Developing event-driven microservices with event sourcing and CQRS

Chris Richardson

Author of POJOs in Action

Founder of the original CloudFoundry.com

@crichardson

chris@chrisrichardson.net

http://plainoldobjects.com

http://microservices.io

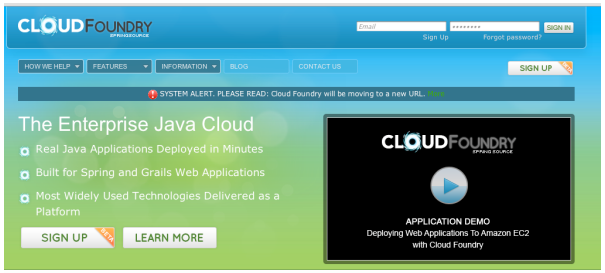
Presentation goal

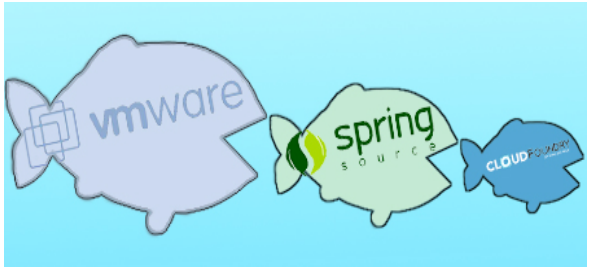
Show how Event Sourcing and

Command Query Responsibility Segregation (CQRS)

are a great way to implement microservices

About Chris


CLOUDFOUNDRY
Email
SIGN IN
Sign Up
Forgot password?
HOW WE HELP
FEATURES
INFORMATION
BLOG
CONTACT US
SIGN UP
SYSTEM ALERT. PLEASE READ: Cloud Foundry will be moving to a new URL.
The Enterprise Java Cloud
Real Java Applications Deployed in Minutes
Built for Spring and Grails Web Applications
Most Widely Used Technologies Delivered as a Platform
SIGN UP
LEARN MORE
CLOUDFOUNDRY
APPLICATION DEMO
Deploying Web Applications To Amazon EC2
with Cloud Foundry


vmware
spring
source
CLOUDFOUNDRY

About Chris

- Founder of a startup that's creating a platform for developing event-driven microservices: http://eventuate.io/
- Consultant helping organizations improve how they architect and deploy applications using cloud, micro services, polyglot applications, NoSQL, ...
- Creator of http://microservices.io

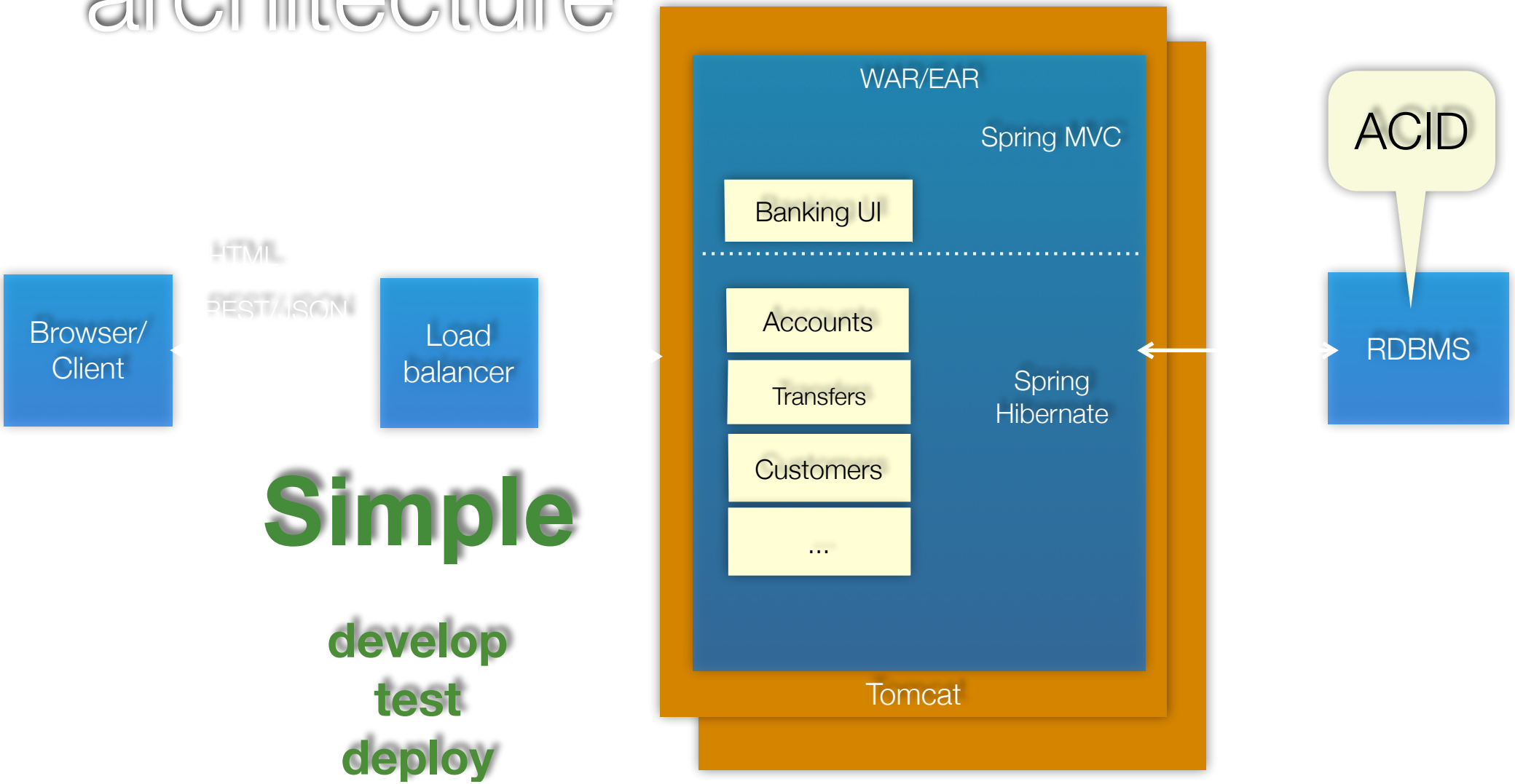
For more information

- http://microservices.io
- http://github.com/cer/microservices-examples
- https://github.com/cer/event-sourcing-examples
- http://plainoldobjects.com/
- https://twitter.com/crichardson

Agenda

- Why build event-driven microservices?
- Overview of event sourcing
- Designing microservices with event sourcing
- Implementing queries in an event sourced application

Traditional monolithic architecture



Simple

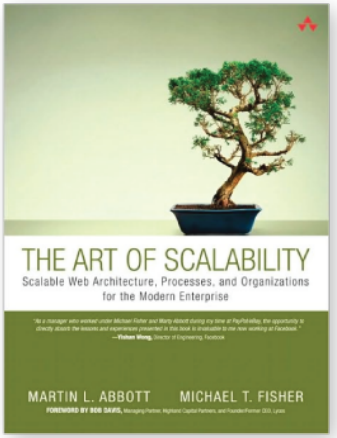
develop
test
deploy
scale

But large and/or complex monolithic applications

=

Trouble!

