Lessons Learned in Deploying Akka Persistence

Duncan K. DeVore

@ironfish

Akka Days 2014

The Challenge

- Turn energy expenditure into energy revenue
- Commercial, Industrials, data centers, universities, etc.
- Help customers manage
  - Forecasting & Optimization
  - Renewables & storage
  - Controllable load
  - Energy assets

The Solution Version 1

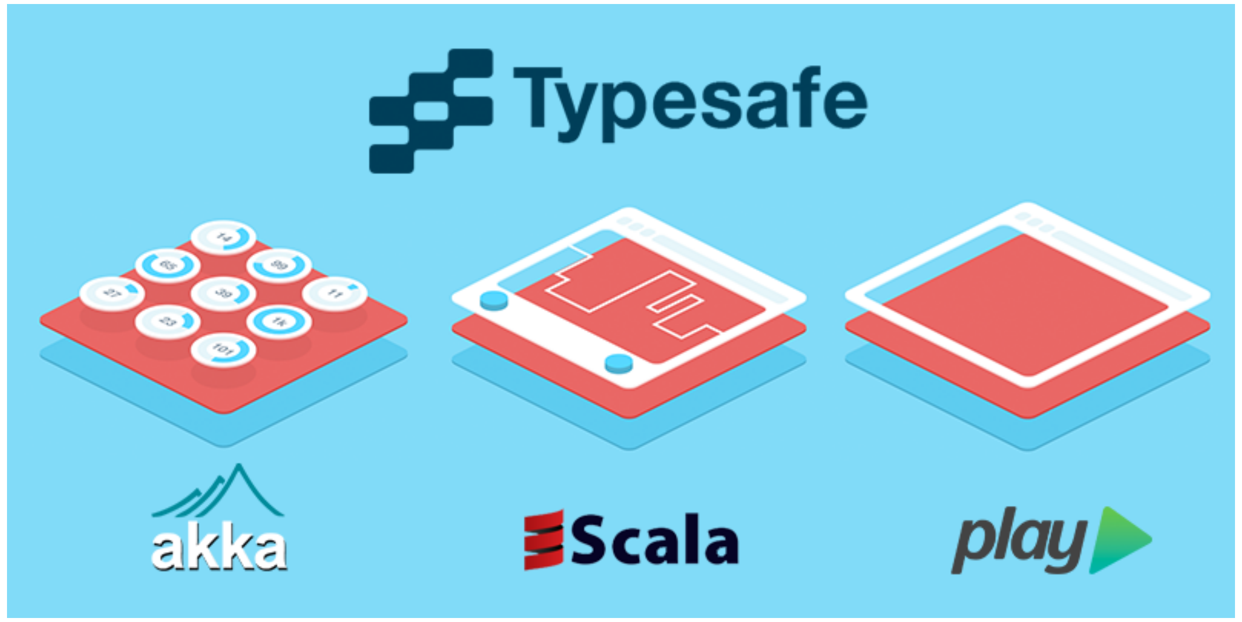
- Monolithic
- Java, Spring, Hibernate, Some(Scala)
- Postgres, SQL Server
- Problems (deployment, responsiveness, scalability, etc.)
- Explicit auditing
- The coming tidal wave - meter data

There Must Be a Better Way!

Thinking Cap Time

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

Vendor Time



mongoDB

Solution Version 2

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

Our Toolkit

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

The 3 Musketeers

- **Event Sourcing**
- **CQRS**
- **Eventual Consistency**

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**

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
7/1/2014	Deposit from 3300	+ 10,000.00	10,000.00
7/3/2014	Check 001	- 4,000.00	6,000.00
7/4/2014	ATM Withdrawal	- 3.00	5,997.00
7/11/2014	Check 002	- 5.00	5,992.00
7/12/2014	Deposit from 3301	+ 2,000.00	7,992.00

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:
    - `OrderCreated`
    - `ItemAdded`

