Building and deploying microservices with event sourcing, CQRS and Docker

Chris Richardson

Author of POJOs in Action

Founder of the original CloudFoundry.com

@crichardson

chris@chrisrichardson.net

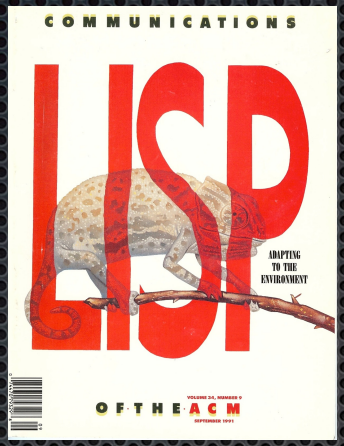
http://plainoldobjects.com

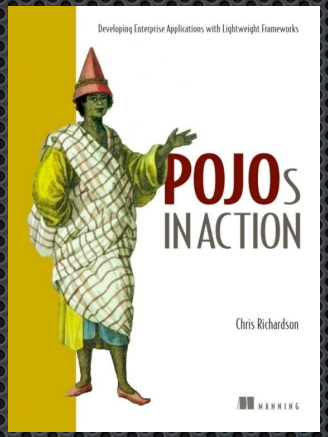
http://microservices.io

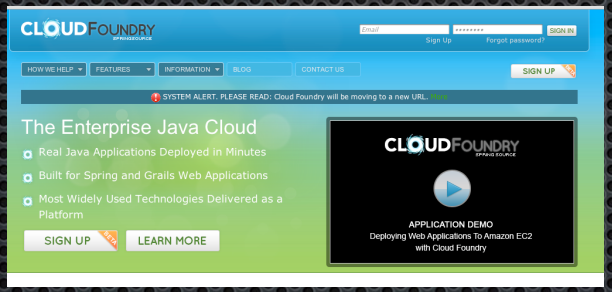
Presentation goal

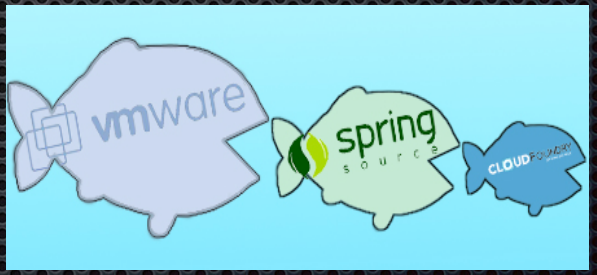
Share my experiences with building and deploying an application using Scala, functional domain models, microservices, event sourcing, CQRS, and Docker

About Chris


COMMUNICATIONS
LISP
ADAPTING TO THE ENVIRONMENT
OF THE ACM


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


CLOUDFOUNDRY
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
APPLICATION DEMO
Deploying Web Applications To Amazon EC2 with Cloud Foundry


vmware
spring
source

About Chris

- Founder of a buzzword compliant (stealthy, social, mobile, big data, machine learning, ...) startup
- Consultant helping organizations improve how they architect and deploy applications using cloud, micro services, polyglot applications, NoSQL, ...
- Creator of http://microservices.io

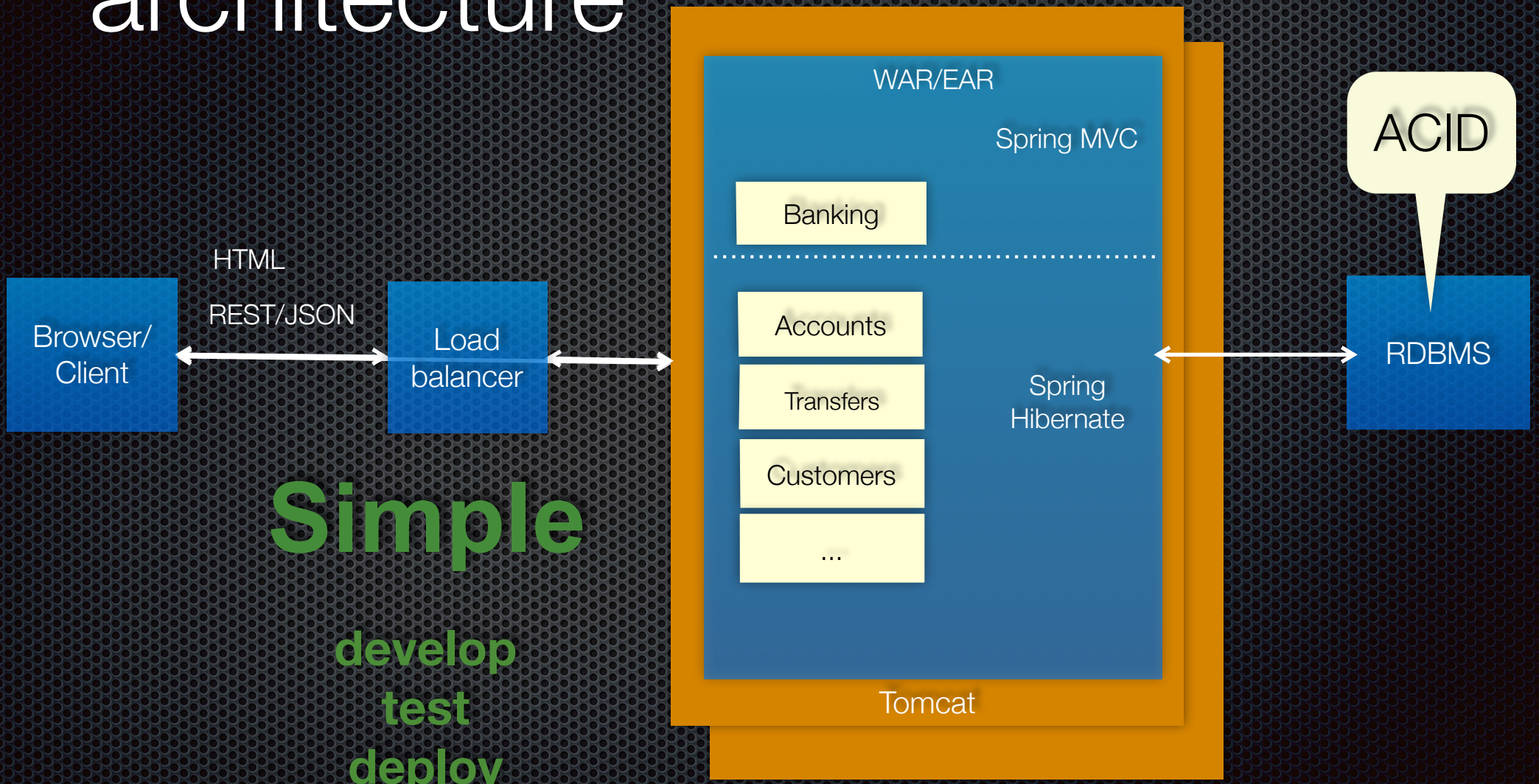
For more information

- https://github.com/cer/event-sourcing-examples
- http://microservices.io
- 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
- Building and deploying microservices

Traditional application architecture



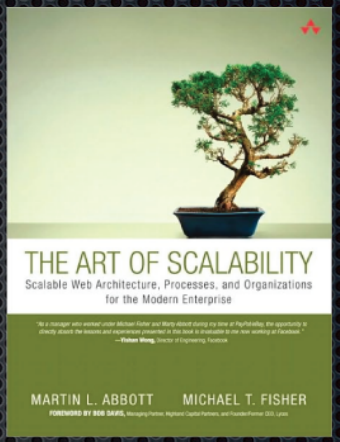
Simple

develop
test
deploy
scale

Limitations of the monolithic architecture

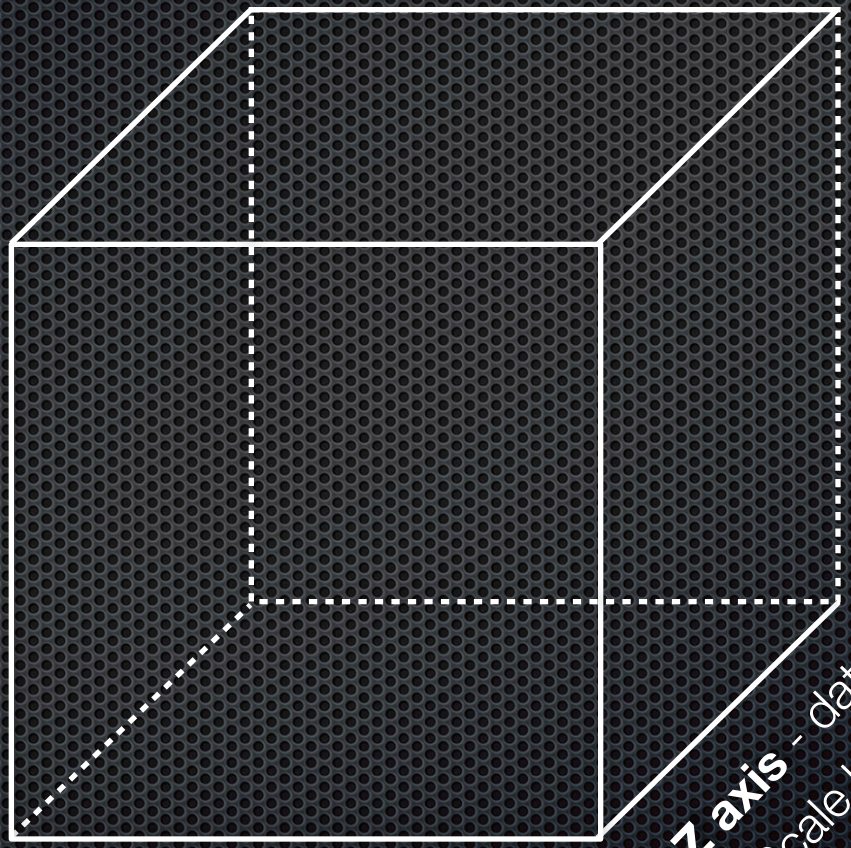
- Intimidates developers
- Obstacle to frequent deployments
- Overloads your IDE and container
- Obstacle to scaling development
- Modules having conflicting scaling requirements
- Requires long-term commitment to a technology stack

