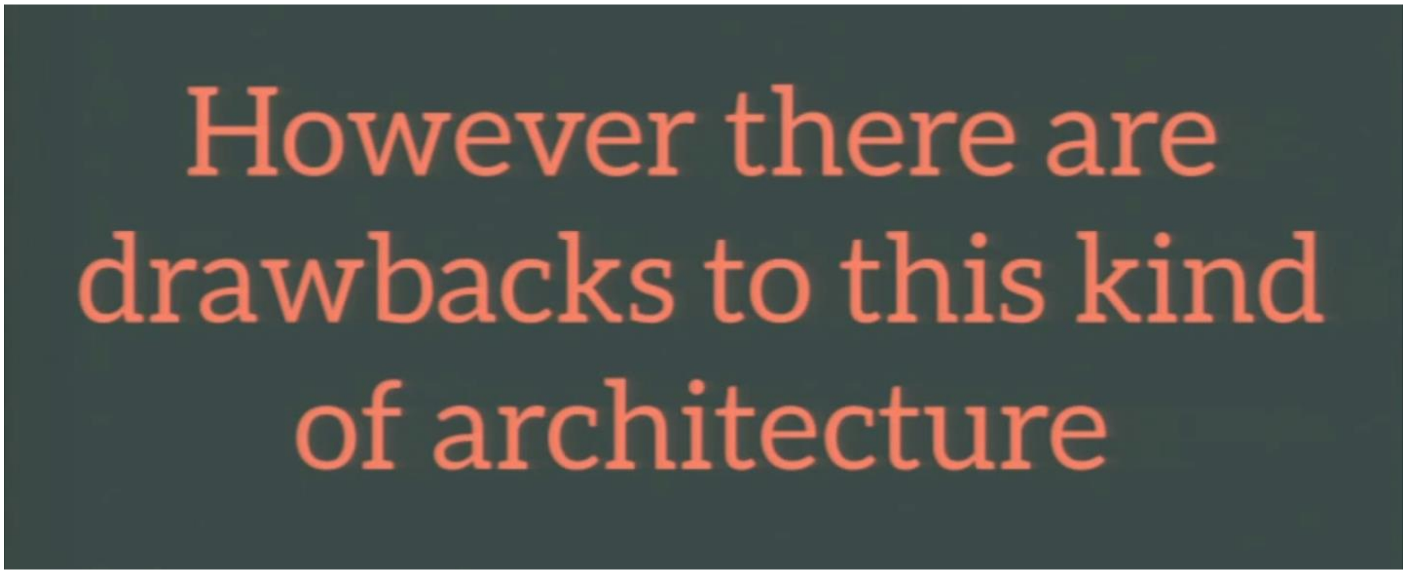
4

We change datasets directly

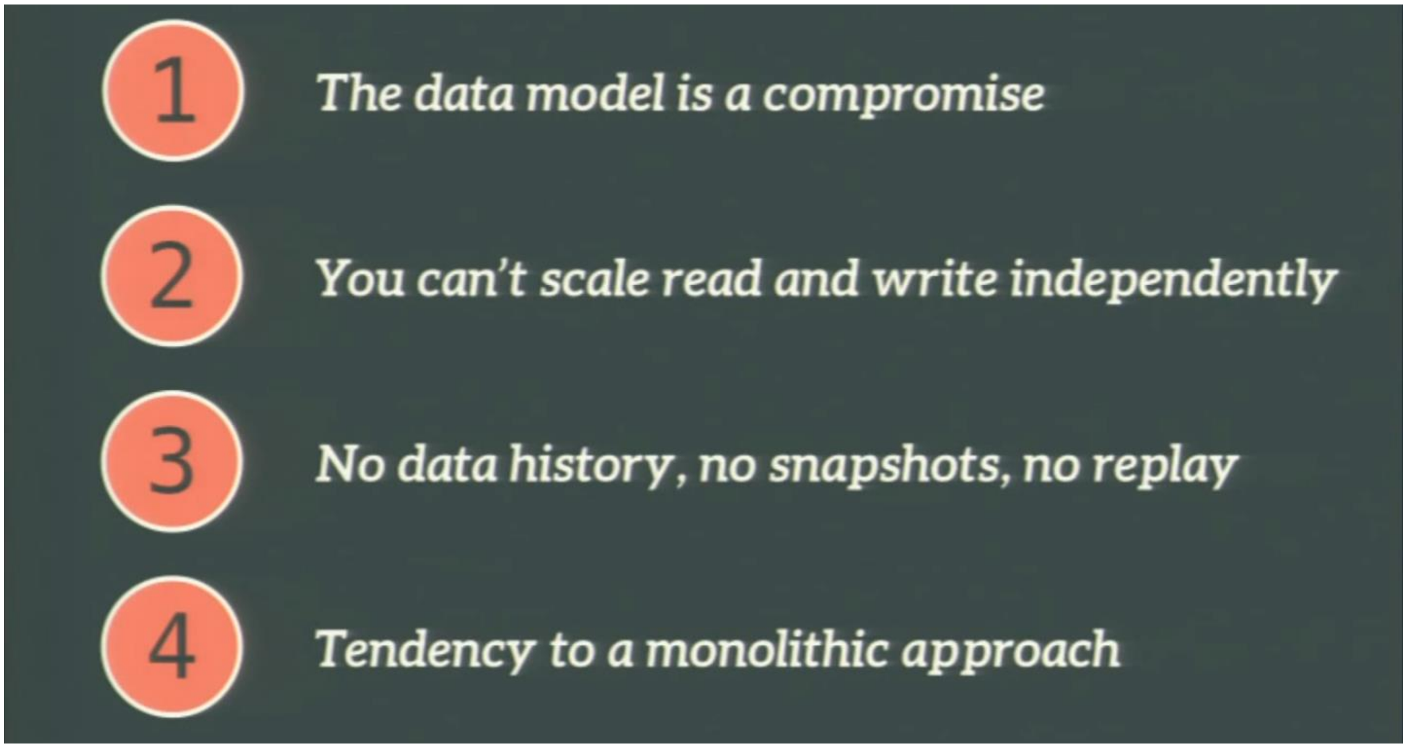
ID	USER_ID	DATE	TEXT
1	23423	11.03.2014	**Mouse is broken**
2	67454	12.03.2014	EMail not working
3	93729	12.03.2014	Office license
...	...	...	...



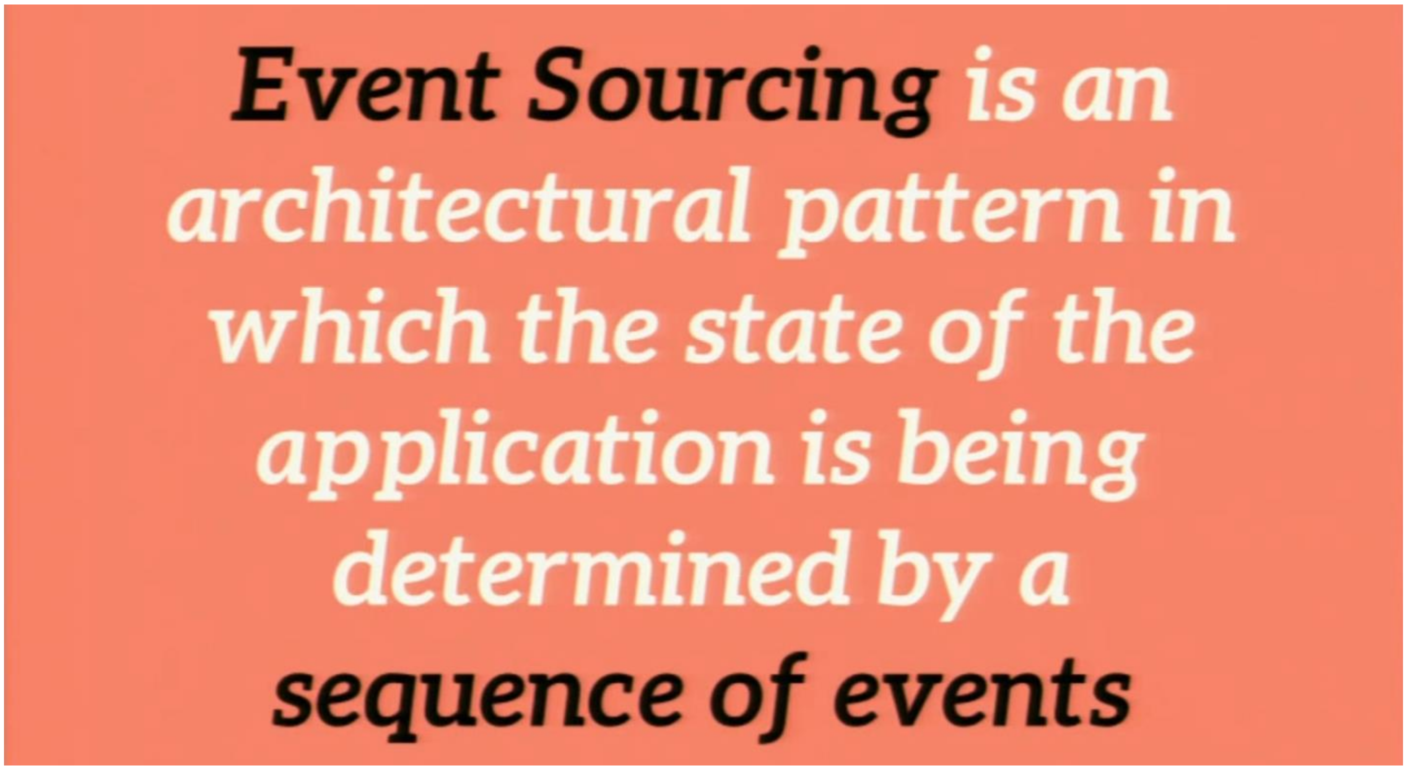
Many applications will run smooth and fine with this kind of approach



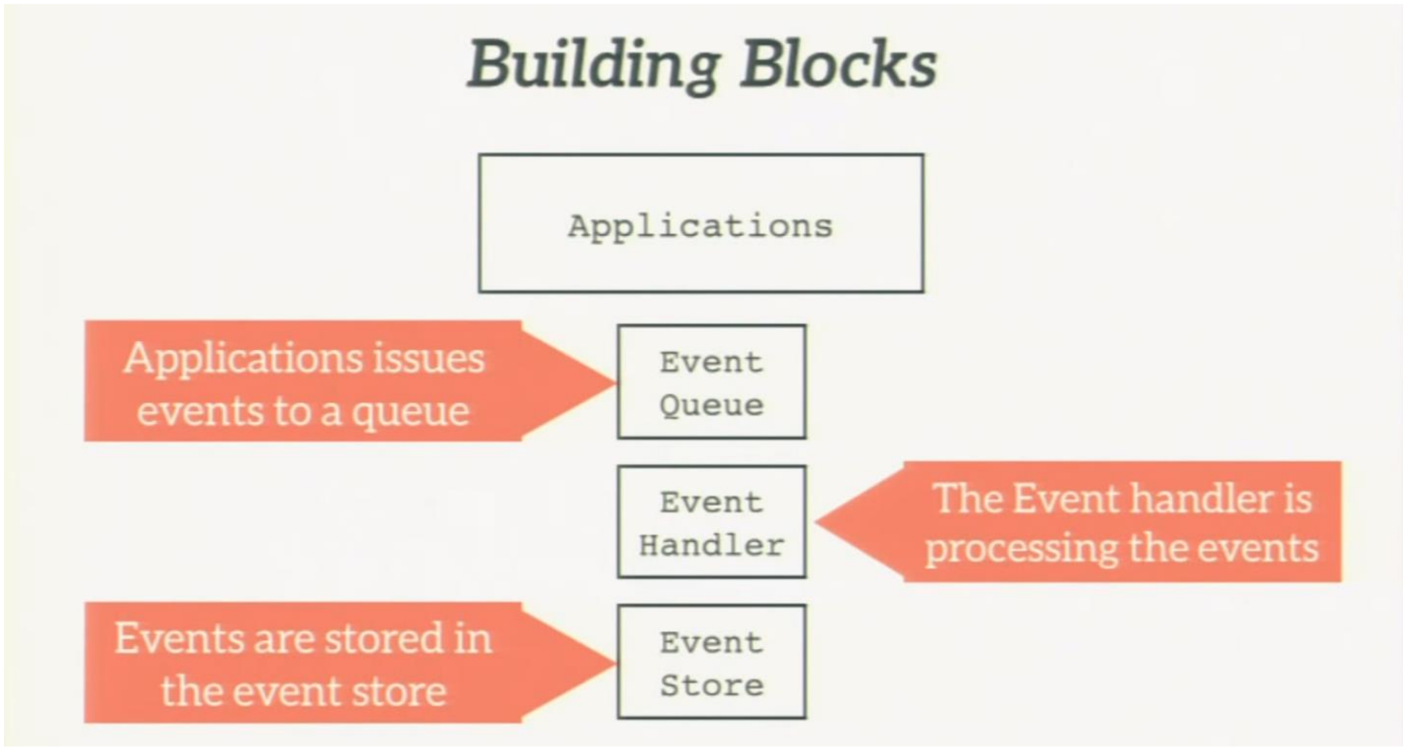
However there are drawbacks to this kind of architecture



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*

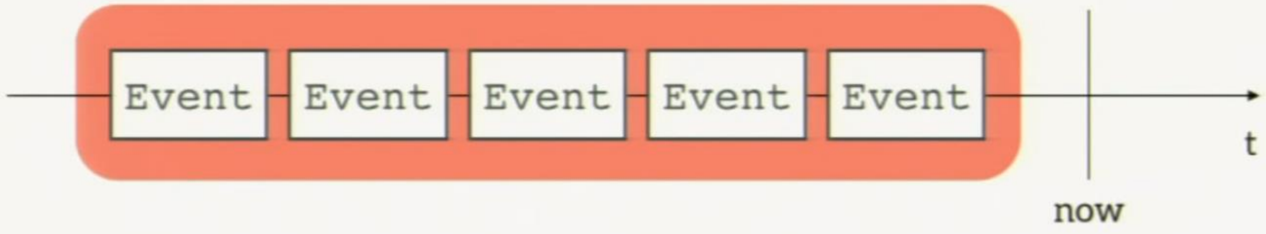


Event Sourcing** is an architectural pattern in which the state of the application is being determined by a **sequence of events

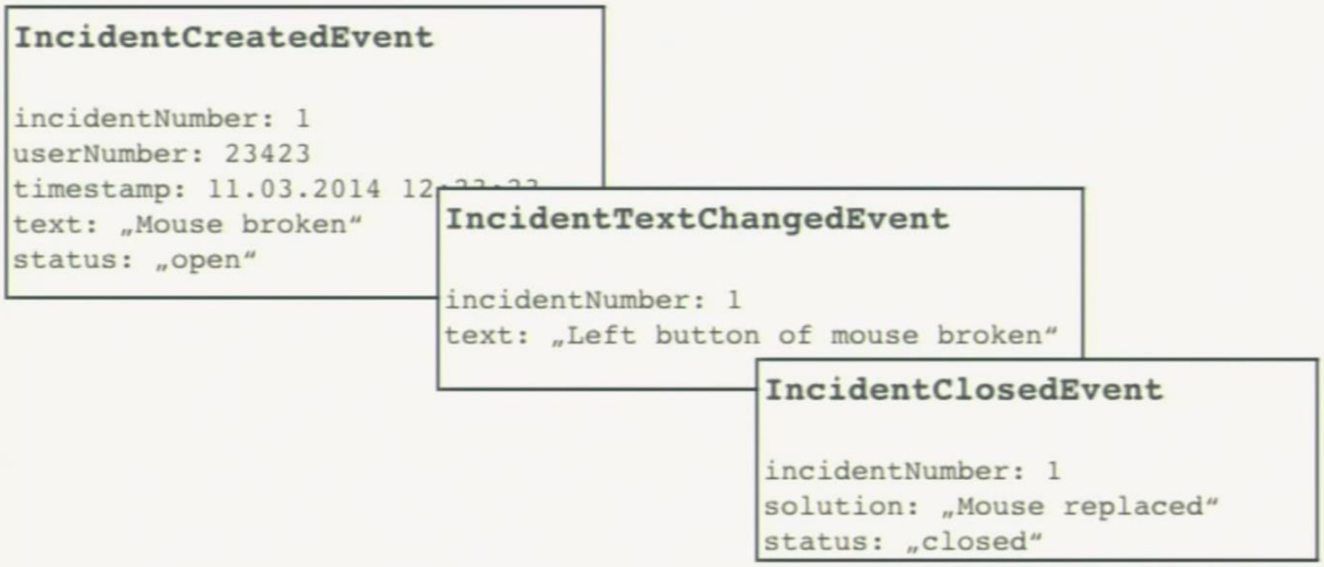


Building Blocks
Applications
Applications issues events to a queue
Event Queue
Event Handler
The Event handler is processing the events
Events are stored in the event store
Event Store

The sequence of events in the queue is called event stream



Event Stream Example

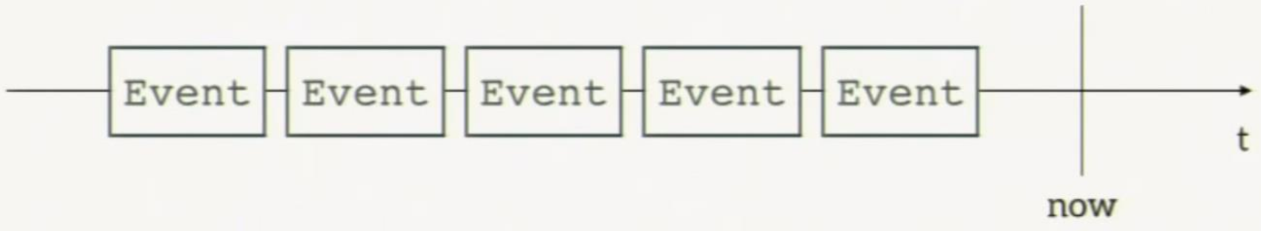


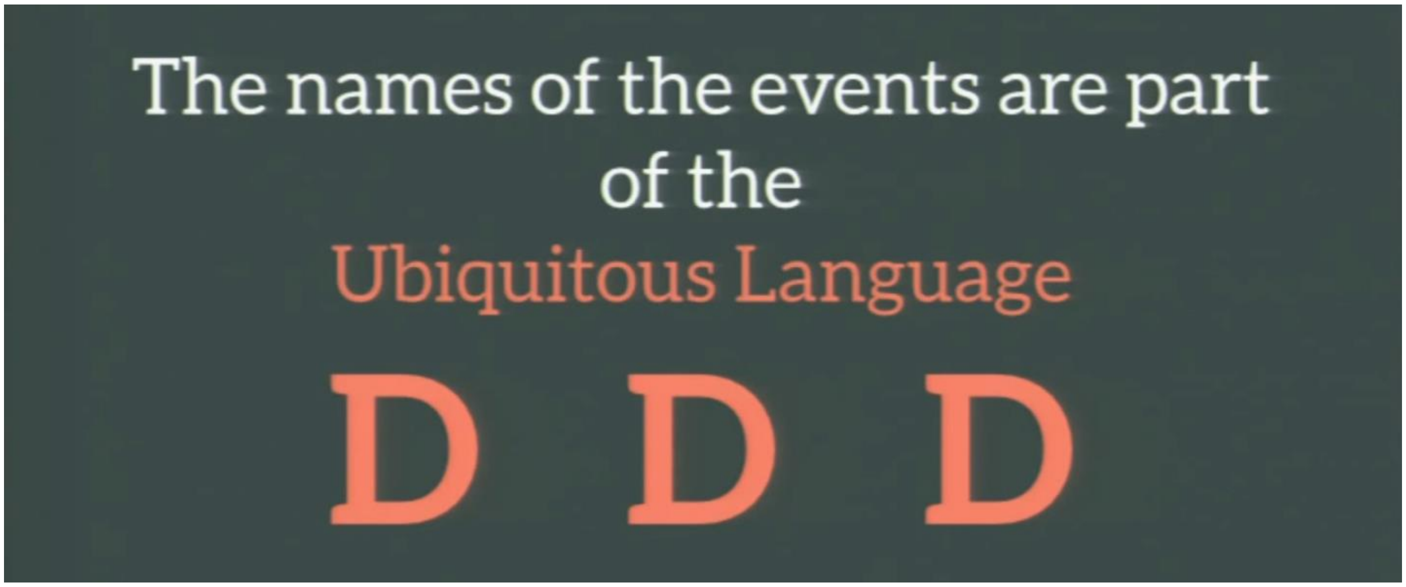
```
IncidentCreatedEvent

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

