Developing functional domain models with event sourcing

Chris Richardson

Author of POJOs in Action

Founder of the original CloudFoundry.com

🐦 @crichardson

chris@chrisrichardson.net

http://plainoldobjects.com

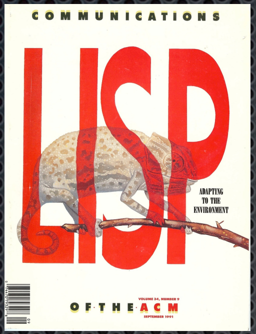
http://microservices.io

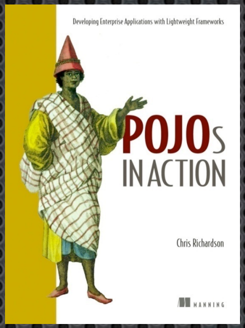
QCon
NEW YORK

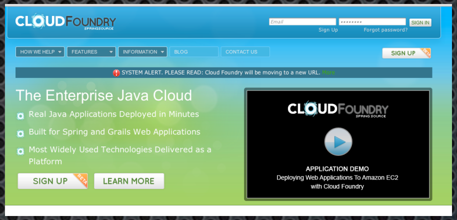
Presentation goal

How to develop functional domain models based on event sourcing

About Chris




Developing Enterprise Applications with Lightweight Frameworks
POJOs IN ACTION
Chris Richardson
MANNING

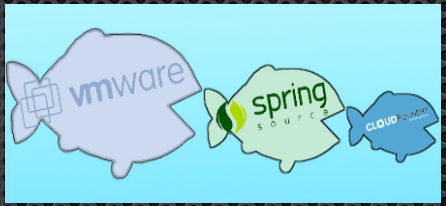


SYSTEM ALERT. PLEASE READ: Cloud Foundry will be moving to a new URL. More

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

APPLICATION DEMO
Deploying Web Applications To Amazon EC2
with Cloud Foundry



Consultant &
Founder of Eventuate.IO

@crichardson

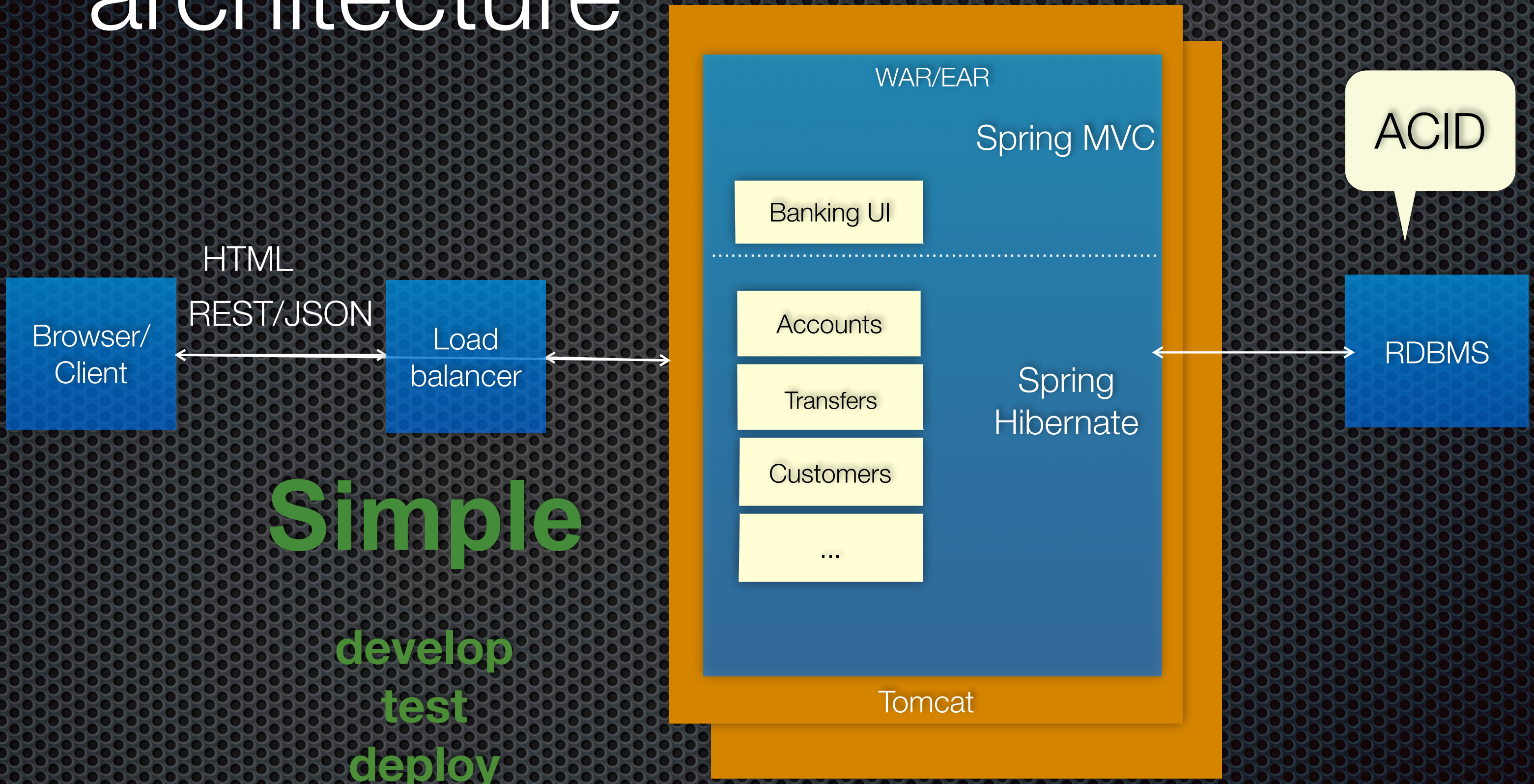
For more information

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

Agenda

- Why event sourcing?
- Designing a domain model based on event sourcing
- Event sourcing and service design
- Microservices and event sourcing

Traditional monolithic architecture



Simple

develop
test
deploy
scale

@crichardson

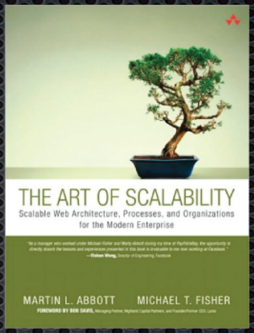
But large and/or complex monolithic applications

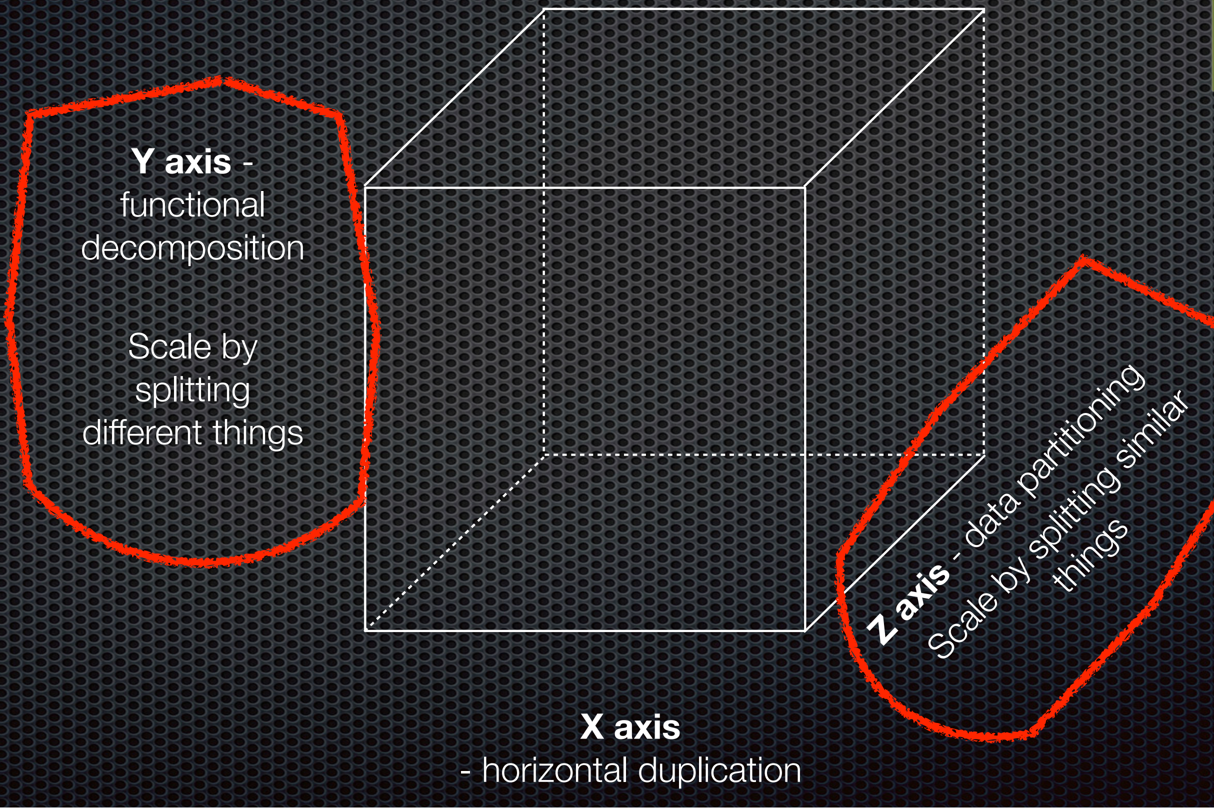
=

Trouble!

Using a single RDBMS has its limitations

Apply the scale cube


THE ART OF SCALABILITY
Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise
MARTIN L. ABBOTT
MICHAEL T. FISHER