Apply the scale cube



Y axis - functional decomposition

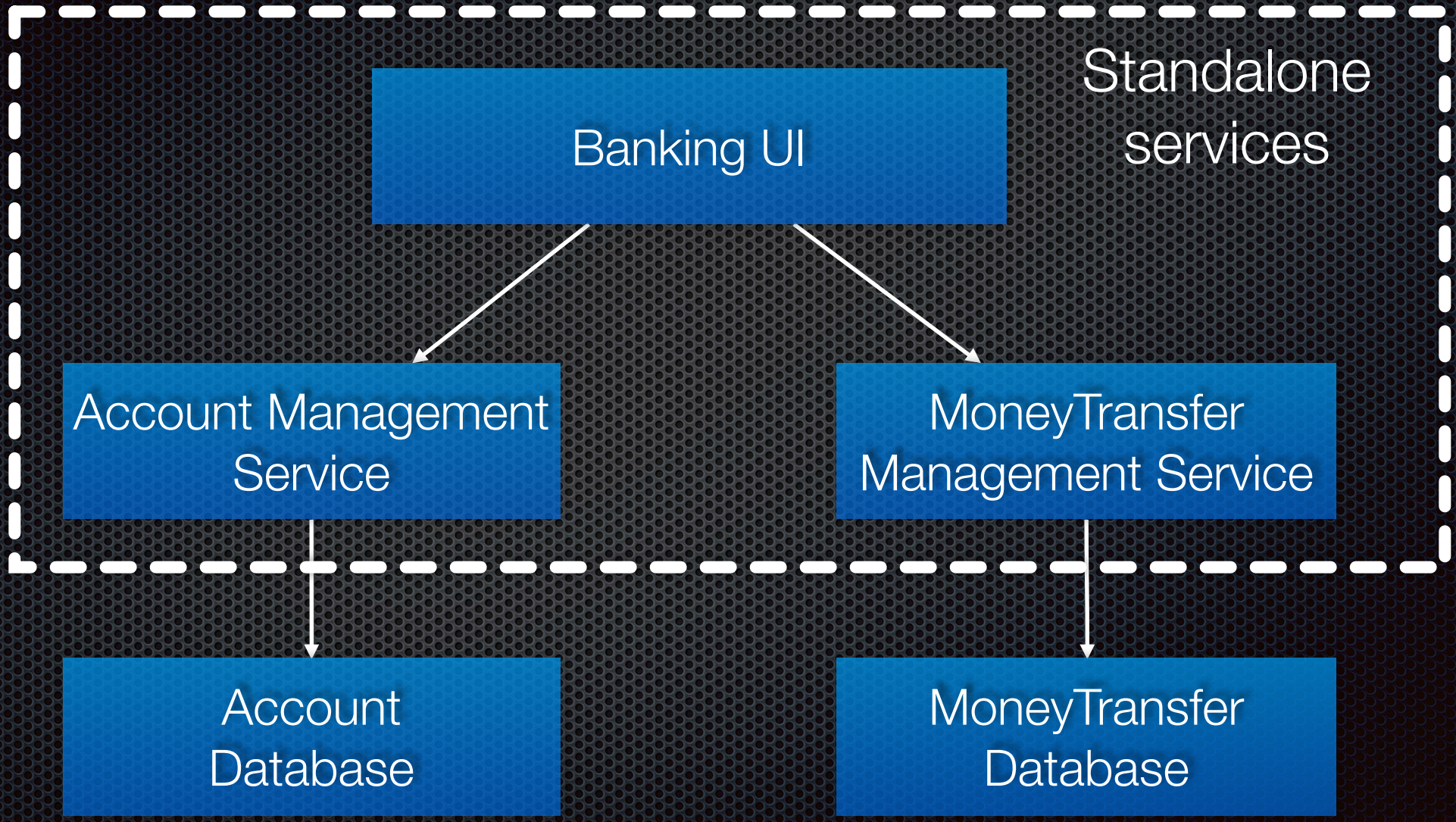
Scale by splitting different things



Z axis - data partitioning
Scale by splitting similar things

X axis
- horizontal duplication

Use a microservice architecture



Limitations of a single relational database

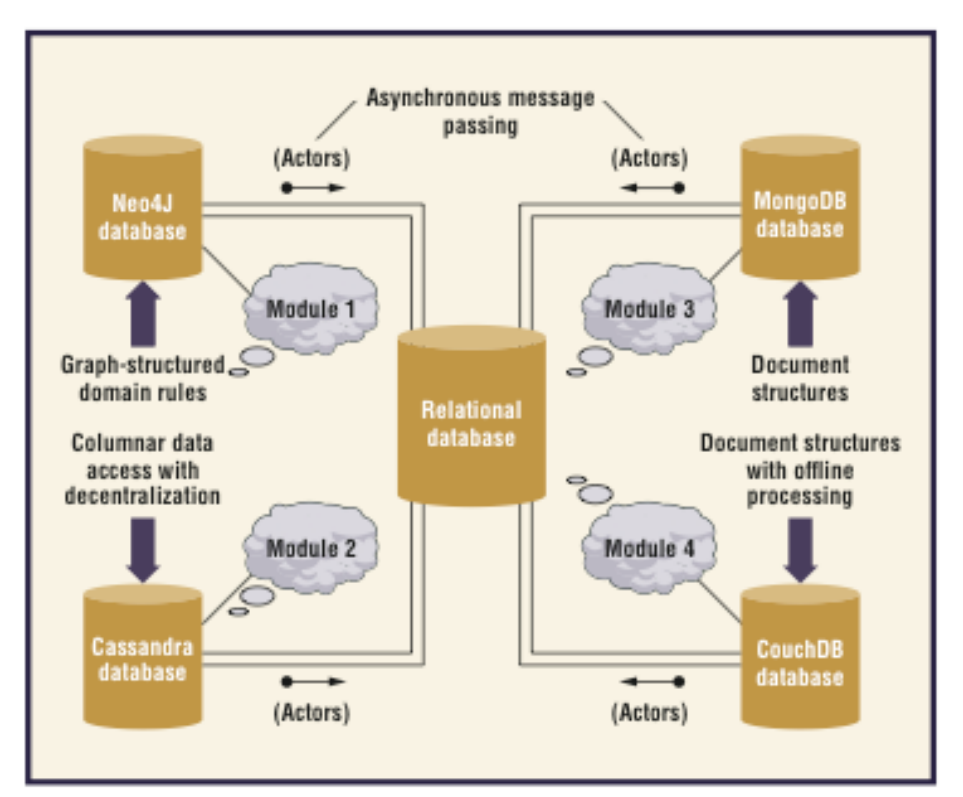
- Scalability
- Distribution
- Schema updates
- O/R impedance mismatch
- Handling semi-structured data