```
IncidentTextChangedEvent

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

```
IncidentClosedEvent

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

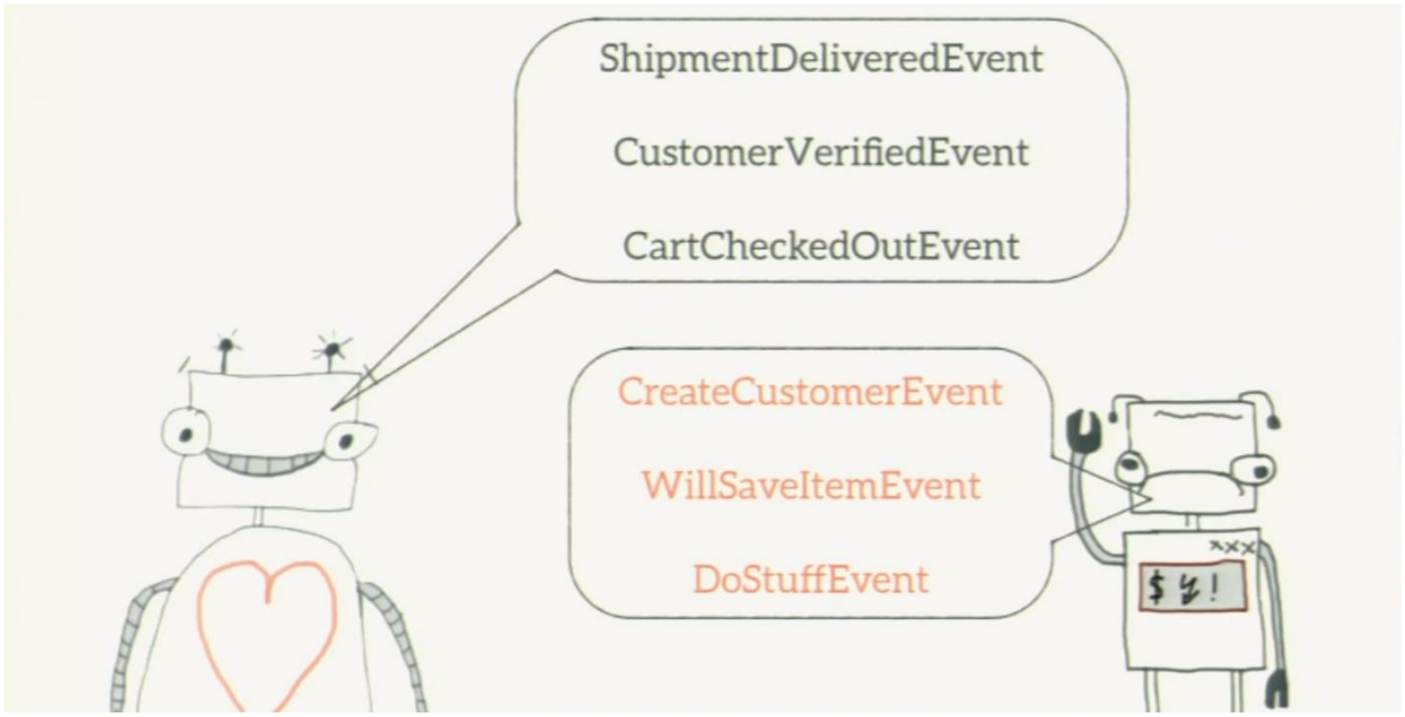
An event is something that happened in the past



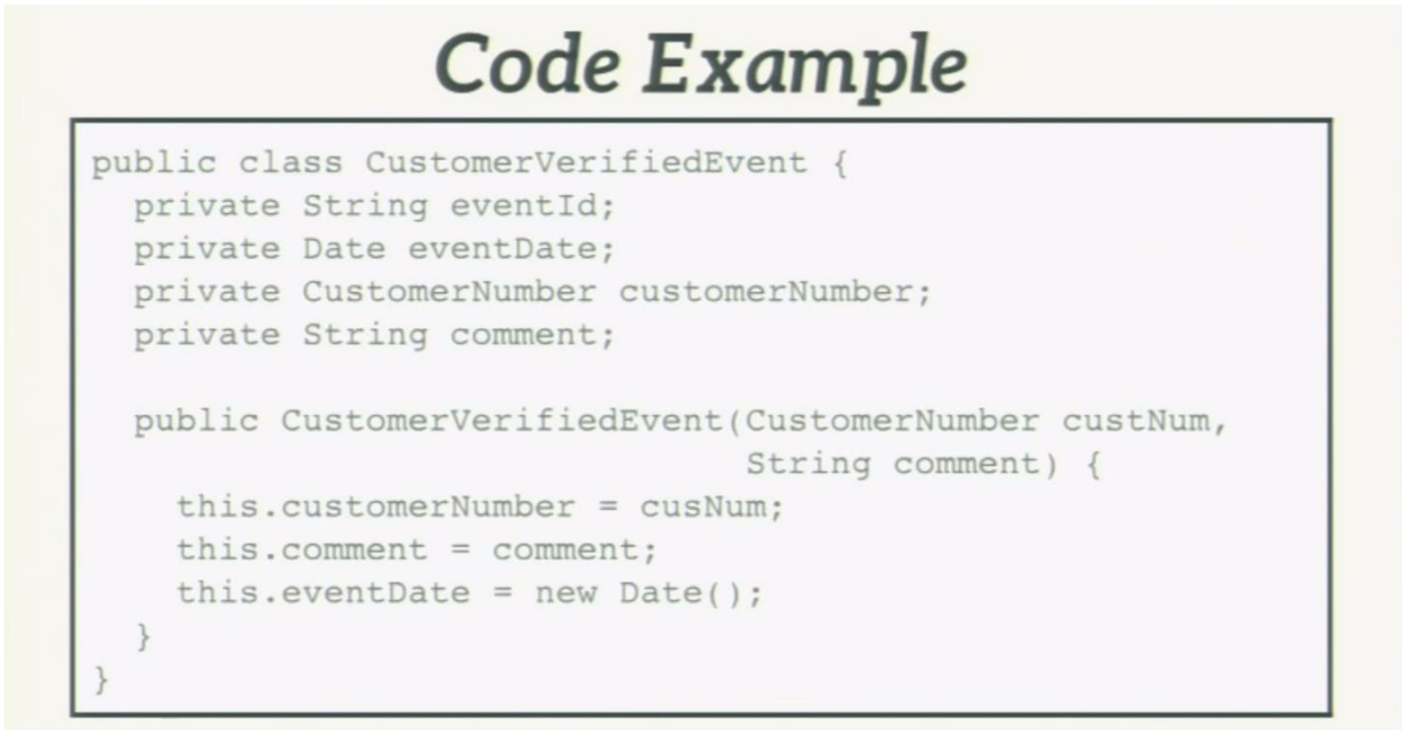


The names of the events are part of the Ubiquitous Language

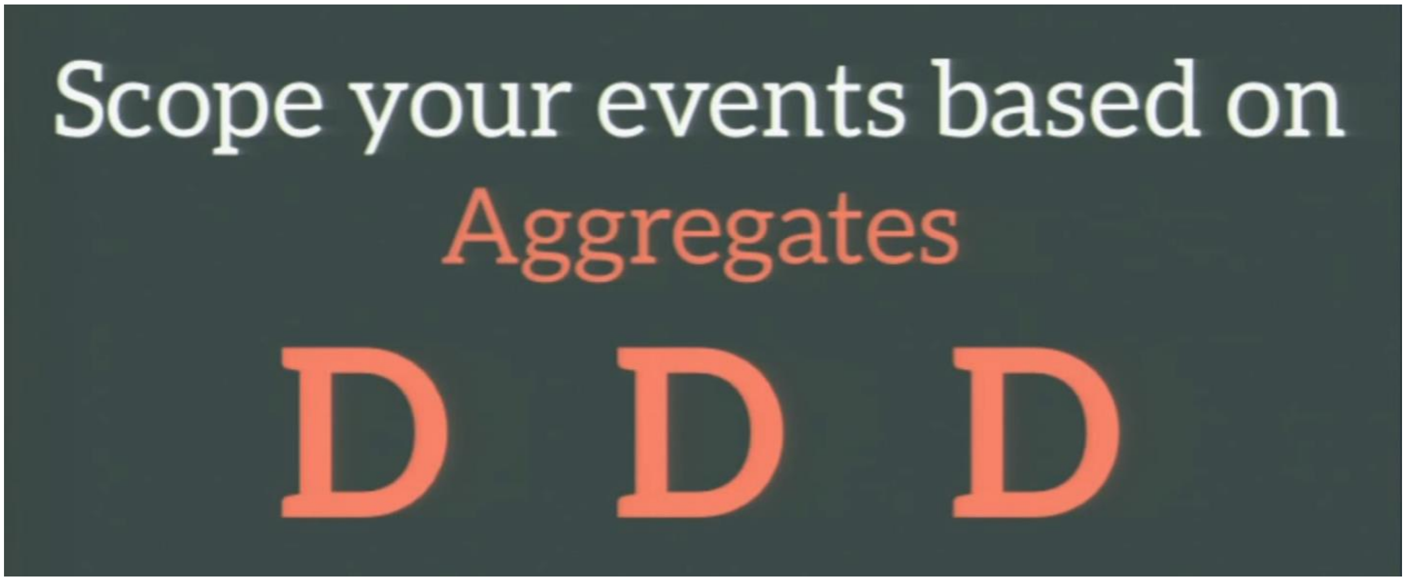
D D D



The bad event names in red are command-style event names that are bad in DDD!

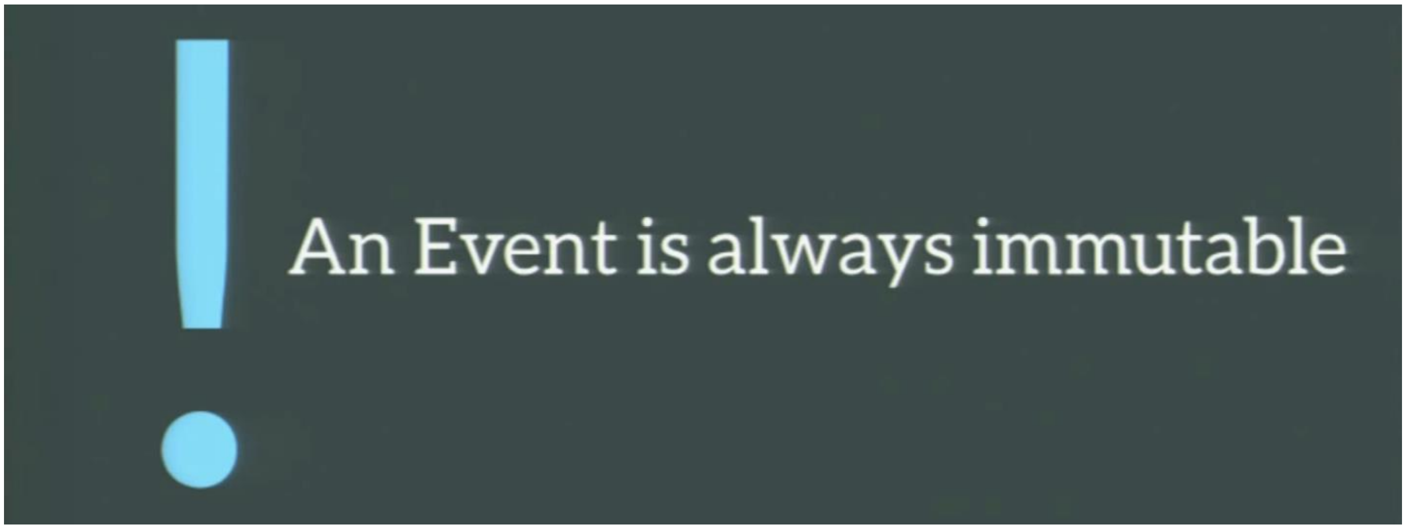

Code Example

```
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();
  }
}
```

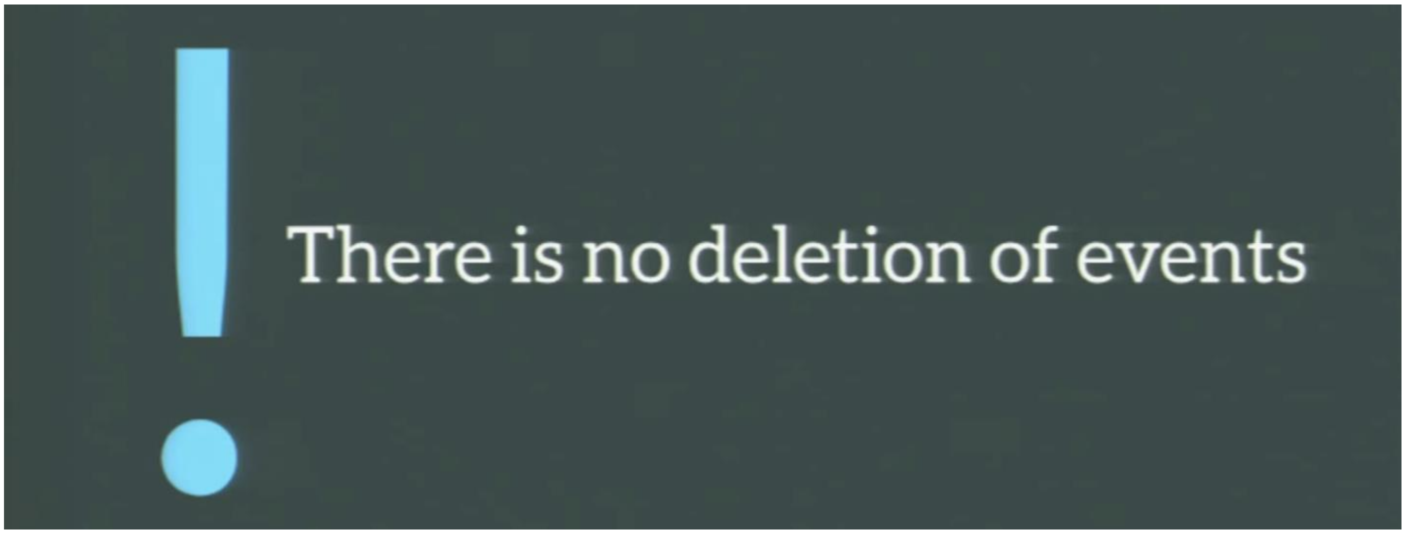

Scope your events based on Aggregates

D D D

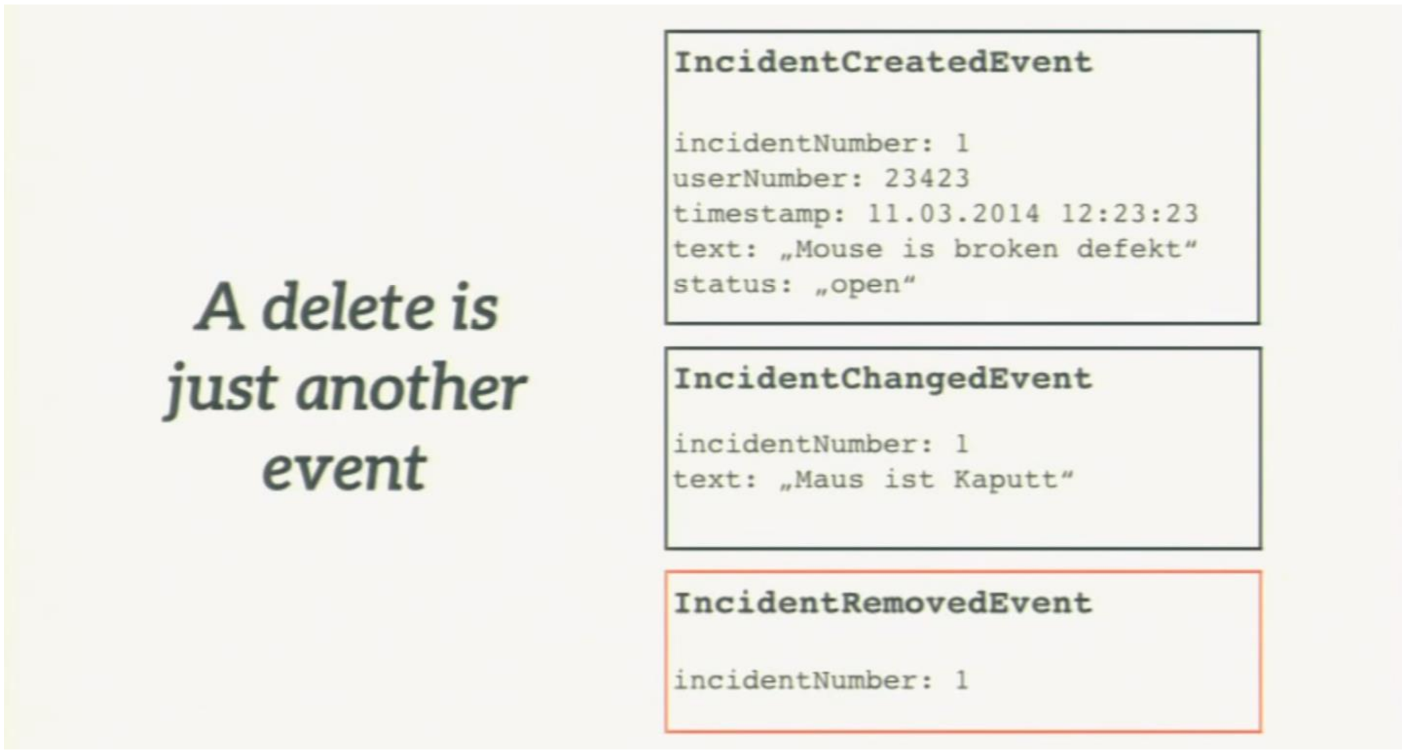
How fine grained should we model our events? Aggregates in DDD are a collection of entities bounded to a specific business model. There is always a root entity that other entities are based on, the root entity is usually he aggregate that we can use to generate events.



An Event is always immutable



There is no deletion of events



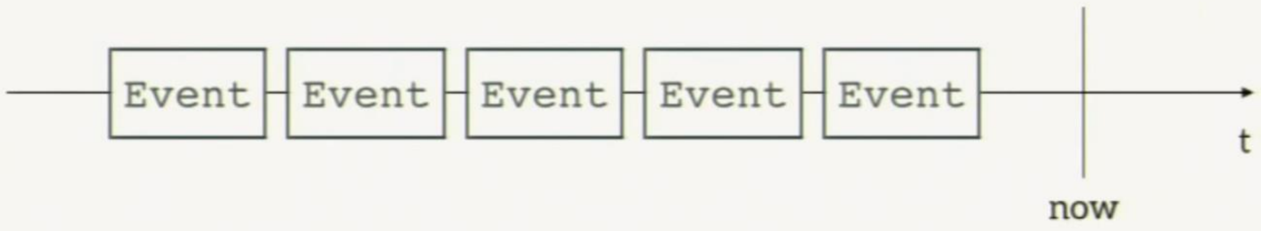
A delete is just another event

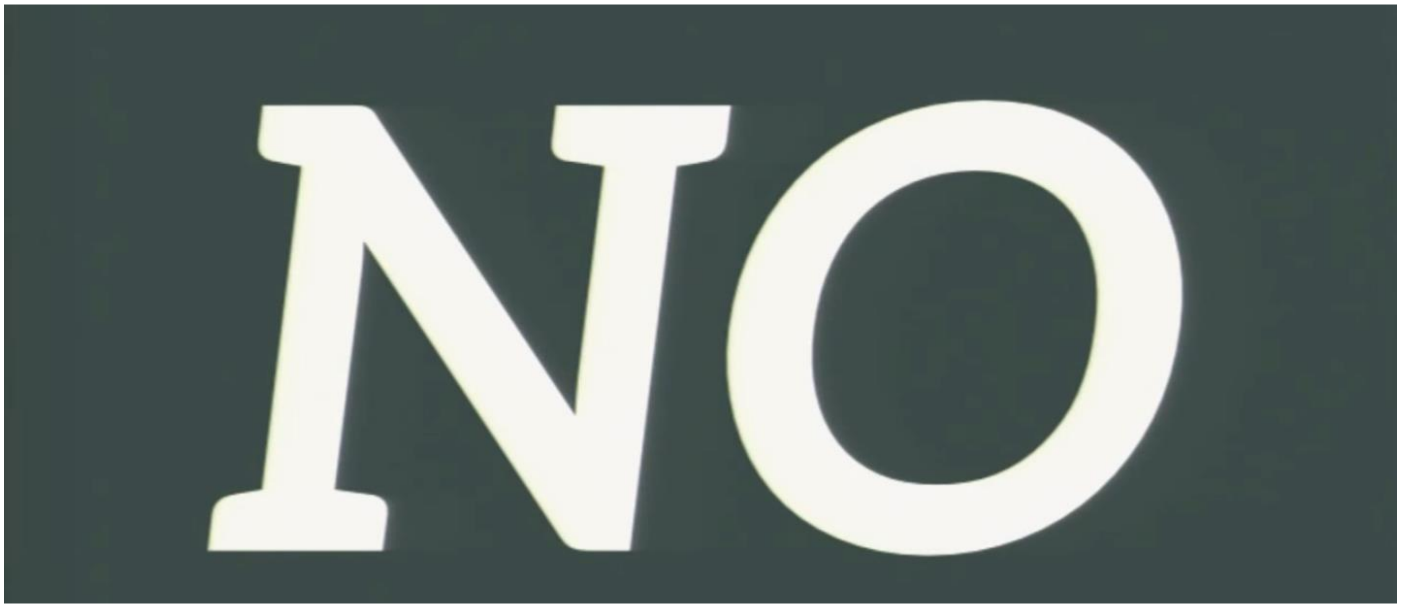
```
IncidentCreatedEvent