Y axis - functional decomposition

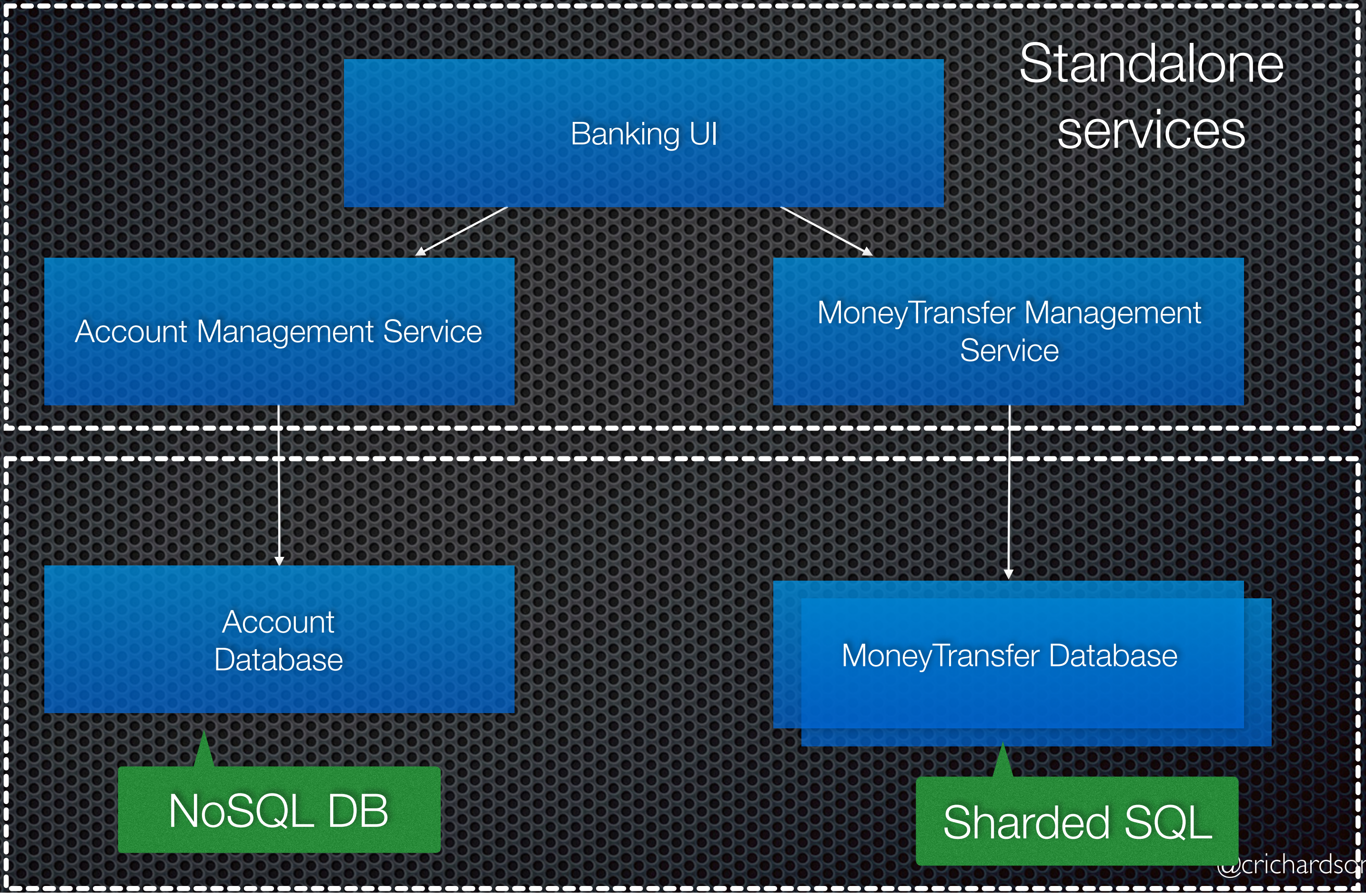
Scale by splitting different things

Z axis - data partitioning

Scale by splitting similar things

X axis - horizontal duplication

Today: use a microservice, polyglot architecture



Standalone services

Banking UI

Account Management Service

MoneyTransfer Management Service

Account Database

MoneyTransfer Database

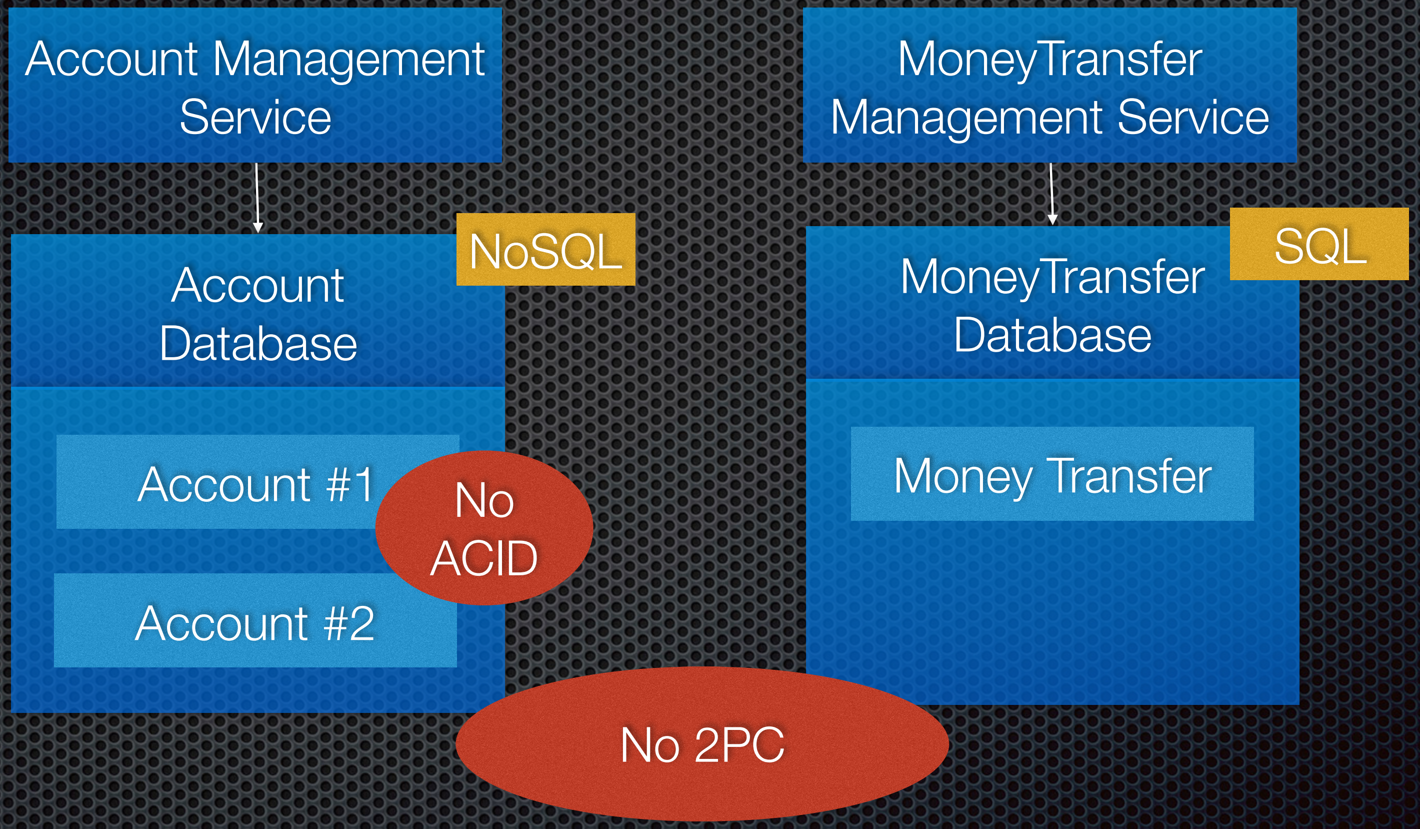
NoSQL DB

Sharded SQL

But now we have distributed data management problems

@crichardson

Example: Money transfer

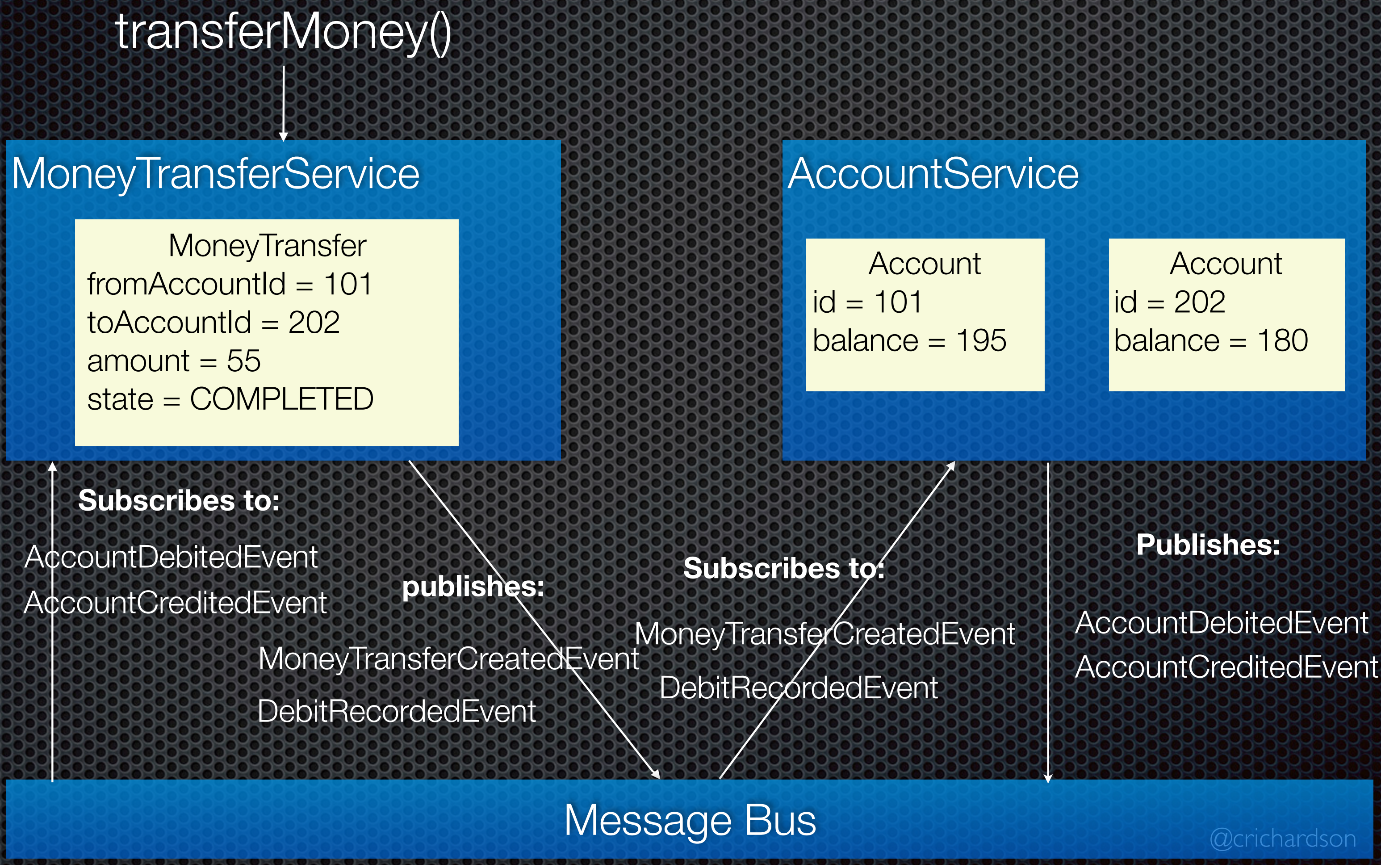


@crichardson

Use an event-driven architecture

- 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



transferMoney()

MoneyTransferService

MoneyTransfer
fromAccountId = 101
toAccountId = 202
amount = 55
state = COMPLETED

AccountService

Account
id = 101
balance = 195

Account
id = 202
balance = 180

Subscribes to:

AccountDebitedEvent
AccountCreditedEvent

publishes:

MoneyTransferCreatedEvent
DebitRecordedEvent

Subscribes to:

MoneyTransferCreatedEvent
DebitRecordedEvent

Publishes:

AccountDebitedEvent
AccountCreditedEvent

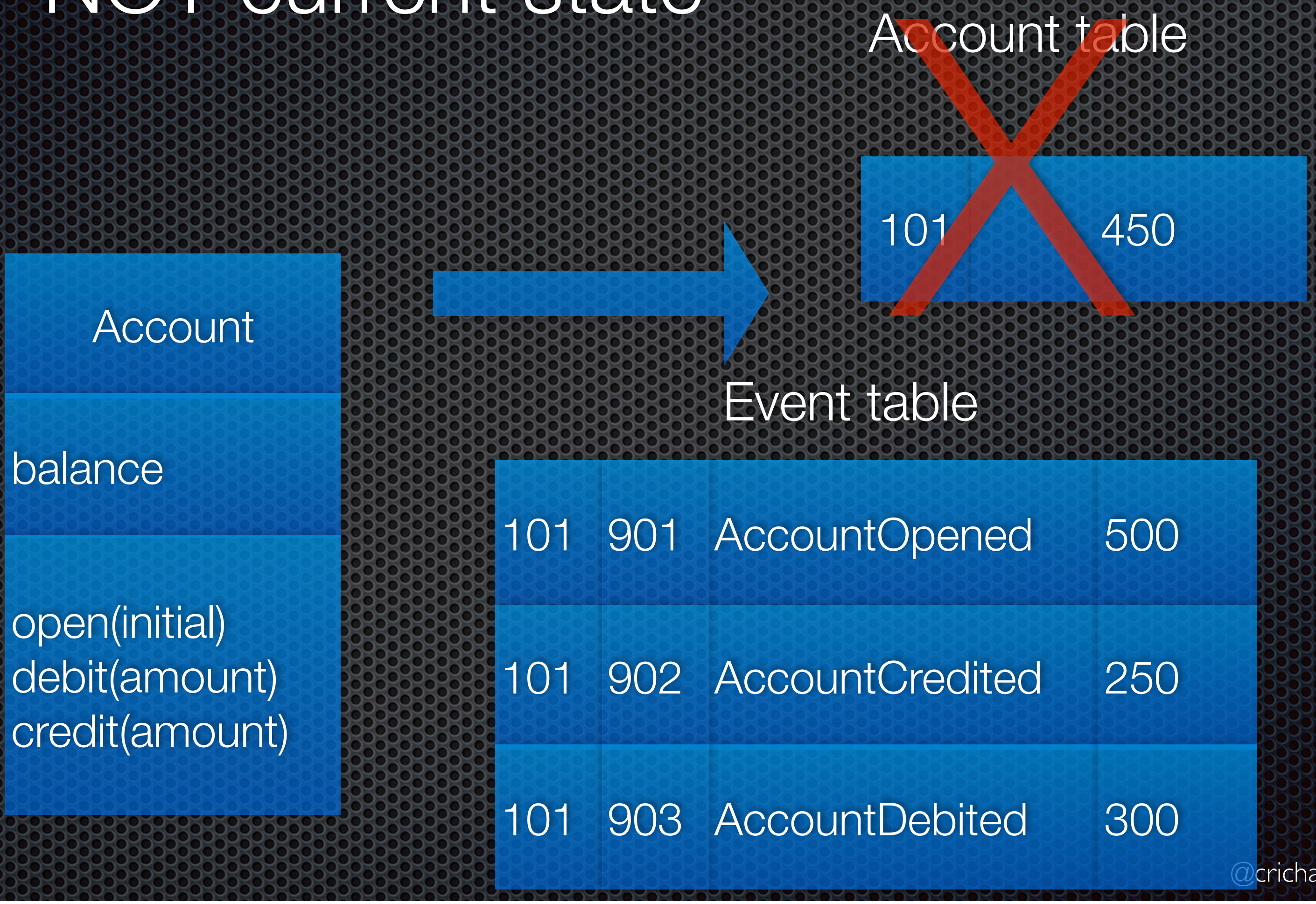
Message Bus

How to
atomically
update the database
and
publish an event
without 2PC?
(dual write problem)

Event sourcing

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

Persists events
NOT current state



Account

balance

open(initial)
debit(amount)
credit(amount)

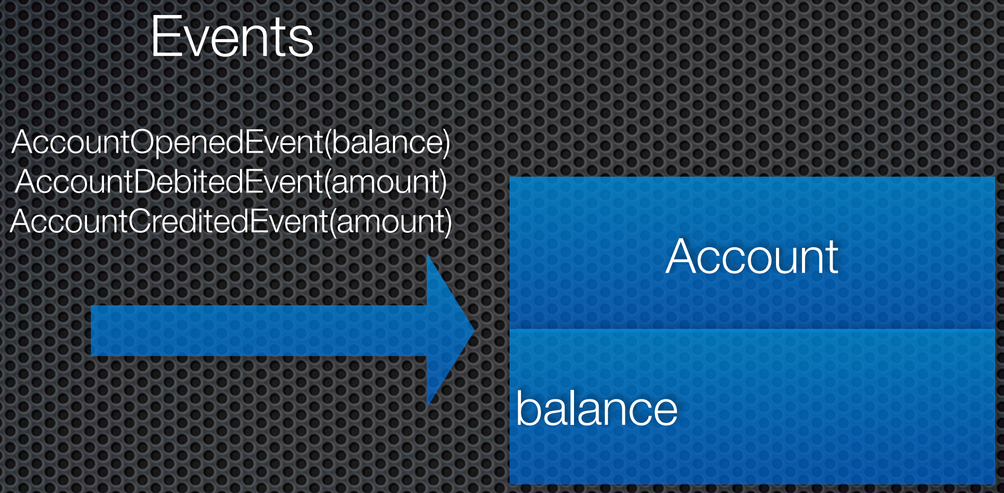
Account table

101	450

Event table

101	901	AccountOpened	500
101	902	AccountCredited	250
101	903	AccountDebited	300

Replay events to recreate state



Events

AccountOpenedEvent(balance)
AccountDebitedEvent(amount)
AccountCreditedEvent(amount)