Use a sharded relational database

Use NoSQL databases

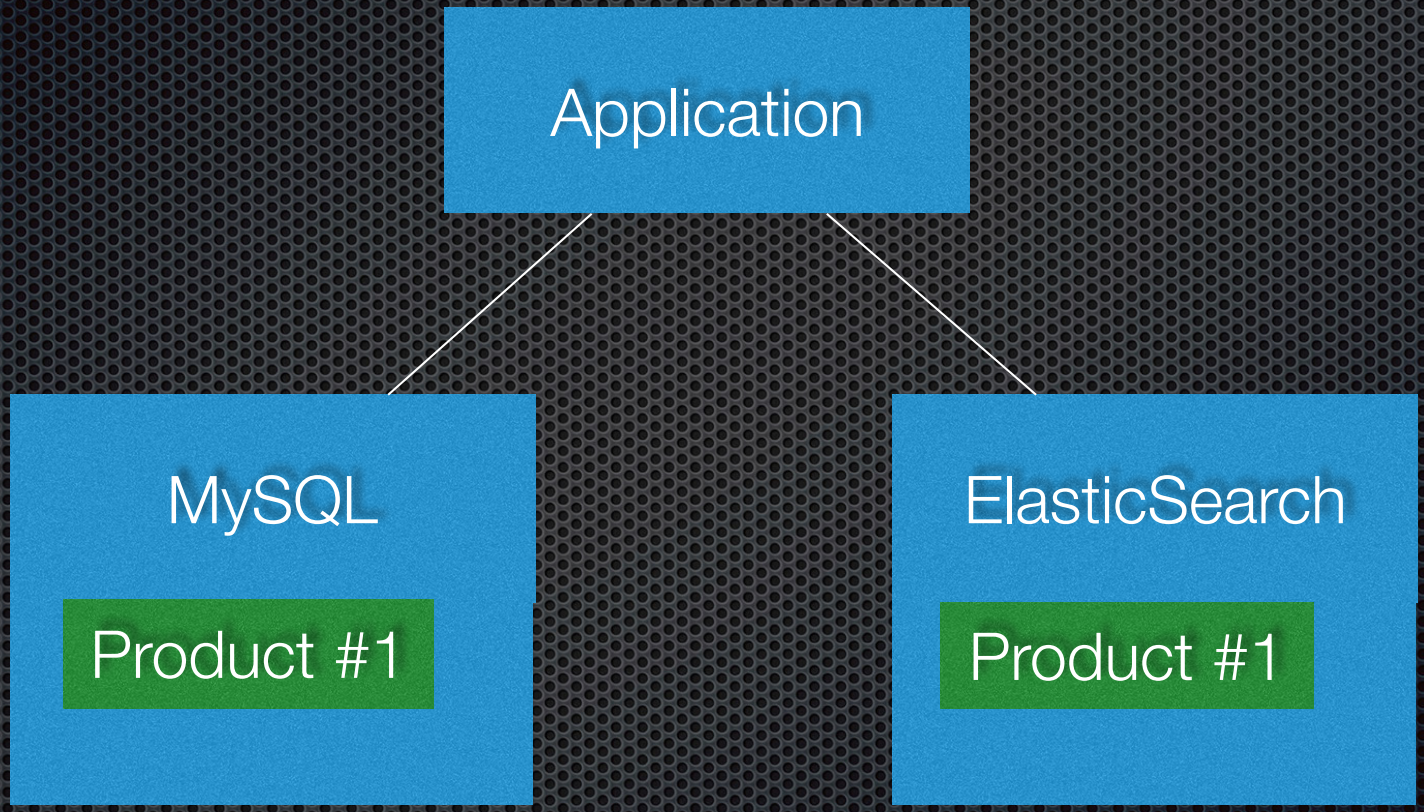
- Avoids the limitations of RDBMS
- For example,
  - text search ⇒ Solr/Cloud Search
  - social (graph) data ⇒ Neo4J
  - highly distributed/available database ⇒ Cassandra
  - ...

Different modules use different types of databases



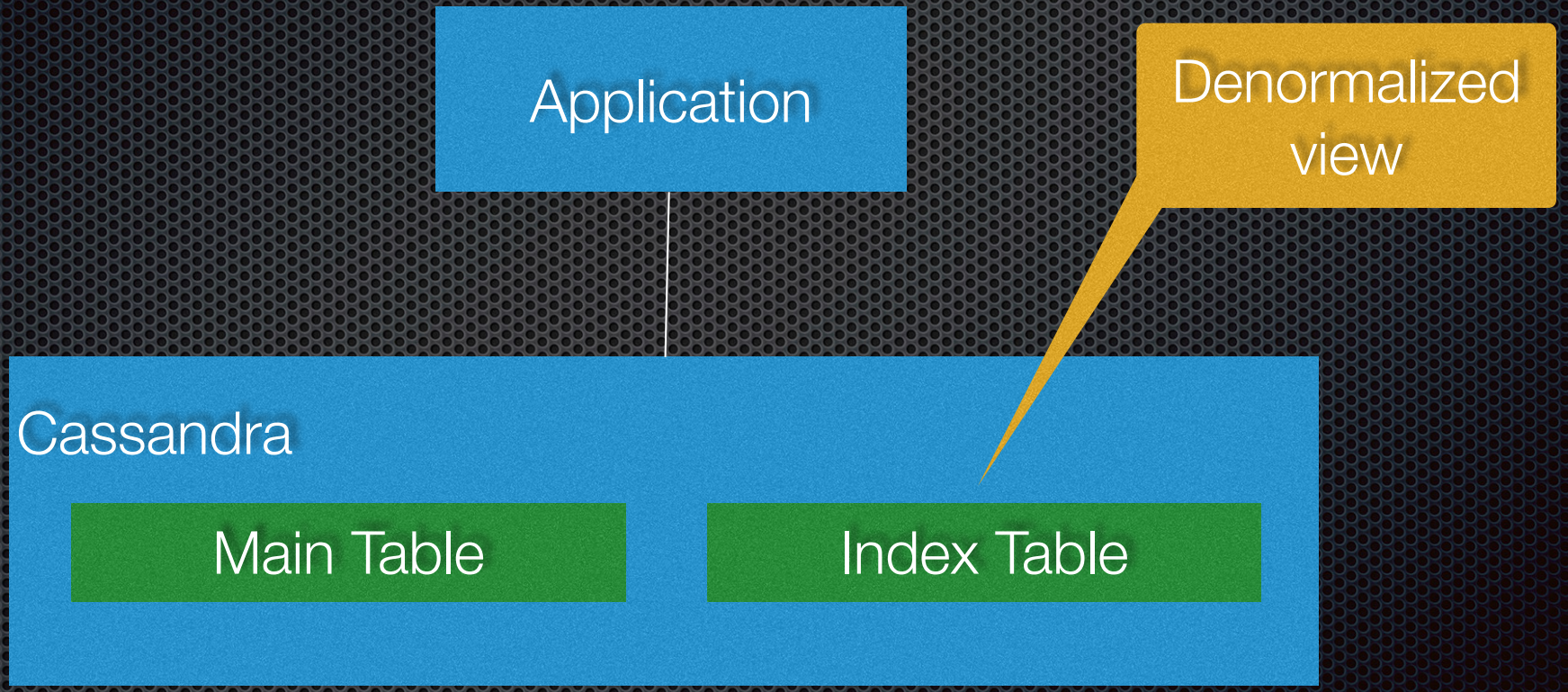
But this results in distributed data management problems

Example #1 - SQL + Text Search engine



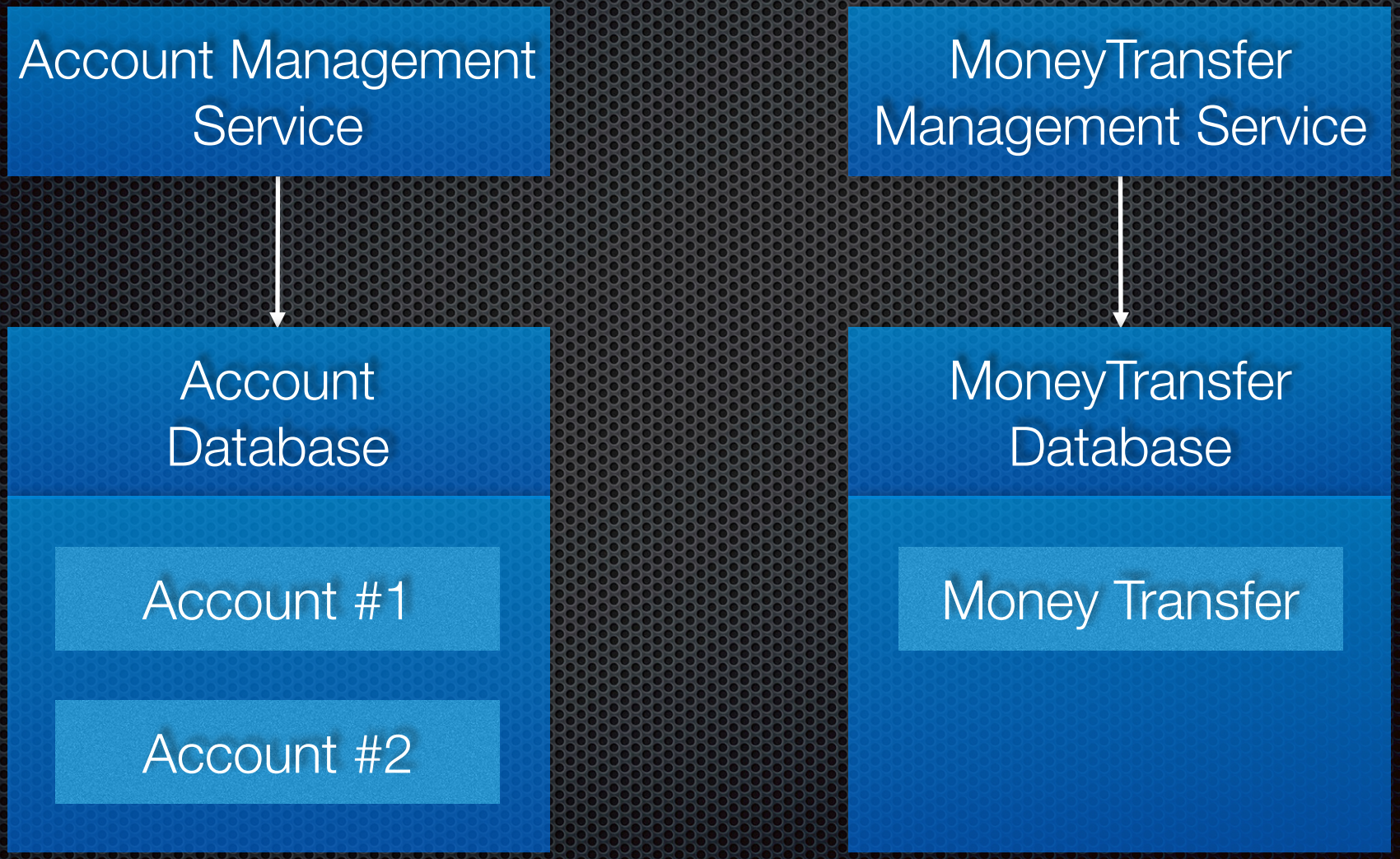
How to maintain consistency without 2PC?