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“
```

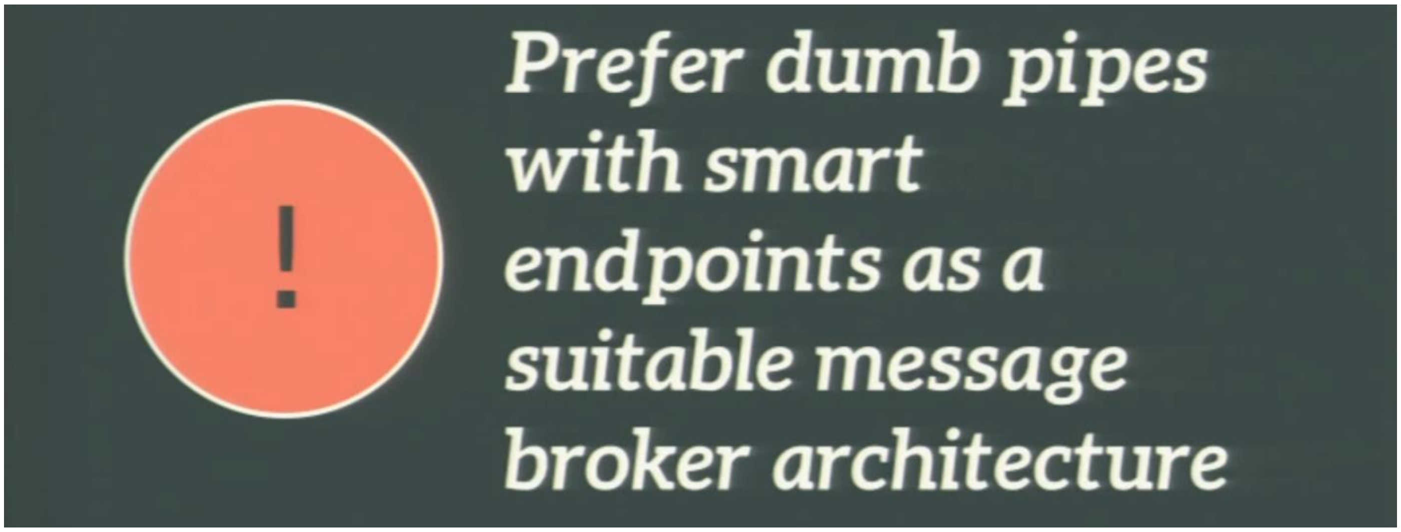
```
IncidentRemovedEvent