Account

balance

Two actions that must be atomic

Before: update state + publish events

↓

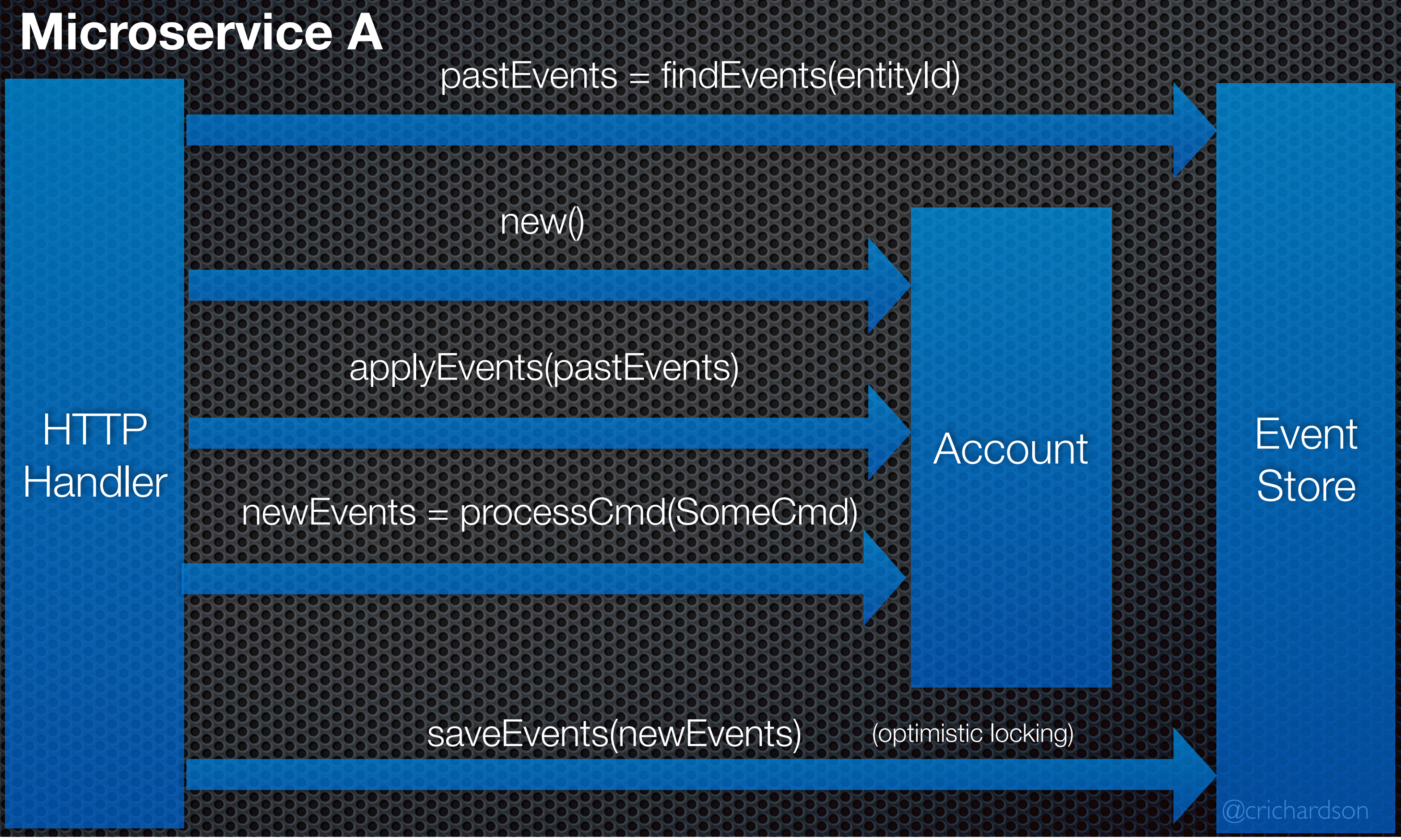
Now: persist (and publish) events

Single action that can be done atomically

Optimizing using snapshots

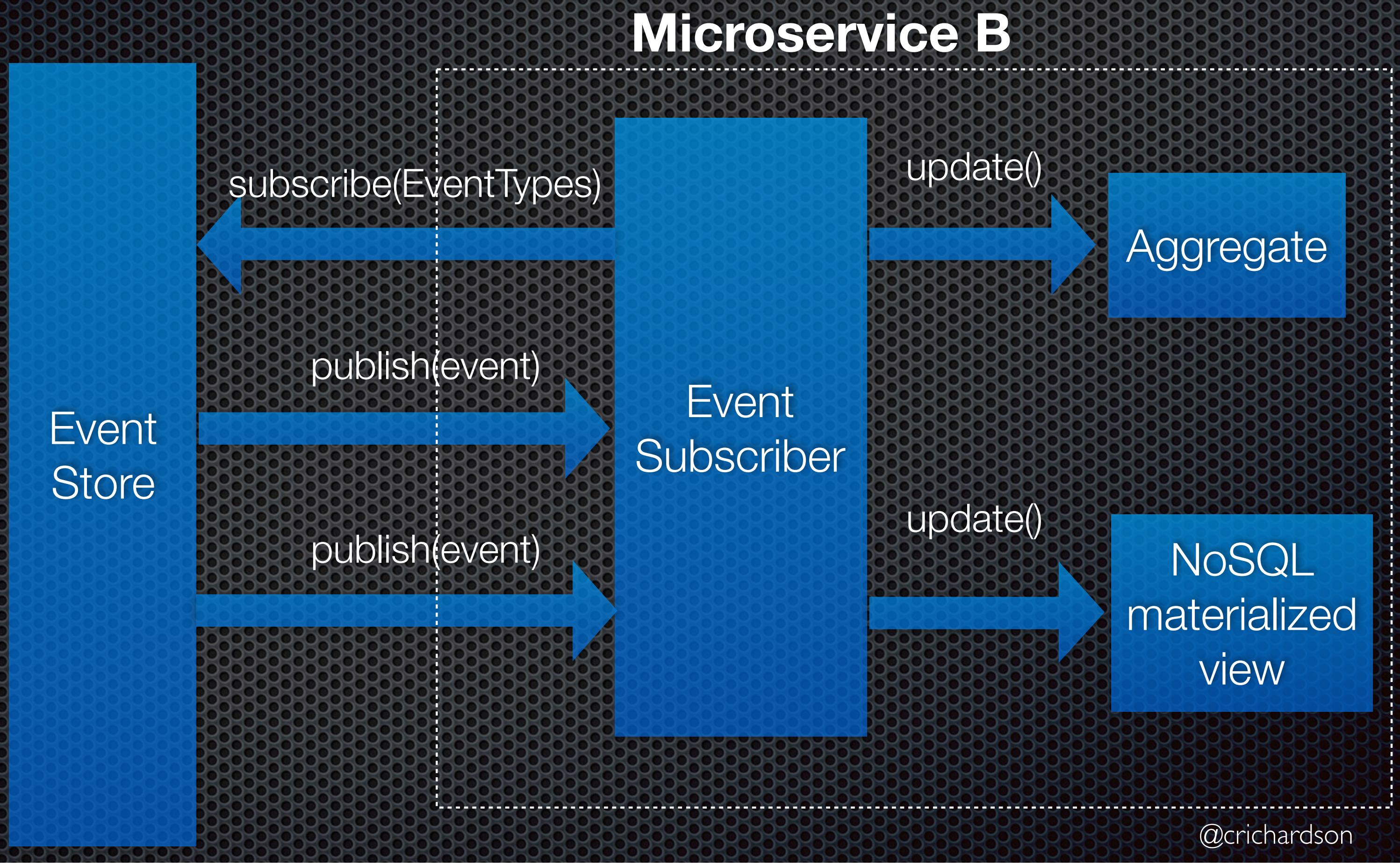
- Most aggregates have relatively few events
- BUT consider a 10-year old Account ⇒ many transactions
- Therefore, use snapshots:
  - Periodically save snapshot of aggregate state
  - Typically serialize a memento of the aggregate
  - Load latest snapshot + subsequent events

Request handling in an event-sourced application



@crichardson

Event Store publishes events - consumed by other services



Microservice B

Event Store

subscribe(EventTypes)

publish(event)

publish(event)

Event Subscriber

update()

Aggregate

update()

NoSQL materialized view

About the event store

- Hybrid database and message broker
  - Retrieve events for an aggregate
  - Append to an aggregates events
  - Subscribe to events
- Event store implementations:
  - Home-grown/DIY
  - geteventstore.com by Greg Young
  - My event store - bit.ly/trialeventuate

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 event sourcing?
- Designing a domain model based on event sourcing
- Event sourcing and service design
- Microservices and event sourcing

Use the familiar building blocks of DDD

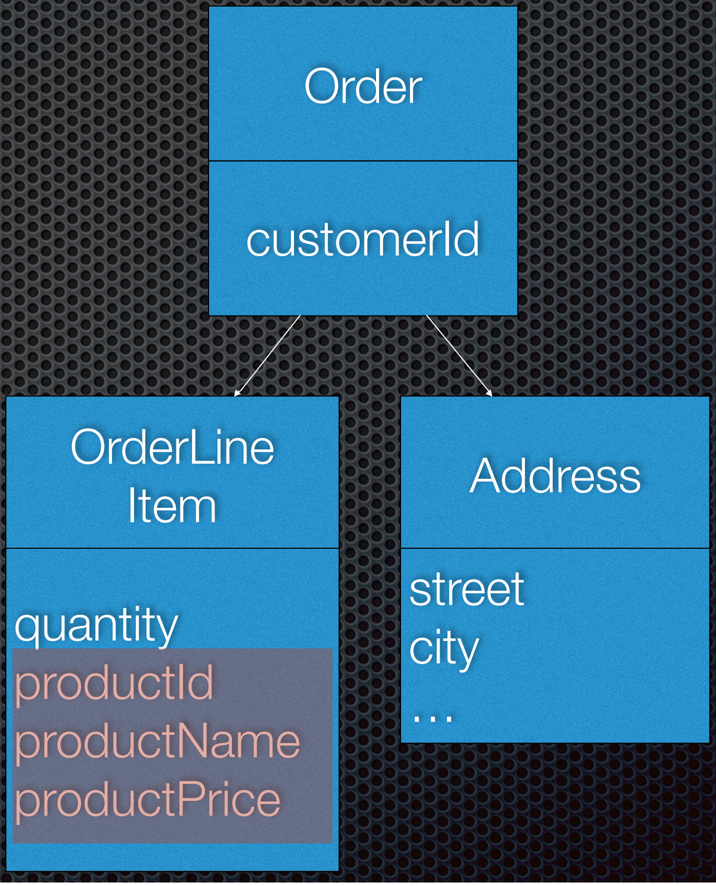
- Entity
- Value object
- Services
- Repositories
- Aggregates

With some differences

Partition the domain model into Aggregates

Aggregate design

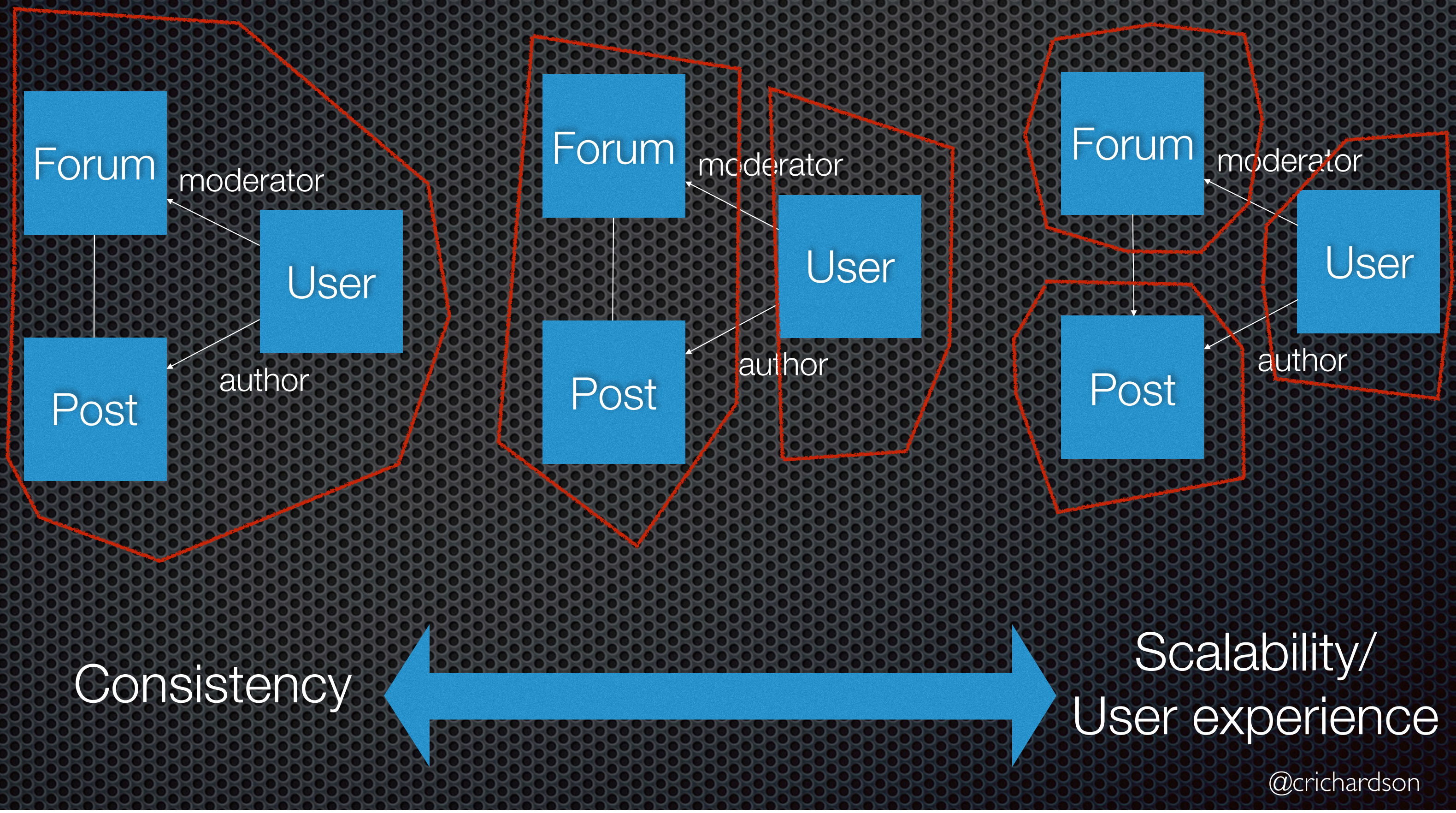
- Graph consisting of a root entity and one or more other entities and value objects
- Each core business entity = Aggregate: e.g. customer, Account, Order, Product, ….
- Reference other aggregate roots via primary key
- Often contains partial copy of other aggregates' data