Example #2 - Cassandra main table <=> index table



How to maintain consistency without 2PC?

Example #3: Money transfer

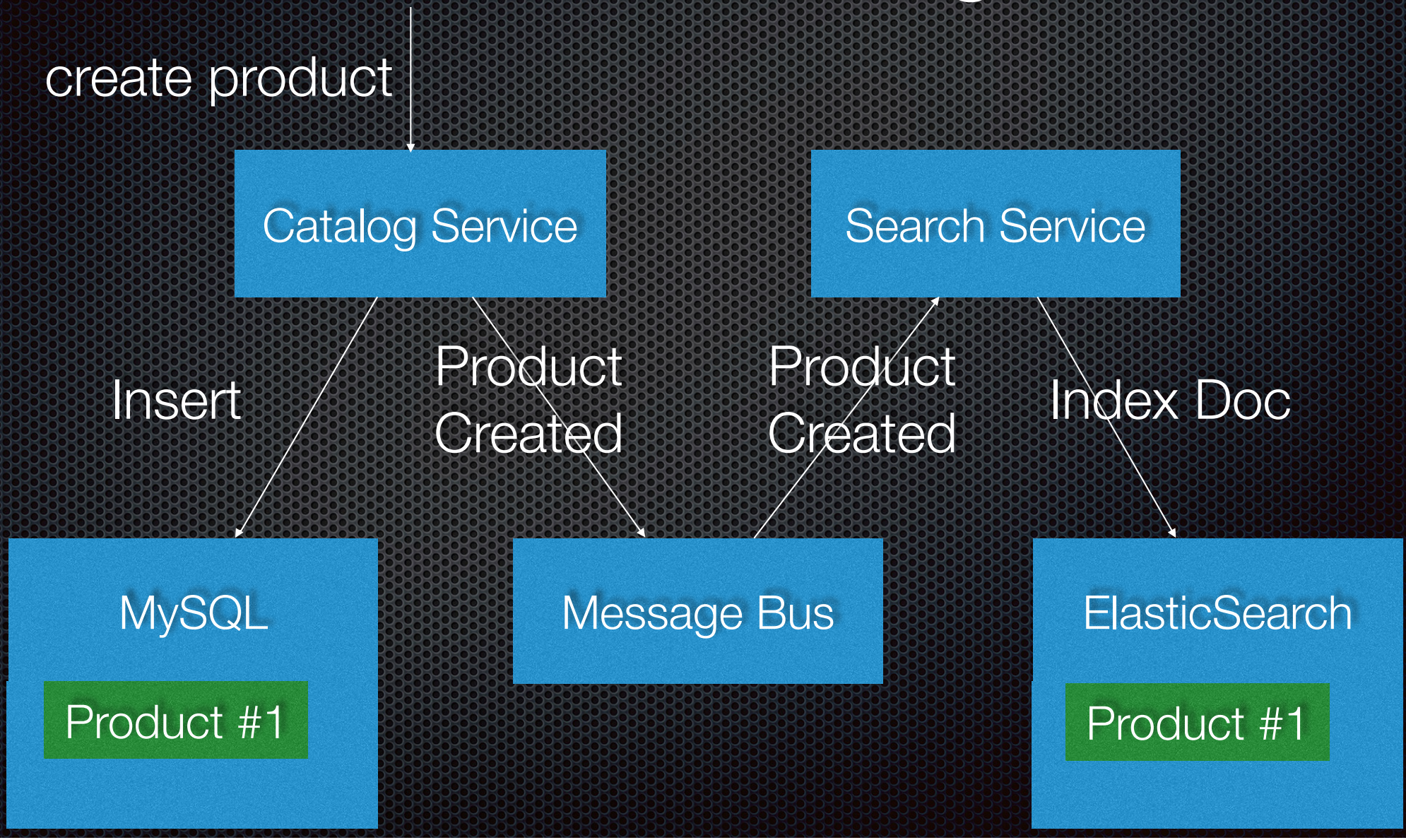


How to maintain consistency without 2PC?

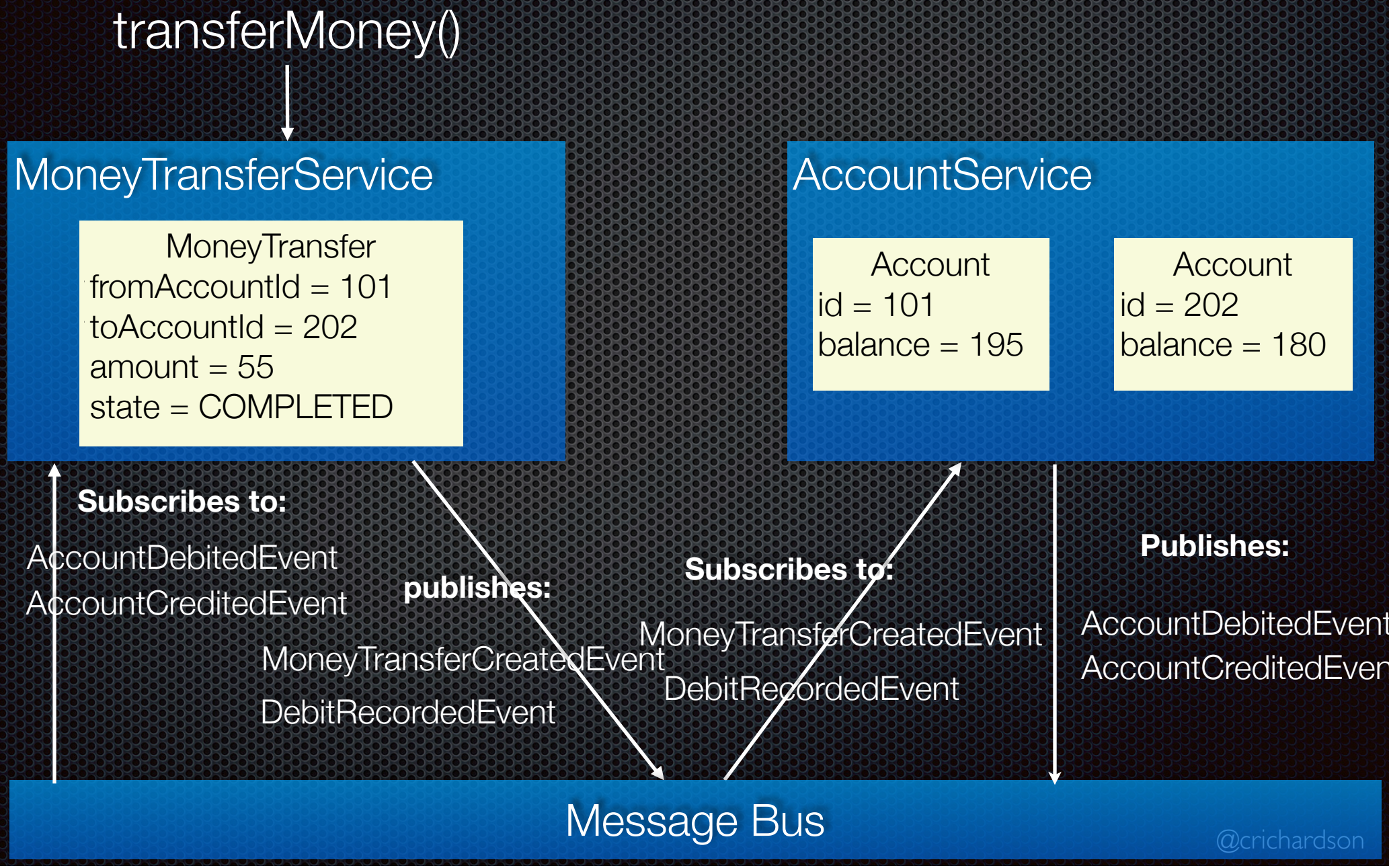
Event-based architecture to the rescue

- Components (e.g. services) publish events when state changes
- Components subscribe to events
  - Maintains eventual consistency across multiple aggregates (in multiple datastores)
  - Synchronize replicated data

Event-driven synchronization: SQL + Text Search engine



Eventually consistent money transfer



To maintain consistency
a service must
atomically publish an event
whenever
a domain object changes

How to atomically update the datastore and publish event(s)?