incidentNumber: 1
```

The event bus is usually implemented by a message broker



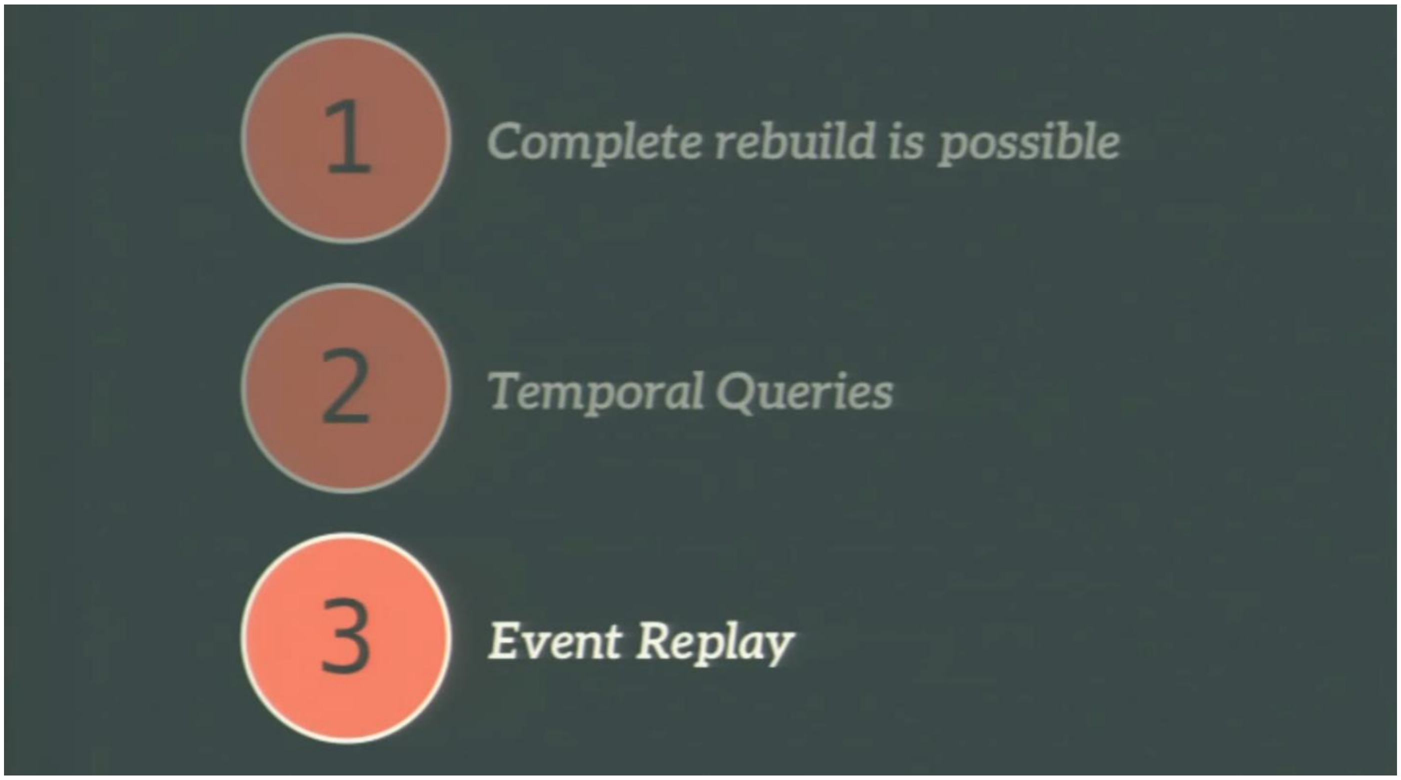


NO



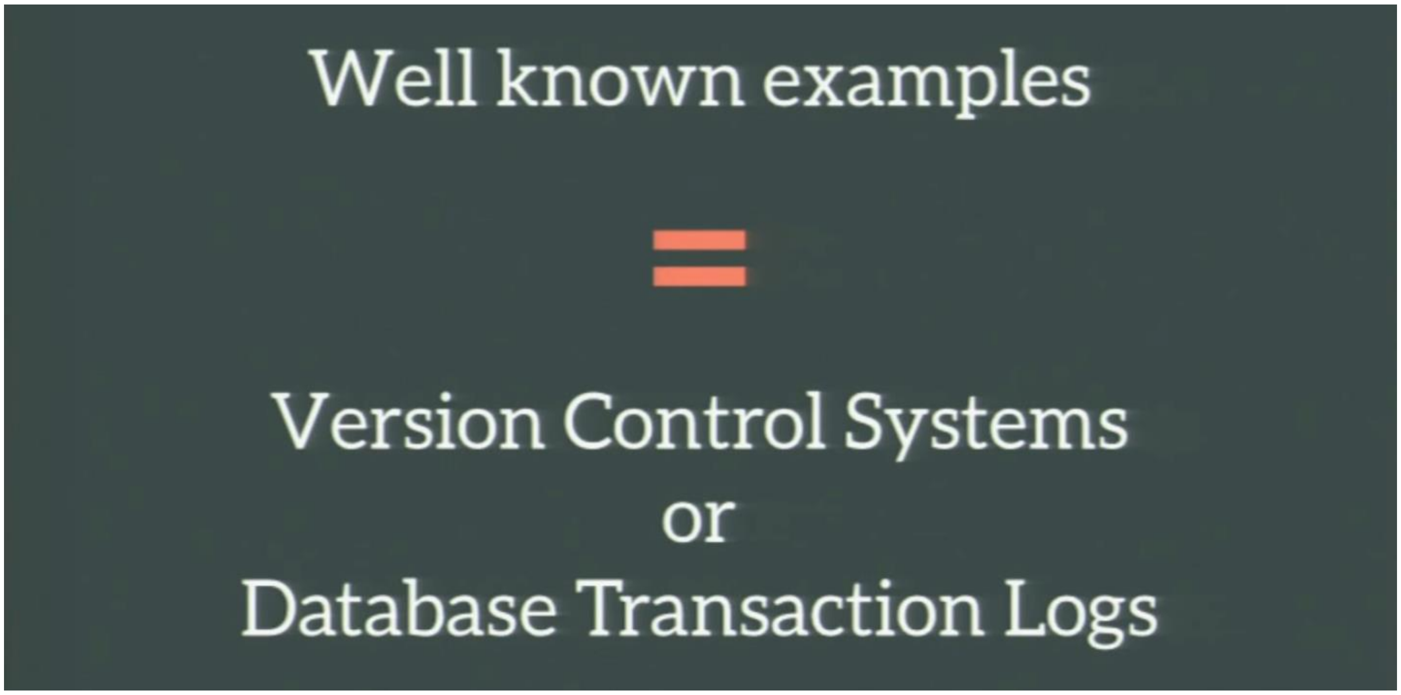
!

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


1 *Complete rebuild is possible*

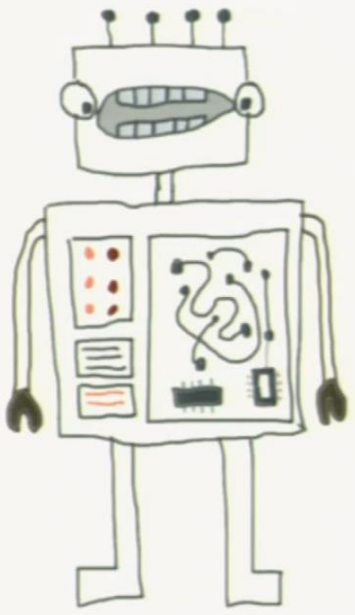
2 *Temporal Queries*

3 ***Event Replay***

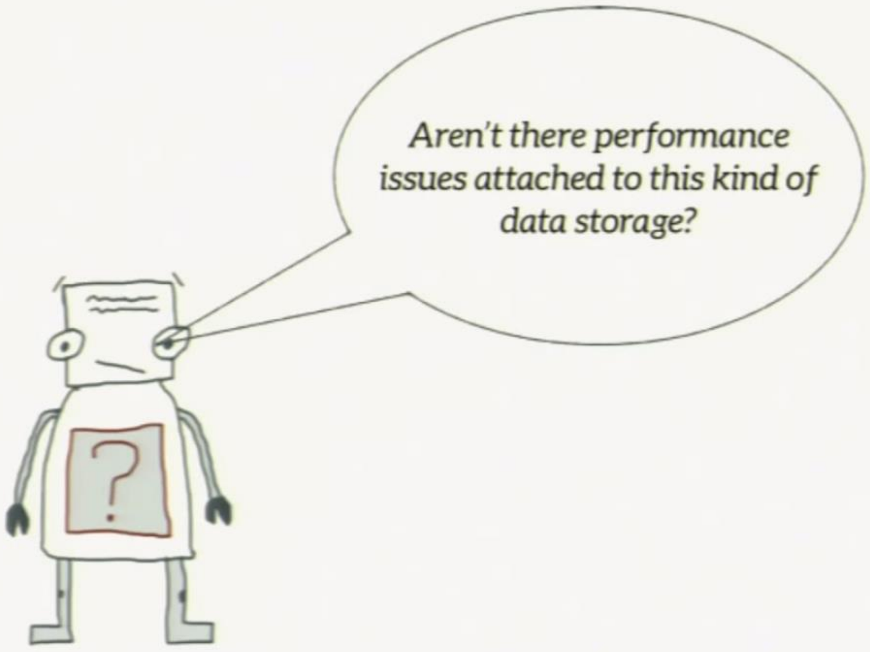

Well known examples

=

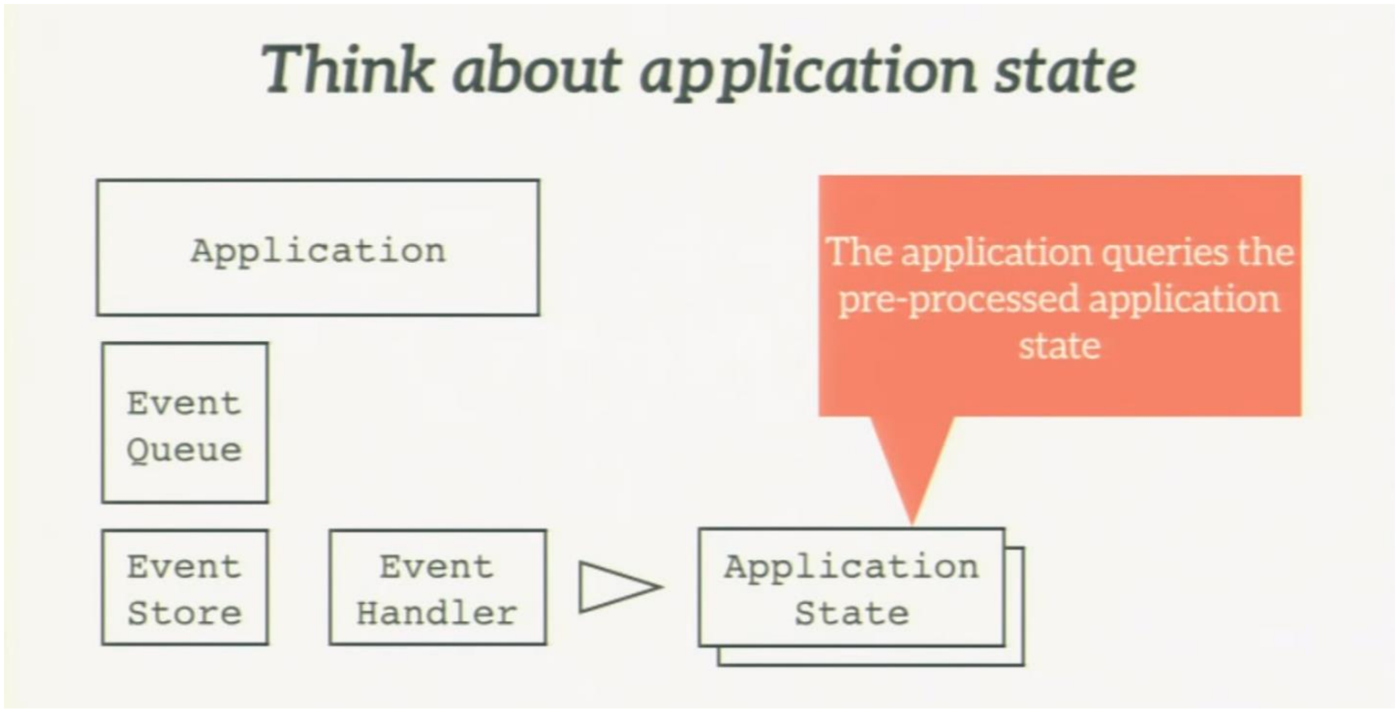
Version Control Systems
or
Database Transaction Logs



The Event Store has a very high business value


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

YES!

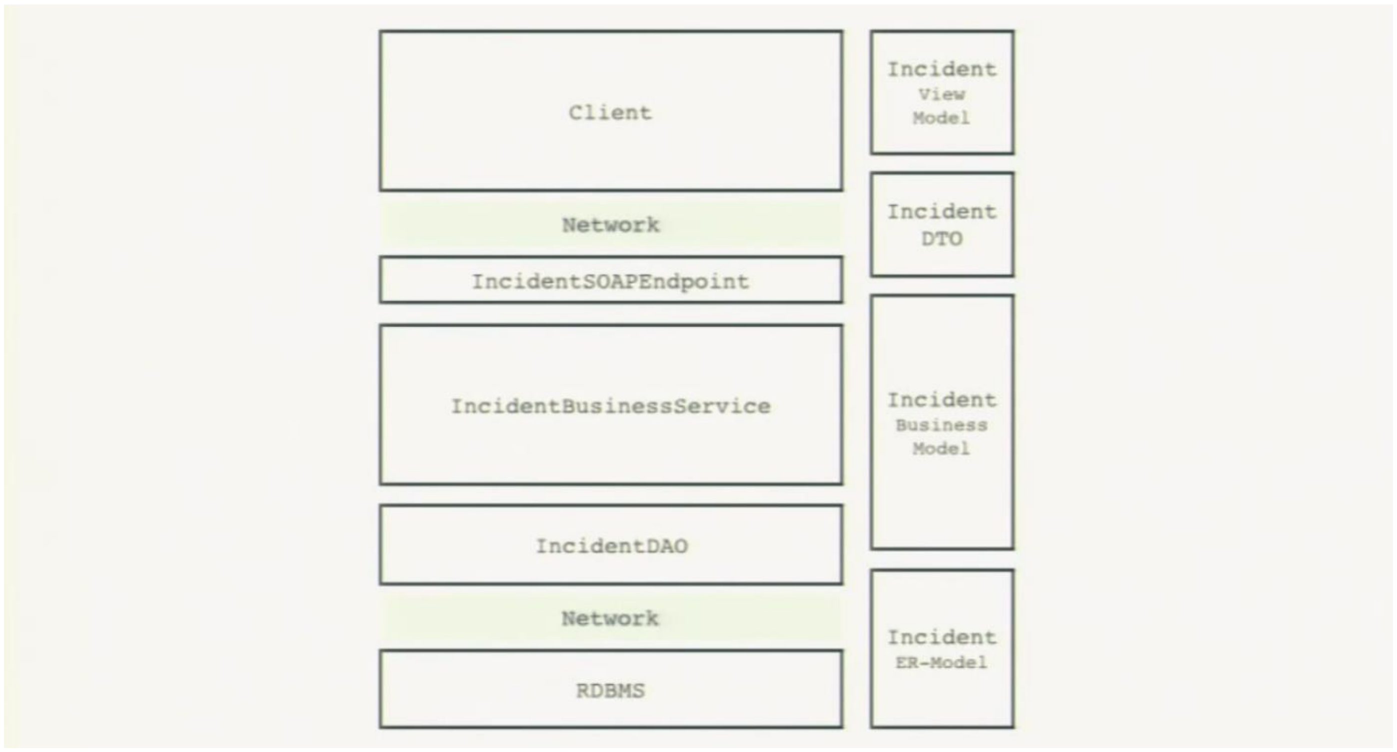


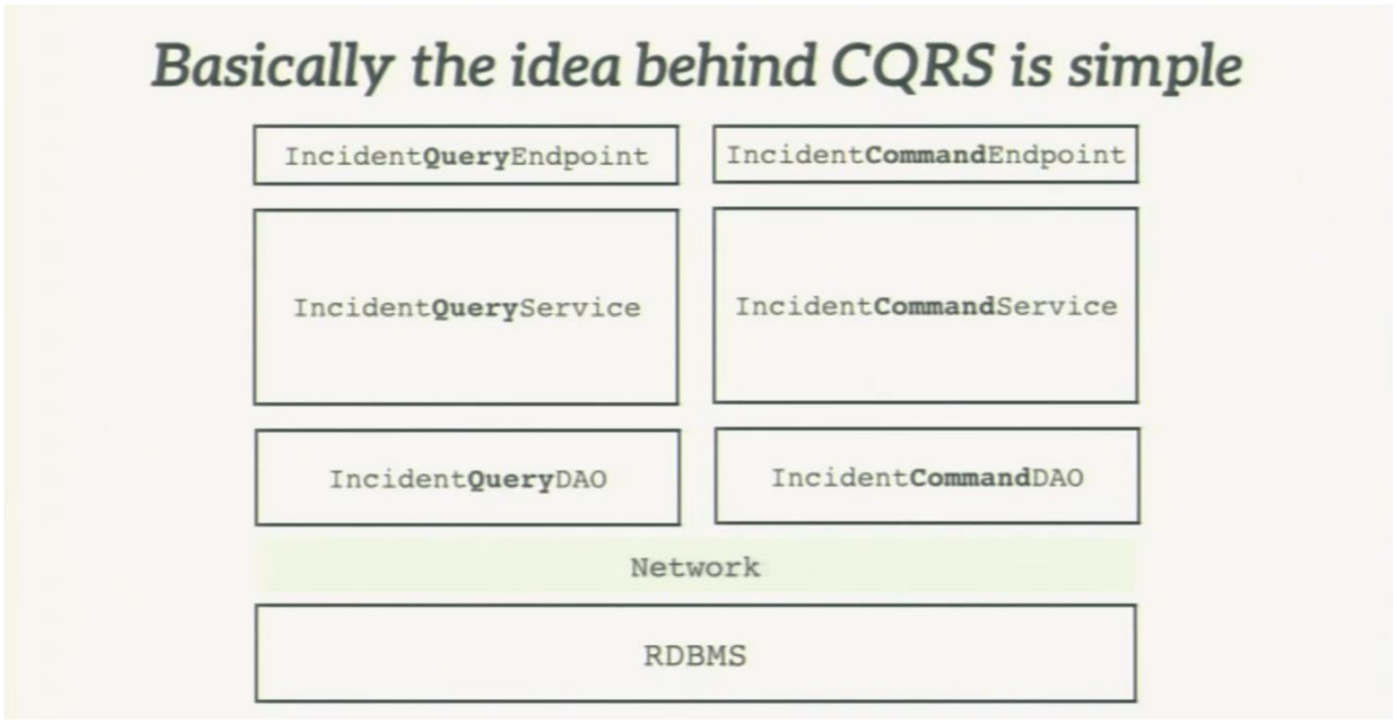
Think about application state

CQRS

Command
Query
Responsibility
Separation

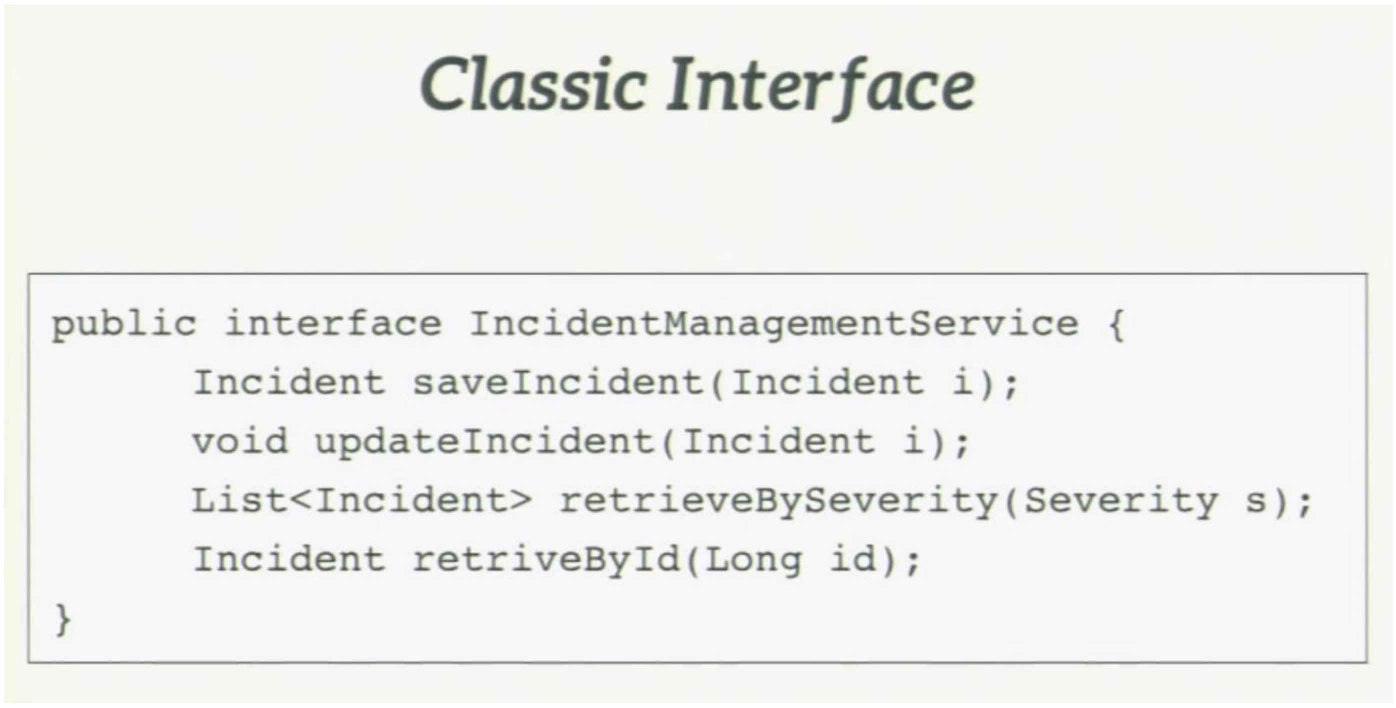
Command are Write operations, Queries and Read operations. We need to separate our Read and Write operations.





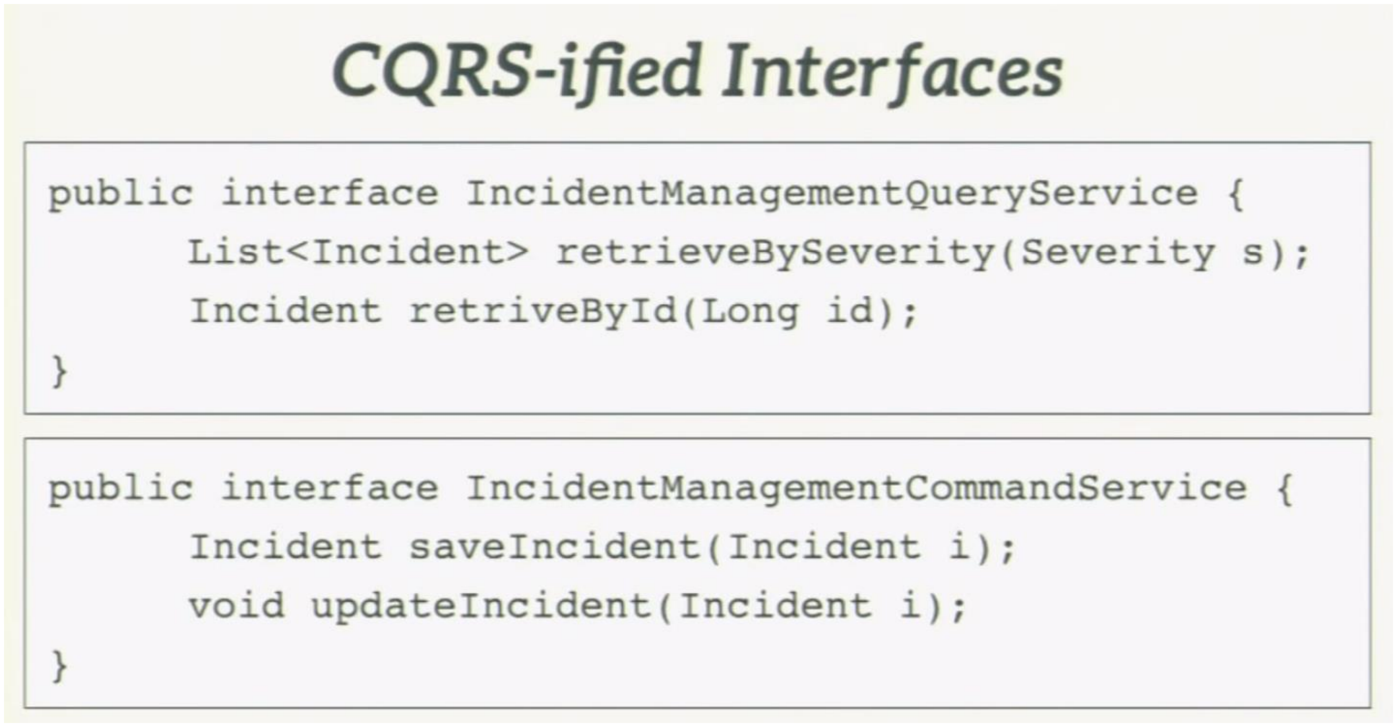
We split up the Read and Write parts of our application into separate logic different from the above. Because we are still working with the same data model

Code Example



Classic Interface

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

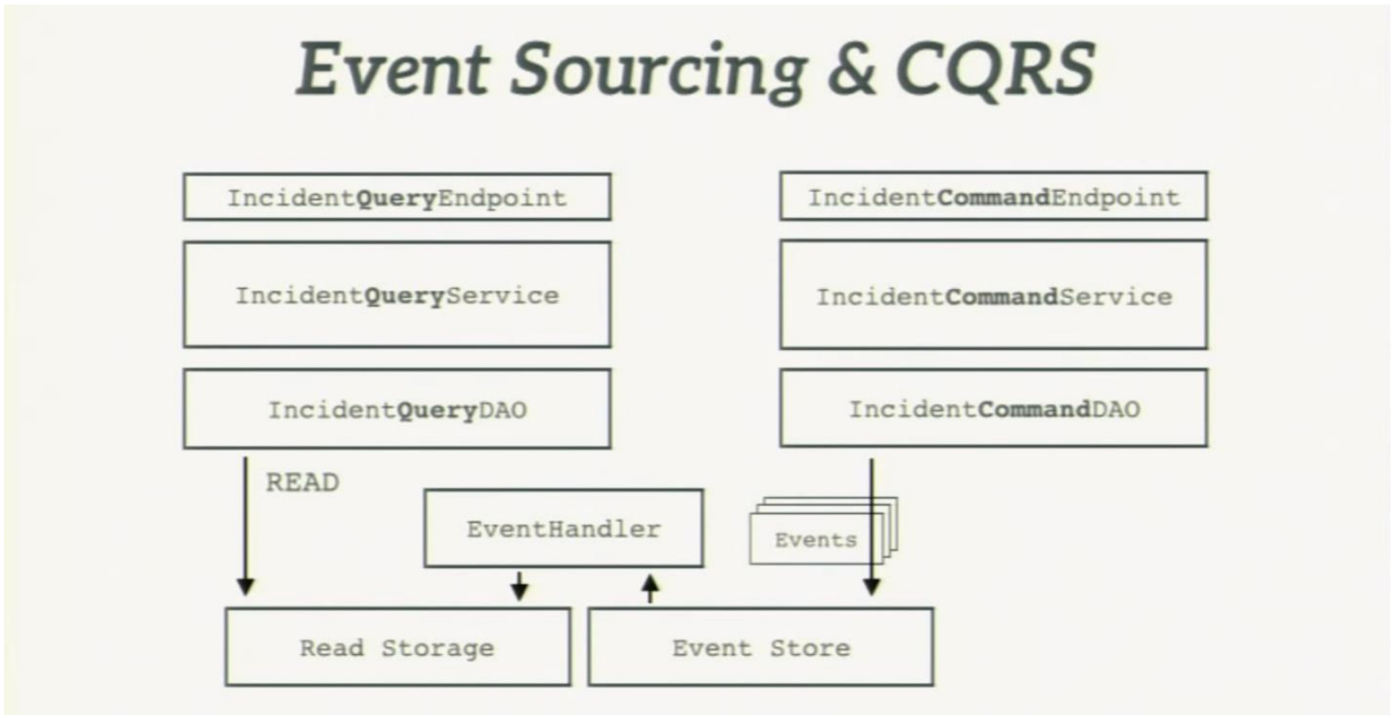


CQRS-ified Interfaces

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

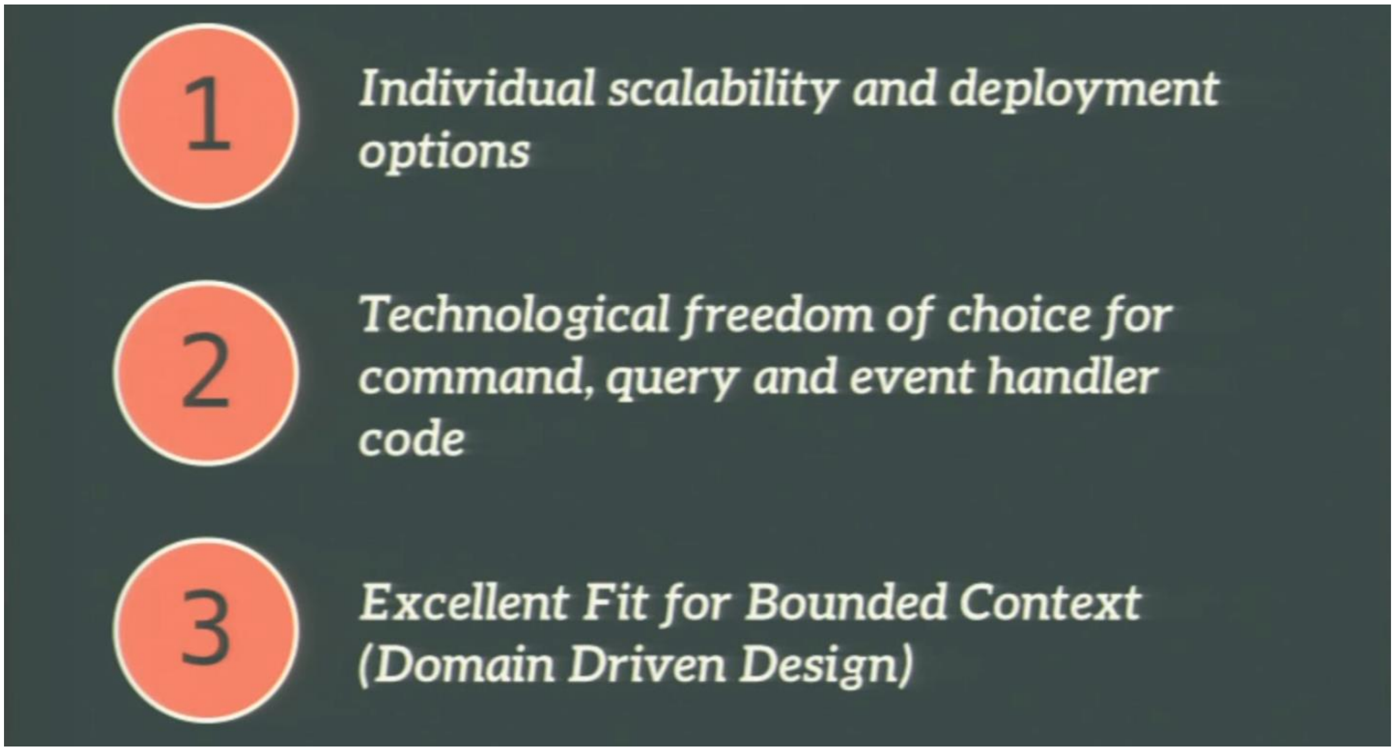
```
public interface IncidentManagementCommandService {
      Incident saveIncident(Incident i);
      void updateIncident(Incident i);
}
```

Split it up into a query service and a command service as above



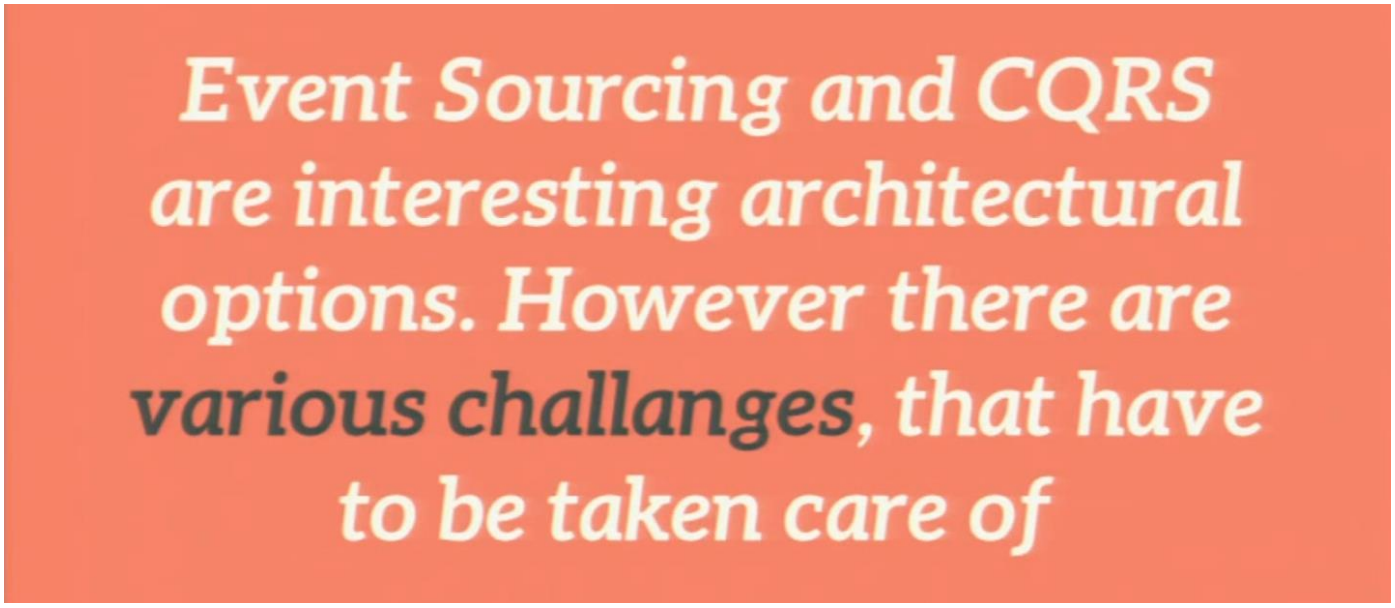
Event Sourcing & CQRS

The Command part issues events to the events store where we have an event handler that is able to derive the application state into a separate Read storage that you query against.

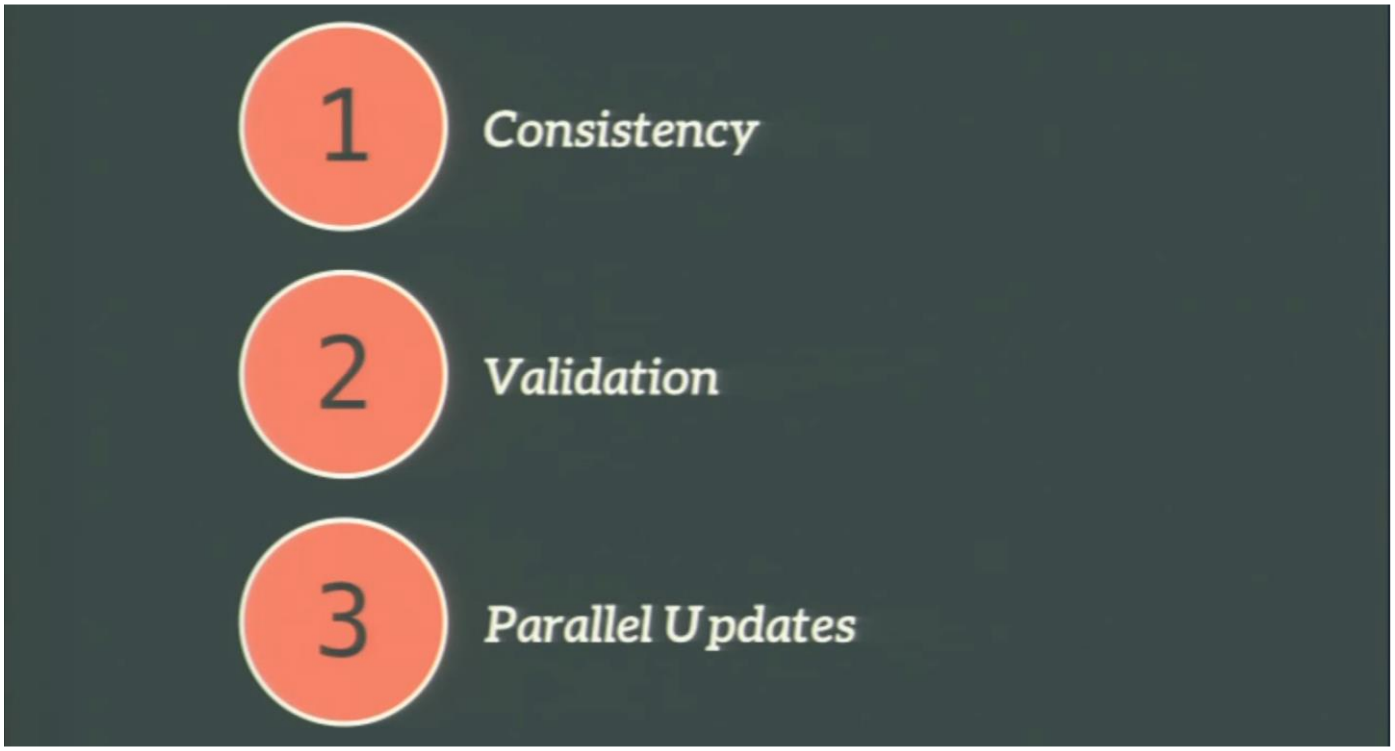


1. *Individual scalability and deployment options*
2. *Technological freedom of choice for command, query and event handler code*
3. *Excellent Fit for Bounded Context (Domain Driven Design)*

A bounded context for a microservice might also include many data aggregates, bind your bounded context of your microservices come from aggregates in your business domain



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

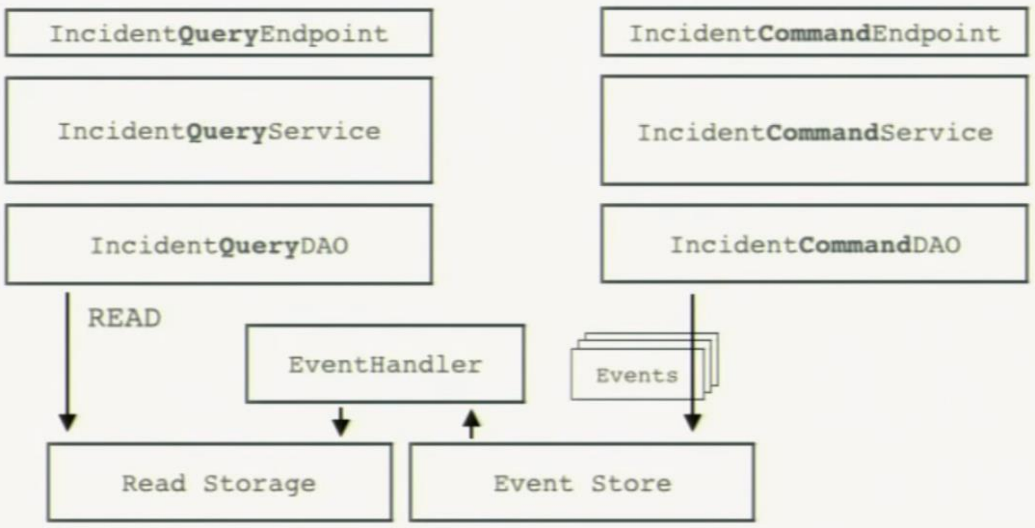
