The Autobiography of Building a Reactive Application

Duncan K. DeVore

@ironfish

ScalaDays 2014

Kudos

- Typesafe
- The Legend of Klang
- Martin Krasser
- Greg Young
- Sean Walsh

About Viridity

- Commercial, Industrials, data centers, universities, etc.
- Help customers manage:
  - Forecasting & Optimization
  - Renewables & storage
  - Controllable load
  - Energy assets

About VPower

- Suite of applications
- Distributed & cloud based
- Micro service architecture
- Reactive: Event-driven, responsive, resilient, scalable
- Transform energy profiles into financial returns
- Production May 2013 - "It just works"

How We Got To Where We Are Today

VPower 1.x

- Monolithic
- Java, Spring, Hibernate, Some(Scala)
- Postgres
- SQL Server
- Problems (deployment, responsiveness, scalability, etc.)
- The coming tidal wave - meter data

There Must Be a Better Way!

What We Wanted

- Scala & Akka
- Modular/Distributed
- Loosely coupled
- Scalable
- Fault tolerant
- Responsive
- Immutable domain model
- Schema-less data model

The Road We Chose

Reactive by Design

- Domain Driven Design
- **CQRS**
- **Eventual Consistency**
- **Event Sourcing**
- Schema-less
- Micro-service based
- Headless via Rest

The Tools We Chose

- Scala
- Akka
- Eventsourced/Akka-Persistence
- Spray.io
- Mongo
- Angular.js
- D3.js

Domain Driven Design

What is Domain Driven Design?

- Tackling Complexity in the Heart of Software by Eric Evans.
- For developing **complex** software.
- Connects implementation to an **evolving** model.
- Not a technology or methodology.
- A structure of practices and terminology.
- Domain Model, Ubiquitous Language, Model Driven Design.

Domain Driven Design

- Patterns of **distribution** emerged from DDD
- Simpler aggregates (es)
- Events conceptually part of aggregates (es)
- No distributed transactions (es)
- No two phased commits (es, cqrs & ec)

CQRS

What is CQRS?

Command Query Responsibility Segregation

Origins from CQS

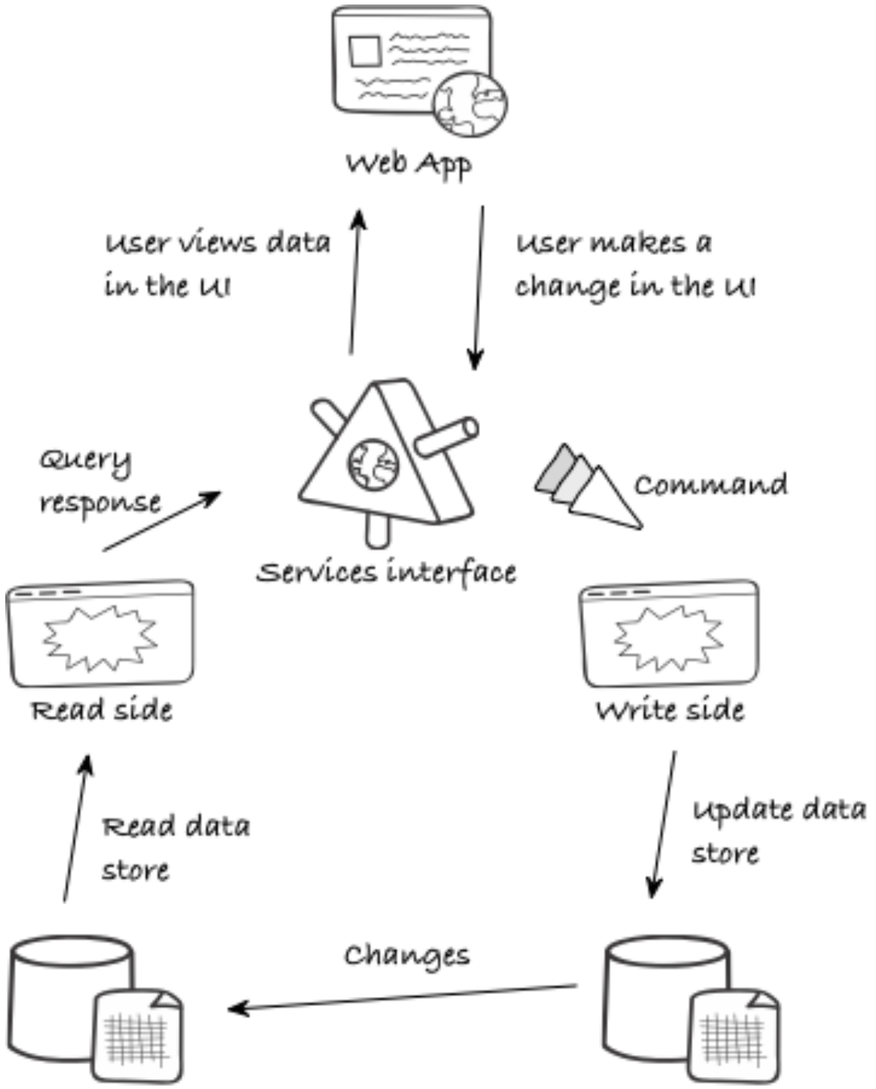
- Command Query Separation
- *Object Oriented Software Construction* by Bertrand Meyer.
- Methods should be either **commands** or **queries**.
- A query **returns** data, does **not** alter the state.
- A command **changes** the state, does **not** return data.
- Becomes clear what **does** and **does not** change state