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

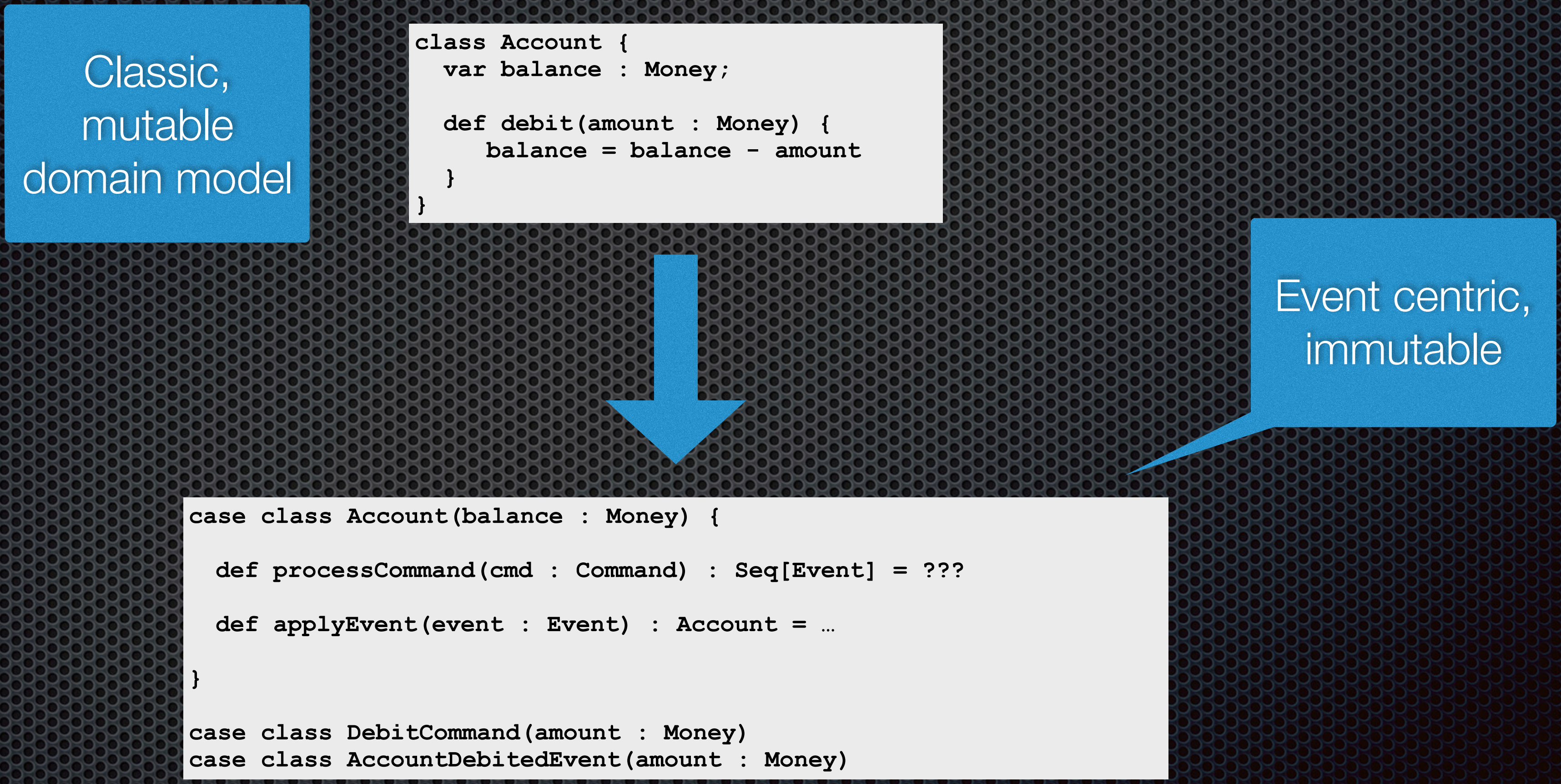


Consistency

Scalability/
User experience

@crichardson

ES-based Aggregate design



Classic, mutable domain model

```
class Account {
  var balance : Money;

  def debit(amount : Money) {
     balance = balance - amount
  }
}
```

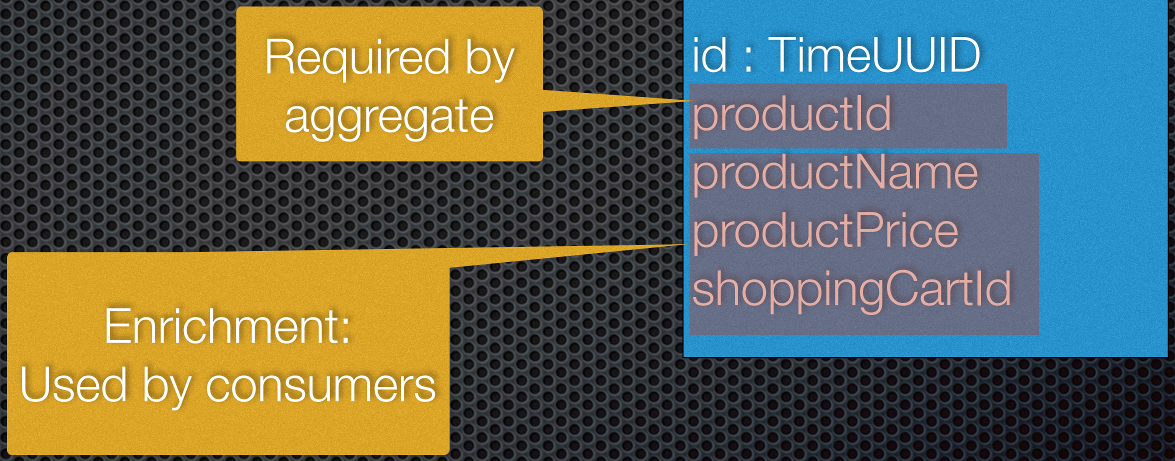
Event centric, immutable

```
case class Account(balance : Money) {

  def processCommand(cmd : Command) : Seq[Event] = ???

  def applyEvent(event : Event) : Account = …

}

case class DebitCommand(amount : Money)
case class AccountDebitedEvent(amount : Money)
```

@crichardson

Designing domain events

- Record state changes for an aggregate
- Part of the public API of the domain model



ProductAddedToCart

id : TimeUUID
productId
productName
productPrice
shoppingCartId

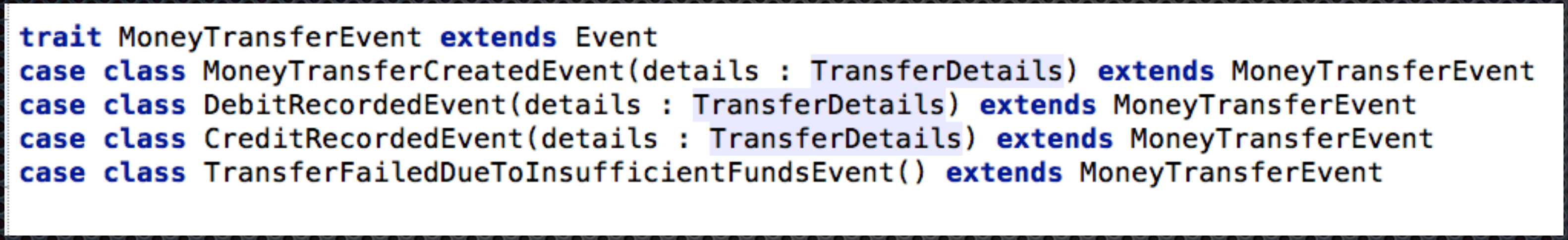
Required by aggregate

Enrichment:
Used by consumers

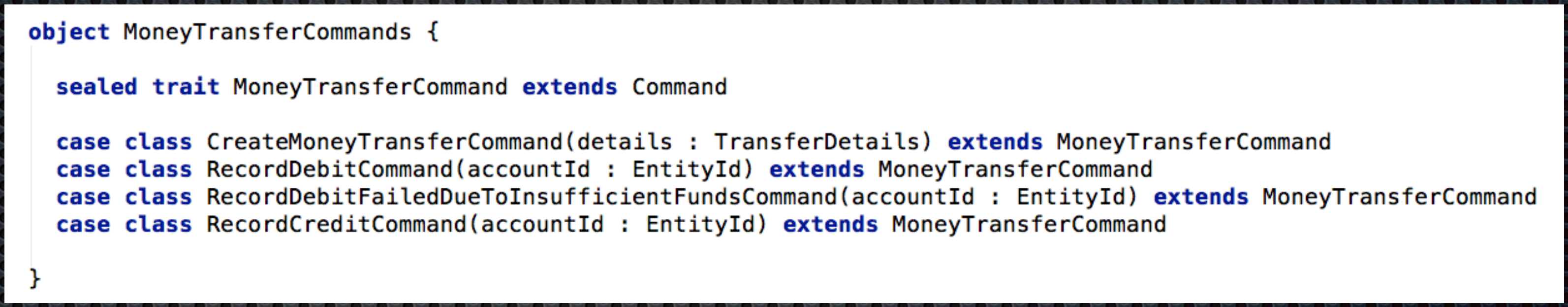
Designing commands

- Created by a service from incoming request
- Processed by an aggregate
- Immutable
- Contains value objects for
  - Validating request
  - Creating event
  - Auditing user activity

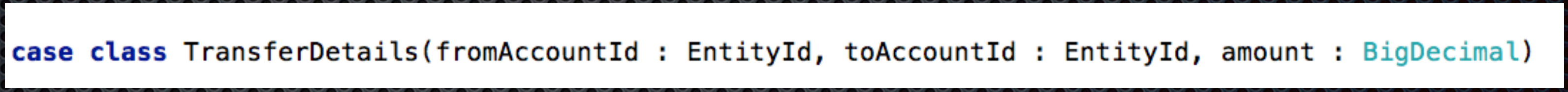
Events and Commands



```
trait MoneyTransferEvent extends Event
case class MoneyTransferCreatedEvent(details : TransferDetails) extends MoneyTransferEvent
case class DebitRecordedEvent(details : TransferDetails) extends MoneyTransferEvent
case class CreditRecordedEvent(details : TransferDetails) extends MoneyTransferEvent
case class TransferFailedDueToInsufficientFundsEvent() extends MoneyTransferEvent
```



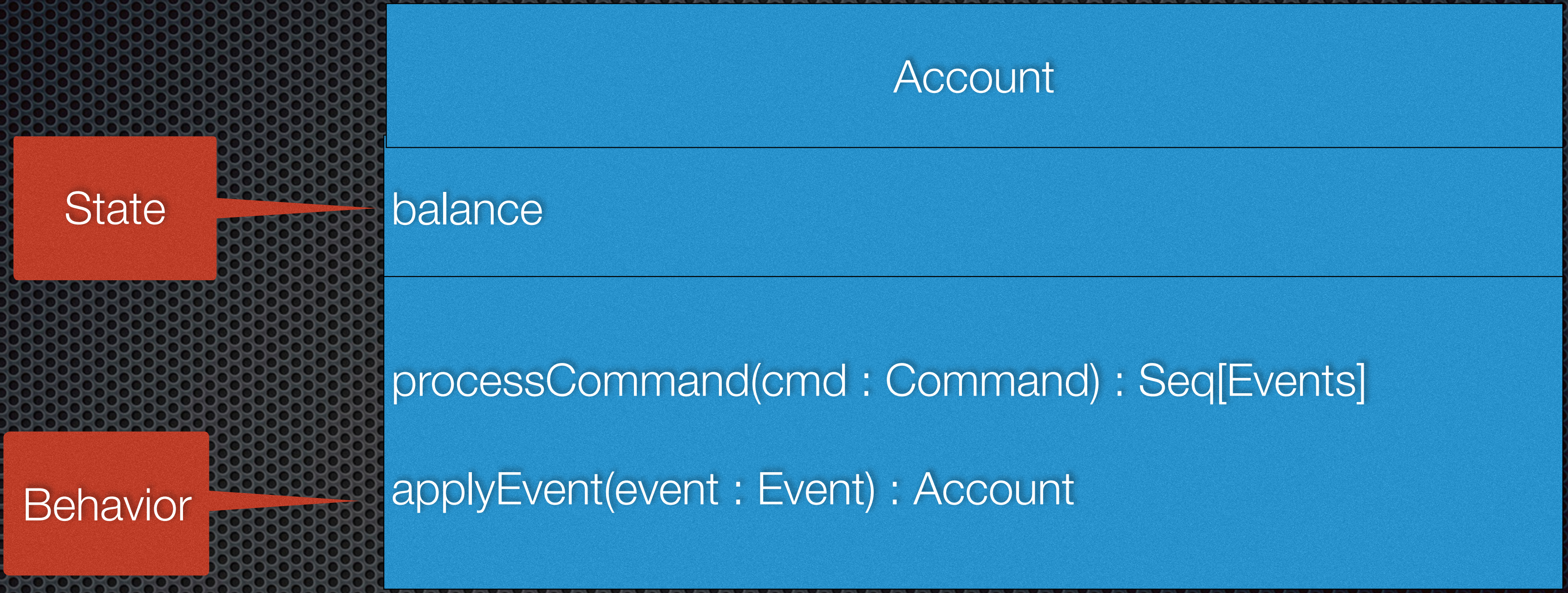
```
object MoneyTransferCommands {

  sealed trait MoneyTransferCommand extends Command

  case class CreateMoneyTransferCommand(details : TransferDetails) extends MoneyTransferCommand
  case class RecordDebitCommand(accountId : EntityId) extends MoneyTransferCommand
  case class RecordDebitFailedDueToInsufficientFundsCommand(accountId : EntityId) extends MoneyTransferCommand
  case class RecordCreditCommand(accountId : EntityId) extends MoneyTransferCommand

}
```