Can we use a RDBMS as our events data store? Where do we validate data about our aggregates in event sourcing? What about parallel updates or processing of events? An event can be a serialized JOSN data, generally a data format that can be easily stored like JSON, XML, or even string.

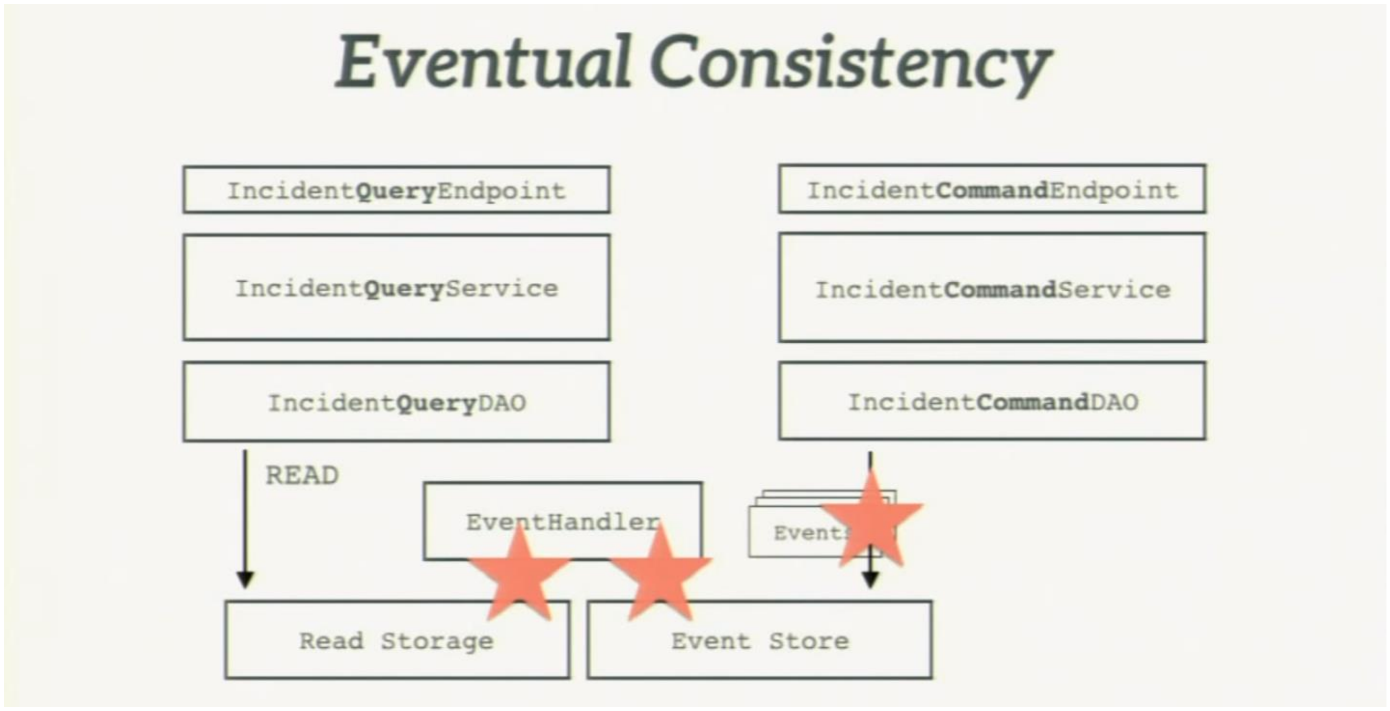
YES

In terms of consistency, the answer is YES

! Systems based on CQRS and Event Sourcing are mostly eventually consistent

Eventual Consistency

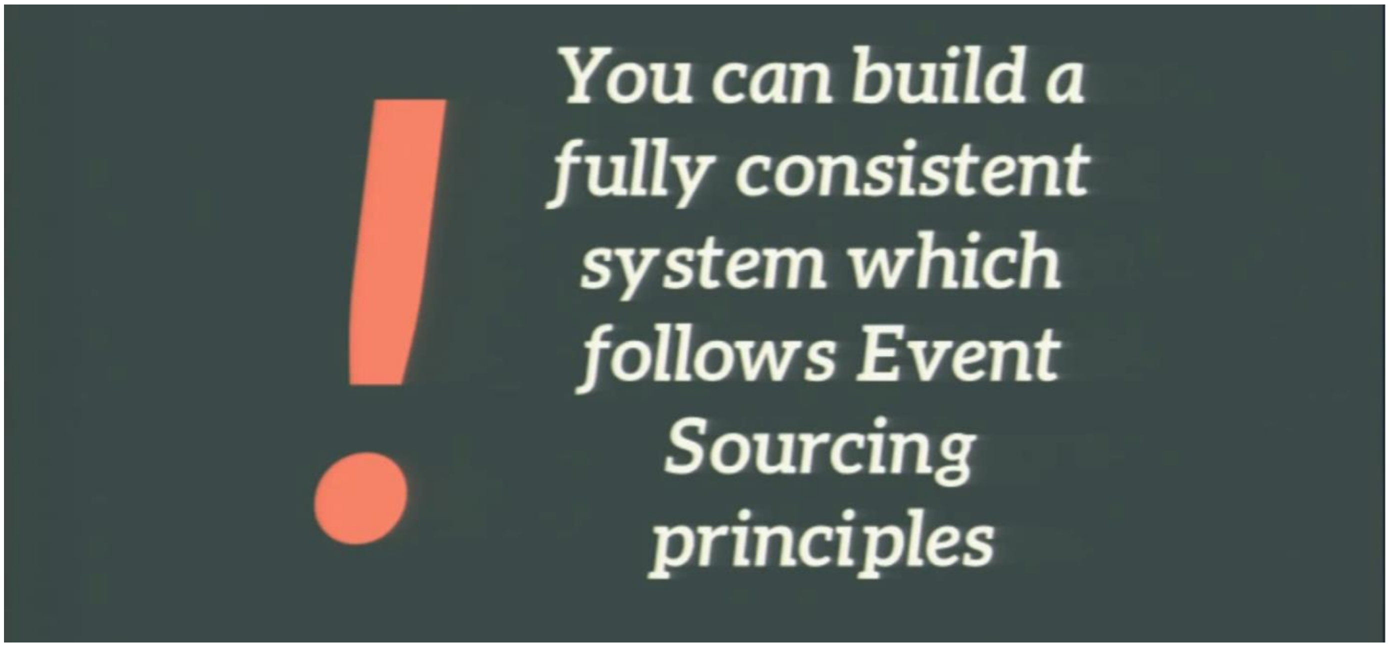




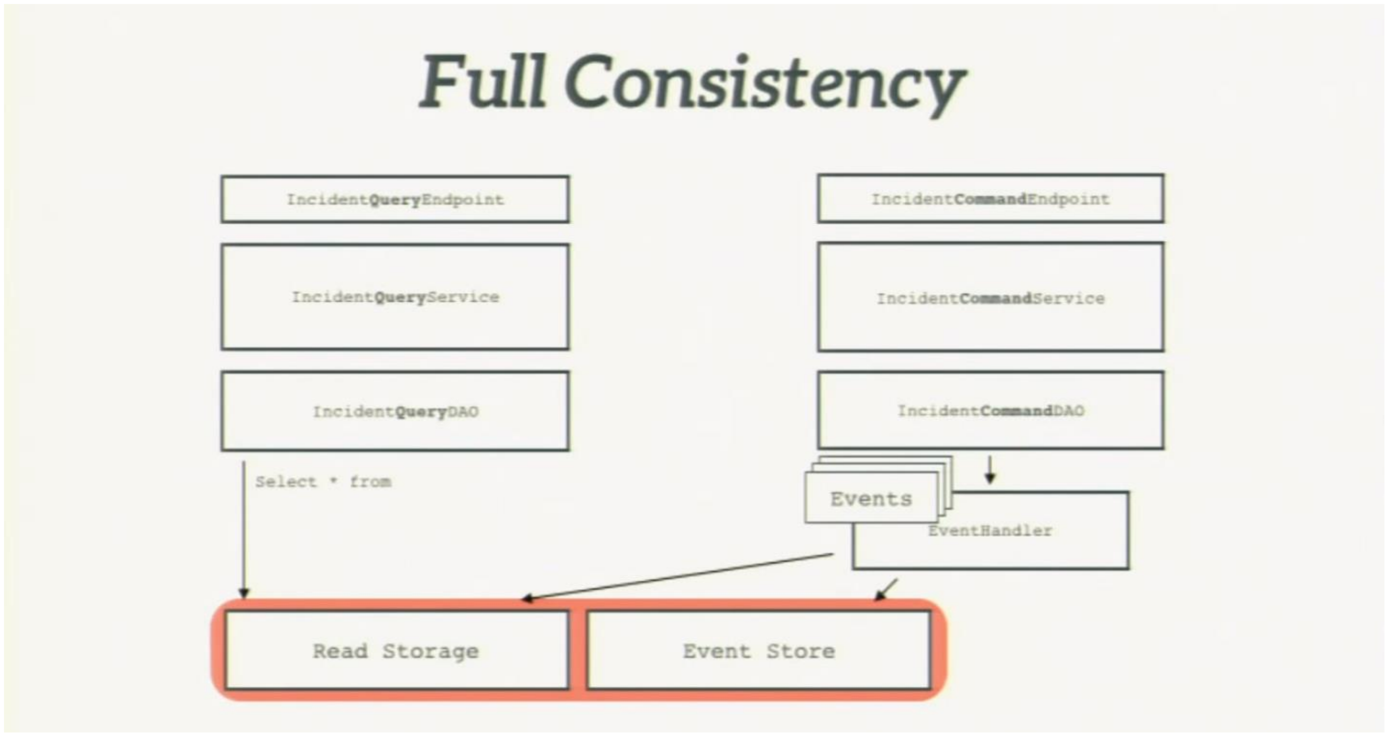
Eventual Consistency

There are consistency risks at the Events level, the EventHandler level, or the Read Storage if we get issues. We can lose event being stored in the if the event store goes does. From the ***CAP Theorem***, Consistency, Availability and Partition Tolerance, ***mostly is that we need to pick between Consistency over Availability***.

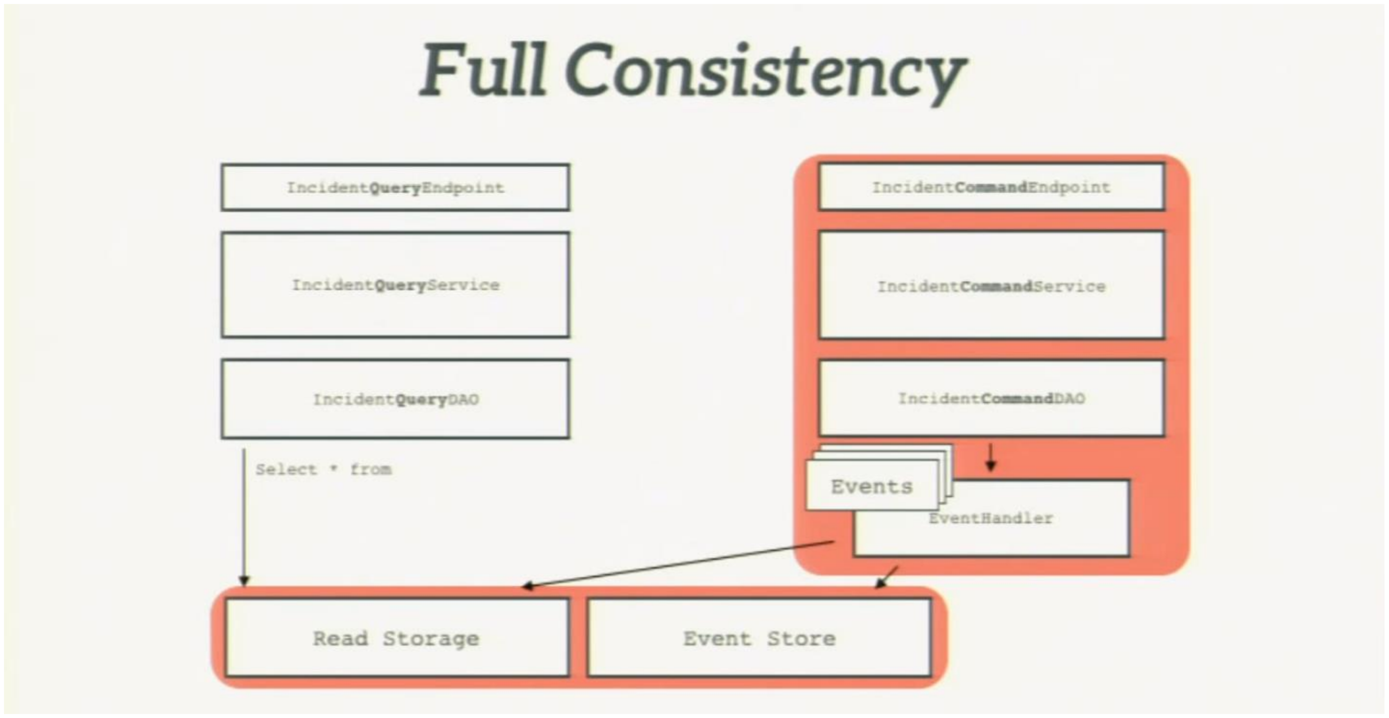
BUT



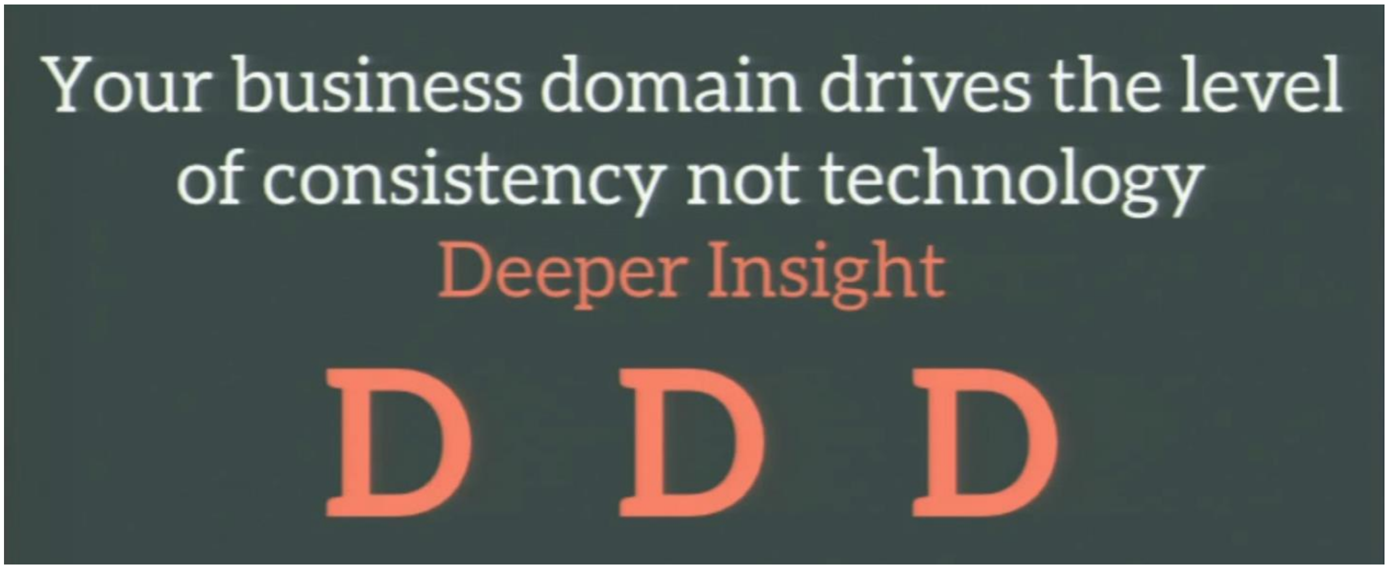
! You can build a fully consistent system which follows Event Sourcing principles



We can go ahead and place the Event Store and the Read Storage in a RDBMS on the same node



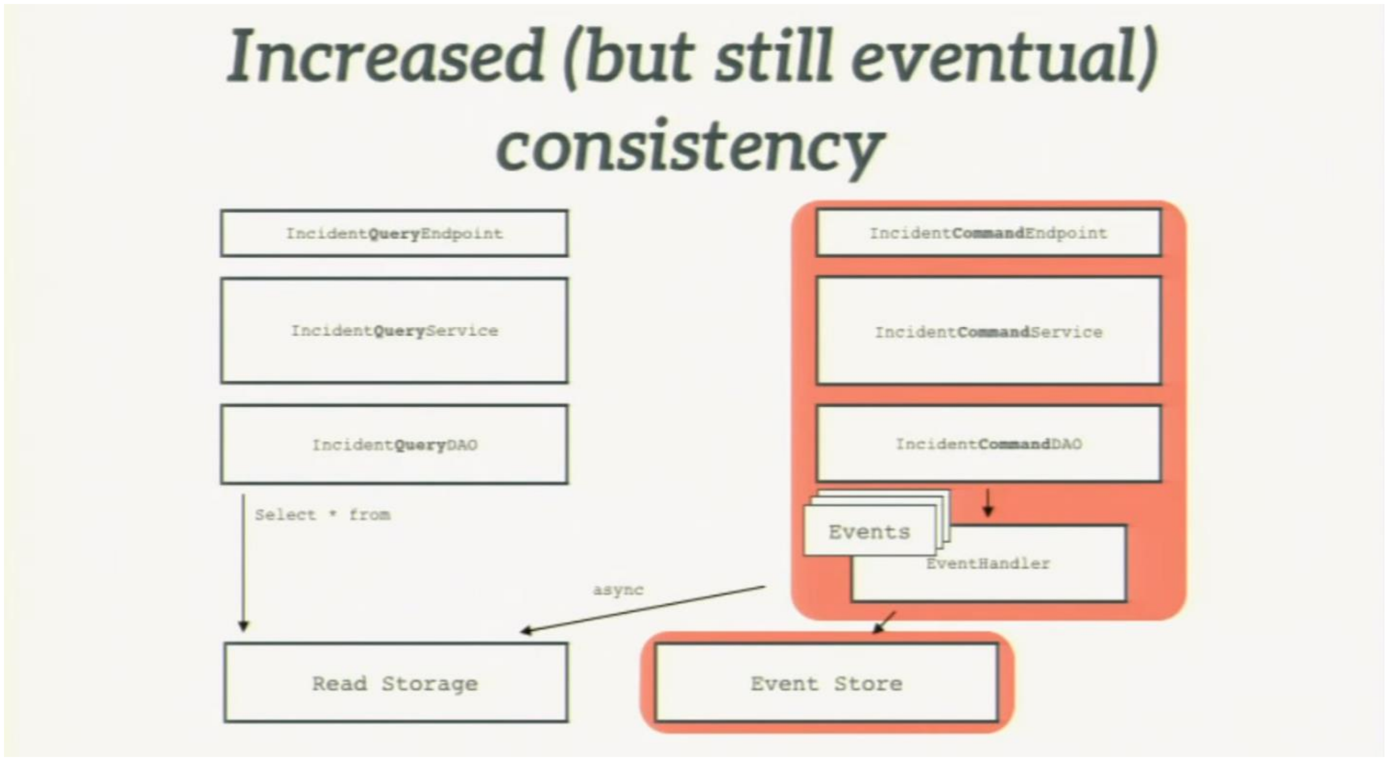
Then we could add the event to both the Event Store and the Read Storage in a single transaction, this will then provide very high transactional guarantee and consistent, but this system will not scale very well for Availability!!!


Your business domain drives the level of consistency not technology

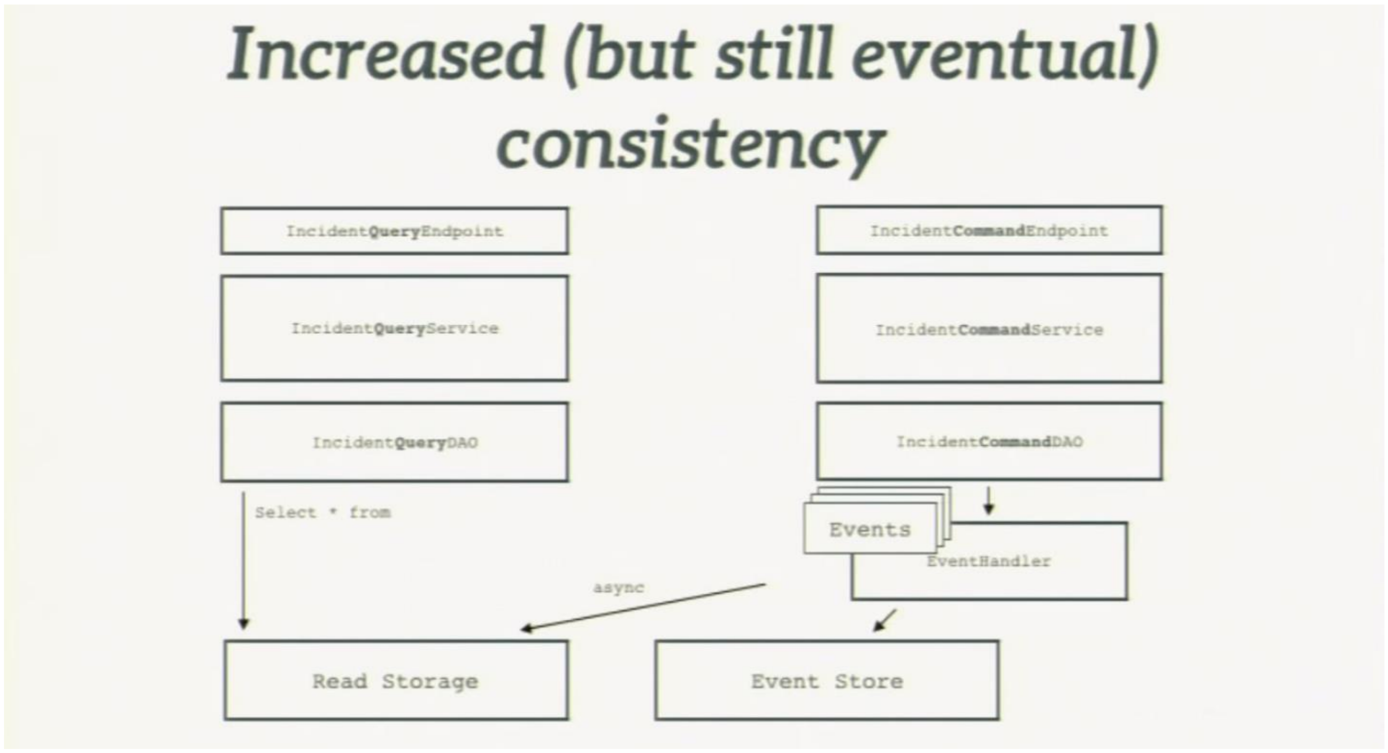
Deeper Insight

D D D

You need to ***consider the business value of failed transactions in your system*** as well as the ***business and regulatory requirements*** for your use case ***before making a decision regarding the level of data consistency*** you want.

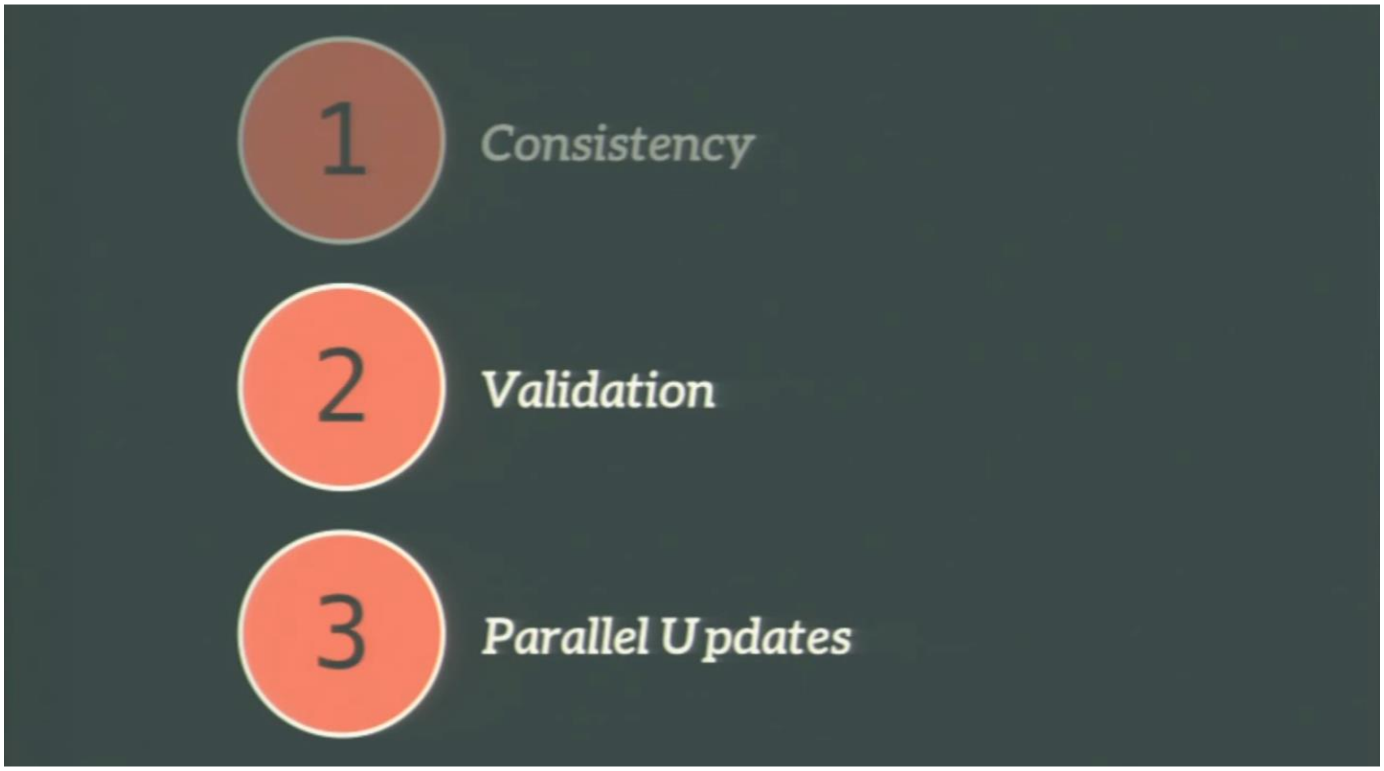


You could decide to add events to the Event store only and do an async storage to the Read Storage as above



We can use updates to the read storage more infrequently

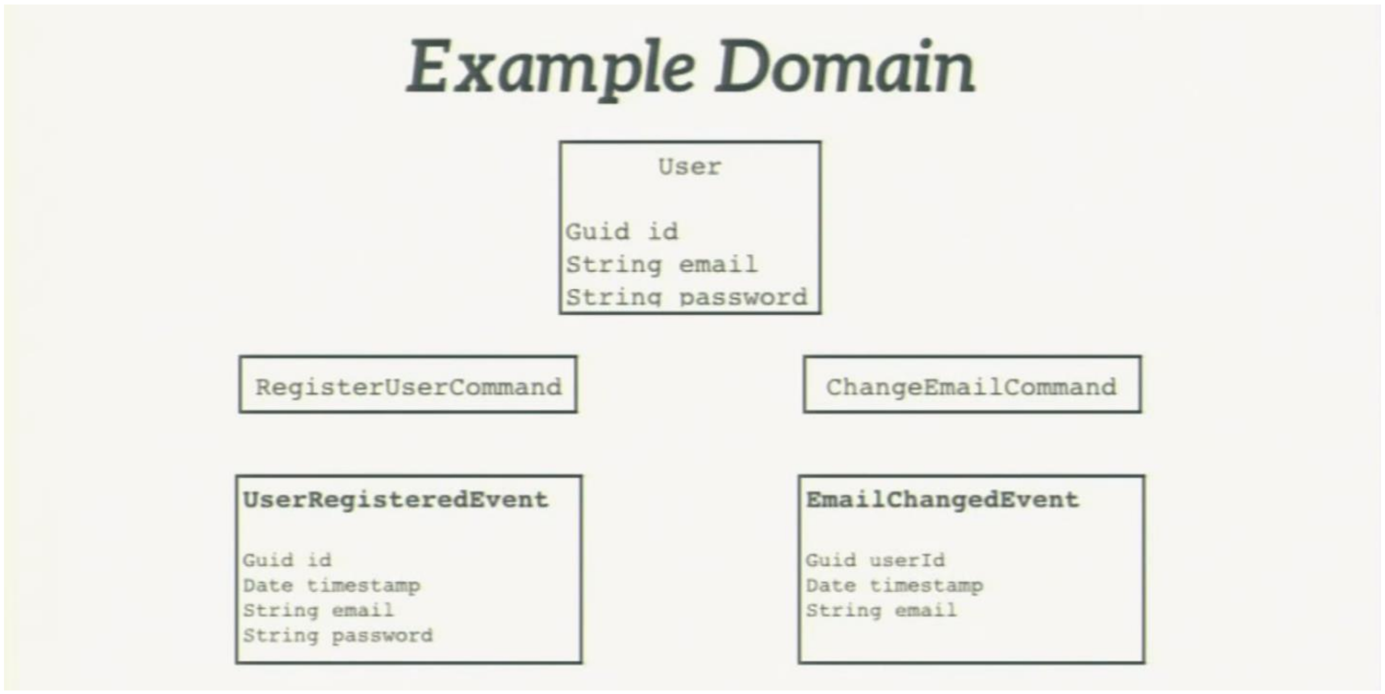
There is no standard solution


1 Consistency

2 Validation

3 Parallel Updates

Validation is easier because we can do things like null checks for some required fields. Uniqueness validations are different



Example Domain