A Step Further

CQRS takes this principle a step further to define a simple pattern.

"CQRS is simply the **creation** of **two objects** where there was previously only one. The separation occurs based upon whether the methods are a command or a query (the same definition that is used by Meyer in Command and Query Separation: a command is any method **(object)** that mutates state and a query is any method **(object)** that returns a value)"

— *Greg Young*



Two Distinct Paths

- One for writes (commands)
- One for reads (queries)
- Allows **separate** optimization
- **Simpler** reasoning about paths

Reason for Segregation

- Large imbalance between the number of reads and writes
- Single model encapsulating reads/writes **does neither** well
- Command side often involves **complex** business logic
- Read side **de-normalized** (redundant) for fast queries
- More atomic and easier to reason about
- Read side easily **re-creatable**

CQRS Commands

Behavior

On the command side its all about **behavior** rather than data centricity. This leads to a more true implementation of DDD.

Commands are a **request** of the system to perform a **task** or **action**. A sample command would be:

- `RegisterClient`
- `ChangeClientLocale`

Commands

- Commands are **imperative**
- They are a request to **mutate state**
- They represent an action the client **would like** to take
- They transfer in the form of **messages** rather than DTOs
- Implies a tasked-based UX

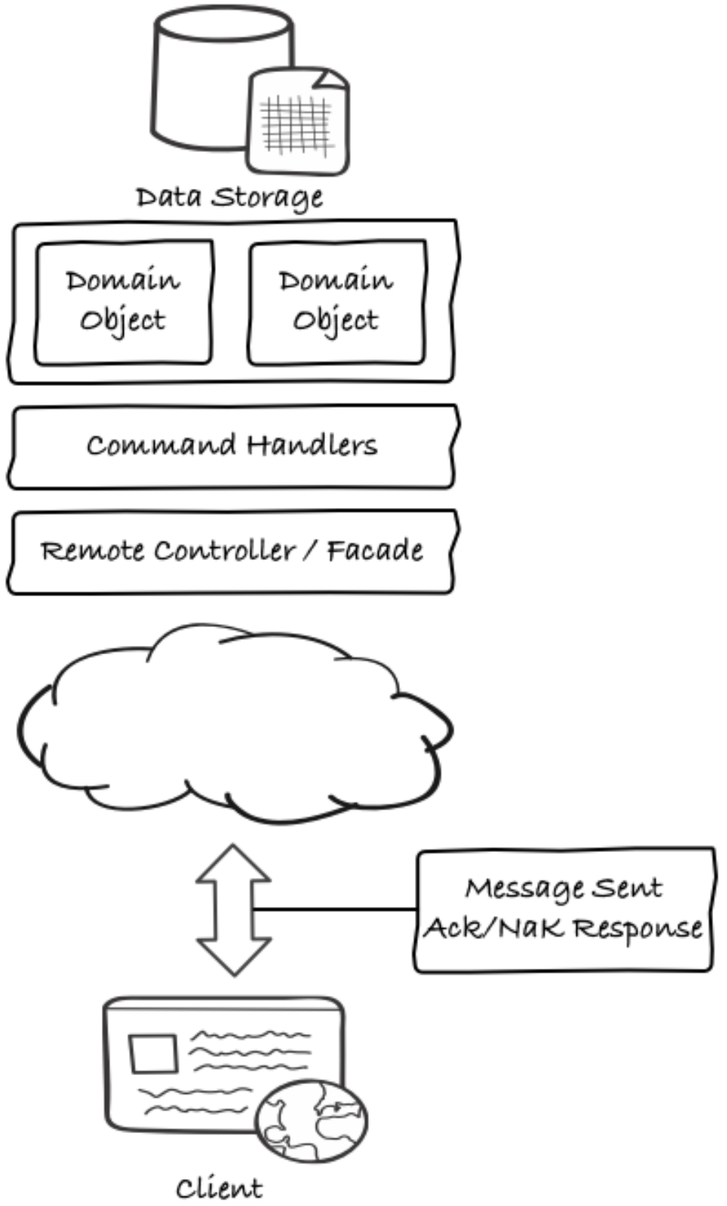
Commands

- Conceptually **not** editing data, rather performing a task
- Can be thought of as serializable method calls
- The command handler can say **NO**
- Internal state not exposed
- Your repository layer is greatly simplified

Command Handler

In CQRS command handlers are the objects that process commands.

- Client sends a command in the form of a message
- That message will be processed by a command handler
- Commands can be rejected
- Turned into one or more events that are persisted



Command Handler

```
class ExampleProcessor extends PersistentActor {
  var state = ExampleState() # <--- mutable state, but NOT shared = OK!

  def updateState(event: Evt): Unit =
    state = state.update(event)

  ...

}
```

Command Handler

```
class ExampleProcessor extends PersistentActor {
  ...

  val receiveRecover: Receive = { # <=== process persisted events on boostrap
    case evt: Evt                                 => updateState(evt)
    case SnapshotOffer(_, snapshot: ExampleState) => state = snapshot
  }

  ...
}
```