Apply the scale cube



Y axis - functional decomposition

Scale by splitting different things

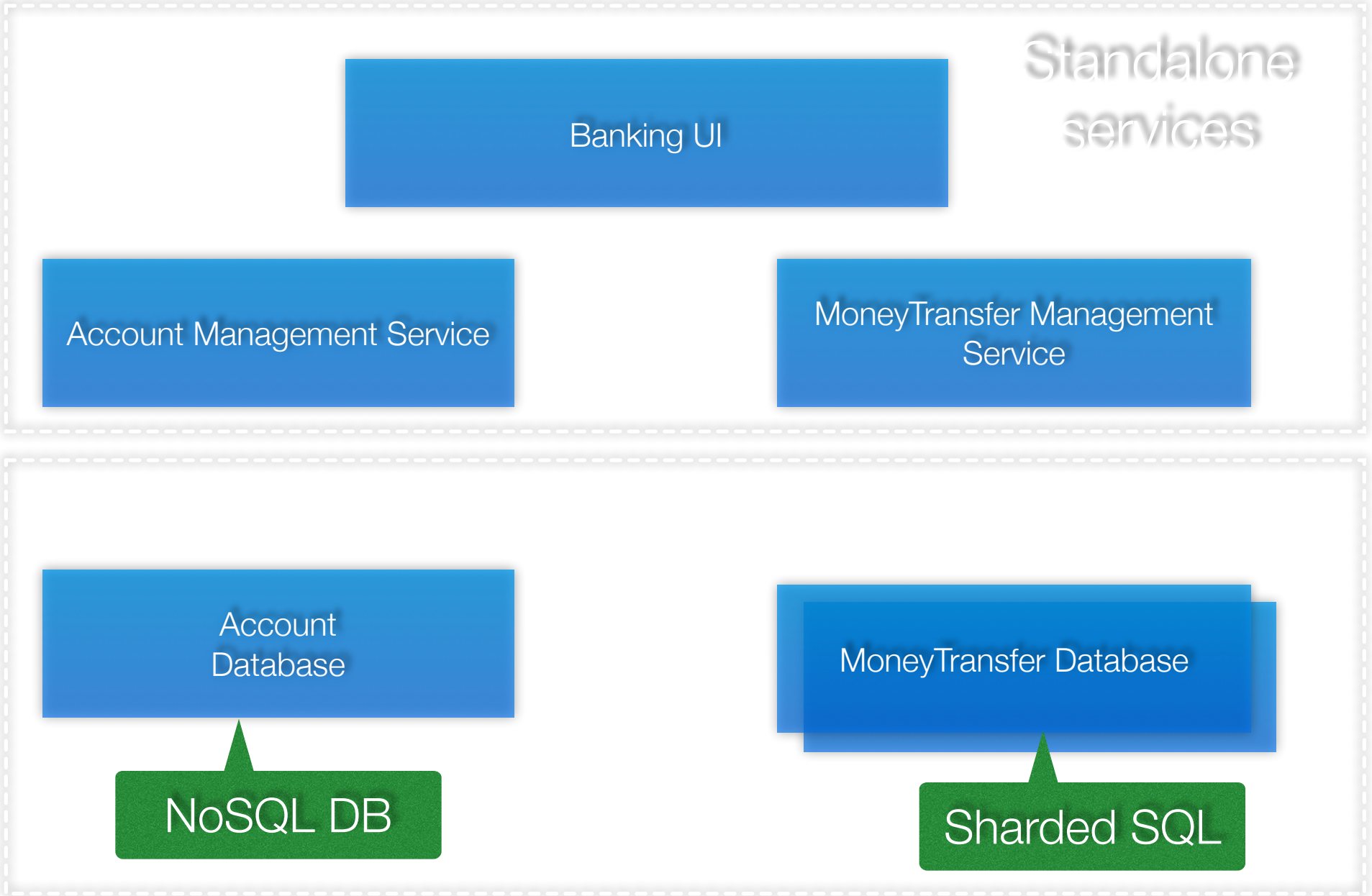
Z axis - data partitioning

Scale by splitting similar things

X axis

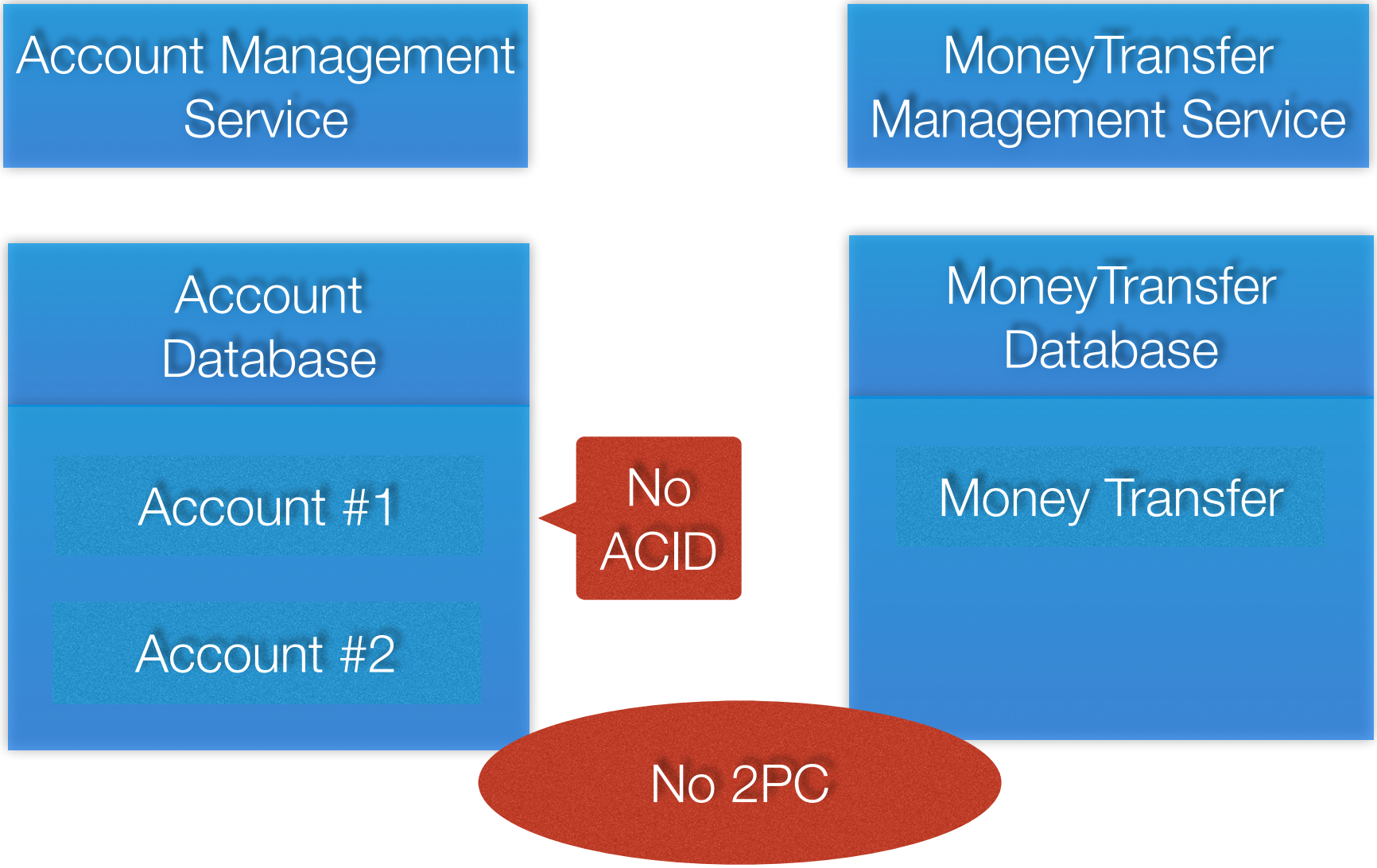
- horizontal duplication

Today: use a microservice, polyglot architecture



But this results in distributed data management problems

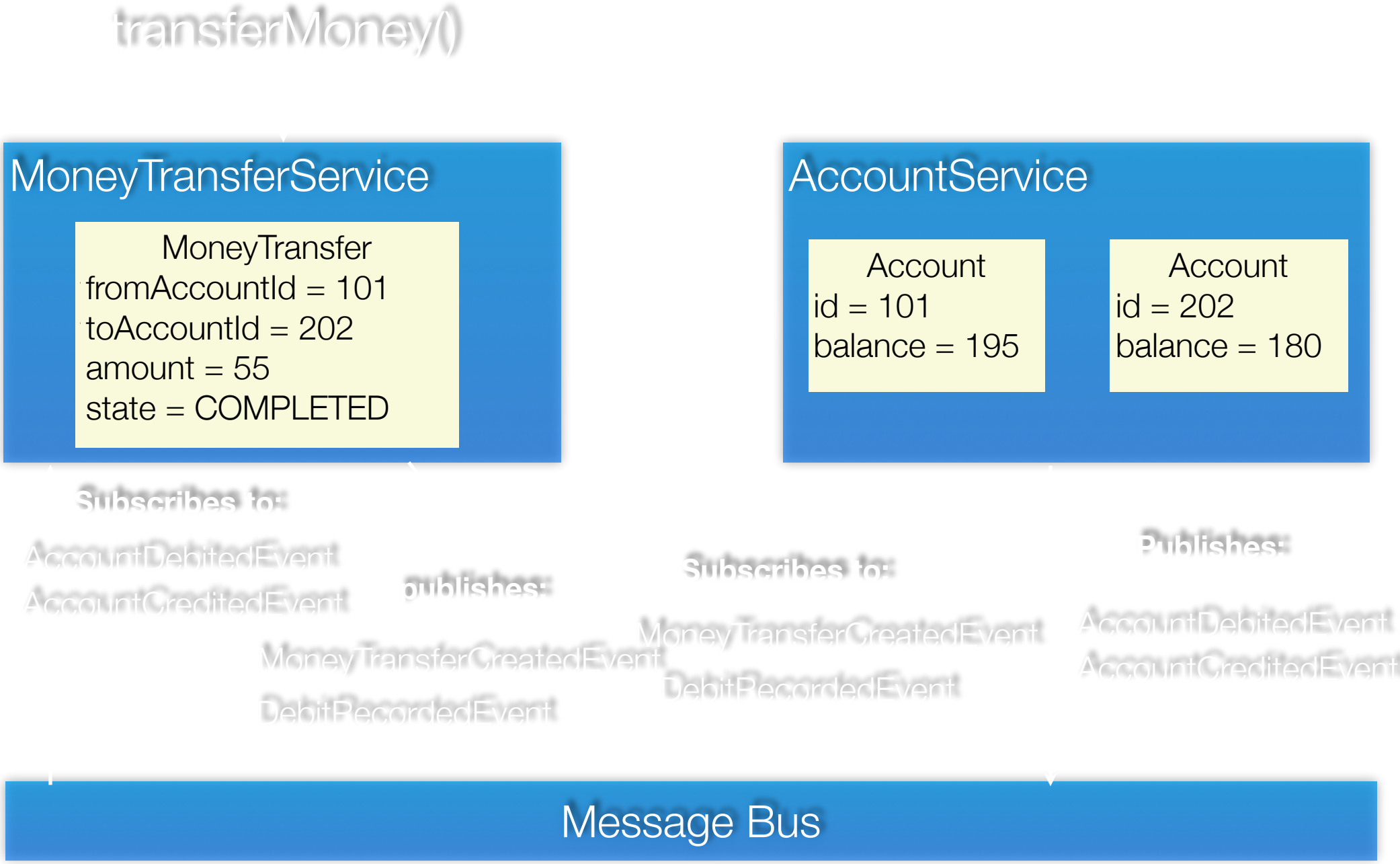
Example: Money transfer



Use an event-driven

- Services publish events when state changes
- Services subscribe to events and update their state
  - Maintain eventual consistency across multiple aggregates (in multiple datastores)
  - Synchronize replicated data