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

Hybrid OO/FP domain objects

OO = State + Behavior



State

Behavior

Account

balance

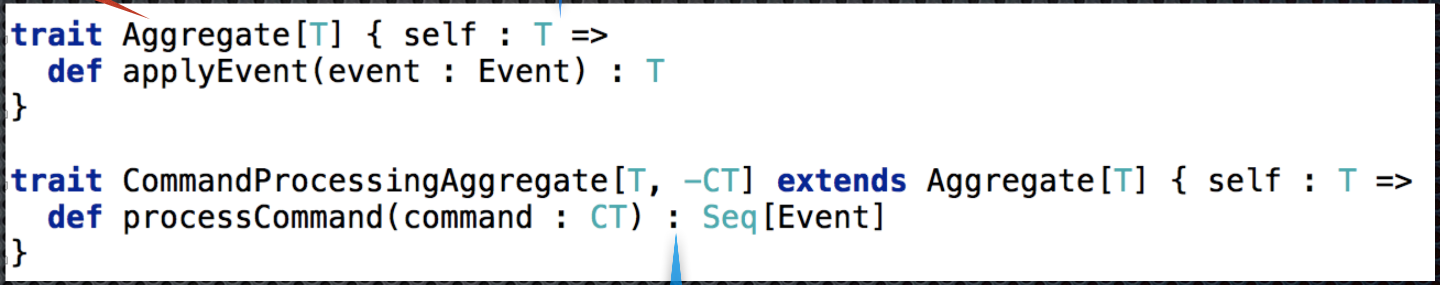
processCommand(cmd : Command) : Seq[Events]

applyEvent(event : Event) : Account

Aggregate traits

Used by Event Store to reconstitute aggregate

Apply event returning updated Aggregate

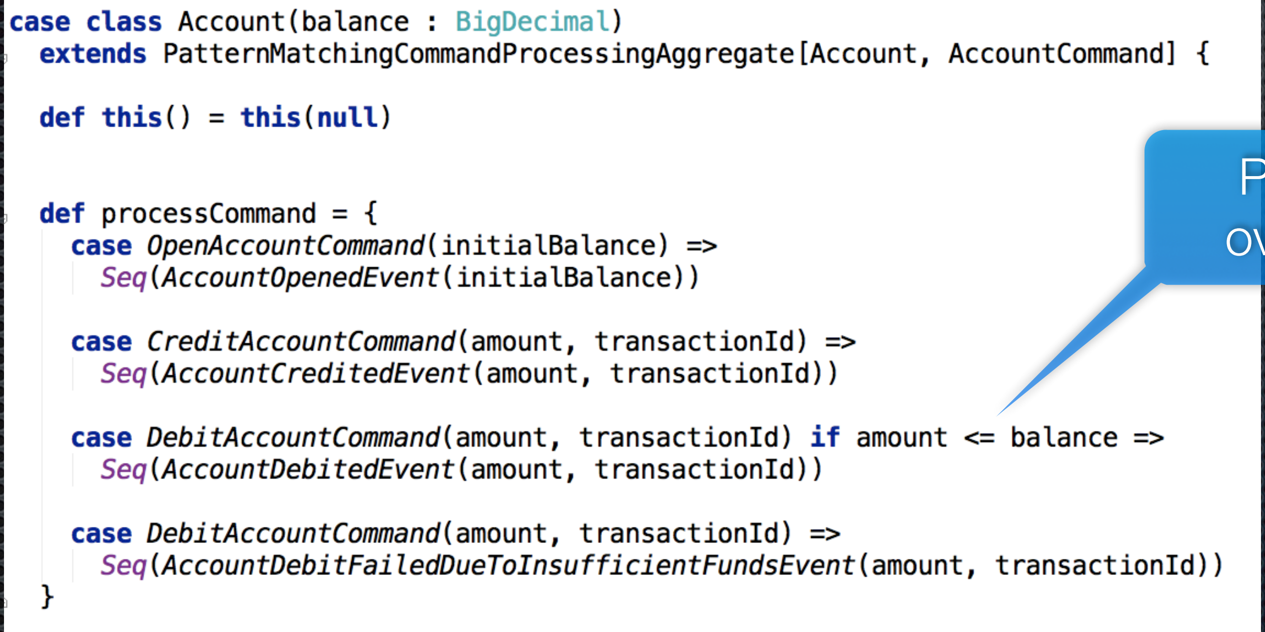


```
trait Aggregate[T] { self : T =>
  def applyEvent(event : Event) : T
}

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

Map Command to Events

@crichardson

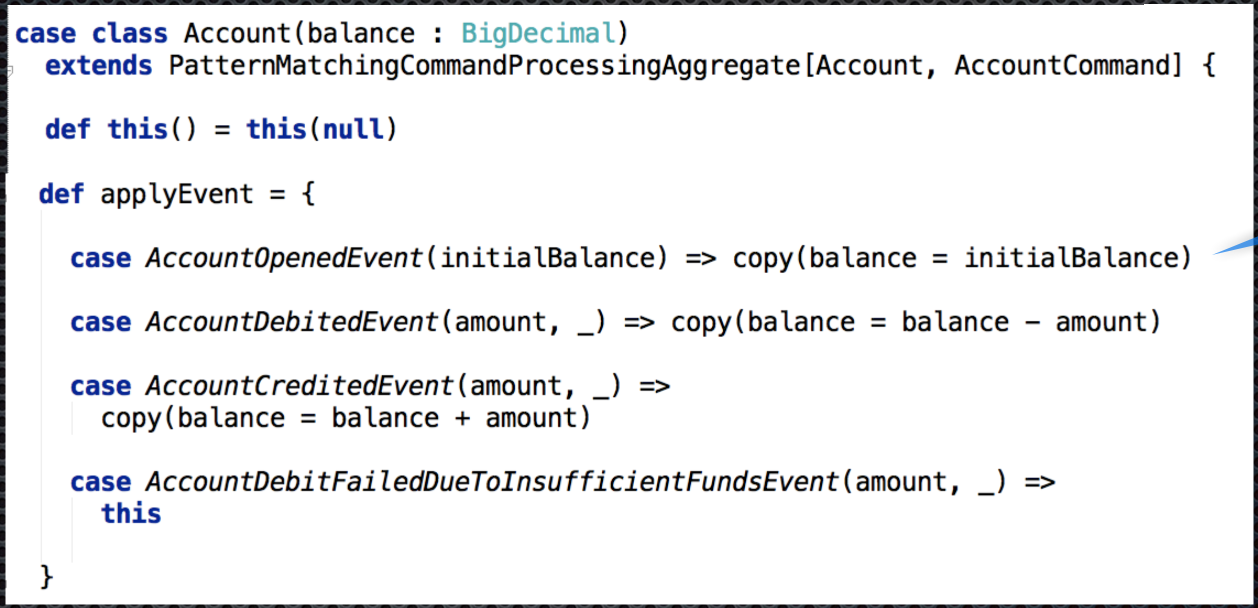
Account - command processing



```
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



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

  def this() = this(null)

  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

  }
```

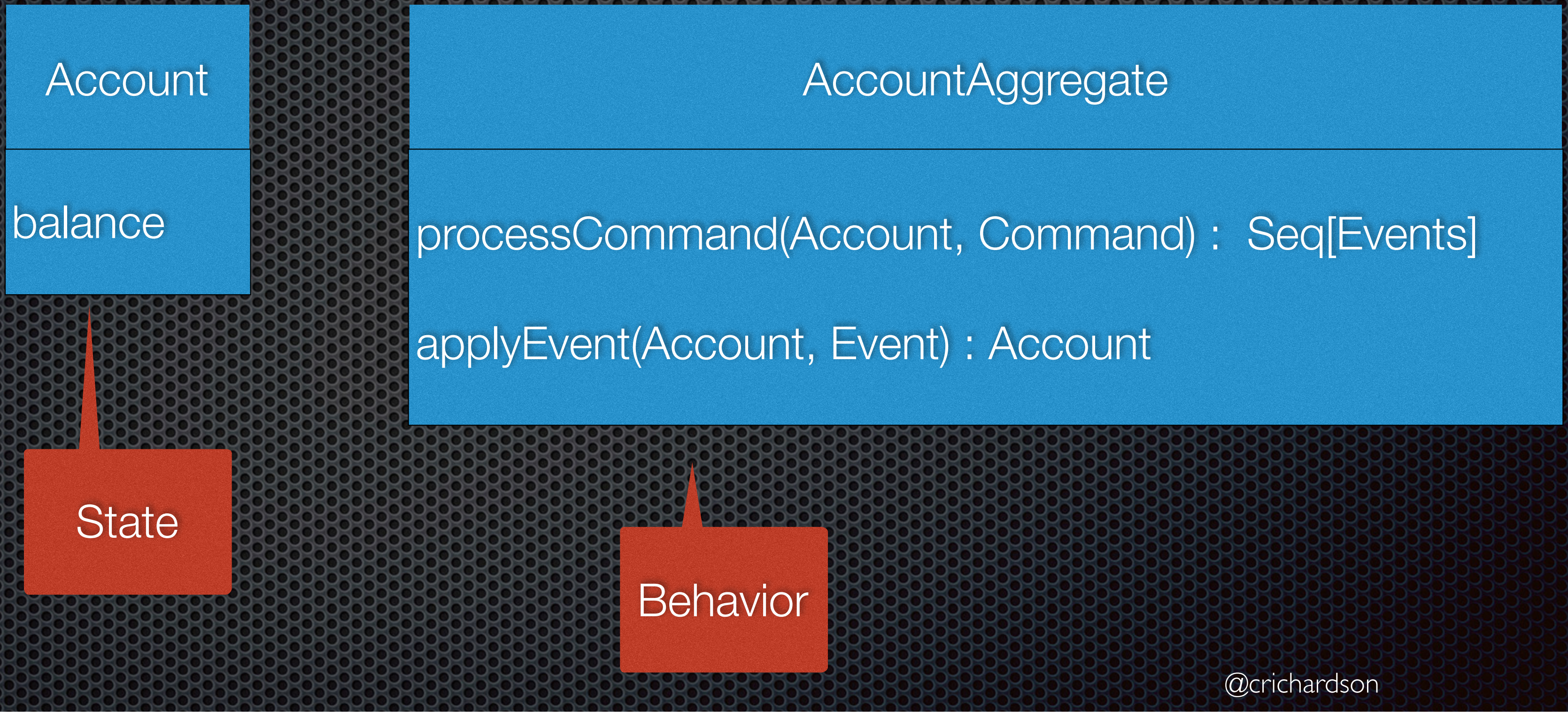
Immutable

Event Store API

```
trait EventStore {

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

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

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

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

  def subscribe(subscriptionId: SubscriptionId):
       Future[AcknowledgableEventStream]
}
```

FP-style domain objects

FP = Separation of State and Behavior



Account

balance

AccountAggregate

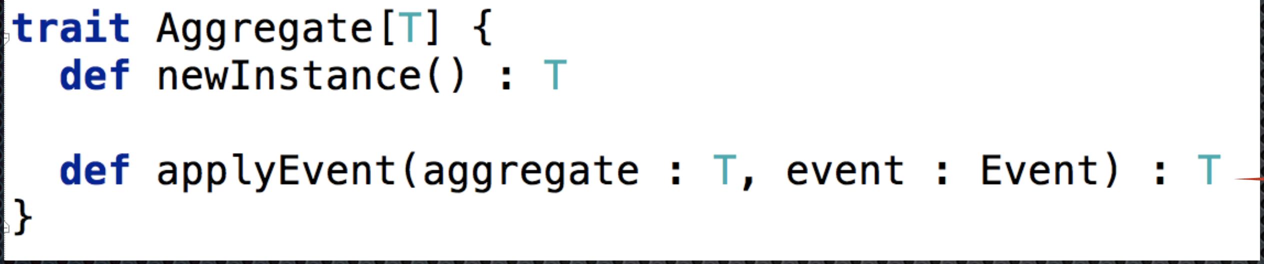
processCommand(Account, Command) : Seq[Events]

applyEvent(Account, Event) : Account

State

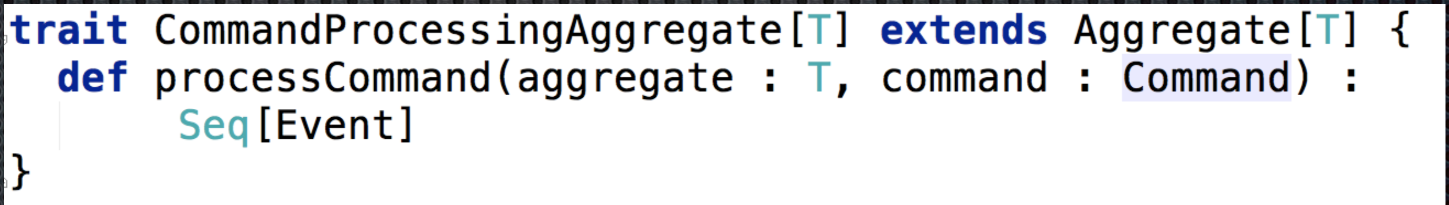
Behavior

Aggregate type classes/implicits



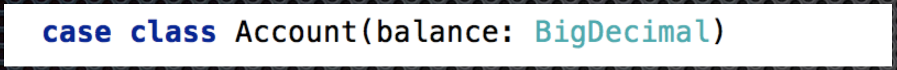
```
trait Aggregate[T] {
  def newInstance() : T