Command Handler

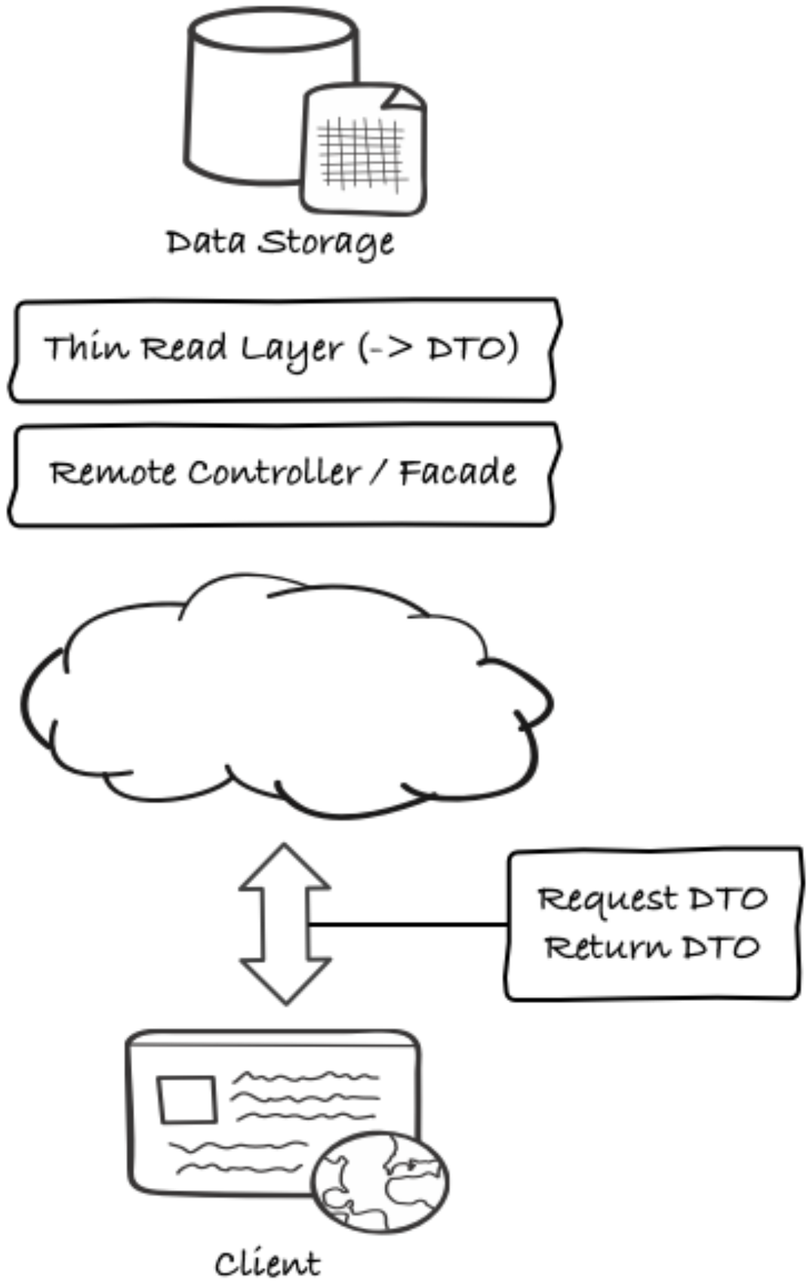
```
class ExampleProcessor extends PersistentActor {
  ...

  val receiveCommand: Receive = { # <=== process commands, if valid persist events
    case Cmd(data) =>
      persist(Evt(s"{data}")) { event =>
        updateState(event)
        context.system.eventStream.publish(event)
      }
    ...
  }
}
```

CQRS Queries

CRUD = Pain

- DTOs projected off domain
- Aggregate getters **expose** internal state
- DTOs **different** model than domain
- Usually require extensive mapping
- Large # of read methods on repos
- **Optimization** of queries becomes hard
- Queries object **!=** to data model
- Object model translated to data model
- Impedance mismatch



Thin Read Layer

- CQRS applies a **natural** boundary
- DTO's **no longer** project off domain
- Reads from the data store
- Read side projects the DTOs
- No need for a complex ORM
- Data stored/fetched **screen** structure
- No more Impedance Mismatch!
- Easier to optimize & faster
- No more **looping** to construct view

What is Eventual Consistency?

By applying CQRS, the concepts of Writes and Reads have been **separated**. If we keep the paths segregated, how do we keep them **consistent**?

Eventual Consistency

- Eventual Consistency
- Business **determines** how long between sync
- Pushed **asynchronously** from the write side
- Read side has listeners
- Queue can be used
- Two phased commits **not needed**
- Use the event store as your queue

Eventually Consistent Read Side

```
class BenefitsView extends View {
  import EmployeeProtocol._
  import BenefitsProtocol._

  ...

  override def processorId = "employee-processor" # <=== identifies the processor
  override def viewId = "benefits-view"

  def receive = {
    case p @ Persistent(payload, _) =>
      payload match {
        case evt: EmployeeHired =>
          val eb = BenefitDates(evt.id, evt.startDate, Nil, Nil, Nil)
  ...
      }
  }
}
```

Eventually Consistent Read Side

```
class BenefitsView extends View {
  import EmployeeProtocol._
  import BenefitsProtocol._

  ...

  override def processorId = "employee-processor" # <=== identifies the processor
  override def viewId = "benefits-view"

  def receive = {
    case p @ Persistent(payload, _) =>
      payload match {
        case evt: EmployeeHired =>
          val eb = BenefitDates(evt.id, evt.startDate, Nil, Nil, Nil)
  ...
      }
  }
}

akka.persistence.view.auto-update-interval = 5s # <=== update intervals are configurable
```

Event Sourcing

What is Event Sourcing?