Eventually consistent money transfer



How to
atomically
update state
and
publish an event

Update and publish using 2PC

- Guaranteed atomicity **BUT**
- Need a distributed transaction manager
- Database and message broker must support 2PC
- Impacts reliability
- Not fashionable
- 2PC is best avoided

Use data store as message queue

- Use datastore as a message queue
  - Txn #1: Update database: new entity state & event
  - Txn #2: Consume event
  - Txn #3: Mark event as consumed
- Eventually consistent mechanism (used by eBay)
- See BASE: An Acid Alternative, http://bit.ly/ebaybase
- **BUT**
- Tangled business logic and event publishing code
- Difficult to implement when using a NoSQL database :-(

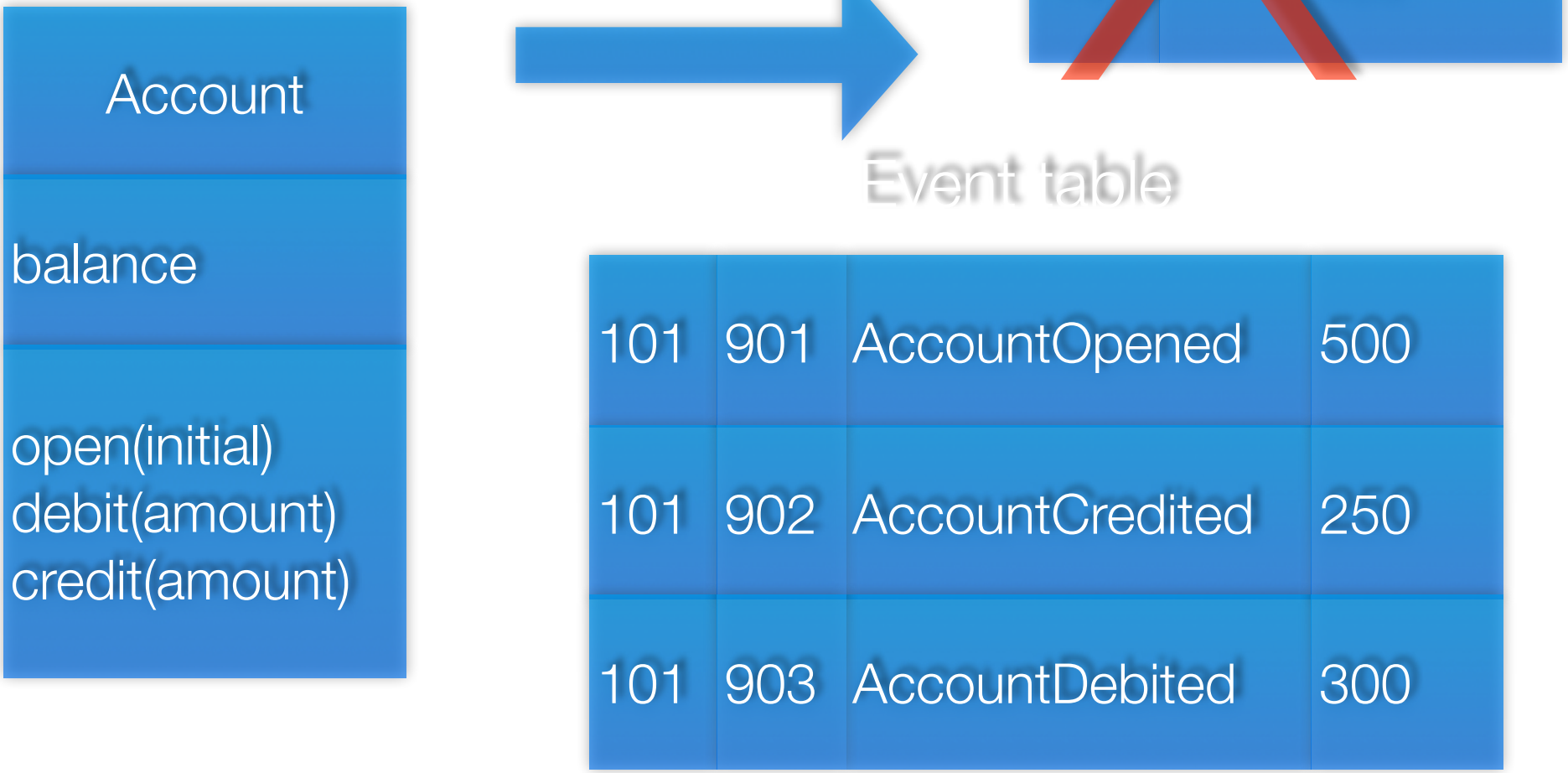
Agenda

- Why build event-driven microservices?
- Overview of event sourcing
- Designing microservices with event sourcing
- Implementing queries in an event sourced application