```
User

Guid id
String email
String password
```

```
RegisterUserCommand
```

```
ChangeEmailCommand
```

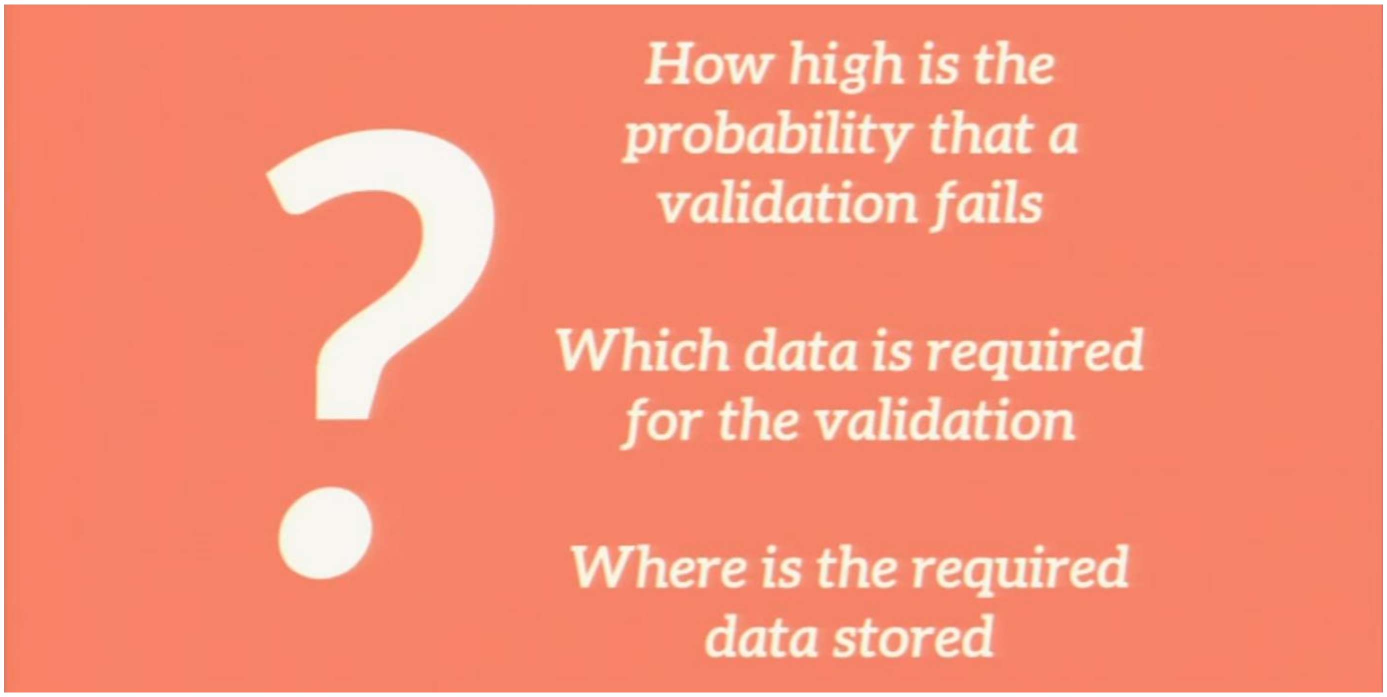
```
UserRegisteredEvent