The **majority** of business applications today rely on storing **current state** in order to process transactions. As a result in order to track history or implement audit capabilities **additional** coding or frameworks are required.

This Was Not Always the Case

- Side-effect of the adoption of **RDBMS** systems
- High performance, mission critical systems **do not** do this
- RDBMS's do not do this **internally**!
- SCADA (System Control and Data Acquisition) Systems

It's About Capturing Events

- Its **behavioral** by nature
- Tracks behavior by transactions
- It **does not** persist current state
- Current state is **derived**

CRUD Shopping Cart

1. Cart created
2. Item 1 @ $30 added
3. Item 2 @ $15 added
4. Item 3 @ $12 added
5. Item 4 @ $5 added
6. Shipping information added
7. Total @ $62 generated
8. Order 123 inserted

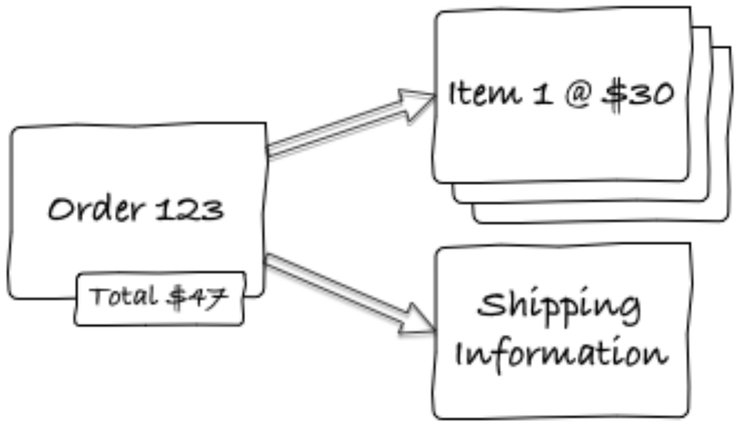
CRUD Shopping Cart

Now at some time in the future **before** the order is shipped, the customer changes their mind and wants to **delete** an item.

1. Order 123 fetched
2. Item 2 @ $15 removed
3. Total @ $47 regenerated
4. Order 123 updated

CRUD Shopping Cart

This is the **current state** persisted



The result of these transactions the current state of the order is 3 items with a total of $47

CRUD Shopping Cart

Now the **manager** ask the development team to give him a **report** of all orders where customers have **removed** items. Since only the **current state** of the data is recorded this cant be done.

- The development team will **add** in a future sprint
- Once added it will only work from **that point forward**
- **Substantial** implications to the value of the data

ORMS & Static State Models

- Work well in most situations, come at a fairly **large cost**
- Query and persist **current state** to database
- Tightly couple domain and data model
- Can lead to leaky abstraction
- Can lead to an anemic domain model
- Lossy and the **intent of the user** is not captured

Static State Models

Consider for a moment the notion of a transaction:

- Represent **change** between two points
- Commonly referred to as Deltas
- In static state models Deltas are **implicit**
- They are left to frameworks such as an ORM
- ORMs save state, calculate differences, update backing model
- As a result much of the **intent** or **behavior** is lost

Tracking Behavior with Events

In a typical CRUD application the **behavior** of the system is create, read, update and delete. This is **not** the only way the data can be viewed.

The Canonical Example

In **mature** business models the notion of tracking **behavior** is very **common**. Consider for example an accounting system.