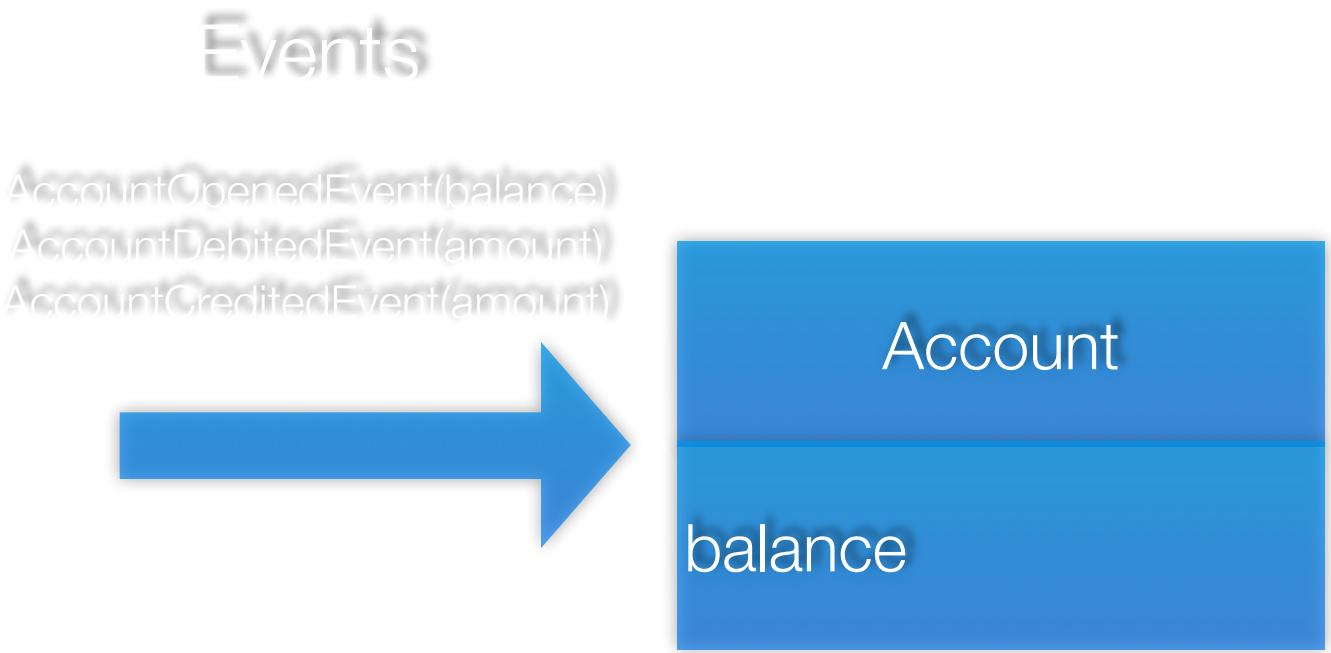
Event sourcing

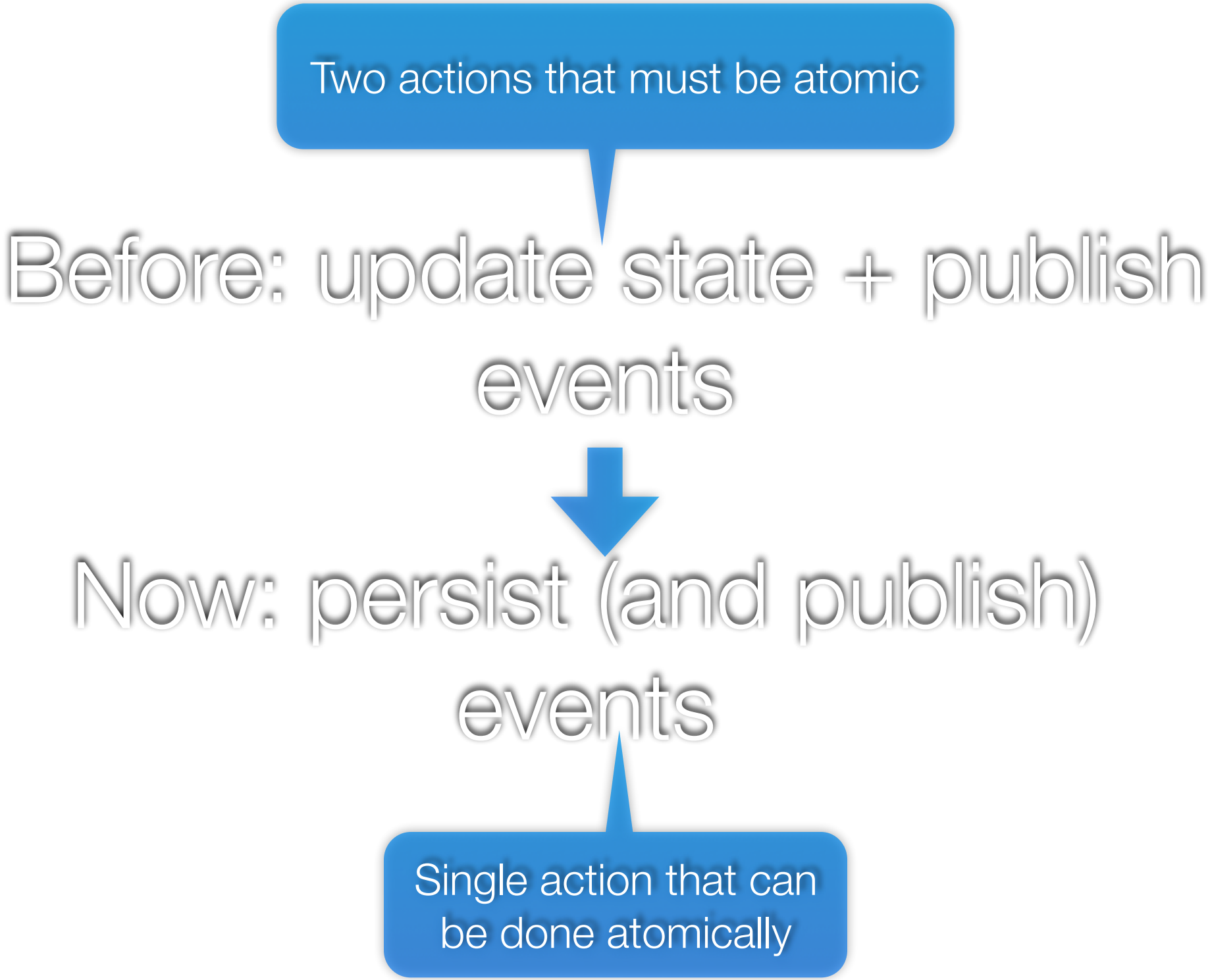
- For each aggregate in your domain model:
  - Identify (state-changing) domain events
  - Define Event classes
- For example,
  - Account: AccountOpenedEvent, AccountDebitedEvent, AccountCreditedEvent
  - ShoppingCart: ItemAddedEvent, ItemRemovedEvent, OrderPlacedEvent

Persists events
NOT current state

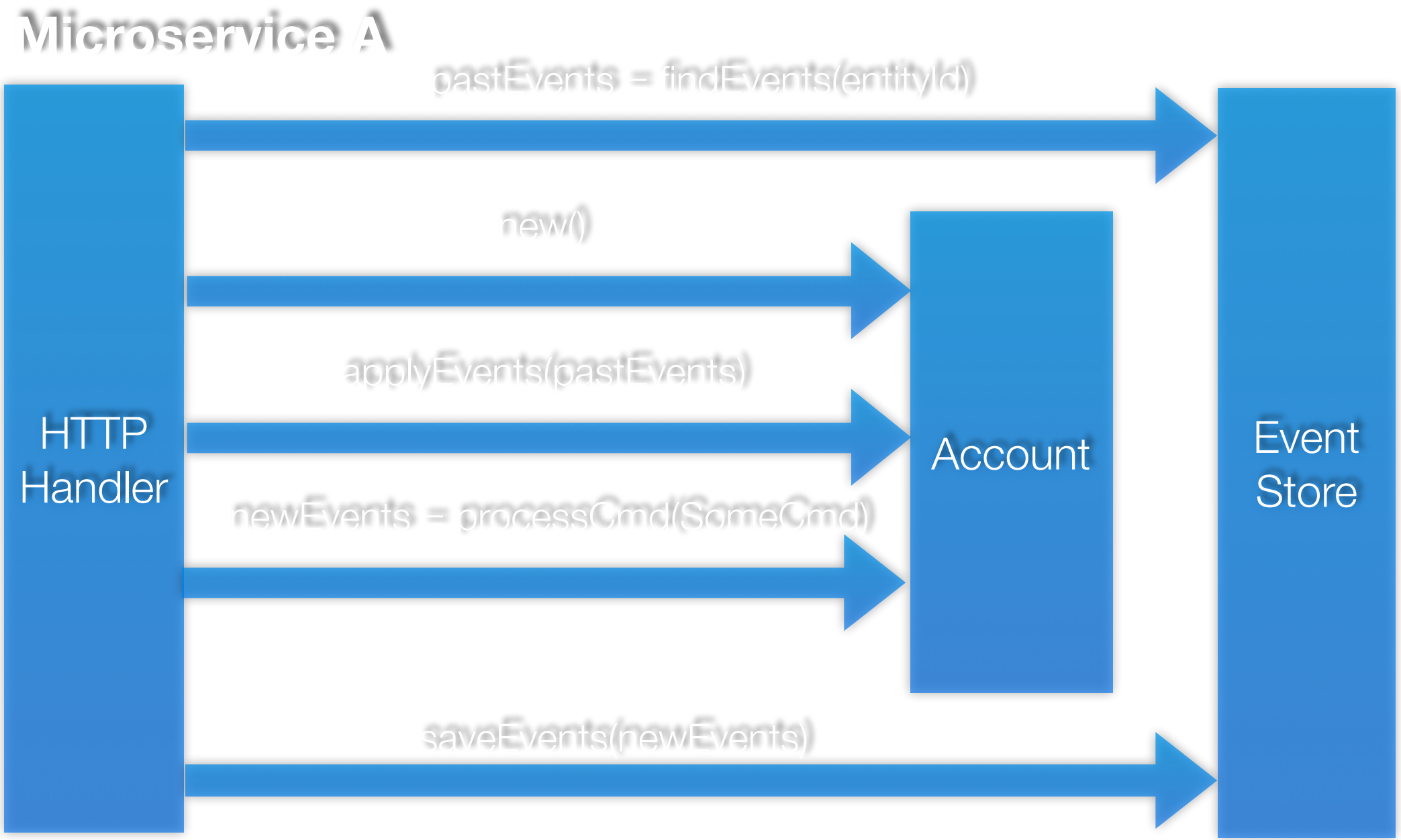


Replay events to recreate state

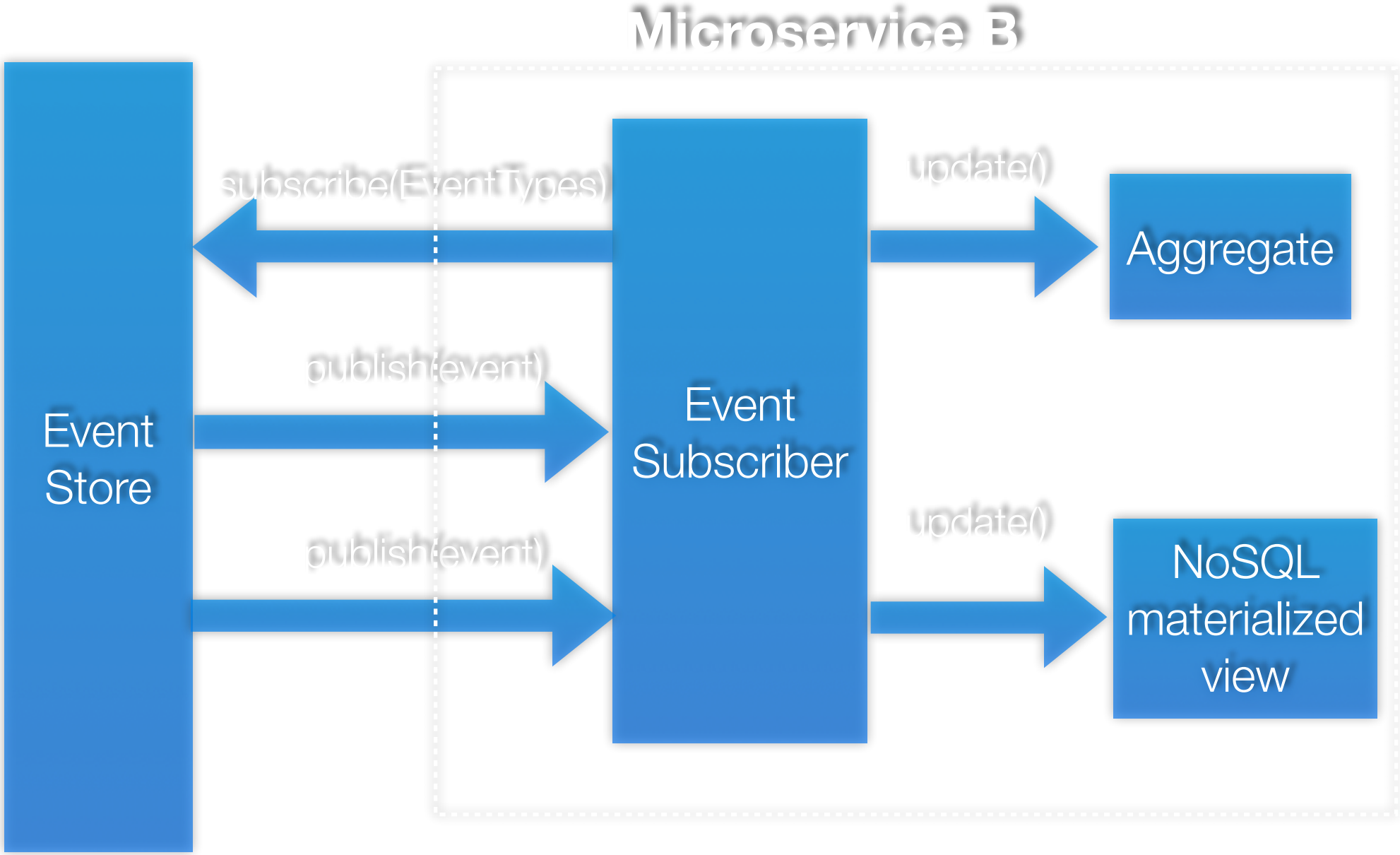



Two actions that must be atomic
Before: update state + publish events
Now: persist (and publish) events
Single action that can be done atomically