- Use 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 datastore as a message queue
  1. Update database: new entity state & event
  2. Consume event & mark event as consumed
  - Eventually consistent mechanism
  - 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
- Building and deploying microservices

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 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)

Two actions that must be atomic

Before: update state + publish events

Now: persist (and publish) events

Single action that can be done atomically

Aggregate traits

Apply event returning updated Aggregate

```
trait Aggregate[T] { self : T =>
  def applyEvent : PartialFunction[Event, T]
}

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

Map Command to Events

Account - command processing

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

  def this() = this(null)

  import net.chrisrichardson.eventstore.examples.bank.accounts.AccountCommands._

  def processCommand = {
    case OpenedAccountCommand(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))
  }
```

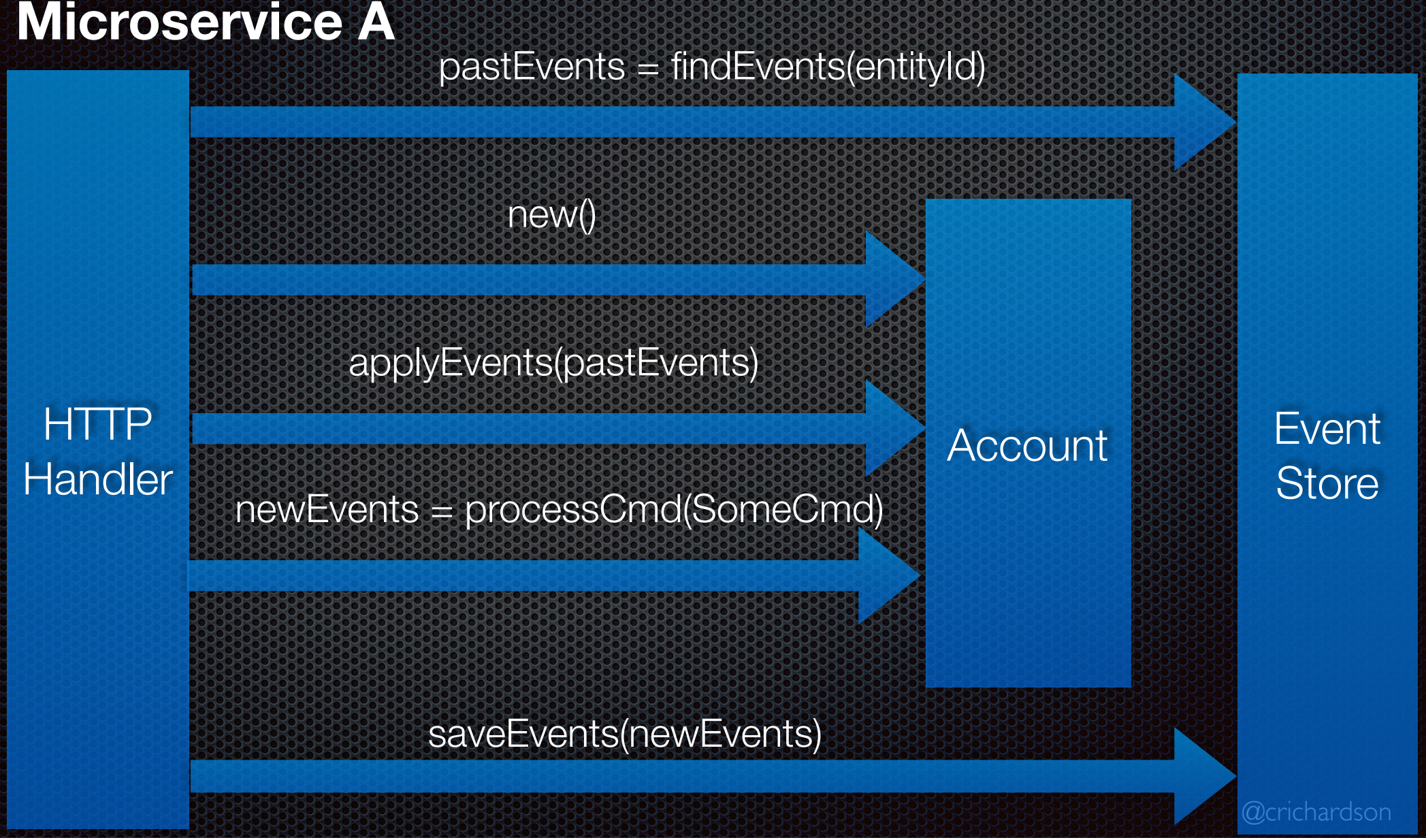
Prevent overdraft

Account - applying events

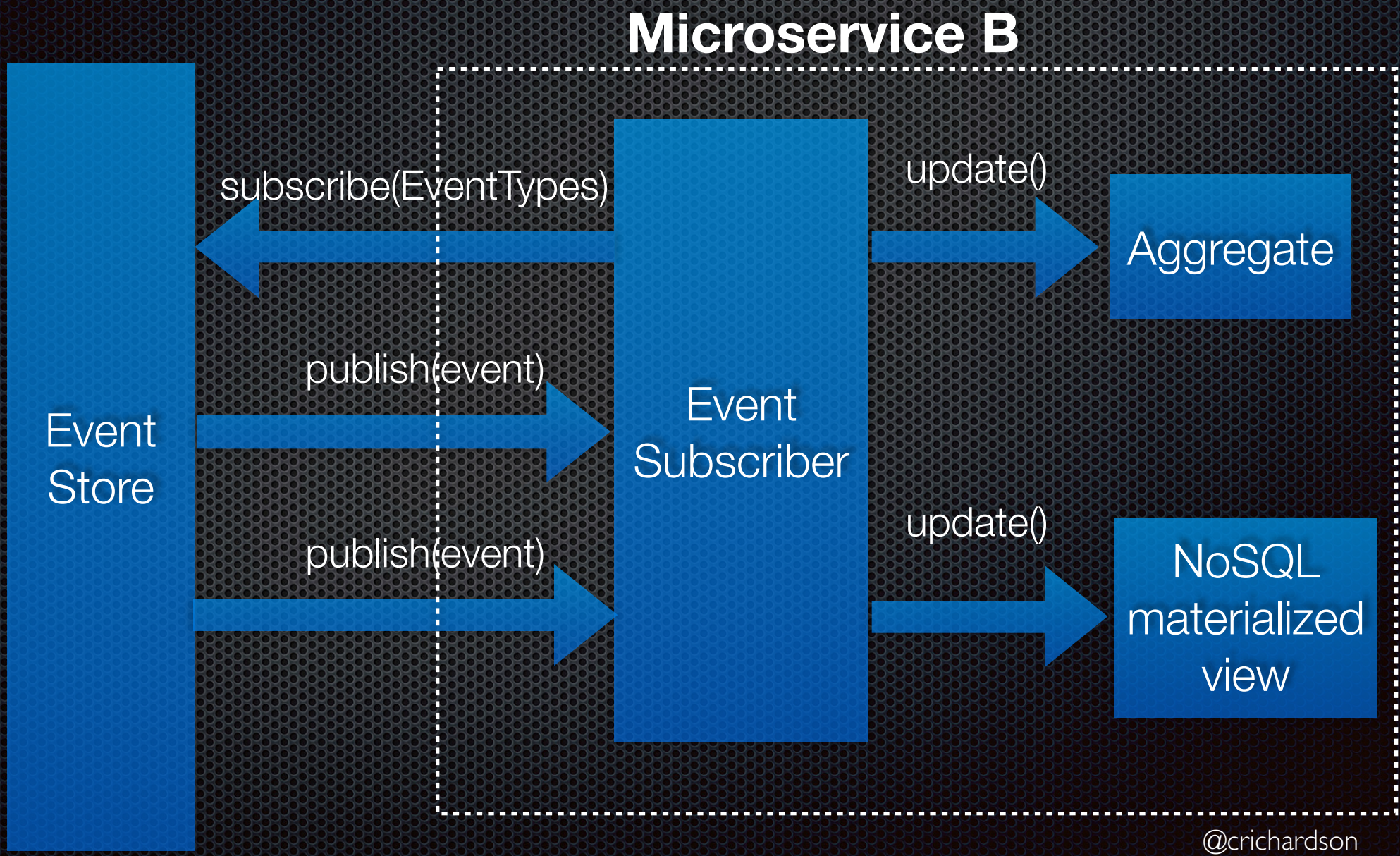
Immutable

```
case class Account(balance : BigDecimal)
  extends CommandProcessingAggregate[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

  }
```