Date	Comment	Change	Balance
1/1/2012	Deposit from 3300	+10000.00	10000.00
1/3/2012	Check 1	-4000.00	6000.00
1/4/2012	ATM withdraw	-3.00	5997.00
1/11/2012	Check 2	-5.00	5992.00
1/12/2012	Deposit from 3301	+2000.00	7992.00

The Canonical Example

- **Each** transaction or delta is being **recorded**
- Next to it is a de-normalized total of the state of the account
- To calculate, the delta is applied to the **last known value**
- The last known value can be **trusted**
- State is **recreated** by replaying all the transactions (events)

The Canonical Example

- Its can be **reconciled** to ensure **validity**
- The data itself is a **verifiable audit log**
- The Current Balance at **any** point can be **derived**
- **State** can be derived for **any** point in time

Events

- Events are **notifications**
- They report on something that has **already** happened
- As such, events cannot be **rejected**
- An event would be something like:
  - `ClientRegistered`
  - `ClientLocaleChanged`

Reactive Shopping Cart

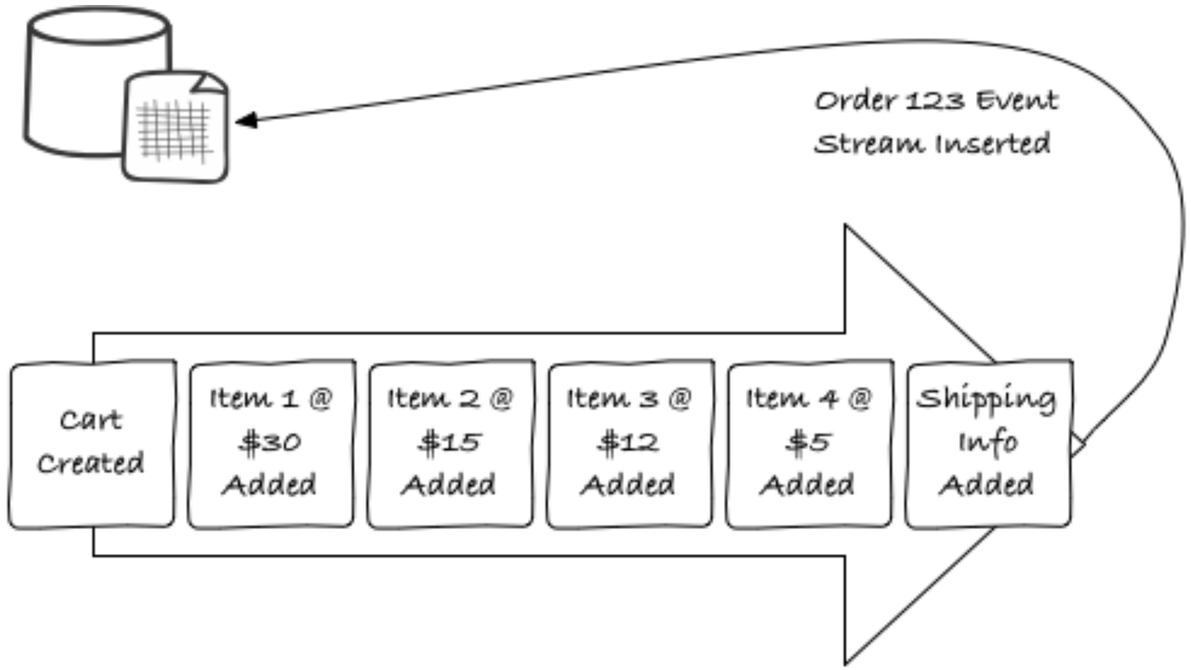
Reactive Shopping Cart

Lets go back and take a look at Shopping Cart example and see how we manage the data from an event based perspective.

Reactive Shopping Cart

1. Cart created
2. Item 1 @ $30 added
3. Item 2 @ $15 added
4. Item 3 @ $12 added
5. Item 4 @ $5 added
6. Shipping information added
7. Order 123 event stream inserted

