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
Presentation goal

Share my experiences with building and deploying an application using Scala, functional domain models, microservices, event sourcing, CQRS, and Docker
About Chris
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
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 are building a banking app...
Domain model

Account

balance

open(initialBalance)
debit(amount)
credit(amount)

MoneyTransfer

fromAccountId
toAccountId

amount
Traditional application architecture

Browser/Client → Load balancer → WAR/EAR → Tomcat → RDBMS

- HTML
- REST/JSON

- Banking UI
- Accounts
- Transfers
- Customers
- ...

Spring MVC

Spring Hibernate

ACID

Simple to develop, test, deploy, scale
Problem #1: 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
Solution #1: use a microservice architecture

- Banking UI
- Account Management Service
  - Account Database
- MoneyTransfer Management Service
  - MoneyTransfer Database

Standalone services
Problem #2: relational databases

- Scalability
- Distribution
- Schema updates
- O/R impedance mismatch
- Handling semi-structured data
Solution #2: use NoSQL databases

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

@crichardson
Different modules use different databases
But now we have problems with data consistency!
Problem #3: Microservices = distributed data management

- Each microservice has its own database
- Business transactions must update data owned by multiple services,
  - e.g. Update MoneyTransfer and from/to Accounts
- Some data is replicated and must be kept in sync
- Tricky to implement reliably without 2PC
Problem #4: NoSQL = ACID-free, denormalized databases

- Limited transactions, i.e. no ACID transactions
  - Tricky to implement business transactions that update multiple rows,
  - e.g. Update MoneyTransfer and from/to Accounts
- Limited querying capabilities
  - Requires denormalized/materialized views that must be synchronized
  - Multiple datastores (e.g. DynamoDB + Cloud Search) that need to be kept in sync
Solution to #3/#4: Event-based architecture to the rescue

- Microservices publish events when state changes
- Microservices subscribe to events
  - Maintains eventual consistency across multiple aggregates (in multiple datastores)
  - Synchronize replicated data
Eventually consistent money transfer

```
MoneyTransferService

MoneyTransfer
fromAccountId = 101
fromAccountId = 202
amount = 55
state = INITIAL

AccountService

Account
id = 101
balance = 250

Account
id = 202
balance = 125

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

Message Bus

publishes:
MoneyTransferCreatedEvent
DebitRecordedEvent

Message Bus

publishes:
MoneyTransferCreatedEvent
DebitRecordedEvent

AccountDebitedEvent
AccountCreditedEvent

Subscribes to:
AccountDebitedEvent
AccountCreditedEvent

Subscribes to:
MoneyTransferCreatedEvent
DebitRecordedEvent

AccountDebitedEvent
AccountCreditedEvent
```

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

Account
id = 101
balance = 195

Account
id = 202
balance = 180
```
To maintain consistency a service must **atomically** publish an event **whenever** a domain object changes.
How to reliably generate events whenever state changes?

- Database triggers, Hibernate event listener, ...
  - Reliable BUT
  - Not with NoSQL
  - Disconnected from the business level event
  - Limited applicability

- Ad hoc event publishing code mixed into business logic
  - Publishes business level events BUT
  - Tangled code, poor separation of concerns
  - Unreliable, e.g. too easy to forget to publish an event
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
  - **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
- balance
- open(initial)
- debit(amount)
- credit(amount)

Event table
- 101 901 AccountOpened 500
- 101 902 AccountCredited 250
- 101 903 AccountDebited 300

Account table
- 101 901
- 101 902
- 101 903
- 101 904
- 101 905
Replay events to recreate state

Events
- AccountOpenedEvent(balance)
- AccountDebitedEvent(amount)
- AccountCreditedEvent(amount)
Before: update state + publish events

Two actions that must be atomic

Now: persist (and publish) events

Single action that can be done atomically
Aggregate traits

Apply event returning updated
Aggregate

Map Command to Events

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]]
}
Account - command processing

```scala
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))
    }
```
Account - applying events

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

Microservice A

- pastEvents = findEvents(entityId)
- new()
- applyEvents(pastEvents)
- newEvents = processCmd(SomeCmd)
- saveEvents(newEvents)
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
Persisting events

- Ideally use a cross platform format
- Use weak serialization:
  - enables event evolution, eg. add memo field to transfer
  - missing field → provide default value
  - unknown field → ignore
- JSON is a good choice
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

```scala
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]
}
```
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)
Example of an eventual consistency problem

- Scenario:
  1. Create the user
  2. Create shopping cart
  3. Update the user with the shopping cart's id
- The user temporarily does not have a shopping cart id!
  - Client might need to retry their request at a later point
  - Server should return status code 418??
Handling duplicate events

- Idempotent operations
  - e.g. add item to shopping cart
- Duplicate detection:
  - e.g. track most recently seen event and discard earlier ones
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

```java
@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

DSL concisely specifies:
1. Creates MoneyTransfer aggregate
2. Processes command
3. Applies events
4. Persists events

```scala
class MoneyTransferService(implicit eventStore : EventStore) {
  def transferMoney(transferDetails : TransferDetails) =
    newEntity[MoneyTransfer] <= CreateMoneyTransferCommand(transferDetails)
}
```
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 Debit RecordedEvent(details : TransferDetails) extends Event
case class Credit RecordedEvent(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 **Command Query Responsibility Separation**
Command Query Responsibility Separation (CQRS)

- **Commands**
  - Aggregate
  - Events

- **Queries**
  - (Denormalized) View
  - Events

- **Event Store**
Query-side microservices

Updater - microservice

View Updater Service

Events

Event Store

View Store
e.g. MongoDB
Neo4J
CloudSearch

View Query Service

Reader - microservice

HTTP GET Request

update

query
Persisting account balance and recent transactions in MongoDB

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

- **Current balance**
- **Transfers that update the account**
- **The sequence of debits and credits**

*Denormalized = efficient lookup*
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
Dealing with eventually consistent views

- **Scenario:**
  - Client creates/updates aggregate
  - Client requests view of aggregate
- **Problem:**
  - The view might not yet have been updated
- **Solution:**
  - Create/Update response contains aggregate version
  - Query request contains desired version
  - Out of date view ⇒ wait or return “out of date view” error code
- Alternatively:
  - “Fake it” in the UI until the view is updated
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

API gateway

Service 1

Service 2

Service ...

Event Store

Event Archiver

Indexer

S3

AWS Cloud Search

NodeJS

Scala/Spring Boot
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}} .
Smoke testing docker images

Docker daemon must listen on TCP port

- POST /containers/create
- POST /containers/{id}/start
- GET /health

Docker daemon creates Service container
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}
CI environment runs on Docker

EC2 Instance

- Jenkins Container
- Artifactory container

EBS volume

- /jenkins-home
- /gradle-home
- /artifactory-home
Updating 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,
  - Mesos and Marathon + Zookeeper, Kubernetes or ???

@crichardson
Summary

- Event sourcing solves key data consistency issues with:
  - Microservices
  - Partitioned SQL/NoSQL databases
- Use CQRS to implement materialized views for queries
- Docker is a great way to package microservices
Questions? Let’s talk at the Open Space

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