  def applyEvent(aggregate : T, event : Event) : T
}
```

Used by Event Store to reconstitute aggregate



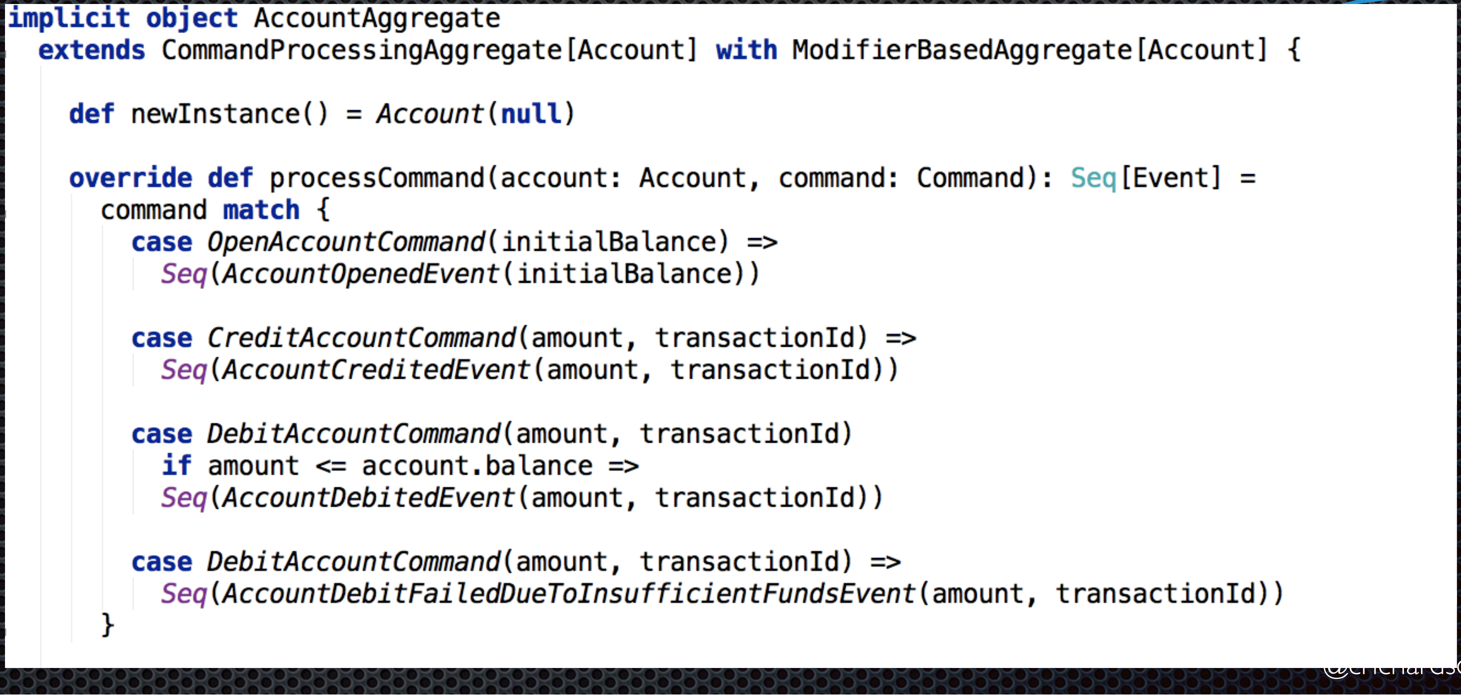
```
trait CommandProcessingAggregate[T] extends Aggregate[T] {
  def processCommand(aggregate : T, command : Command) :
       Seq[Event]
}
```

Functional-style Account Aggregate



```scala
case class Account(balance: BigDecimal)
```

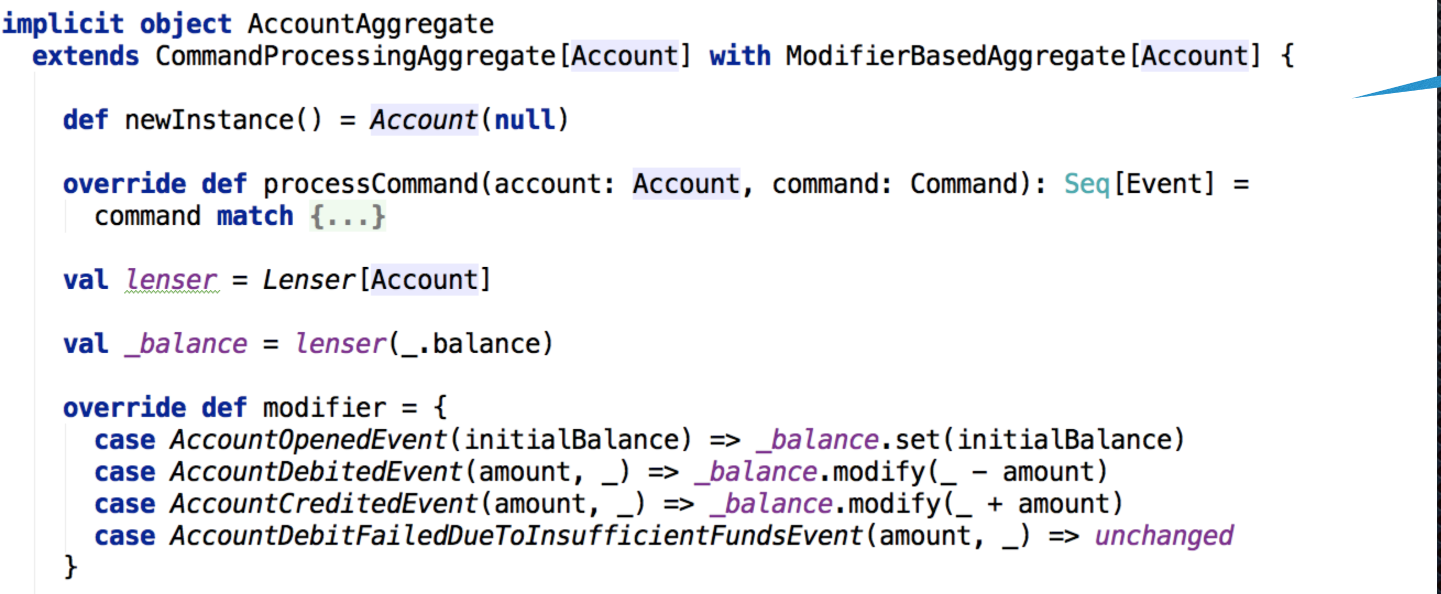
State



```scala
implicit object AccountAggregate
  extends CommandProcessingAggregate[Account] with ModifierBasedAggregate[Account] {

    def newInstance() = Account(null)

    override def processCommand(account: Account, command: Command): Seq[Event] =
      command match {
        case OpenAccountCommand(initialBalance) =>
          Seq(AccountOpenedEvent(initialBalance))

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

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

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

Behavior

Functional-style Account Aggregate

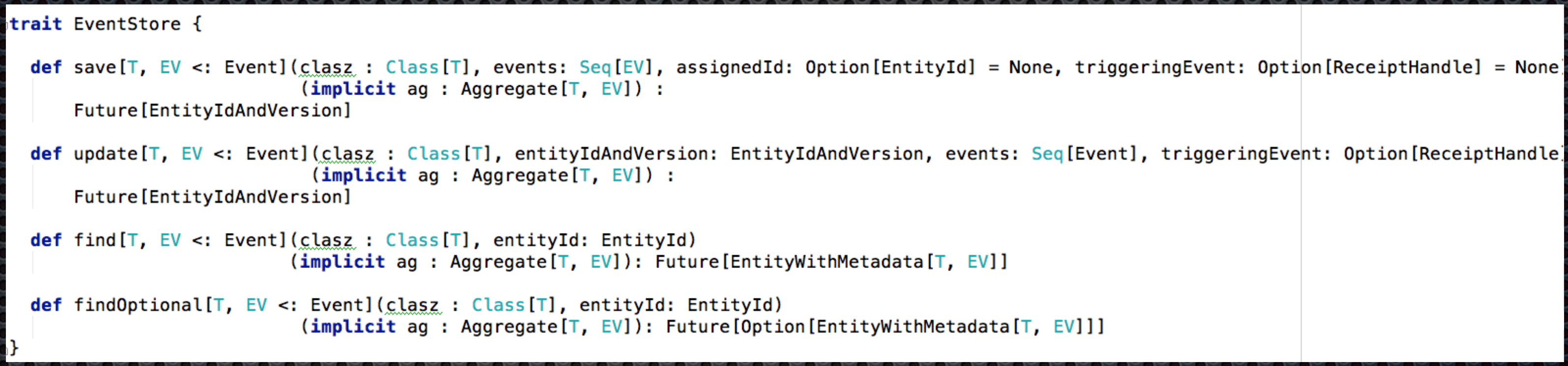


```
implicit object AccountAggregate
  extends CommandProcessingAggregate[Account] with ModifierBasedAggregate[Account] {

    def newInstance() = Account(null)

    override def processCommand(account: Account, command: Command): Seq[Event] =
      command match {...}

    val lenser = Lenser[Account]

    val _balance = lenser(_.balance)

    override def modifier = {
      case AccountOpenedEvent(initialBalance) => _balance.set(initialBalance)
      case AccountDebitedEvent(amount, _) => _balance.modify(_ - amount)
      case AccountCreditedEvent(amount, _) => _balance.modify(_ + amount)
      case AccountDebitFailedDueToInsufficientFundsEvent(amount, _) => unchanged
    }
```

Behavior

FP-style event store

Enables inference of T, and EV

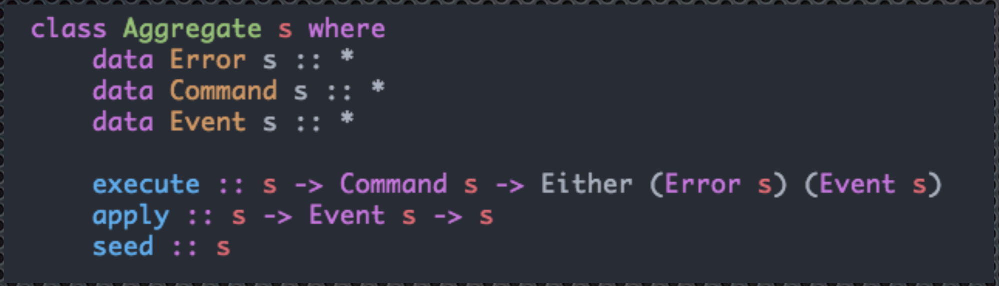


```
trait EventStore {

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

  def update[T, EV <: Event](clasz : Class[T], entityIdAndVersion: EntityIdAndVersion, events: Seq[Event], triggeringEvent: Option[ReceiptHandle
                            (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
=
Strategy

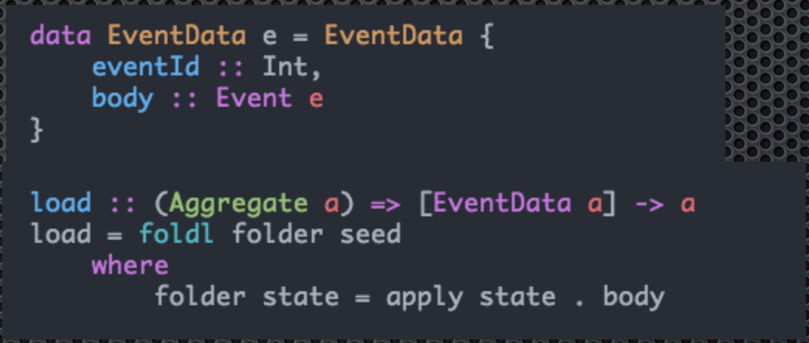
@crichardson

Haskell aggregate