Request handling in an event-sourced application



Event Store publishes events - consumed by other services



Event store implementations

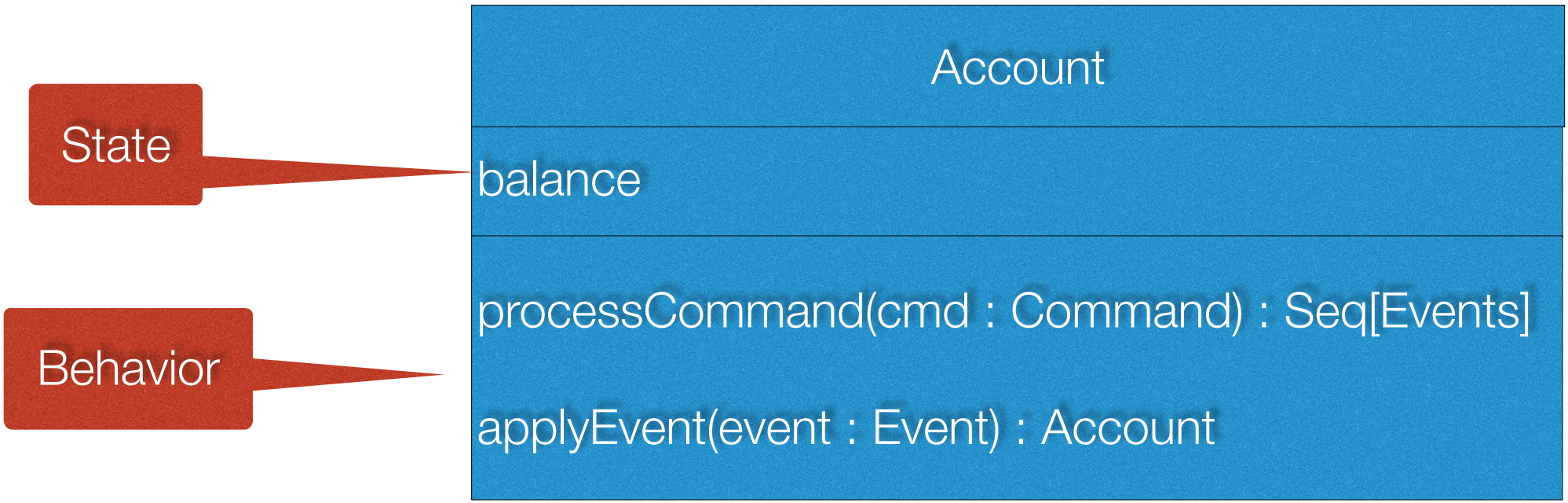
- Home-grown/DIY
- geteventstore.com by Greg Young
- My event store - http://bit.ly/trialeventuate

Optimizing using snapshots

- Most aggregates have relatively few events
- BUT consider a 10-year old Account ⇒ many transactions
- Therefore, use snapshots:
  - Periodically save snapshot of aggregate state
  - Typically serialize a memento of the aggregate
  - Load latest snapshot + subsequent events

Hybrid OO/Functional style example aggregate

OO = State + Behavior



Aggregate traits

Used by Event Store to reconstitute aggregate

Apply event returning updated Aggregate

```
trait Aggregate[T] { self : T =>
  def applyEvent(event : Event) : T
}

trait CommandProcessingAggregate[T, -CT] extends Aggregate[T] { self : T =>
  def processCommand(command : CT) : Seq[Event]
}
```

Map Command to Events

Account - command processing

```
case class Account(balance : BigDecimal)
  extends PatternMatchingCommandProcessingAggregate[Account, AccountCommand] {

  def this() = this(null)


  def processCommand = {
    case OpenAccountCommand(initialBalance) =>
      Seq(AccountOpenedEvent(initialBalance))

    case CreditAccountCommand(amount, transactionId) =>
      Seq(AccountCreditedEvent(amount, transactionId))

    case DebitAccountCommand(amount, transactionId) if amount <= balance =>
      Seq(AccountDebitedEvent(amount, transactionId))

    case DebitAccountCommand(amount, transactionId) =>
      Seq(AccountDebitFailedDueToInsufficientFundsEvent(amount, transactionId))
  }
```

Prevent overdraft

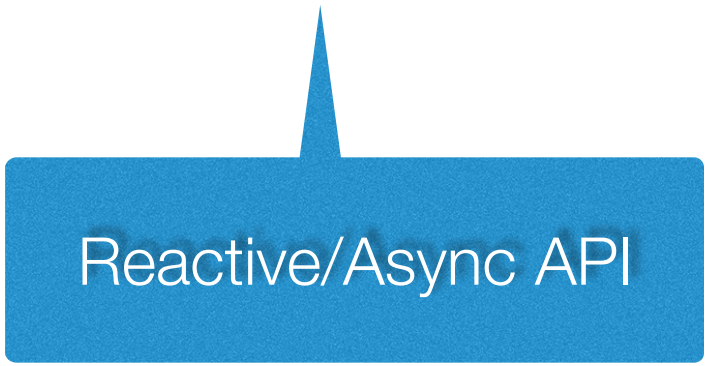
Account - applying events

Immutable

```
case class Account(balance : BigDecimal)
  extends PatternMatchingCommandProcessingAggregate[Account, AccountCommand] {

  def applyEvent = {

    case AccountOpenedEvent(initialBalance) => copy(balance = initialBalance)

    case AccountDebitedEvent(amount, _) => copy(balance = balance - amount)

    case AccountCreditedEvent(amount, _) =>
      copy(balance = balance + amount)

    case AccountDebitFailedDueToInsufficientFundsEvent(amount, _) =>
      this

  }
```

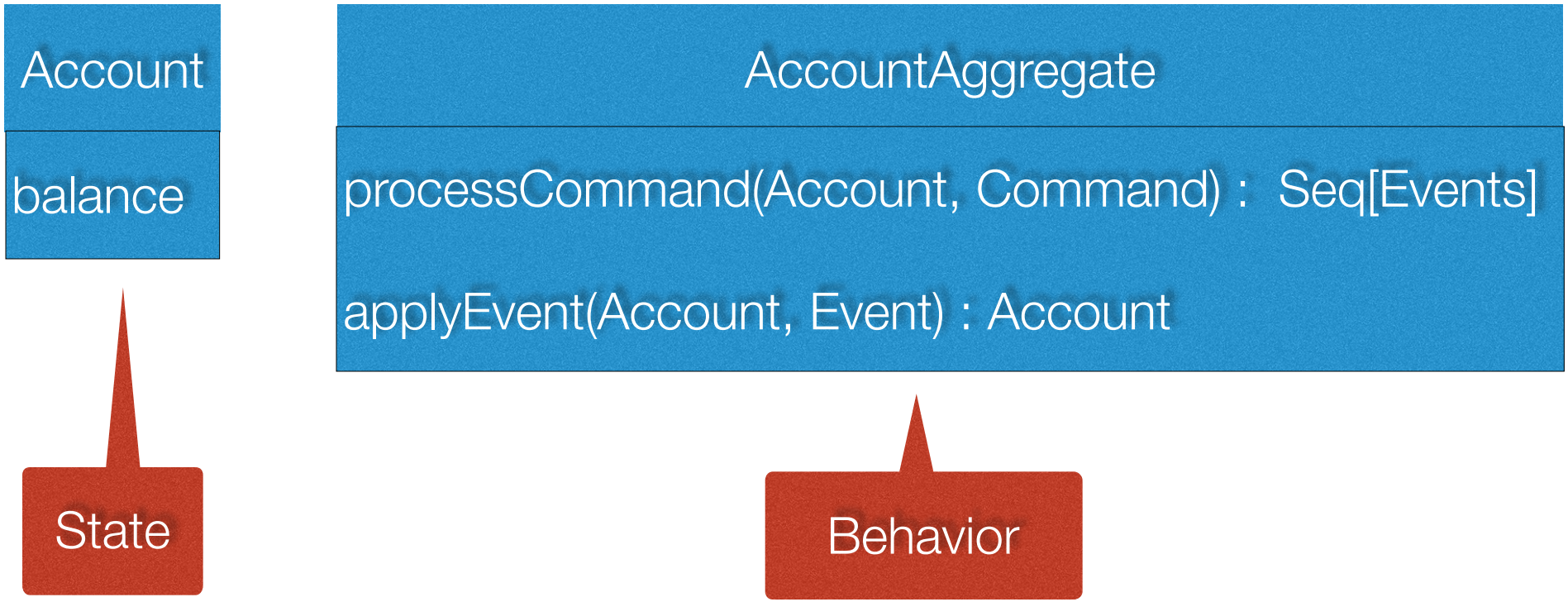
Event Store API

```
trait EventStore {

  def save[T <: Aggregate[T] : ClassTag](events: Seq[Event], assignedId: Option[EntityId] = None)
    : Future[EntityIdAndVersion]

  def update[T <: Aggregate[T] : ClassTag](entityIdAndVersion: EntityIdAndVersion, events: Seq[Event])
    : Future[EntityIdAndVersion]

  def find[T <: Aggregate[T] : ClassTag](entityId: EntityId)
    : Future[EntityWithMetadata[T]]

  def findOptional[T <: Aggregate[T] : ClassTag](entityId: EntityId)
    : Future[Option[EntityWithMetadata[T]]]
```