Request handling in an event-sourced application



Event Store publishes events - consumed by other services



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

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]
}
```

Event store implementations

- Home-grown/DIY
- geteventstore.com by Greg Young
- Talk to me about my project :-)

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

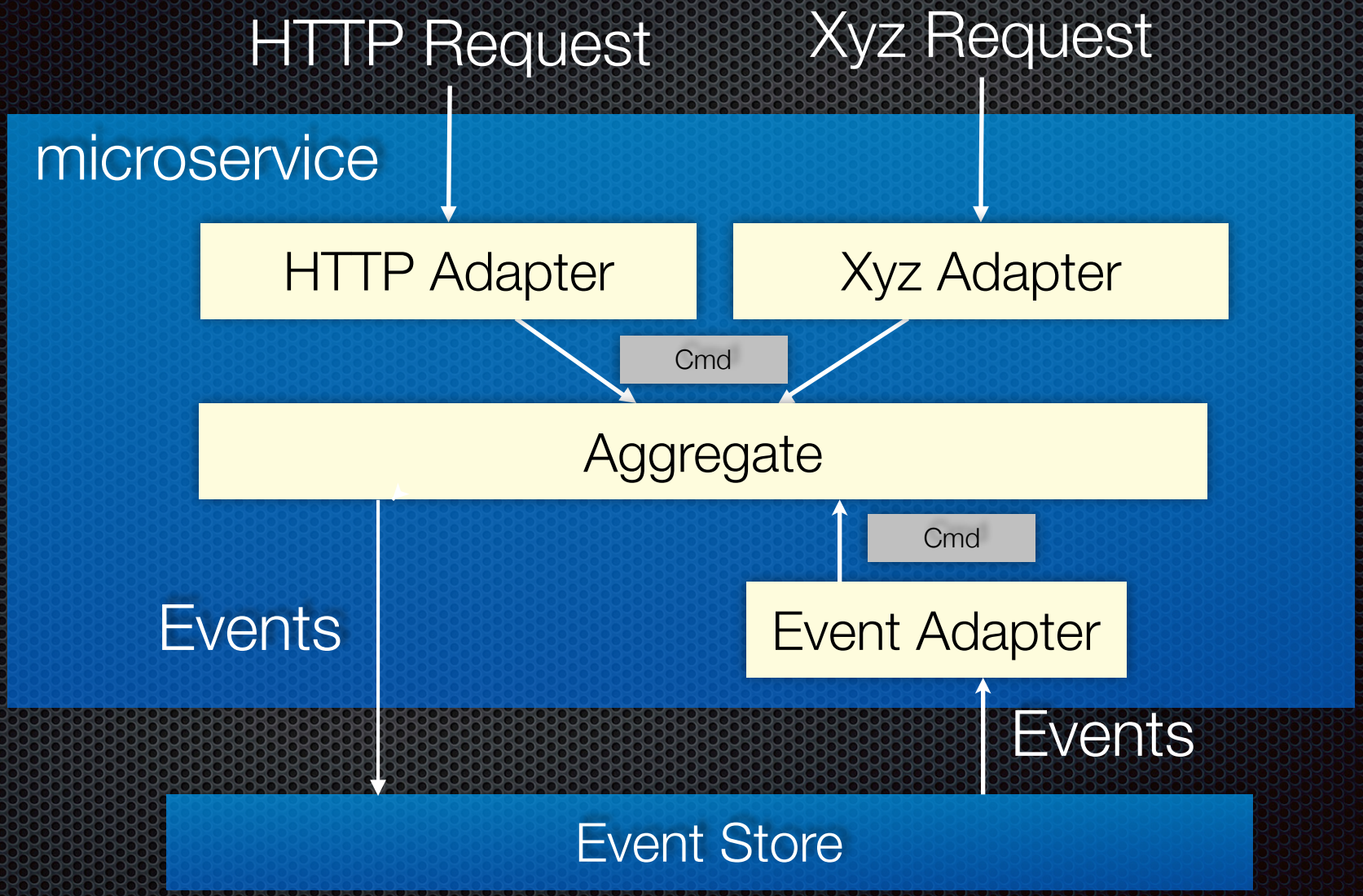
Drawbacks of event sourcing

- 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
- Building and deploying microservices

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)
  }
```

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

MoneyTransfer Aggregate

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

case class MoneyTransfer(state : TransferStates.State, details : TransferDetails)
  extends CommandProcessingAggregate[MoneyTransfer, MoneyTransferCommand] {

  def this() = this(TransferStates.NEW, null)

  import net.chrisrichardson.eventstore.examples.bank.transactions.MoneyTransferCommands._

  def processCommand = {...}

  def applyEvent = {...}

}
```

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

Handling events published by Accounts

```
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
- Building and deploying microservices

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, ...)