```haskell
class Aggregate s where
    data Error s :: *
    data Command s :: *
    data Event s :: *

    execute :: s -> Command s -> Either (Error s) (Event s)
    apply :: s -> Event s -> s
    seed :: s
```

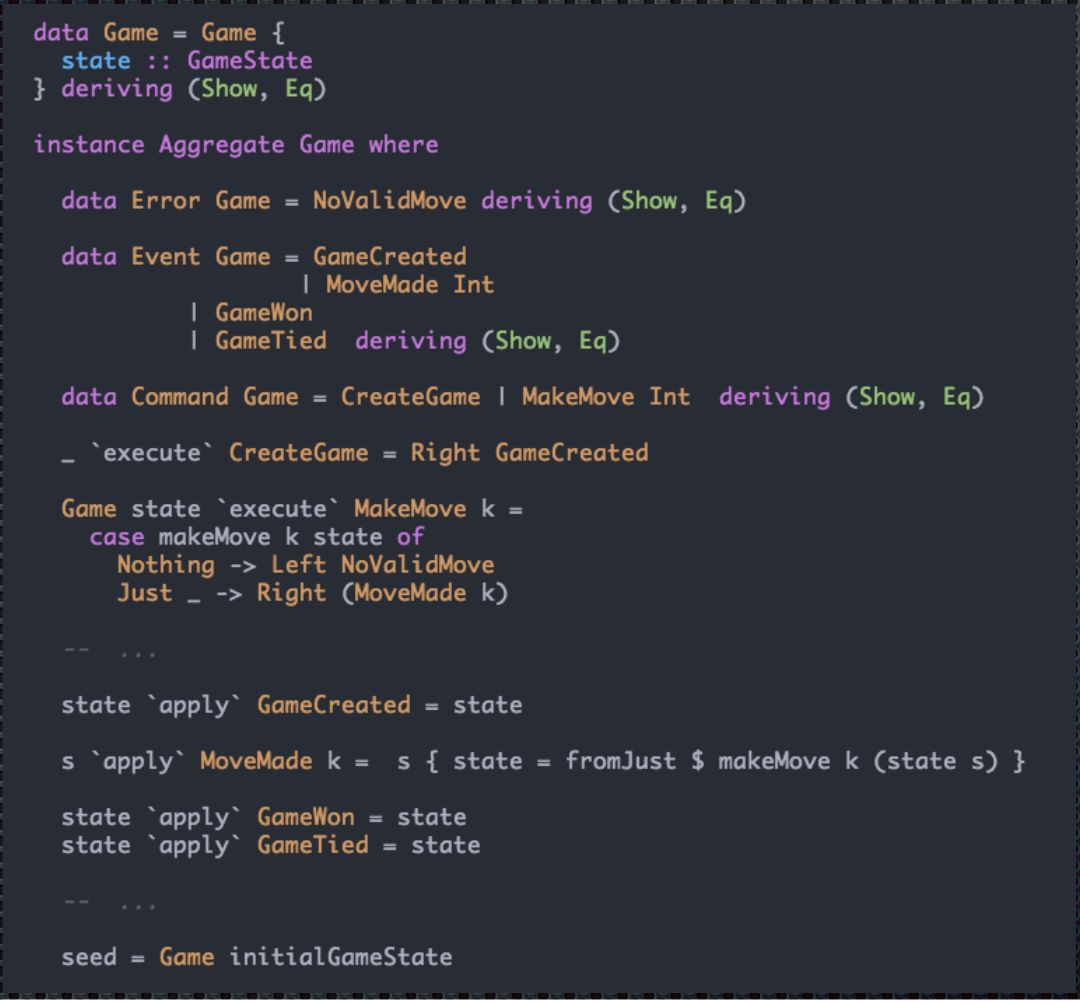
https://gist.github.com/Fristi/7327904



```haskell
data EventData e = EventData {
    eventId :: Int,
    body :: Event e
}

load :: (Aggregate a) => [EventData a] -> a
load = foldl folder seed
    where
        folder state = apply state . body
```

@crichardson

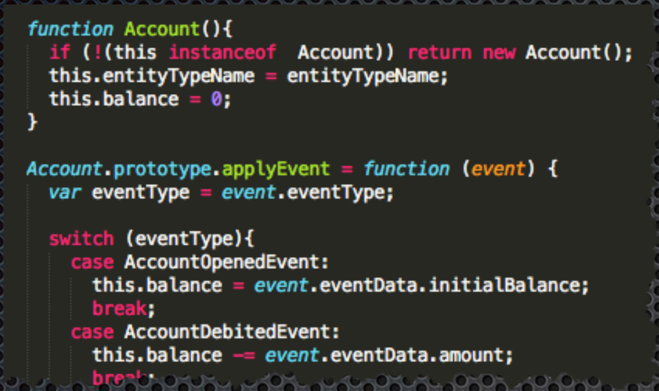
Haskell TicTacToe aggregate



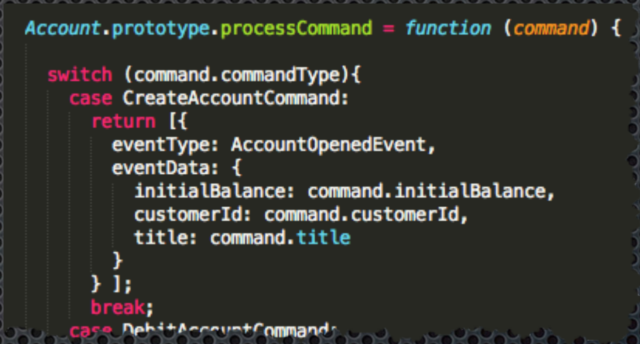
```
data Game = Game {
  state :: GameState
} deriving (Show, Eq)

instance Aggregate Game where

  data Error Game = NoValidMove deriving (Show, Eq)

  data Event Game = GameCreated
                   | MoveMade Int
           | GameWon
           | GameTied  deriving (Show, Eq)

  data Command Game = CreateGame | MakeMove Int  deriving (Show, Eq)

  _ `execute` CreateGame = Right GameCreated

  Game state `execute` MakeMove k =
    case makeMove k state of
      Nothing -> Left NoValidMove
      Just _ -> Right (MoveMade k)

  --  ...

  state `apply` GameCreated = state

  s `apply` MoveMade k =  s { state = fromJust $ makeMove k (state s) }

  state `apply` GameWon = state
  state `apply` GameTied = state

  --  ...

  seed = Game initialGameState
```

@crichardson

JavaScript aggregate



```
function Account(){
  if (!(this instanceof  Account)) return new Account();
  this.entityTypeName = entityTypeName;
  this.balance = 0;
}

Account.prototype.applyEvent = function (event) {
  var eventType = event.eventType;

  switch (eventType){
    case AccountOpenedEvent:
      this.balance = event.eventData.initialBalance;
      break;
    case AccountDebitedEvent:
      this.balance -= event.eventData.amount;
      break;
```



```
Account.prototype.processCommand = function (command) {

  switch (command.commandType){
    case CreateAccountCommand:
      return [{
        eventType: AccountOpenedEvent,
        eventData: {
          initialBalance: command.initialBalance,
          customerId: command.customerId,
          title: command.title
        }
      } ];
      break;
    case DebitAccountCommand:
```

@crichardson

Agenda

- Why event sourcing?
- Designing a domain model based on event sourcing
- Event sourcing and service design
- Microservices and event sourcing

Designing services

- Responsibilities and collaborations
  - Invoked by adapter, e.g. HTTP controller
  - Creates a Command
  - Selects new aggregate or existing aggregate to process command
- Load aggregate from same bounded context, e.g. Add a Post to a Forum - load forum
- Load data from another other bounded context, e.g. addProductToCart()
  - Requests ProductInfo from ProductService
  - Invokes PricingService to calculate discount price
- Sometimes loads target aggregate before creating command
  - e.g. addProductToCart() needs contents of shopping cart to calculate discounted price of product to add

Money transfer example

Story

As a customer of the bank
I want to transfer money between two bank accounts
So that I don't have to write a check

Scenario

Given that my open savings account balance is $150
Given that my open checking account balance is $10
When I transfer $50 from my savings account
to my checking account
Then my savings account balance is $100
Then my checking account balance is $60
Then a MoneyTransfer was created

Pre conditions

Post conditions

Old-style ACID…

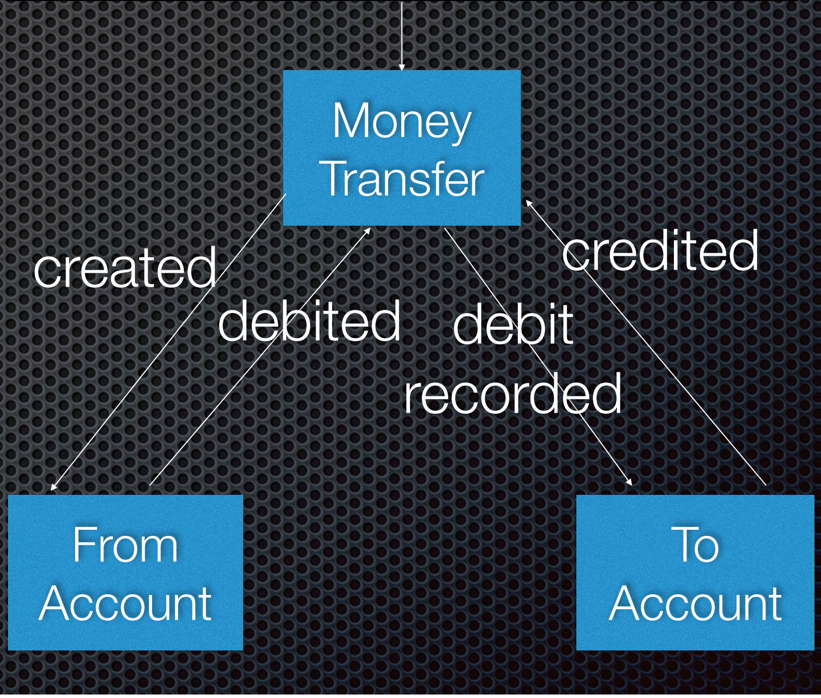
```
public class MoneyTransferServiceImpl …{

   private final AccountRepository accountRepository;
   private final MoneyTransferRepository moneyTransferRepository;
    …
   @Transactional
   public MoneyTransfer transfer(
            String fromAccountId, String toAccountId,
            double amount) throws MoneyTransferException {
    Account fromAccount =
         accountRepository.findAccount(fromAccountId);
    Account toAccount =
         accountRepository.findAccount(toAccountId);
    // … Verify accounts are open …
    fromAccount.debit(amount);
    toAccount.credit(amount);
    return moneyTransferRepository.createMoneyTransfer(
                        fromAccount, toAccount, amount);
    }
}
```

@crichardson

… becomes eventually consistent (BASE)

- Updating multiple aggregates
  - multi-step, event-driven flow
  - each step updates one Aggregate
- Service creates saga to coordinate workflow
  - A state machine
  - Part of the domain, e.g. MoneyTransfer aggregate
  - OR Synthetic aggregate
- Post-conditions eventually true

```
public MoneyTransfer transfer() {
    … Creates MoneyTransfer …
}
```

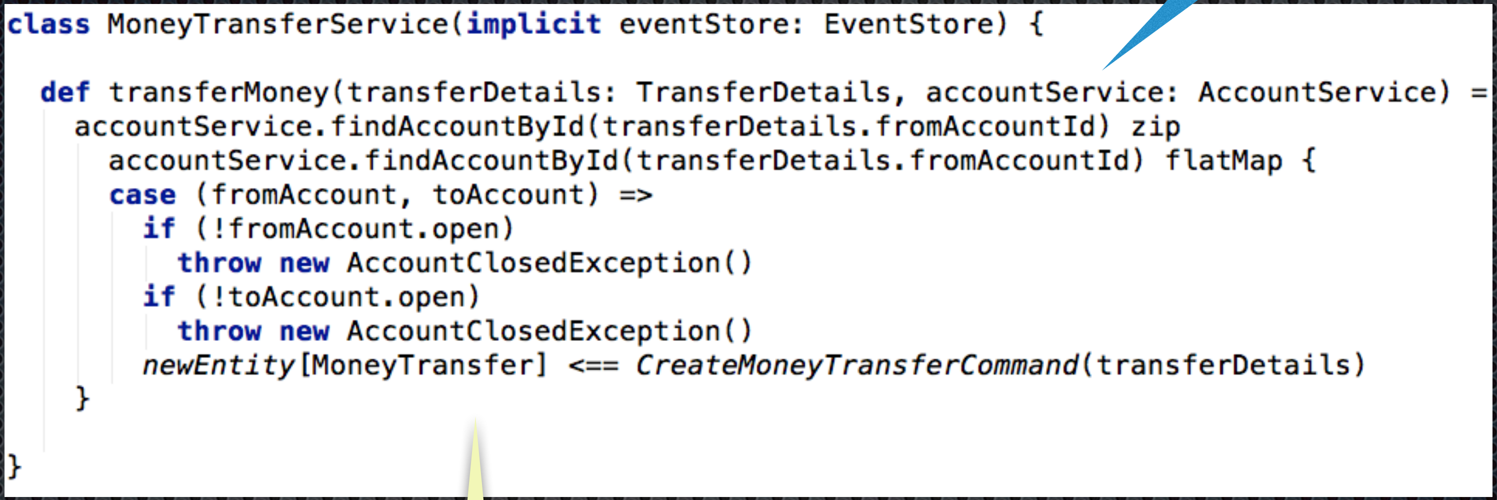


Need compensating transactions

- Pre-conditions might be false when attempting to update an aggregate
- For example: an account might be closed transferring money:
  - *from account* when debiting ⇒ stop transfer
  - *to account* ⇒ reverse the debit
  - *from account* when attempting reversal ⇒ bank wins!

MoneyTransferService

Remoting proxy



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

  def transferMoney(transferDetails: TransferDetails, accountService: AccountService) =
    accountService.findAccountById(transferDetails.fromAccountId) zip
      accountService.findAccountById(transferDetails.fromAccountId) flatMap {
      case (fromAccount, toAccount) =>
        if (!fromAccount.open)
          throw new AccountClosedException()
        if (!toAccount.open)
          throw new AccountClosedException()
        newEntity[MoneyTransfer] <== CreateMoneyTransferCommand(transferDetails)
    }

}
```