Reactive Shopping Cart



Reactive Shopping Cart

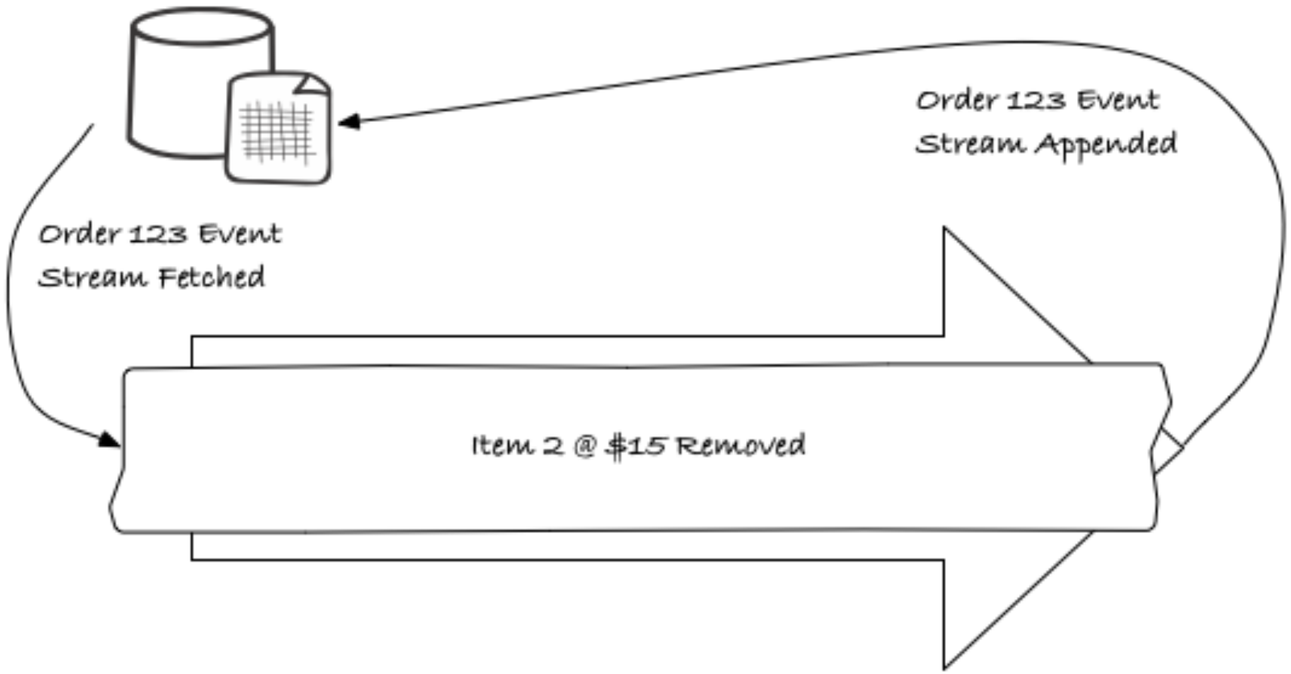
Now at some time in the future **before** the order is shipped, the customer changes their mind and wants to **delete** an item.

1. Order 123 event stream fetched

2. Item 2 @ $15 removed event

3. Order 123 event stream appended

Reactive Shopping Cart



Reactive Shopping Cart

By replaying the event stream the object can be returned to the **last known** state.

- There is a structural representation of the object
- It exists by **replaying** previous transactions
- Data is **not** persisted structurally
- It is a **series** of transactions
- **No coupling** between current state in the domain and storage

No CRUD Except Create & Read

- There are no **updates** or **deletes**
- **Everything** is persisted as an event
- Its stored in **append only** fashion
- Delete & update are simply **events** that gets appended

Technology Implications

- The storage system becomes an **additive only** architecture
- Append-only architectures **distribute**
- Far **fewer** locks to deal with
- Horizontal Partitioning is difficult for a relational model
  - What **key** do you partition on in a complex relational model?
- When using an Event Store there is only **1 key**!

Business Implications

- Criteria is **tracked** from inception as an event stream
- You can answer questions form the **origin** of the system
- You can answer questions **not asked yet**!
- Natural audit log

Tying It All Together with CQRS/ES

- Event-driven by nature
- Natural boundaries for isolation & partitioning for scale
- Baked in recovery for resilience
- Read side optimized for responsiveness

Tying It All Together with CQRS/ES

I believe that CQRS and Event Sourcing when combined, provide a clear and concise way to build distributed applications that adhere to the reactive manifesto.

— *me*

Coming Soon, Summer 2014…

Building Reactive Applications

Duncan DeVore and Sean Walsh

Manning Publication, Co.

The End