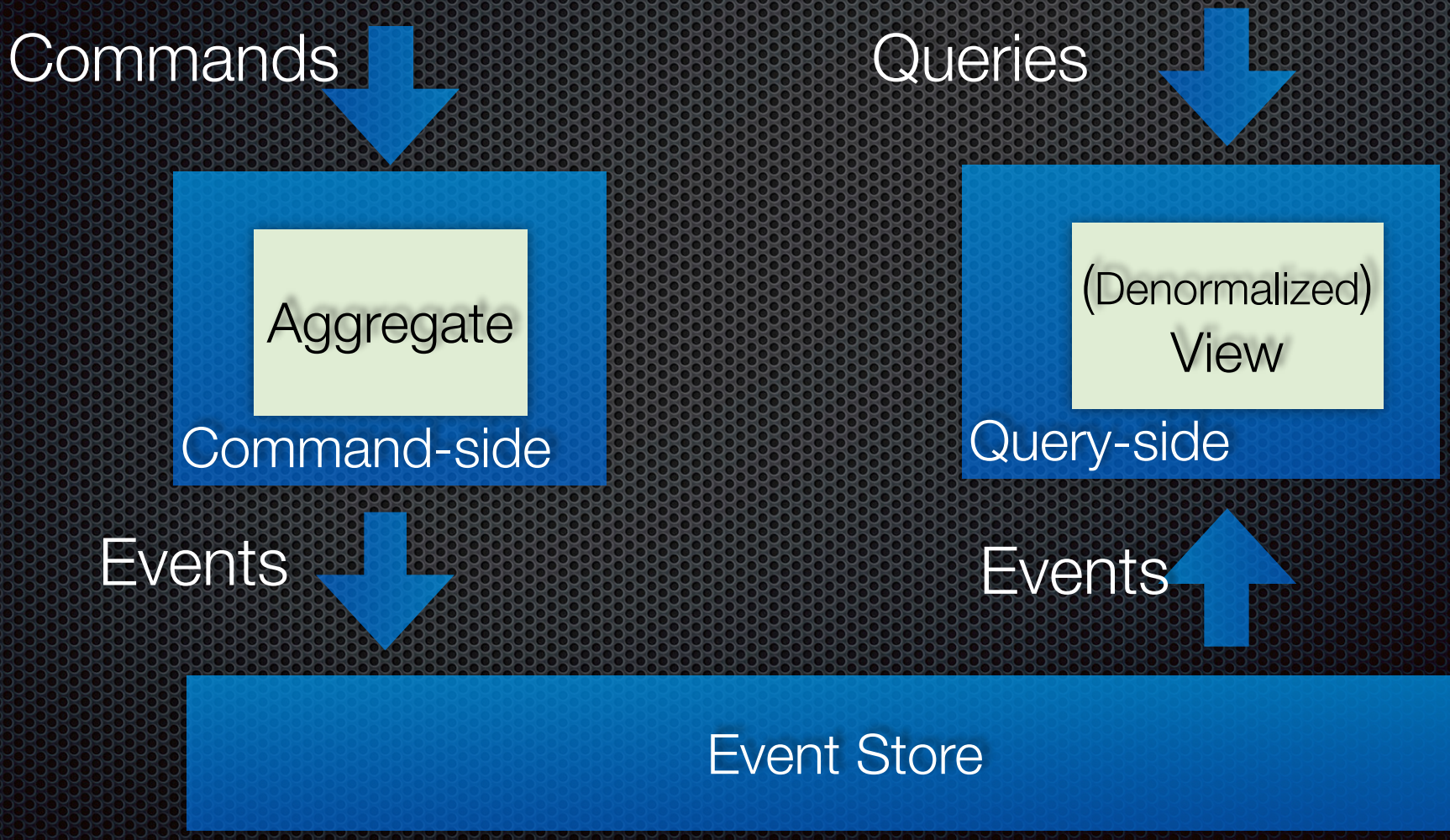
BUT

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

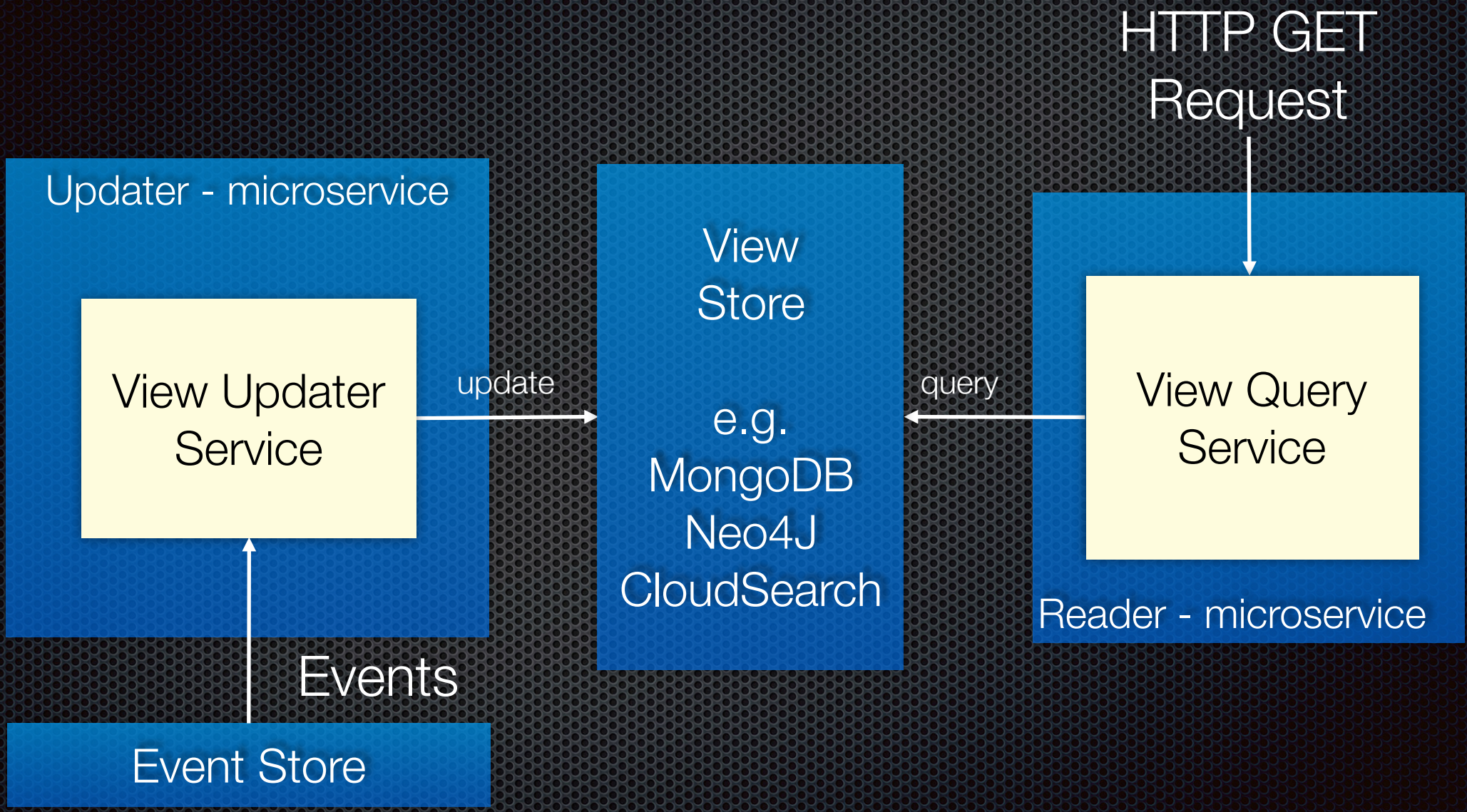
⇒

- 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]) = …


}
```

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

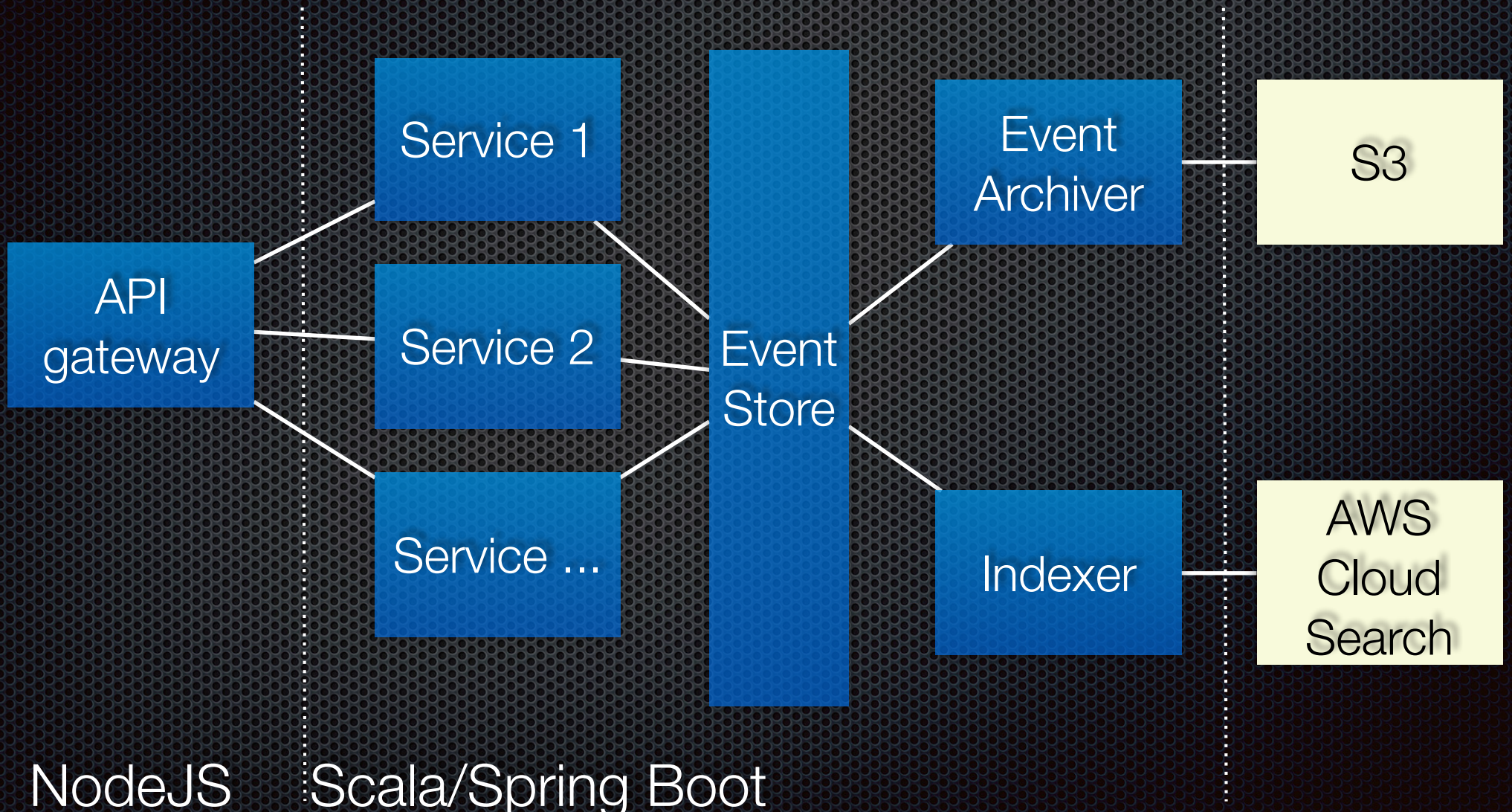
Drawbacks

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

Agenda

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

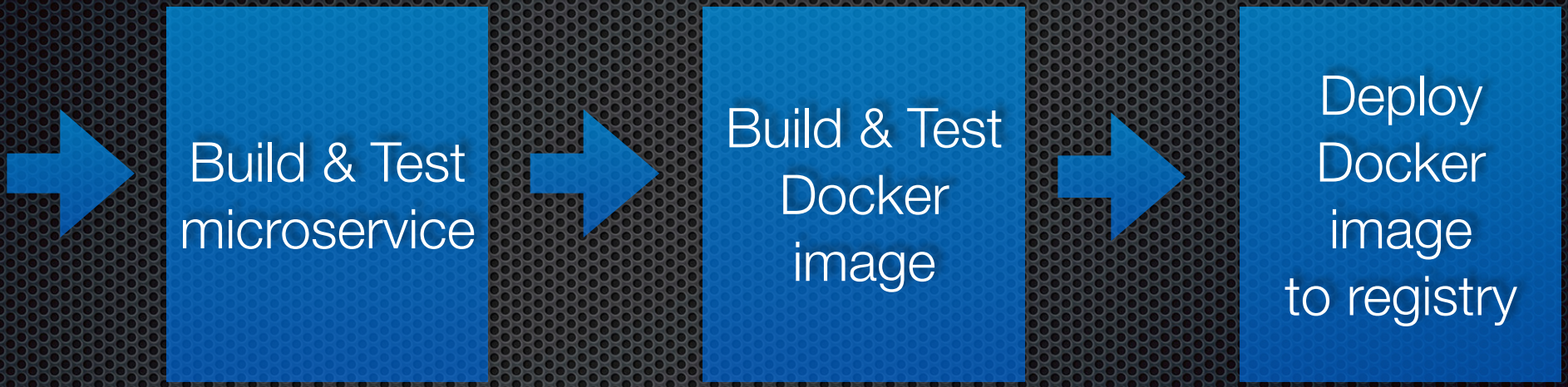
My application architecture



Spring Boot based micro services

- Makes it easy to create stand-alone, production ready microservices
- Automatically configures Spring using Convention over Configuration
- Externalizes configuration
- Built-in health checks and (Codahale) metrics
- Generates standalone executable JARs with embedded web server

Jenkins-based deployment pipeline



One pipeline per microservice

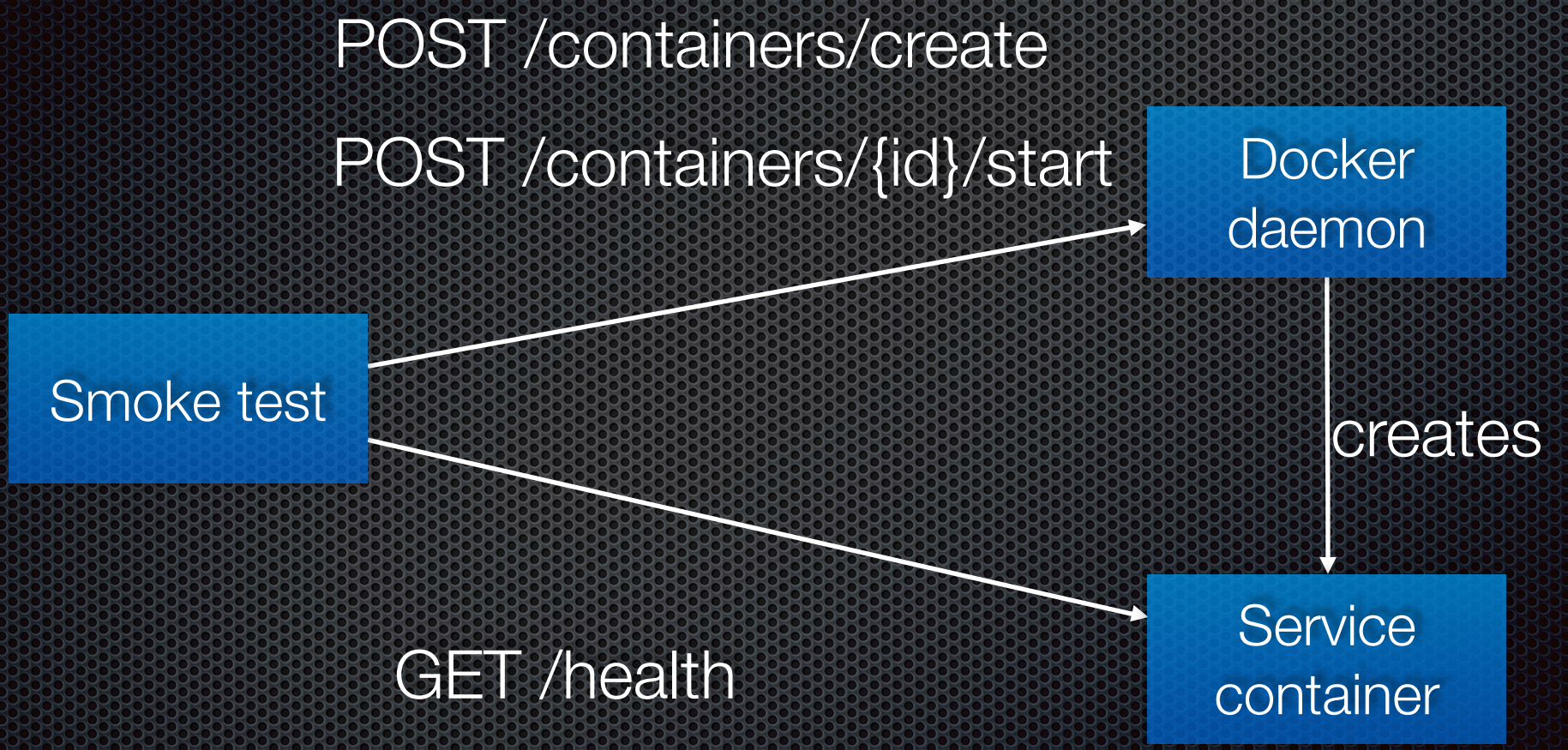
Building Docker images

docker/build.sh