Functional example aggregate

FP = Separation of State and Behavior



Aggregate type classes/implicits

```scala
trait Aggregate[T, -EV <: Event] {
  def newInstance() : T

  def applyEvent(aggregate : T, event : EV) : T

  def applyEvents(aggregate: T, events: Seq[EV]) : T =
    events.foldLeft(aggregate)(applyEvent)
}
```

```scala
trait Command

trait AggregateCommandProcessor[T , -CT <: Command, EV <: Event] {

  def processCommand(aggregate : T, command : CT) : Seq[EV]
}
```

Functional-style MoneyTransfer Aggregate

State

Behavior

```
case class TransferDetails(fromAccountId : EntityId, toAccountId : EntityId, amount : Money)

case class MoneyTransfer(state : TransferStates.State, details : TransferDetails)

implicit object MoneyTransferAggregate extends Aggregate[MoneyTransfer, MoneyTransferEvent]
        with AggregateCommandProcessor[MoneyTransfer, MoneyTransferCommand, MoneyTransferEvent] {

  override def newInstance() = MoneyTransfer(TransferStates.NEW, null)

  override def processCommand(mt: MoneyTransfer, command: MoneyTransferCommand): Seq[MoneyTransferEvent] =
    command match {...}

  override def applyEvent(mt: MoneyTransfer, event: MoneyTransferEvent): MoneyTransfer =
    event match {...}
```

FP-style event store

Enables inference of T, and EV

```
trait EventStore {

  def save[T, EV <: Event](clasz : Class[T], events: Seq[EV], assignedId: Option[EntityId] = None)
                          (implicit ag : Aggregate[T, EV]) :
      Future[EntityIdAndVersion]

  def update[T, EV <: Event](clasz : Class[T], entityIdAndVersion: EntityIdAndVersion, events: Seq[EV])
                           (implicit ag : Aggregate[T, EV]) :
      Future[EntityIdAndVersion]

  def find[T, EV <: Event](clasz : Class[T], entityId: EntityId)
                         (implicit ag : Aggregate[T, EV]): Future[EntityWithMetadata[T, EV]]

  def findOptional[T, EV <: Event](clasz : Class[T], entityId: EntityId)
                           (implicit ag : Aggregate[T, EV]): Future[Option[EntityWithMetadata[T, EV]]]
}
```

Tells ES how to instantiate aggregate and apply events

Business benefits of event sourcing

- Built-in, reliable audit log
- Enables temporal queries
- Publishes events needed by big data/predictive analytics etc.
- Preserved history ⇒ More easily implement future requirements

Technical benefits of event sourcing

- Solves data consistency issues in a Microservice/NoSQL-based architecture:
  - Atomically save and publish events
  - Event subscribers update other aggregates ensuring eventual consistency
  - Event subscribers update materialized views in SQL and NoSQL databases (more on that later)
- Eliminates O/R mapping problem

Drawbacks of event sourcing

- Weird and unfamiliar
- Events = a historical record of your bad design decisions
- Handling duplicate events can be tricky
- Application must handle eventually consistent data
- Event store only directly supports PK-based lookup (more on that later)

Agenda

- Why build event-driven microservices?
- Overview of event sourcing
- Designing microservices with event sourcing
- Implementing queries in an event sourced application

Use the familiar building blocks of DDD

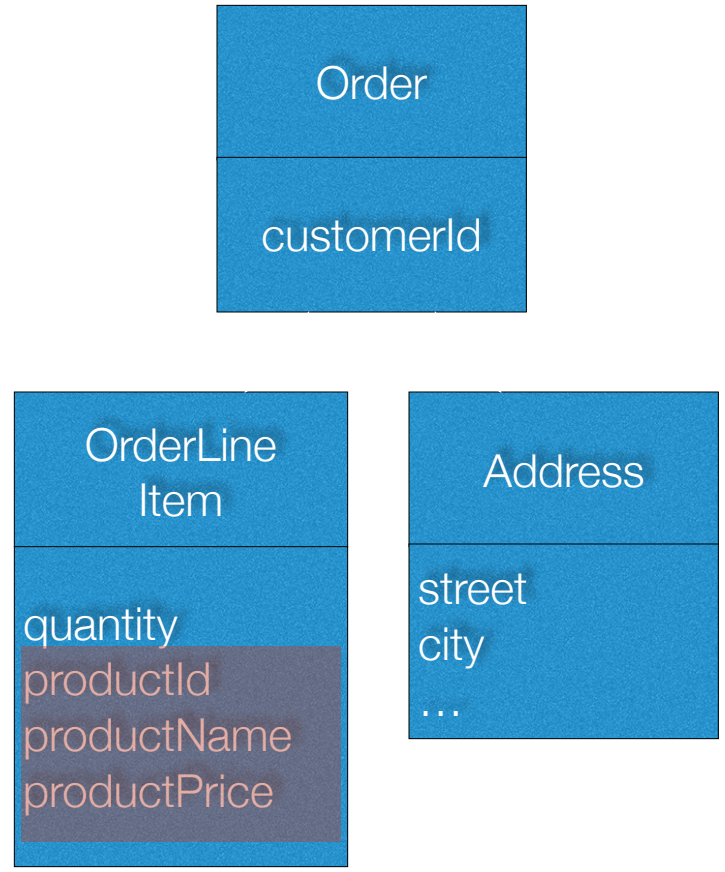
- Entity
- Value object
- Services
- Repositories
- Aggregates

With some differences

Partition a bounded context's domain model into Aggregates

Aggregate design

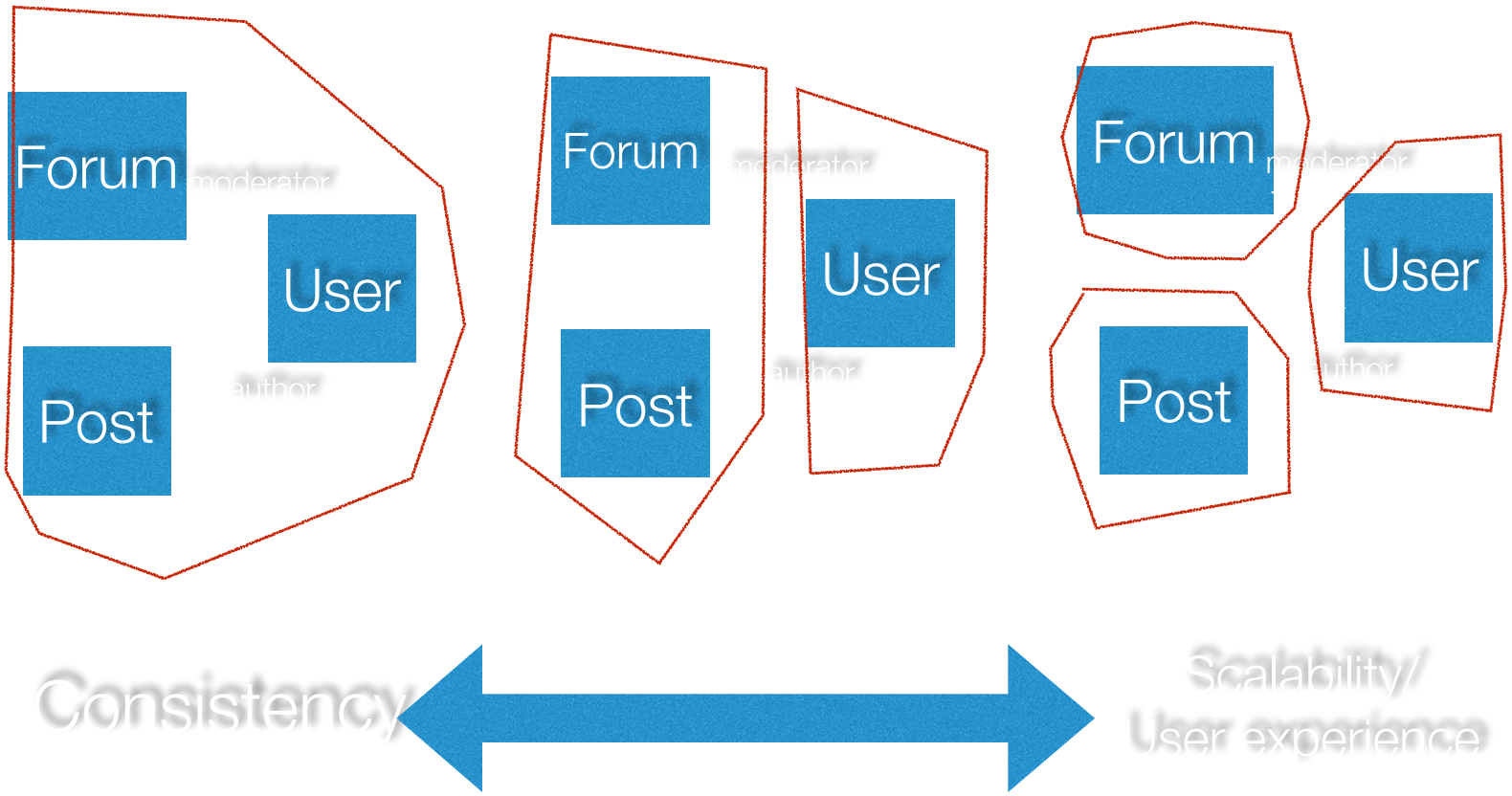
- Graph consisting of a root entity and one or more other entities and value objects
- Each core business entity = Aggregate: e.g. customer, Account, Order, Product, ….
- Reference other aggregate roots via primary key
- Often contains partial copy of other aggregates' data