Guid id
Date timestamp
String email
String password
```

```
EmailChangedEvent

Guid userId
Date timestamp
String email
```

We have 2 commands based on the User object as above,

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

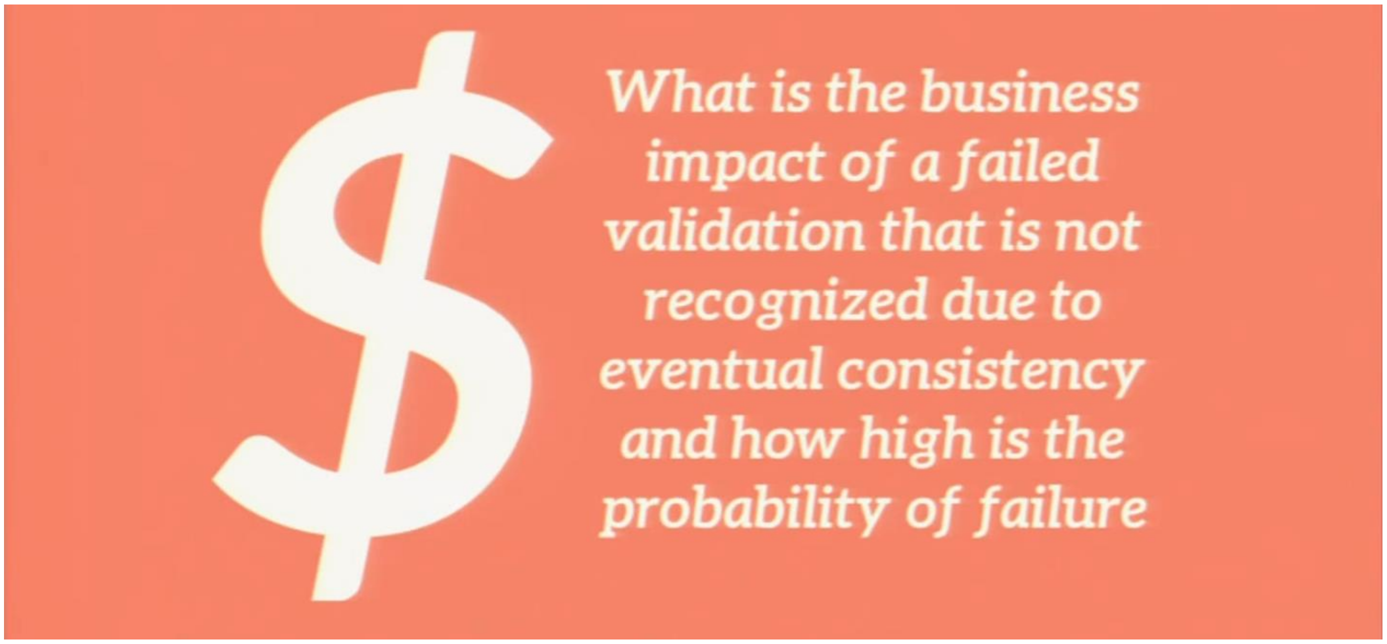
Then we have the requirement above for our event source system. We have a risk of having wrong validation if we do not update our Read Storage immediately after an event occurred, we might not be able to guarantee email uniqueness. We might have to sacrifice some availability in order to increase the consistency level in this use case.



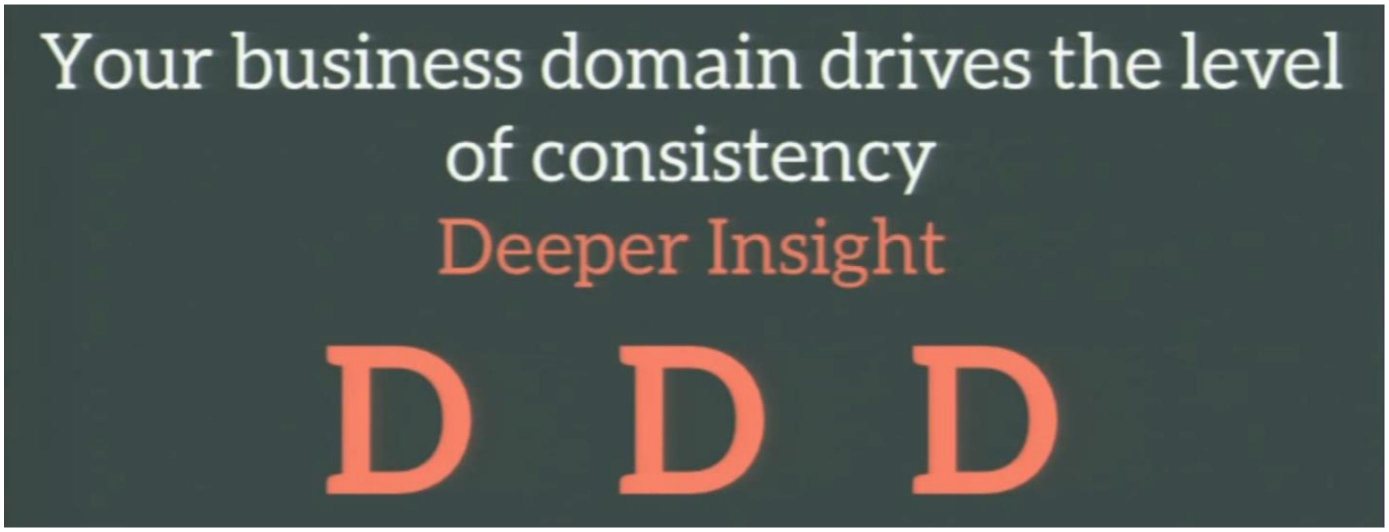
How high is the probability that a validation fails

Which data is required for the validation

Where is the required data stored



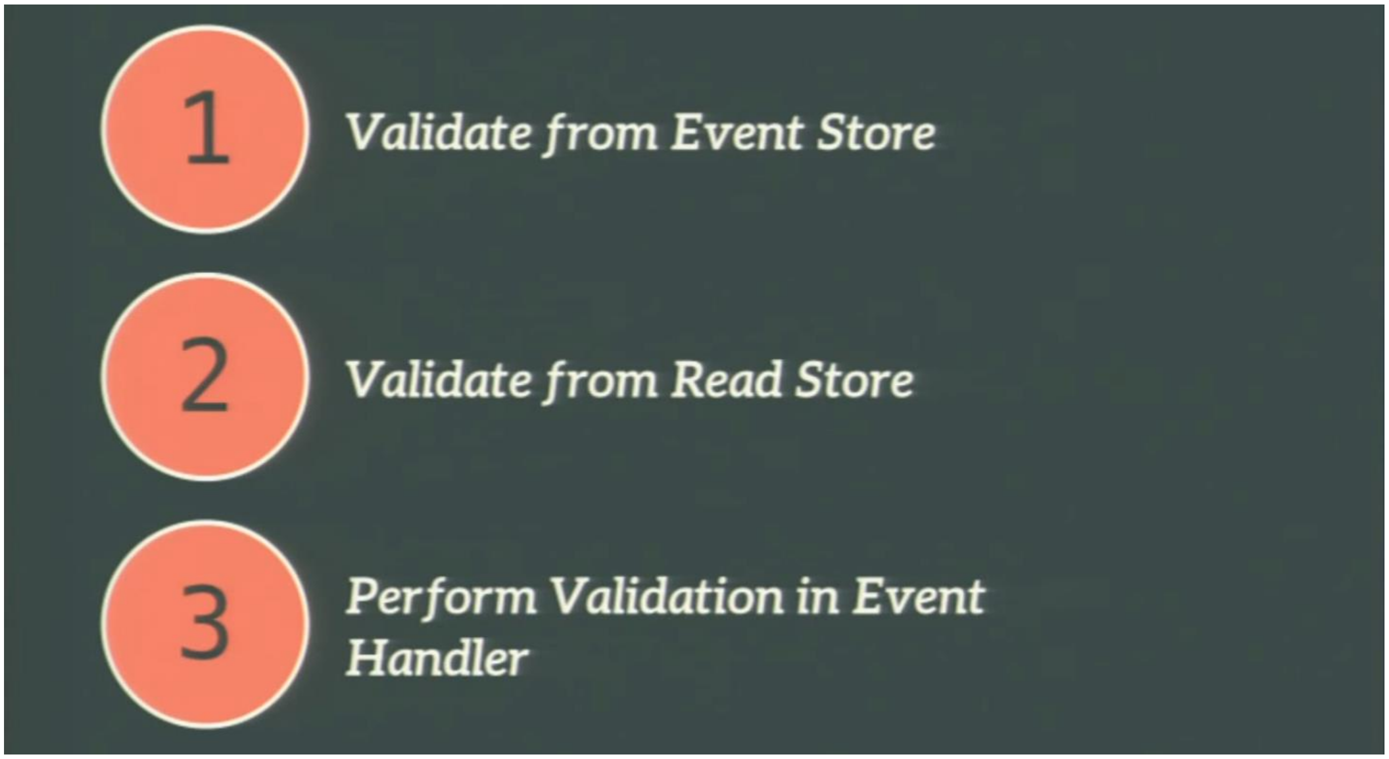
What is the business impact of a failed validation that is not recognized due to eventual consistency and how high is the probability of failure



Your business domain drives the level of consistency

Deeper Insight

D D D

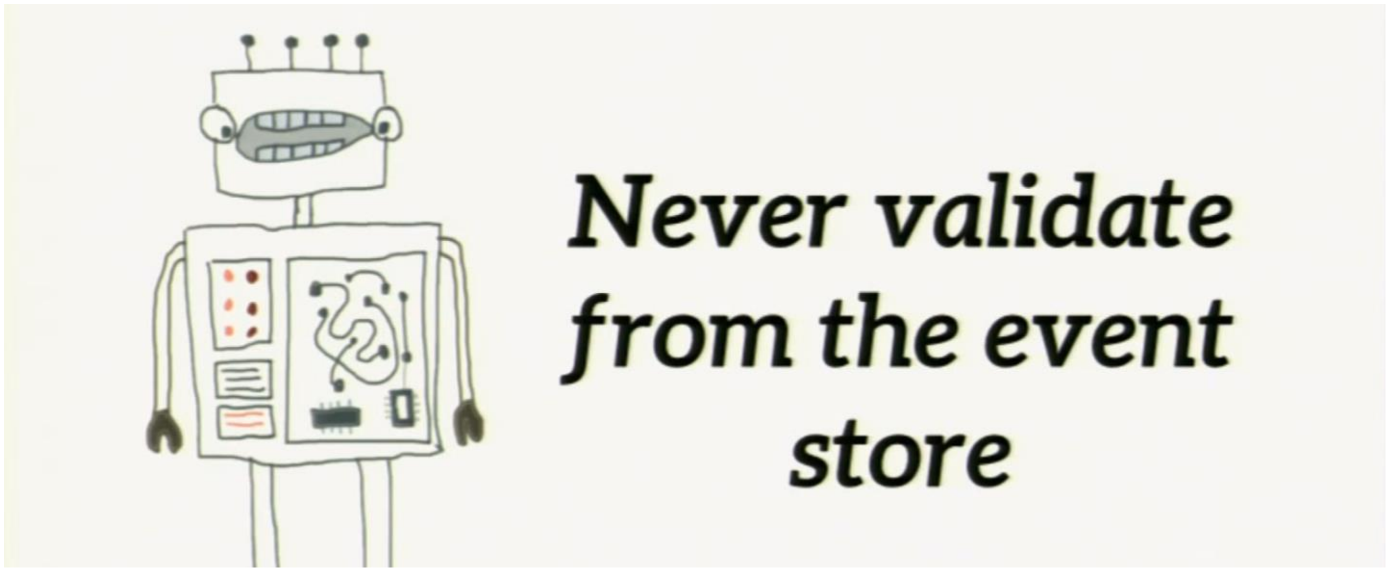


1 Validate from Event Store

2 Validate from Read Store

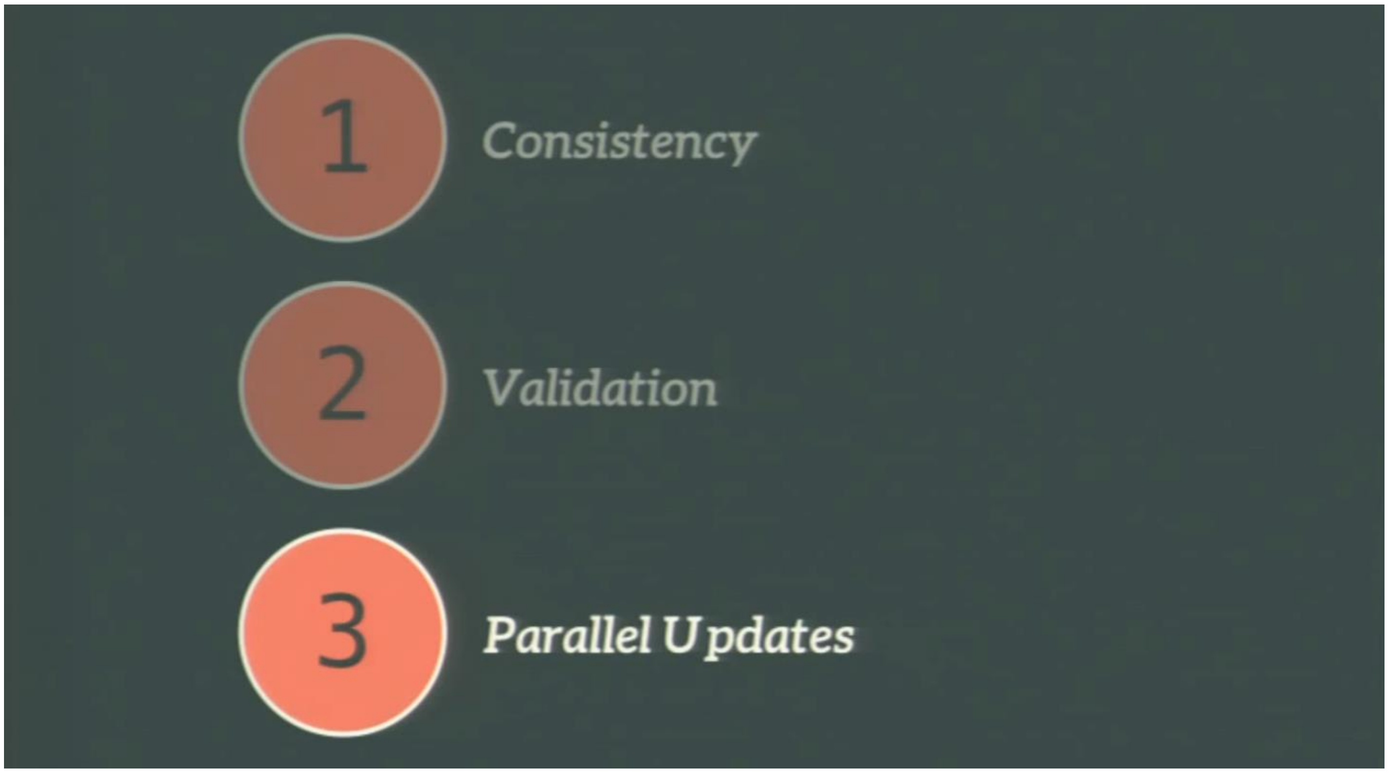
3 Perform Validation in Event Handler

We should strongly validate from the Read Store and update it based on the criticality of the business operation



Never validate from the event store

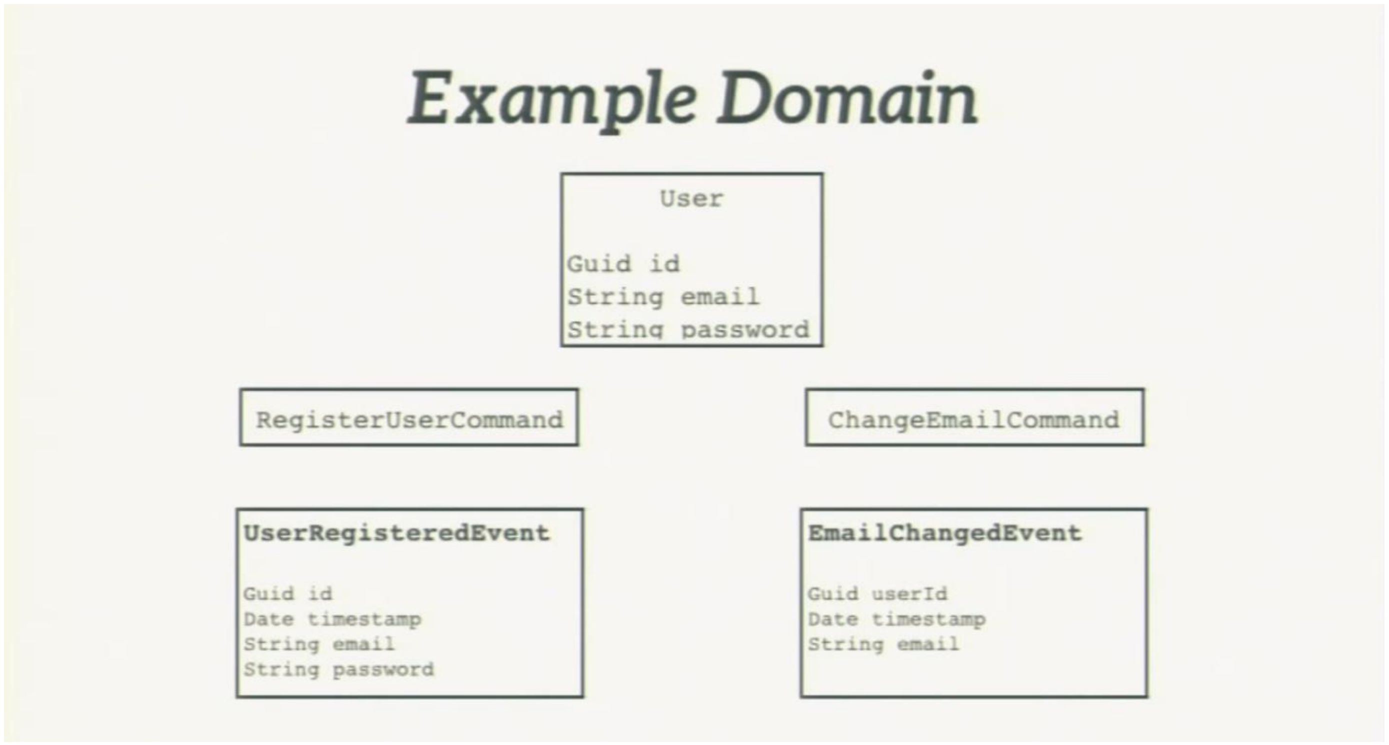
This will create problems in high volume, high transaction systems



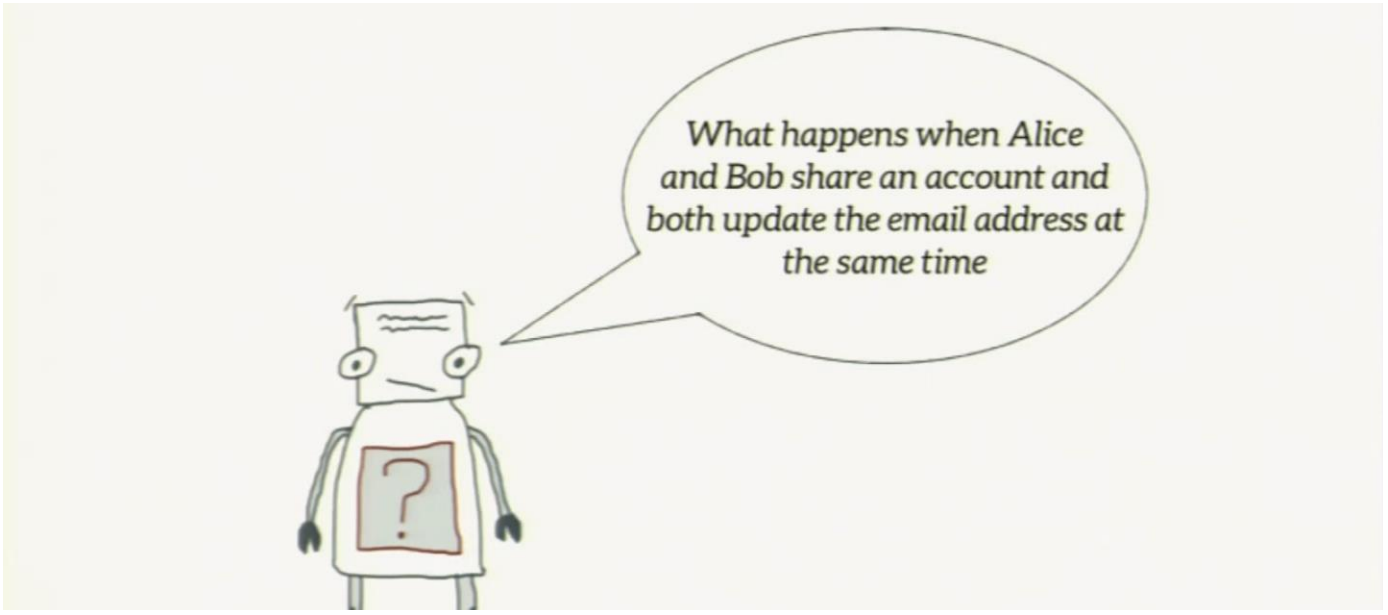
1 Consistency

2 Validation

3 Parallel Updates

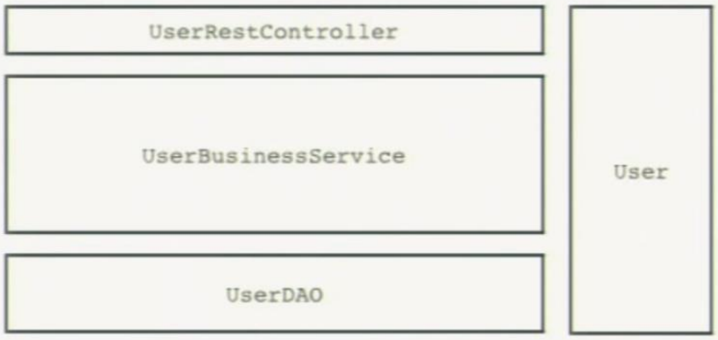


Example Domain

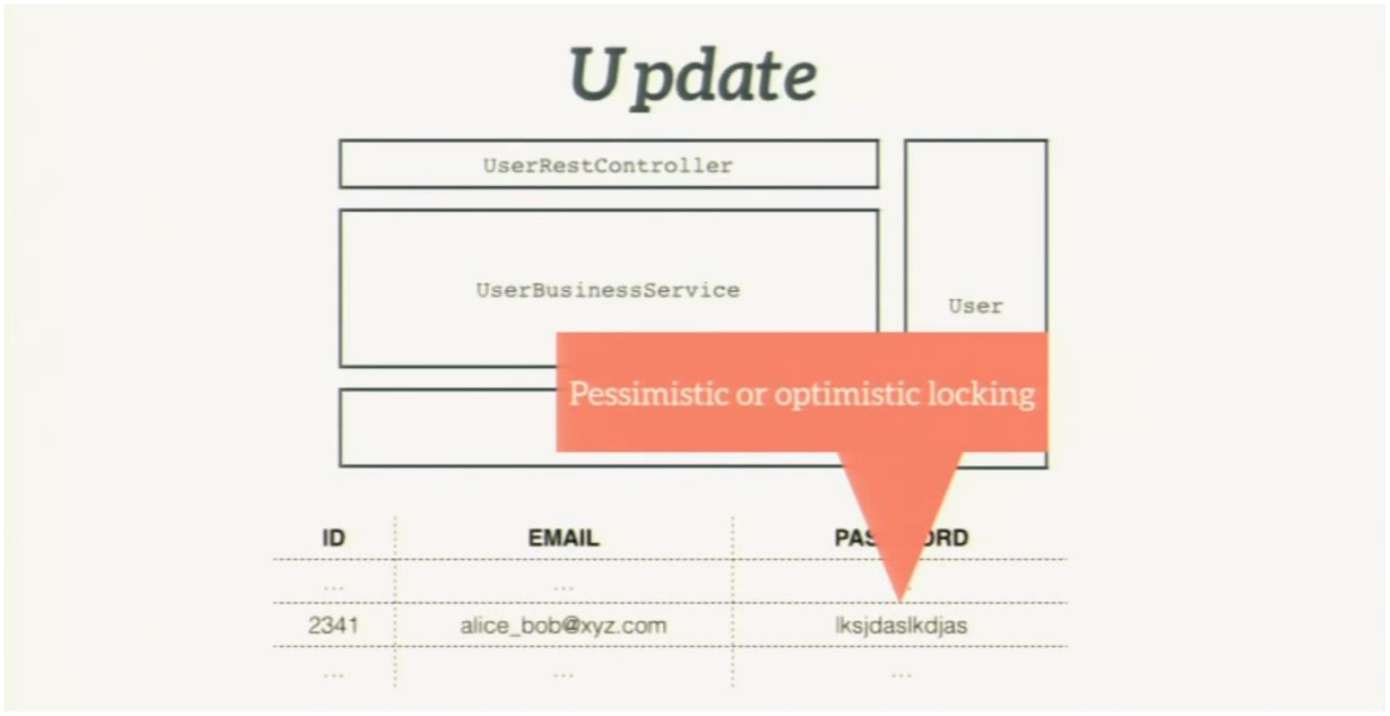

What happens when Alice and Bob share an account and both update the email address at the same time

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

Update



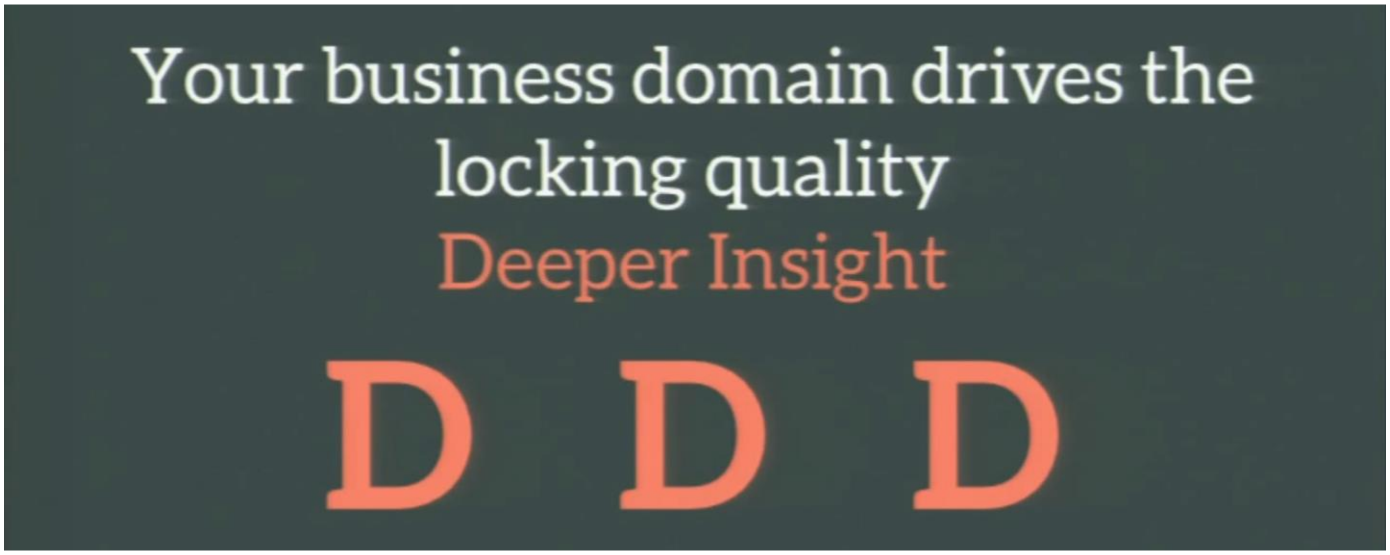
ID	EMAIL	PASSWORD
...	...	...