```
cp ../build/libs/service.${1}.jar build/service.jar


docker build -t service-${VERSION} .
```

Building only takes 5 seconds!

Smoke testing docker images



Docker daemon must listen on TCP port

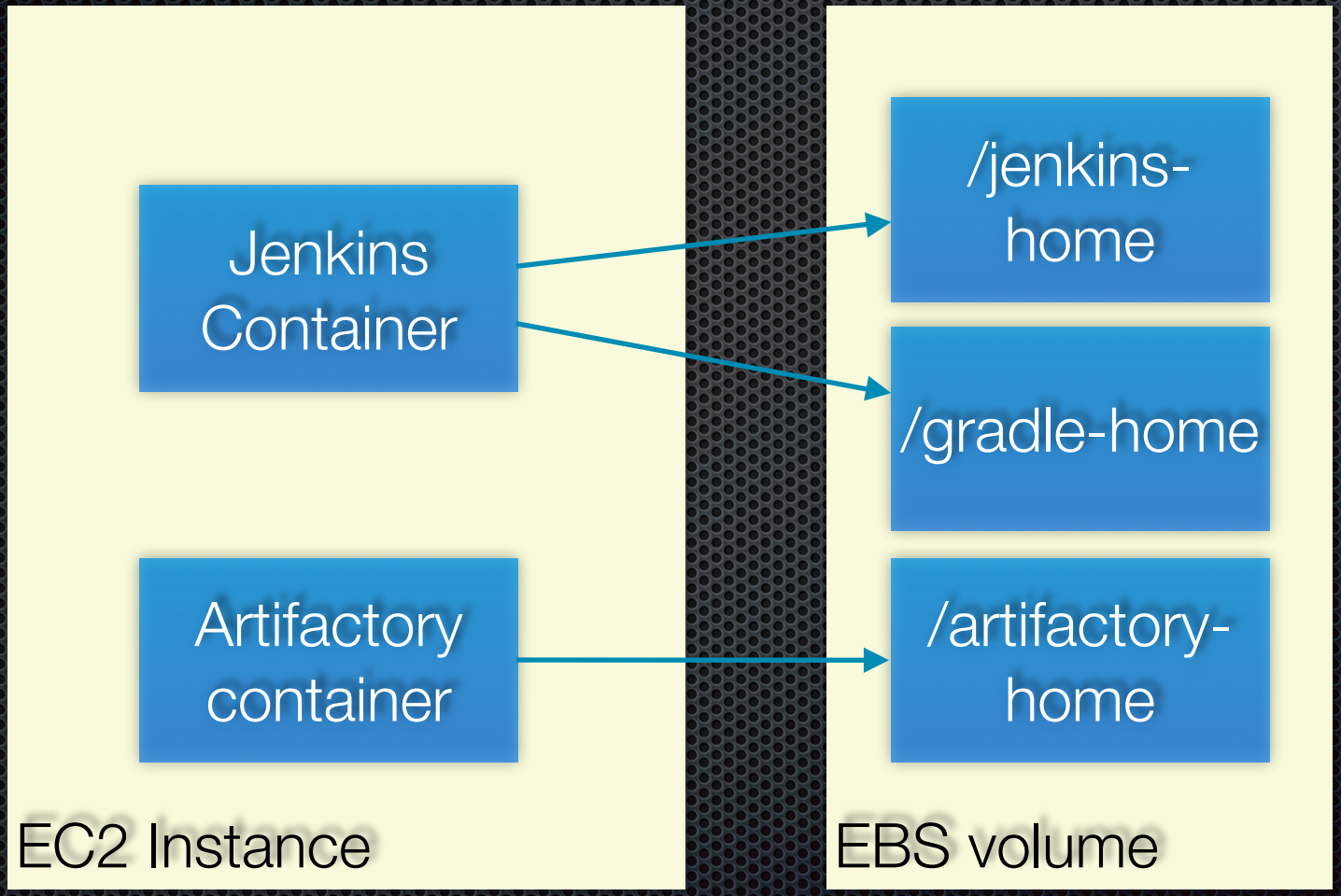
Publishing Docker images

docker/publish.sh

```
docker tag service-${VERSION}:latest \
    ${REGISTRY_HOST_AND_PORT}/service-${VERSION}


docker push ${REGISTRY_HOST_AND_PORT}/service-${VERSION}
```

Pushing only takes 25 seconds!

CI environment runs on Docker



Updating the production environment

- Large EC2 instance running Docker
- Deployment tool:
    1. Compares running containers with what's been built by Jenkins
    2. Pulls latest images from Docker registry
    3. Stops old versions
    4. Launches new versions
- One day: use Docker clustering solution and a service discovery mechanism,
    - Most likely, AWS container service
    - Mesos and Marathon + Zookeeper, Kubernetes or ???

Summary

- Event sourcing solves key data consistency issues with:
  - Microservices
  - Partitioned SQL/NoSQL databases
- Use CQRS to implement materialized views for queries
- Spring Boot is a great foundation for microservices
- Docker is a great way to package microservices

@crichardson chris@chrisrichardson.net

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