Aggregate granularity is important

- Transaction = processing one command by one aggregate
- No opportunity to update multiple aggregates within a transaction
- If an update must be atomic (i.e. no compensating transaction) then it must be handled by a single aggregate
  - e.g. scanning boarding pass at security checkpoint or when entering jetway

Aggregate granularity

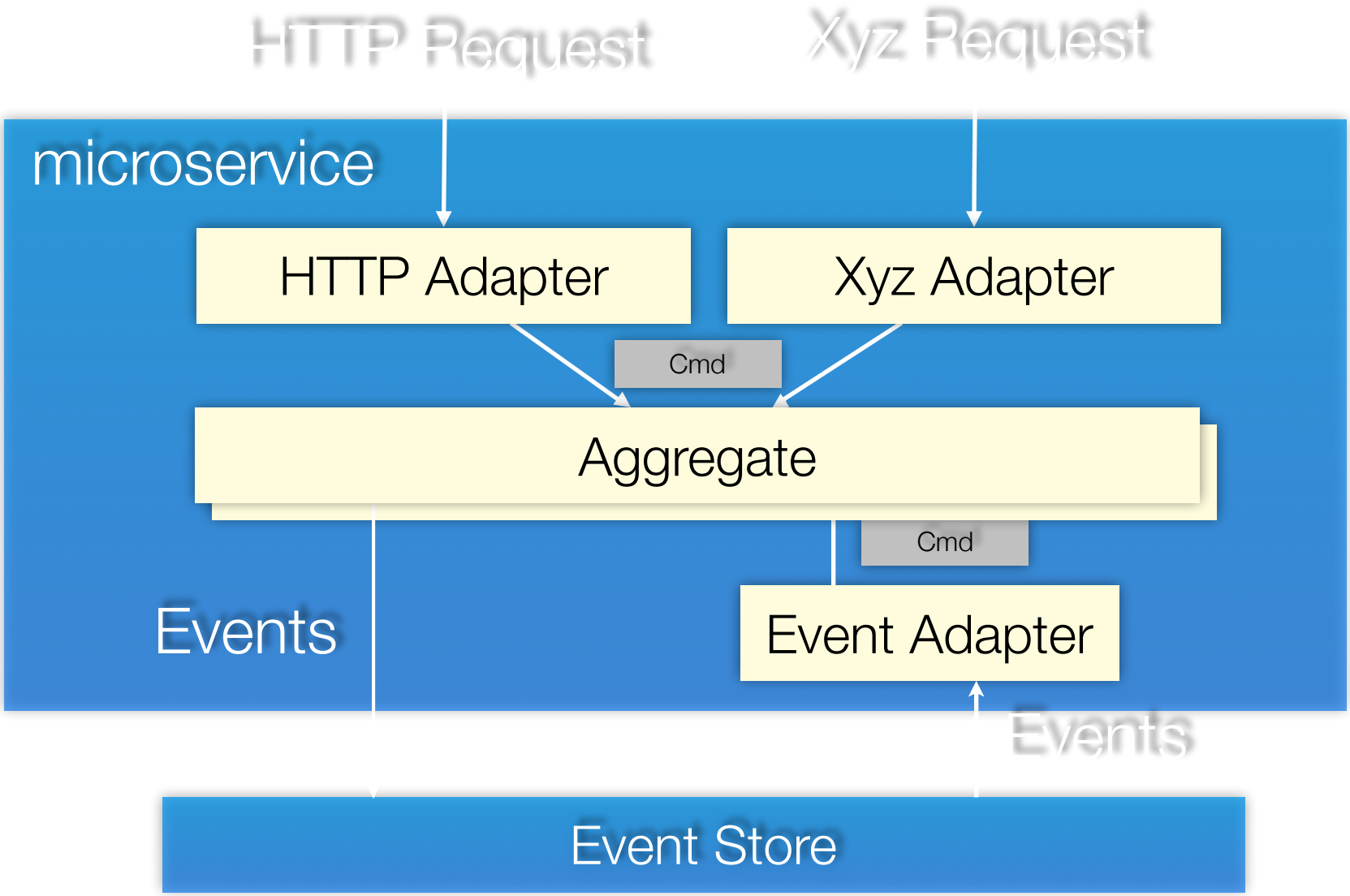


Identify the state changing events for each Aggregate

Designing domain events

- Naming
  - Past tense to reflect that something occurred
  - Ideally specific: AccountOpened/Debited/Credited
  - Sometimes vague: FooUpdated
- Event attributes
  - Id - TimeUUID
  - Other attributes - from command, required to persist entity
- Event enrichment
  - ProductAddedToCart(**productId**) vs. ProductAddedCart(**productInfo**)
  - Extra data to support event consumers

The anatomy of a microservice



Asynchronous Spring MVC controller

```
@RestController
class MoneyTransferController @Autowired()(moneyTransferService : MoneyTransferService,
                                           eventStore : EventStore) {

  @RequestMapping(value=Array("/transfers"), method = Array(RequestMethod.POST))
  def create(@RequestBody transferDetails : TransferDetails) = WebUtil.toDeferredResult {
    for (transaction <- moneyTransferService.transferMoney(transferDetails))
      yield CreateMoneyTransferResponse(transaction.entityId.id)
  }
```

Scala Future => Spring MVC DeferredResult

MoneyTransferService

```
class MoneyTransferService(implicit eventStore : EventStore) {

  def transferMoney(transferDetails : TransferDetails) =
    newEntity[MoneyTransfer] <== CreateMoneyTransferCommand(transferDetails)

}
```

DSL concisely specifies:

1. Creates MoneyTransfer aggregate
2. Processes command
3. Applies events
4. Persists events

Handling events published by Accounts

```
@EventSubscriber (id = "transactionEventHandlers")
class MoneyTransferEventHandlers(implicit eventStore: EventStore)
  extends CompoundEventHandler {

  val recordDebit =
    handlerForEvent[AccountDebitedEvent] { de =>
      existingEntity[MoneyTransfer](de.event.transactionId) <==
        RecordDebitCommand(de.entityId)
    }
```

1. Load MoneyTransfer aggregate
2. Processes command
3. Applies events
4. Persists events

Agenda

- Why build event-driven microservices?
- Overview of event sourcing
- Designing microservices with event sourcing
- Implementing queries in an event sourced application

Let’s imagine that you want to display an account and it’s recent transactions…

Displaying balance + recent credits and debits

- We need to do a “join: between the Account and the corresponding MoneyTransfers
- (Assuming Debit/Credit events don’t include other account, ...)