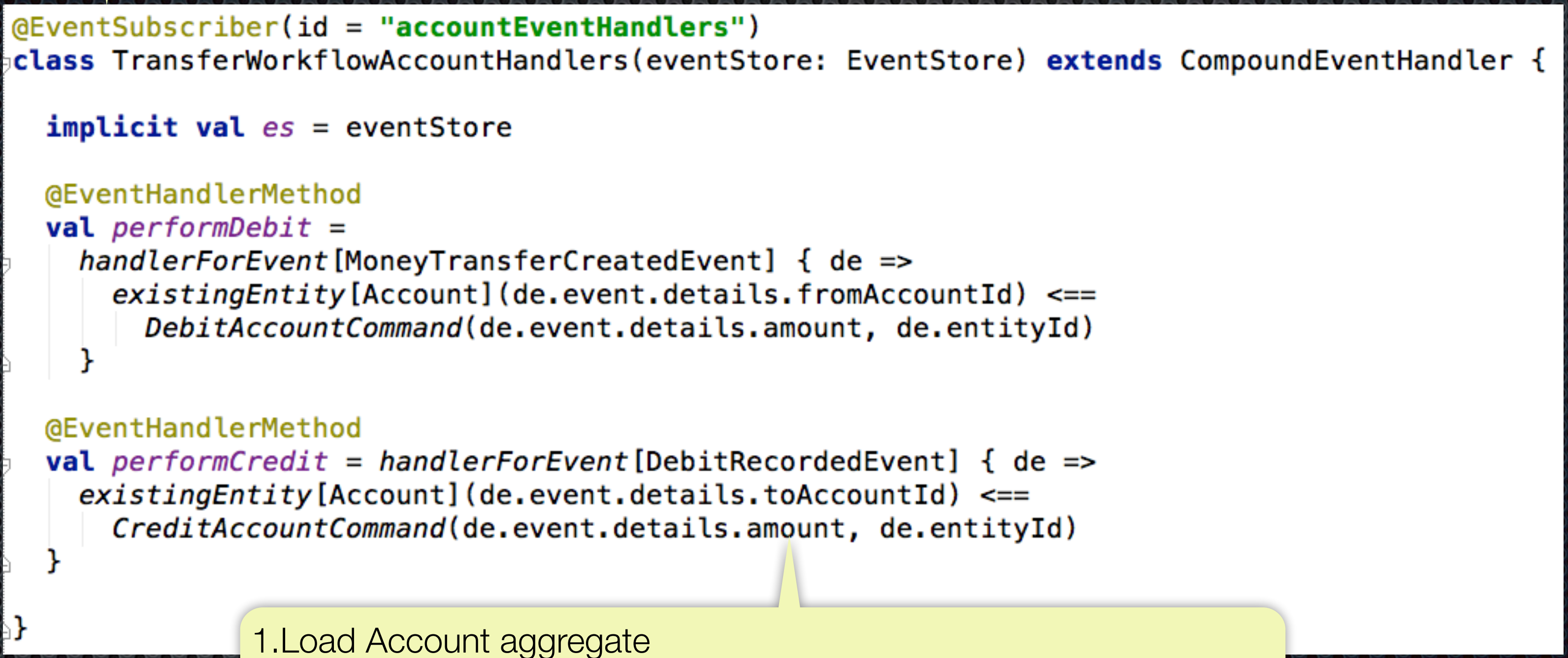
DSL concisely specifies:

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

Event handling in Account

Triggers BeanPostProcessor

Durable subscription name

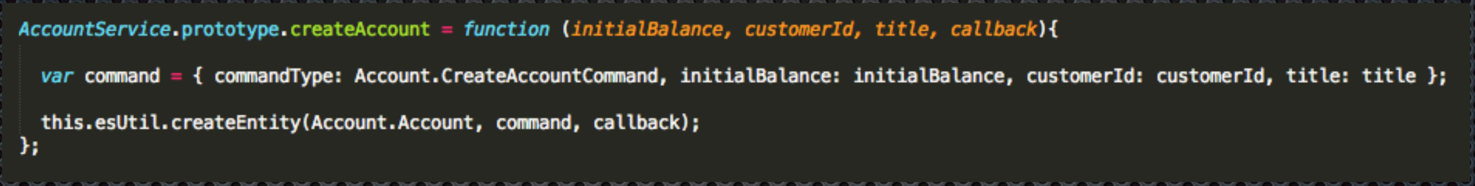


```
@EventSubscriber(id = "accountEventHandlers")
class TransferWorkflowAccountHandlers(eventStore: EventStore) extends CompoundEventHandler {

  implicit val es = eventStore

  @EventHandlerMethod
  val performDebit =
    handlerForEvent[MoneyTransferCreatedEvent] { de =>
      existingEntity[Account](de.event.details.fromAccountId) <==
        DebitAccountCommand(de.event.details.amount, de.entityId)
    }

  @EventHandlerMethod
  val performCredit = handlerForEvent[DebitRecordedEvent] { de =>
    existingEntity[Account](de.event.details.toAccountId) <==
      CreditAccountCommand(de.event.details.amount, de.entityId)
  }

}
```

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

@crichardson

JavaScript service



```
AccountService.prototype.createAccount = function (initialBalance, customerId, title, callback){

  var command = { commandType: Account.CreateAccountCommand, initialBalance: initialBalance, customerId: customerId, title: title };

  this.esUtil.createEntity(Account.Account, command, callback);
};
```

@crichardson

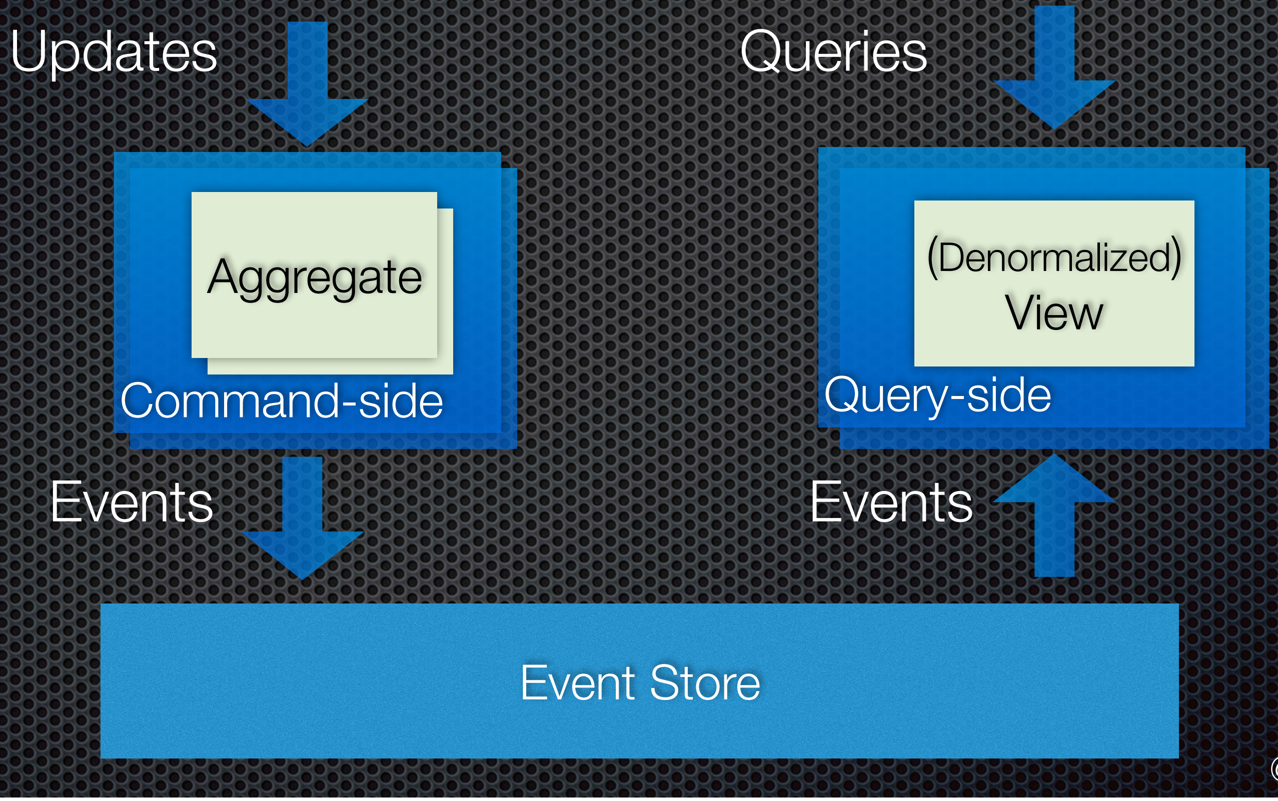
Agenda

- Why event sourcing?
- Designing a domain model based on event sourcing
- Event sourcing and service design
- Microservices and event sourcing

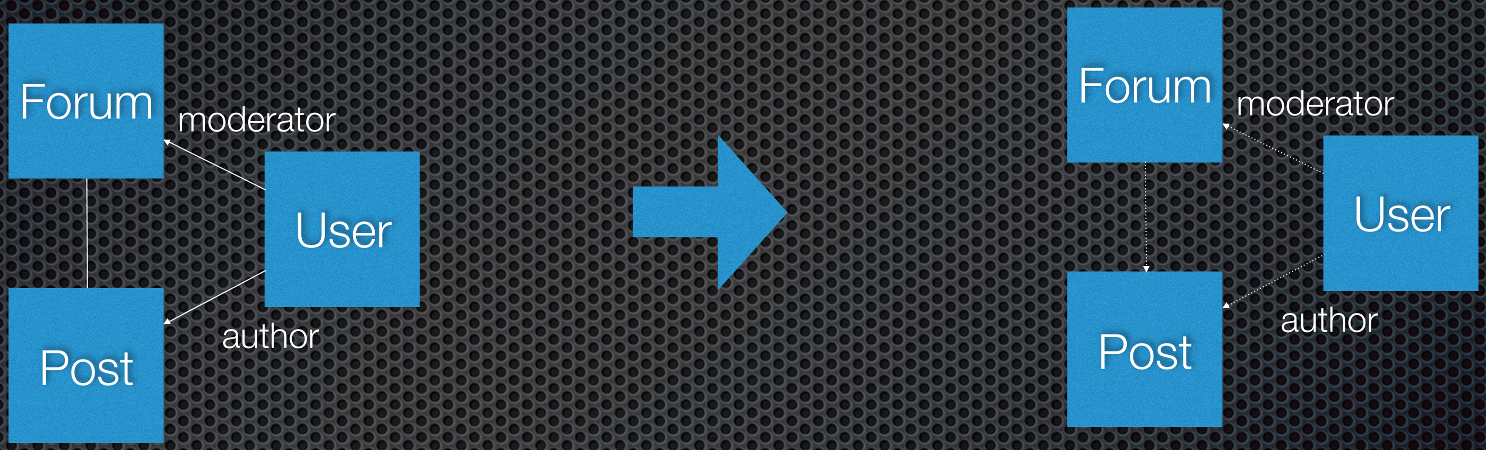
Event Store only supports PK-based lookup

Therefore….

ES+CQRS-based microservices architecture



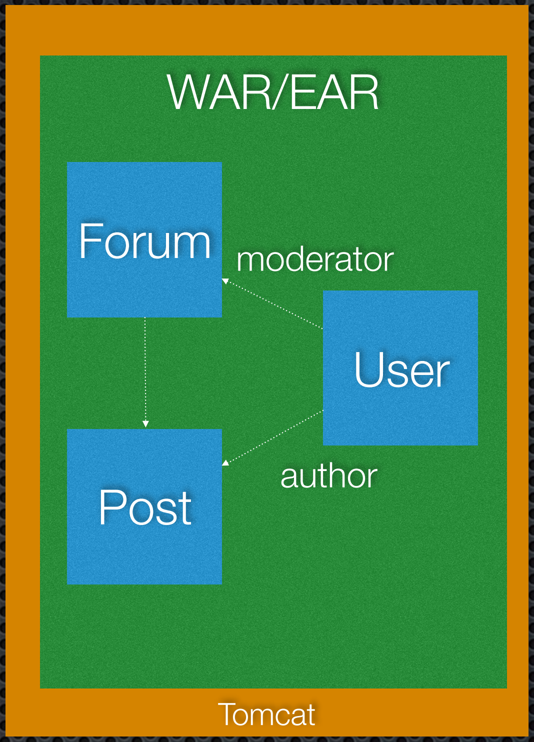
Modular domain model



Tightly coupled
ACID

Loosely coupled aggregates
Eventually consistent

MonolithicFirst approach

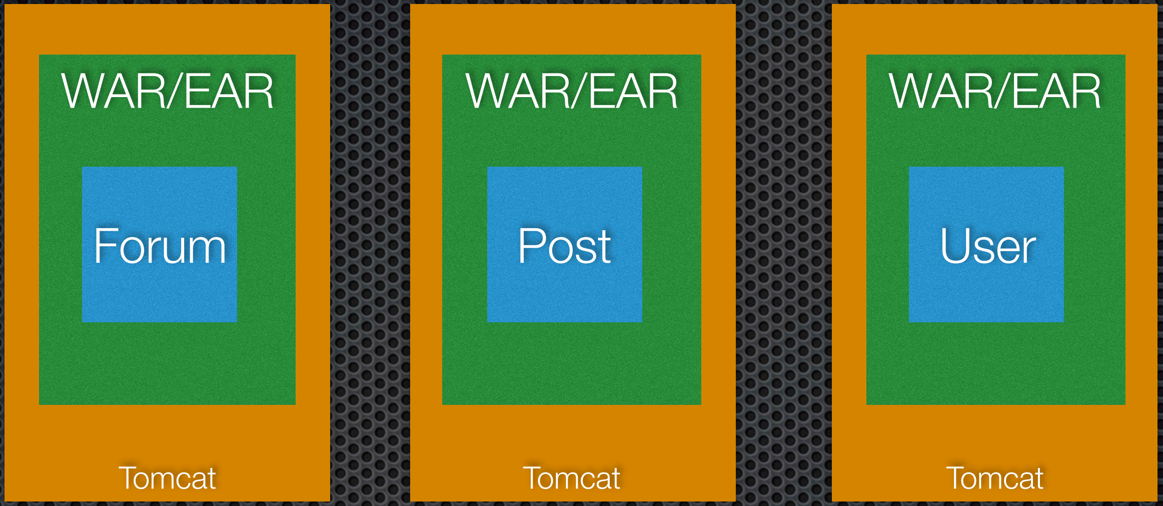


Not entirely free though -
Event Sourcing premium

@crichardson

But no Big Ball of Mud to untangle

Microservices deployment



Much higher - microservices premium

@crichardson

Summary

- Event sourcing solves a variety of problems in modern application architectures
- Scala is a great language for implementing ES-based domain models:
  - Case classes
  - Pattern matching
  - Recreating state = functional fold over events
- But Java, JavaScript and Haskell work too!
- ES-based architecture = flexible deployment

@crichardson chris@chrisrichardson.net

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