2341	alice_bob@xyz.com	lksjdaslkdjas
...	...	...



Update

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

We usually issue the classical 'SELECT FOR' update command while locking that field



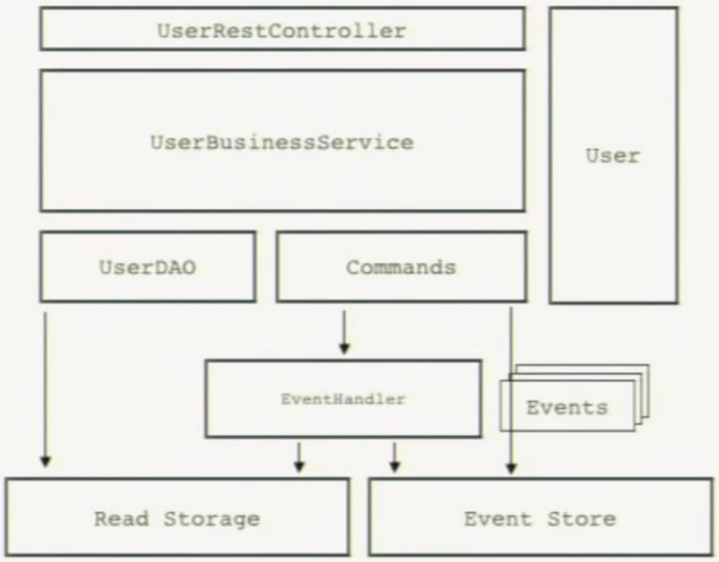
Your business domain drives the locking quality

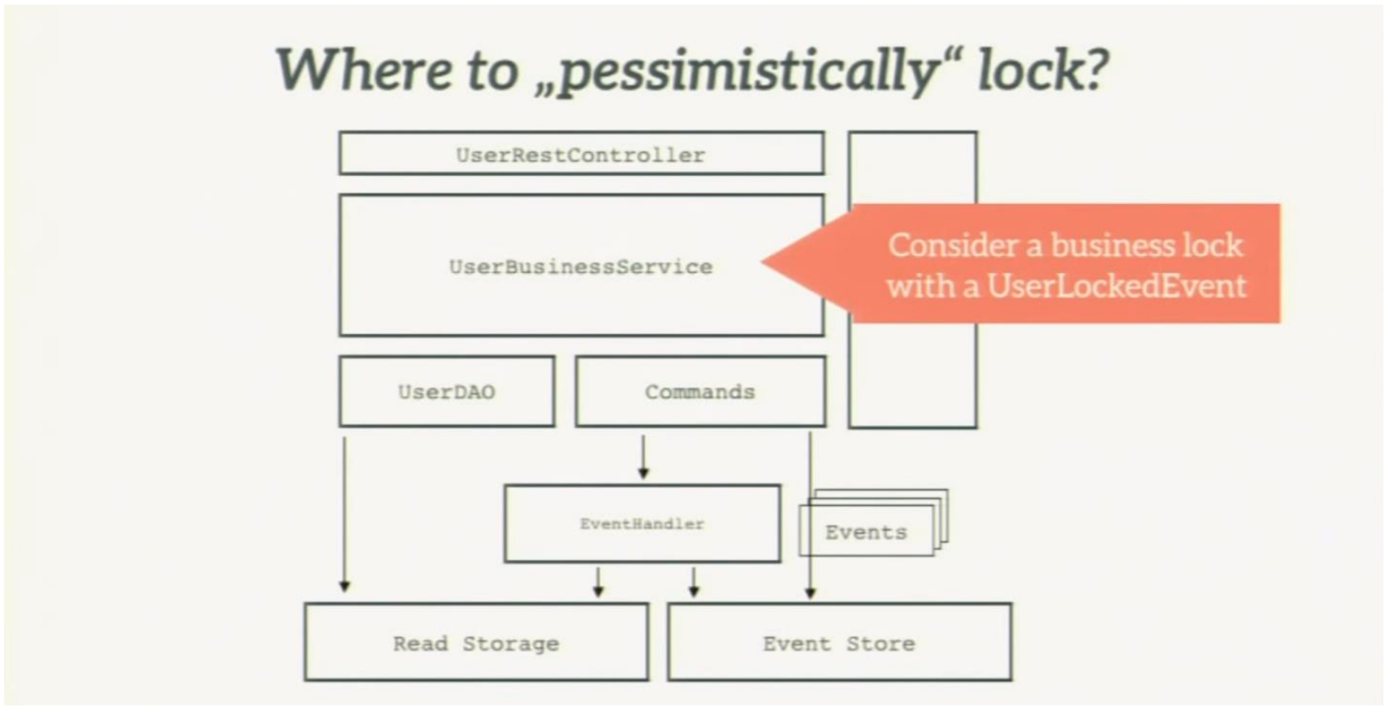
Deeper Insight

D D D

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

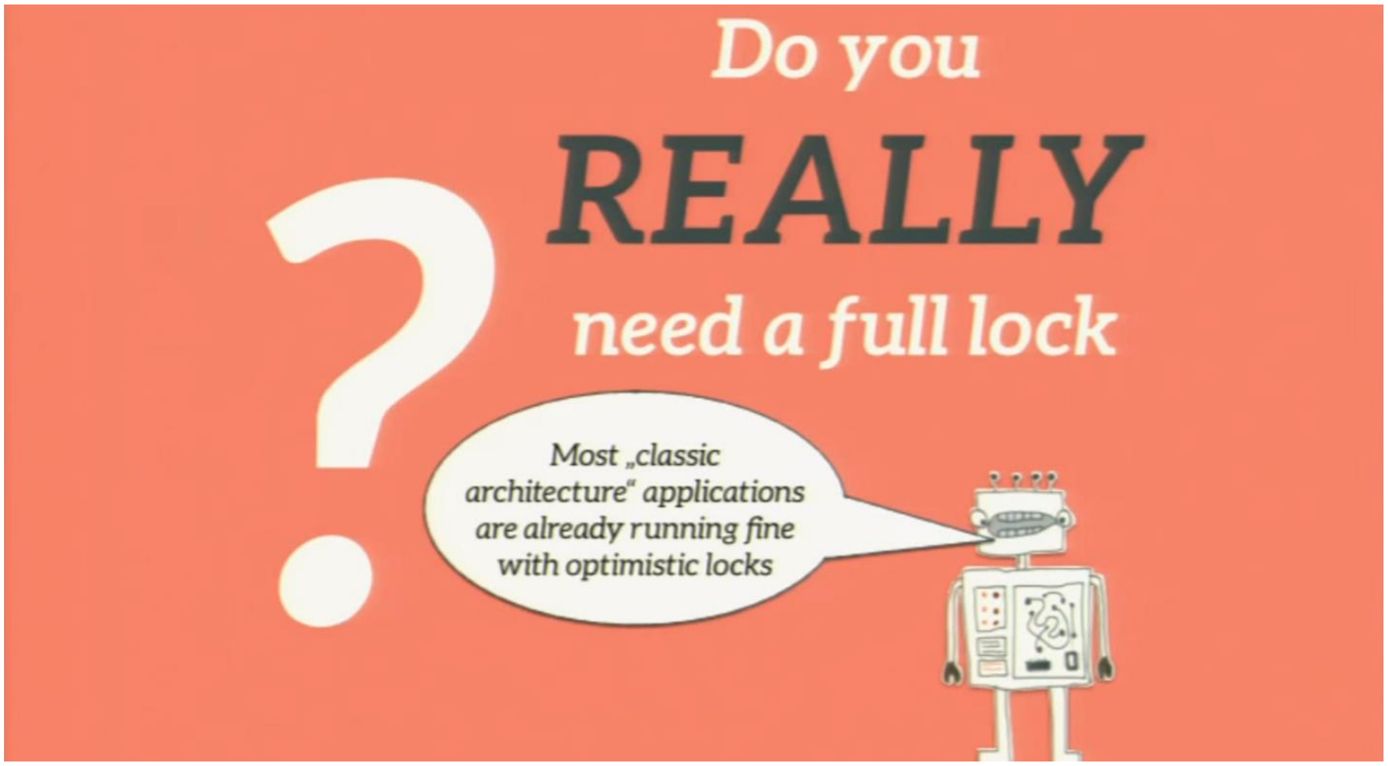
Where to „pessimistically“ lock?





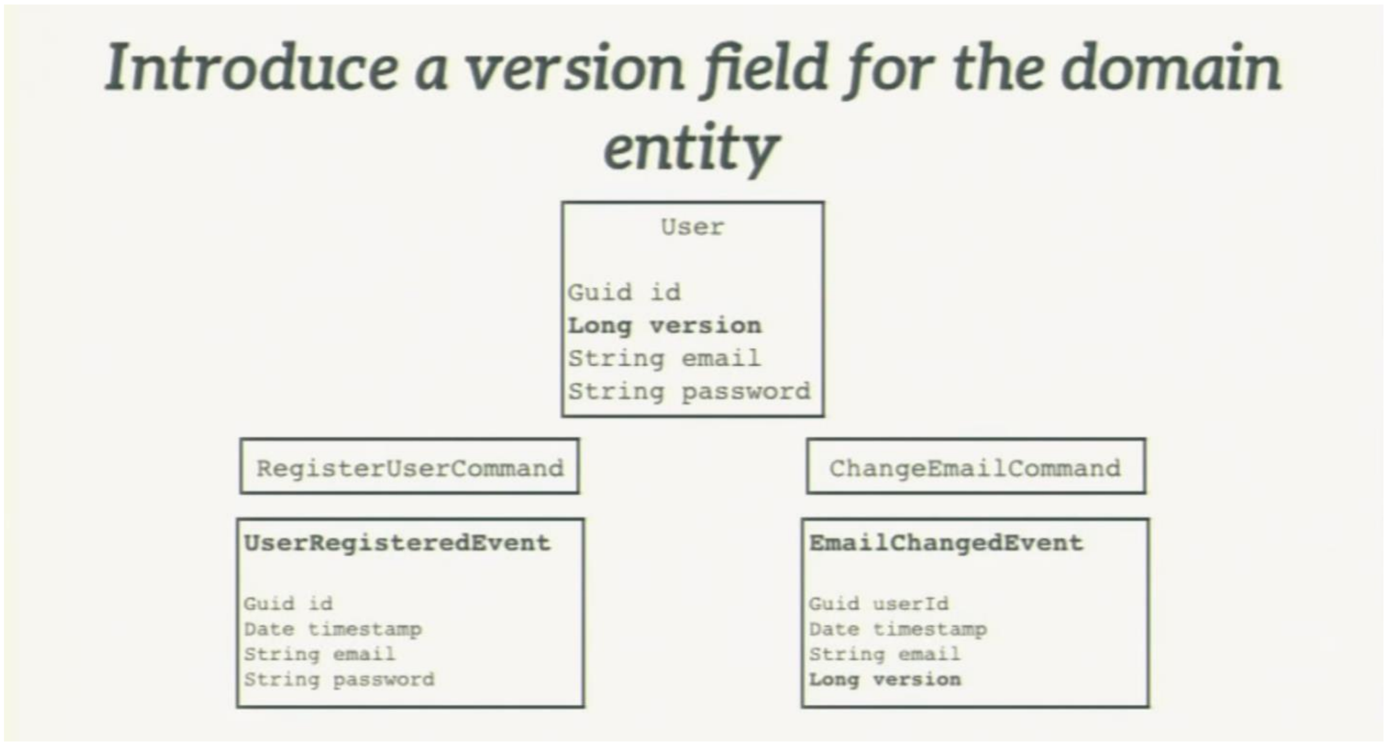
Where to „pessimistically“ lock?

This ‘pessimistic lock’ can be permitted in a mortgage application business scenario



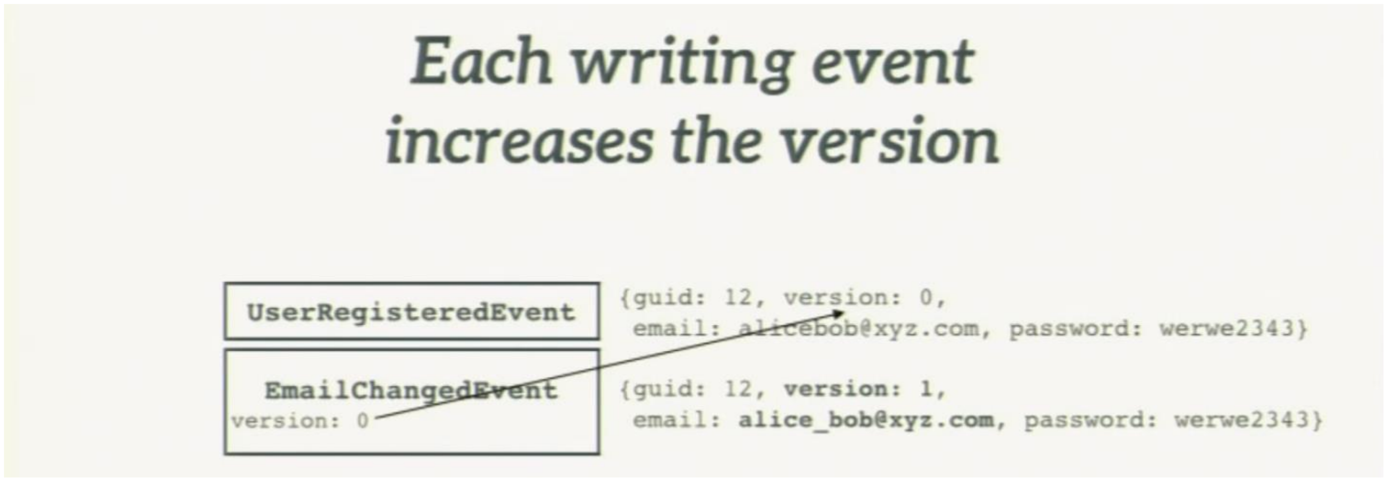
Do you **REALLY** need a full lock

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



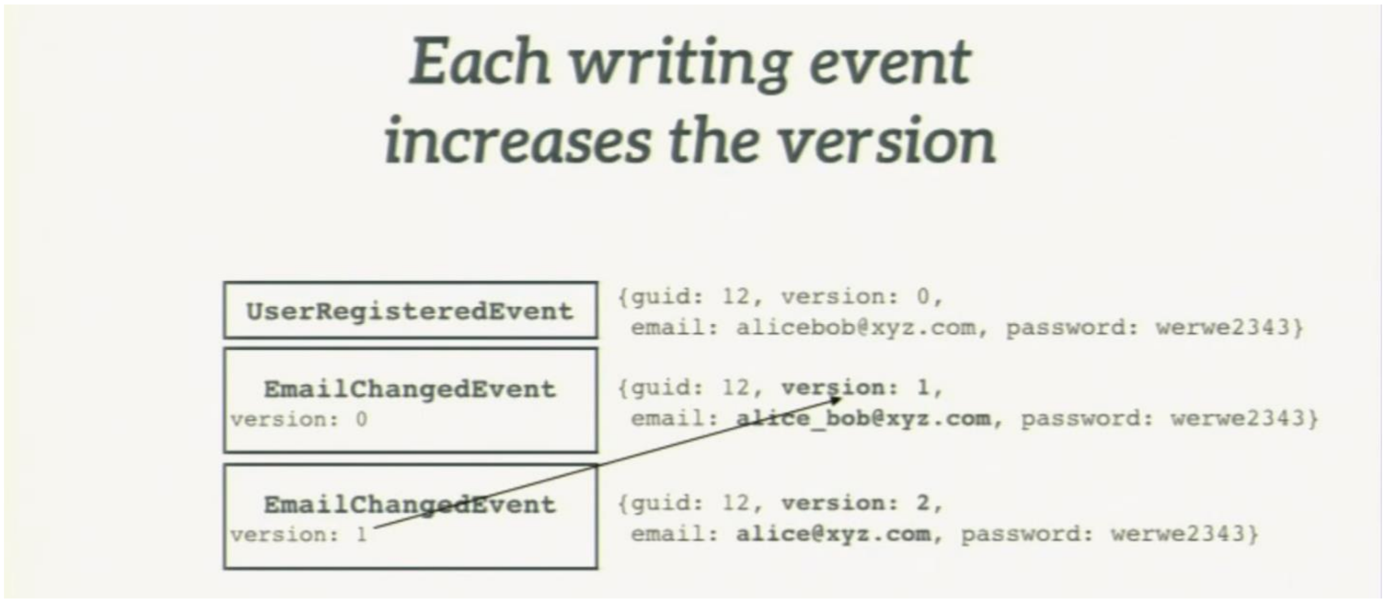
Introduce a version field for the domain entity

This will be an ‘optimistic locking’ scenario

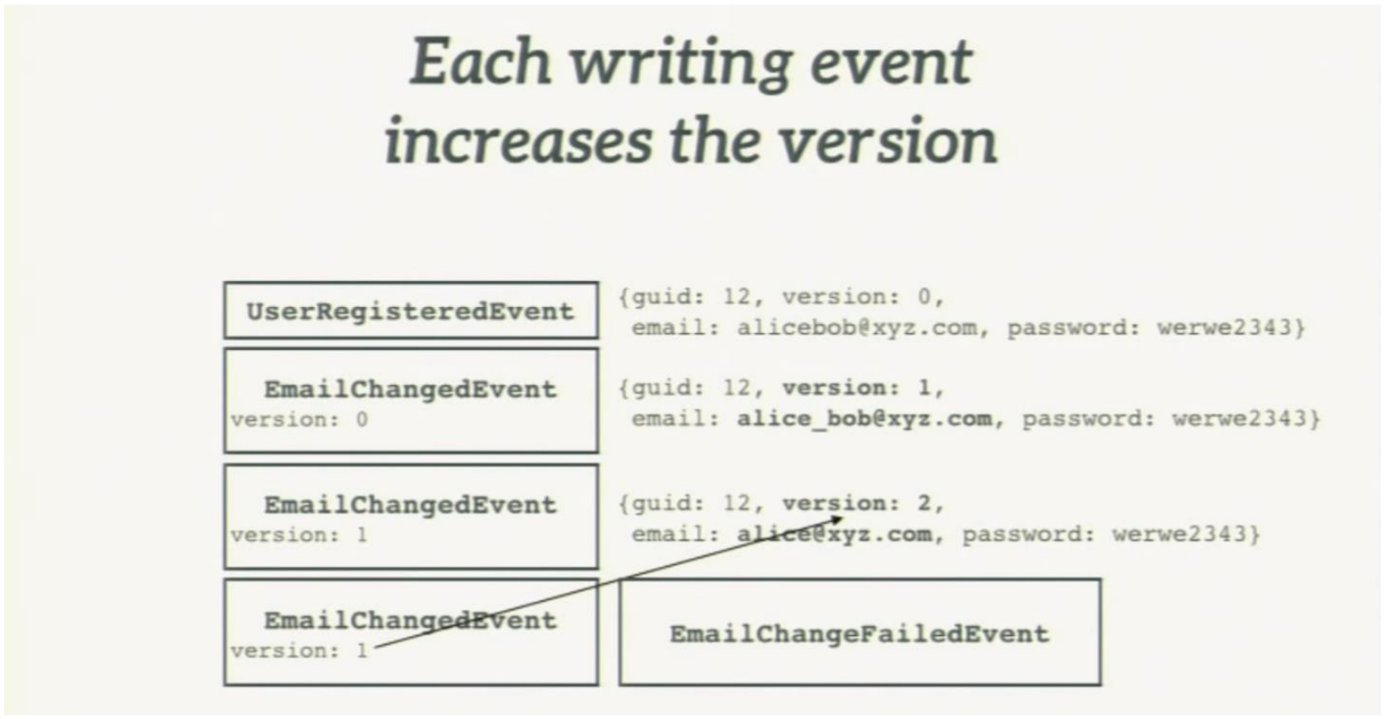


Each writing event increases the version

UserRegisteredEvent
{guid: 12, version: 0,
email: alicebob@xyz.com, password: werwe2343}
EmailChangedEvent
version: 0
{guid: 12, version: 1,
email: alice_bob@xyz.com, password: werwe2343}



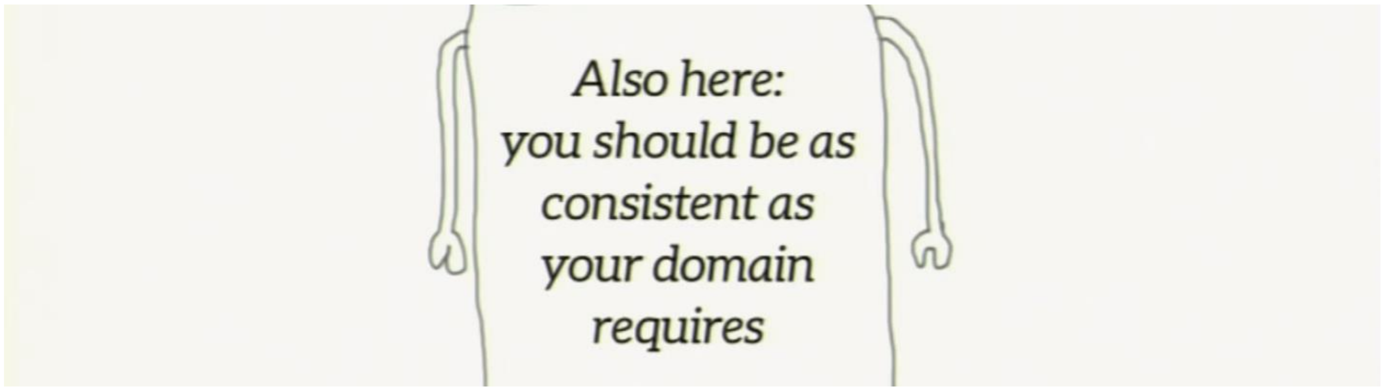
Each writing event increases the version



Each writing event increases the version

```
UserRegisteredEvent        {guid: 12, version: 0,
                            email: alicebob@xyz.com, password: werwe2343}
EmailChangedEvent          {guid: 12, version: 1,
version: 0                  email: alice_bob@xyz.com, password: werwe2343}
EmailChangedEvent          {guid: 12, version: 2,
version: 1                  email: alice@xyz.com, password: werwe2343}
EmailChangedEvent          EmailChangeFailedEvent
version: 1
```

This will be detected when we update our Read Store and we can raise another event to notify the user to go and change their email again



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