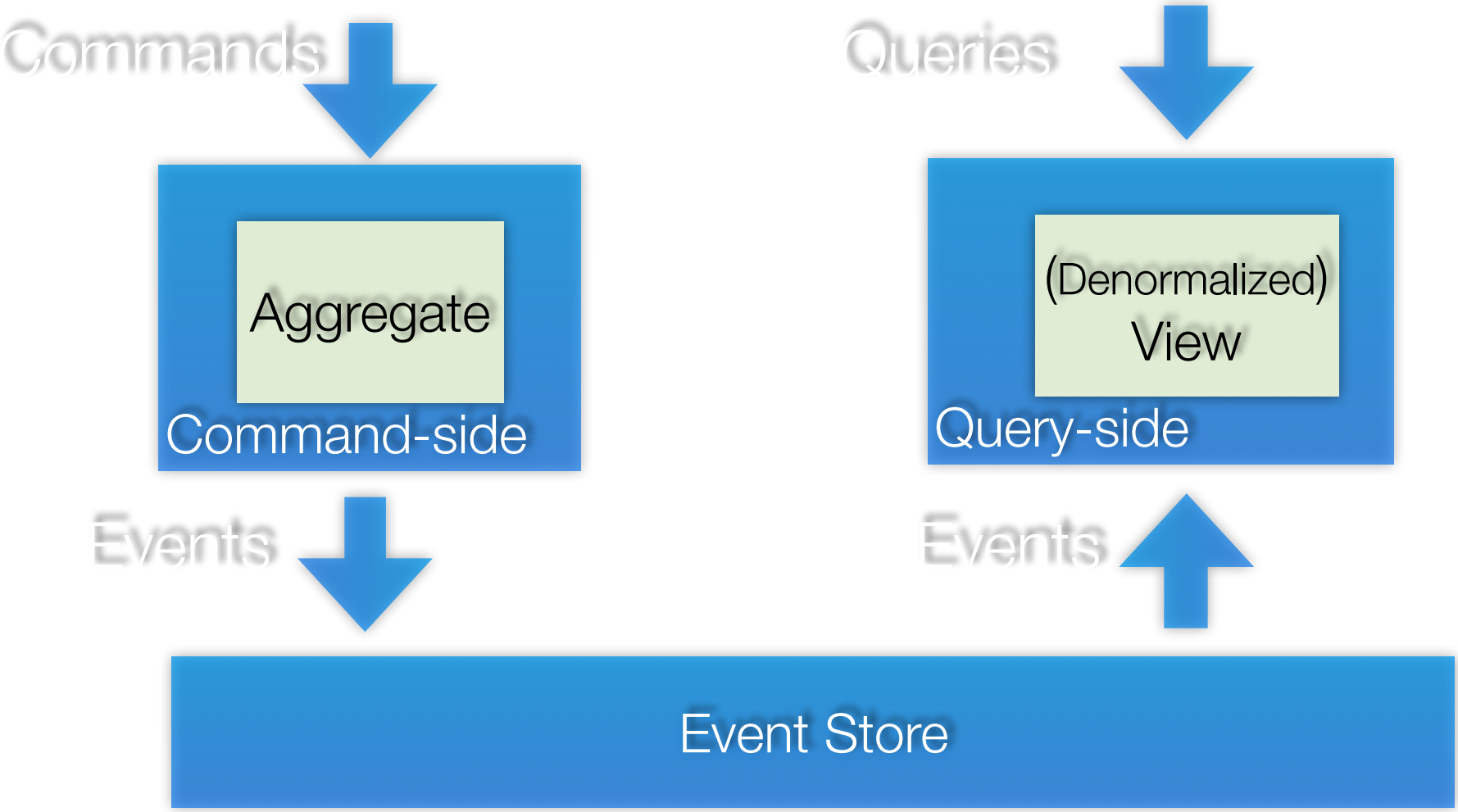
BUT

- Event Store = primary key lookup of individual aggregates, ...

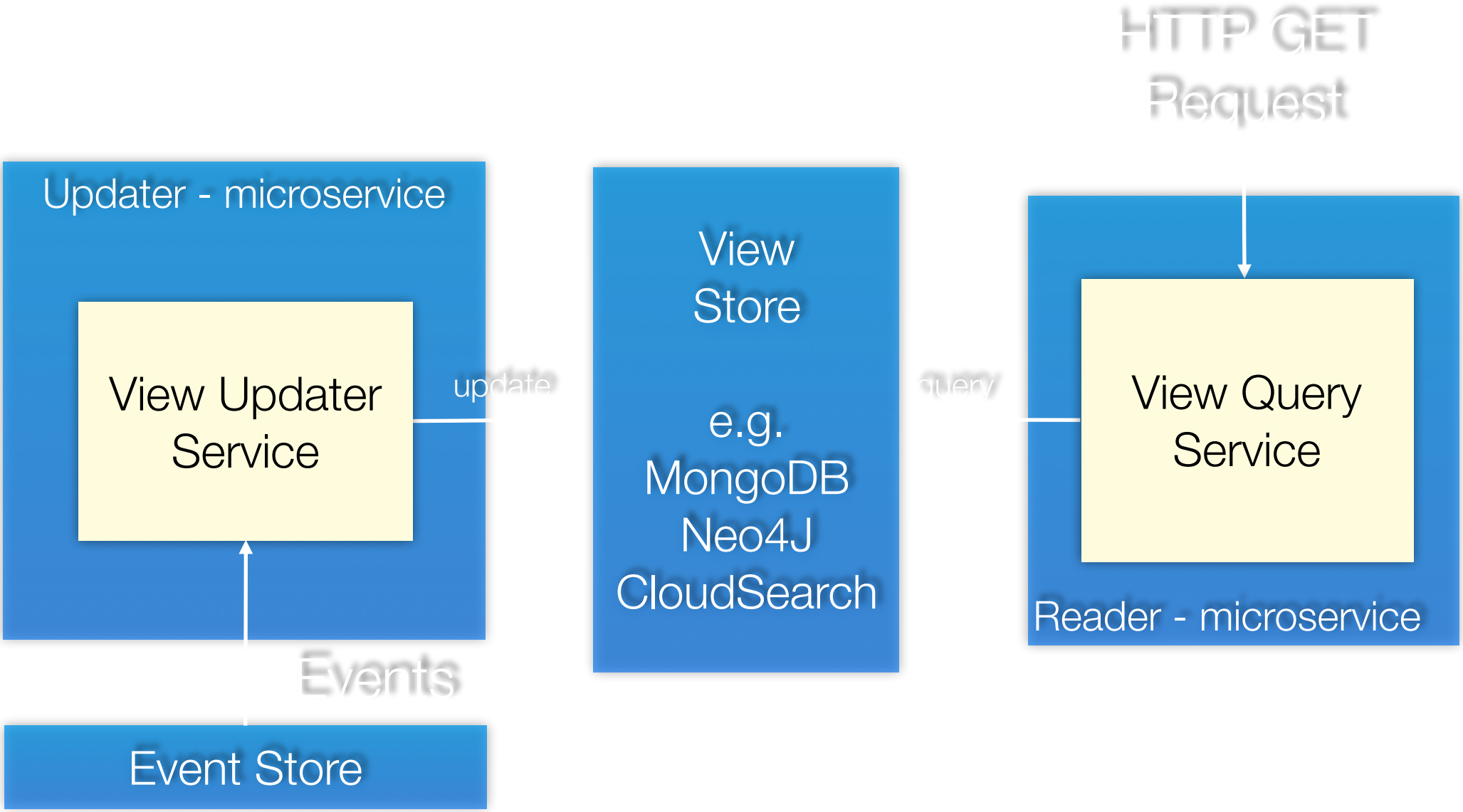
⇒

- Use **C**ommand **Q**uery **R**esponsibility **S**egregation

Command Query Responsibility Segregation (CQRS)



Query-side microservices



Persisting account balance and recent transactions in MongoDB

```
{
   id: "298993498",
   balance: 100000,
   transfers : [
      {"transferId" : "4552840948484",
       "fromAccountId" : 298993498,
       "toAccountId" : 3483948934,
       "amount" : 5000}, ...
   ],
   changes: [
     {"changeId" : "93843948934",
      "transferId" : "4552840948484",
      "transactionType" : "AccountDebited",
      "amount" : 5000}, ...
   ]
}
```

Current balance

MoneyTransfers that update the account

The debits and credits

Denormalized = efficient lookup

Persisting account info using MongoDB...

```
class AccountInfoUpdateService
    (accountInfoRepository : AccountInfoRepository, mongoTemplate : MongoTemplate)
          extends CompoundEventHandler {


  @EventHandlerMethod
  def created(de: DispatchedEvent[AccountOpenedEvent]) = …

  @EventHandlerMethod
  def recordDebit(de: DispatchedEvent[AccountDebitedEvent]) = …

  @EventHandlerMethod
  def recordCredit(de: DispatchedEvent[AccountCreditedEvent]) = …

  @EventHandlerMethod
  def recordTransfer(de: DispatchedEvent[MoneyTransferCreatedEvent]) = …


}
```

Persisting account info using NodeJS and MongoDB...

```
  this.handlers[accountEvents.entityTypeName][accountEvents.AccountOpenedEvent] =
     function (event, callback){
        accountViewUpdaterService.createAccount(event, callback)
  };

  this.handlers[accountEvents.entityTypeName][accountEvents.AccountDebitedEvent] =
    function (event, callback) {
    accountViewUpdaterService.saveAccountChange(event, -1, callback);
   };
```

```
exports.saveAccountChange = function(event, delta, callback){
…
 var update = {
    $inc: { balance: amount * delta },
    $push: { changes: ci },
    $set: { version: changeId }
  };

  var options = { multi: true };

  db.AccountModel.update(conditions, update, options, callback);
};
```

Other kinds of views

- AWS Cloud Search
  - Text search as-a-Service
  - View updater batches aggregates to index
  - View query service does text search
- AWS DynamoDB
  - NoSQL as-a-Service
  - On-demand scalable - specify desired read/write capacity
  - Document and key-value data models
  - Useful for denormalized, UI oriented views

Benefits and drawbacks of CQRS

Benefits

- Necessary in an event-sourced architecture
- Separation of concerns = simpler command and query models
- Supports multiple denormalized views
- Improved scalability and performance

Drawbacks

- Complexity
- Potential code duplication
- Replication lag/eventually consistent views

Summary

- Event sourcing solves key data consistency issues with:
  - Microservices
  - Partitioned SQL/NoSQL databases
- Apply strategic DDD to identify microservices
- Apply tactical DDD to design individual services
- Use CQRS to implement materialized views for queries

@crichardson chris@chrisrichardson.net

http://plainoldobjects.com http://microservices.io