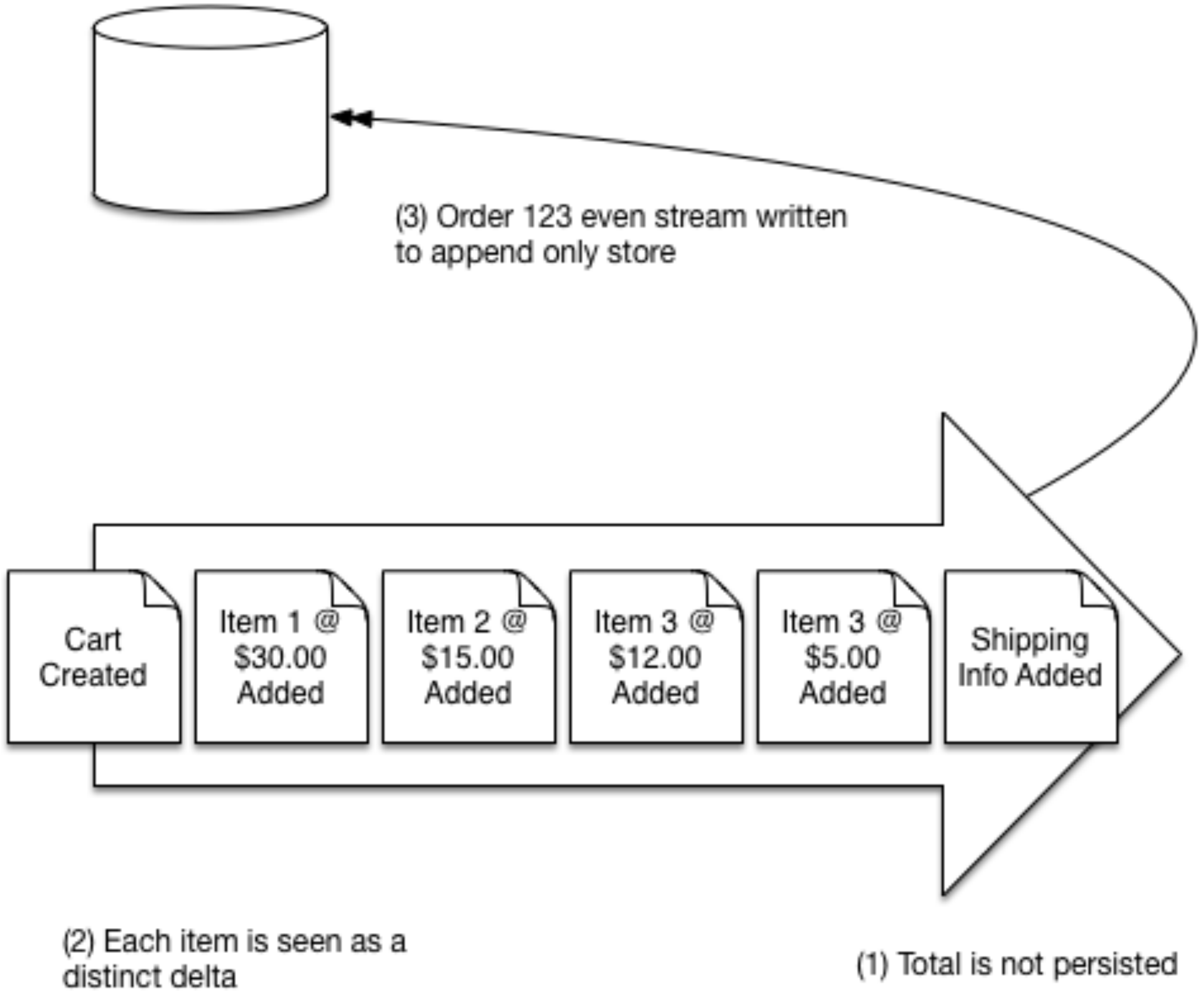
Reactive Shopping Cart

Lets 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

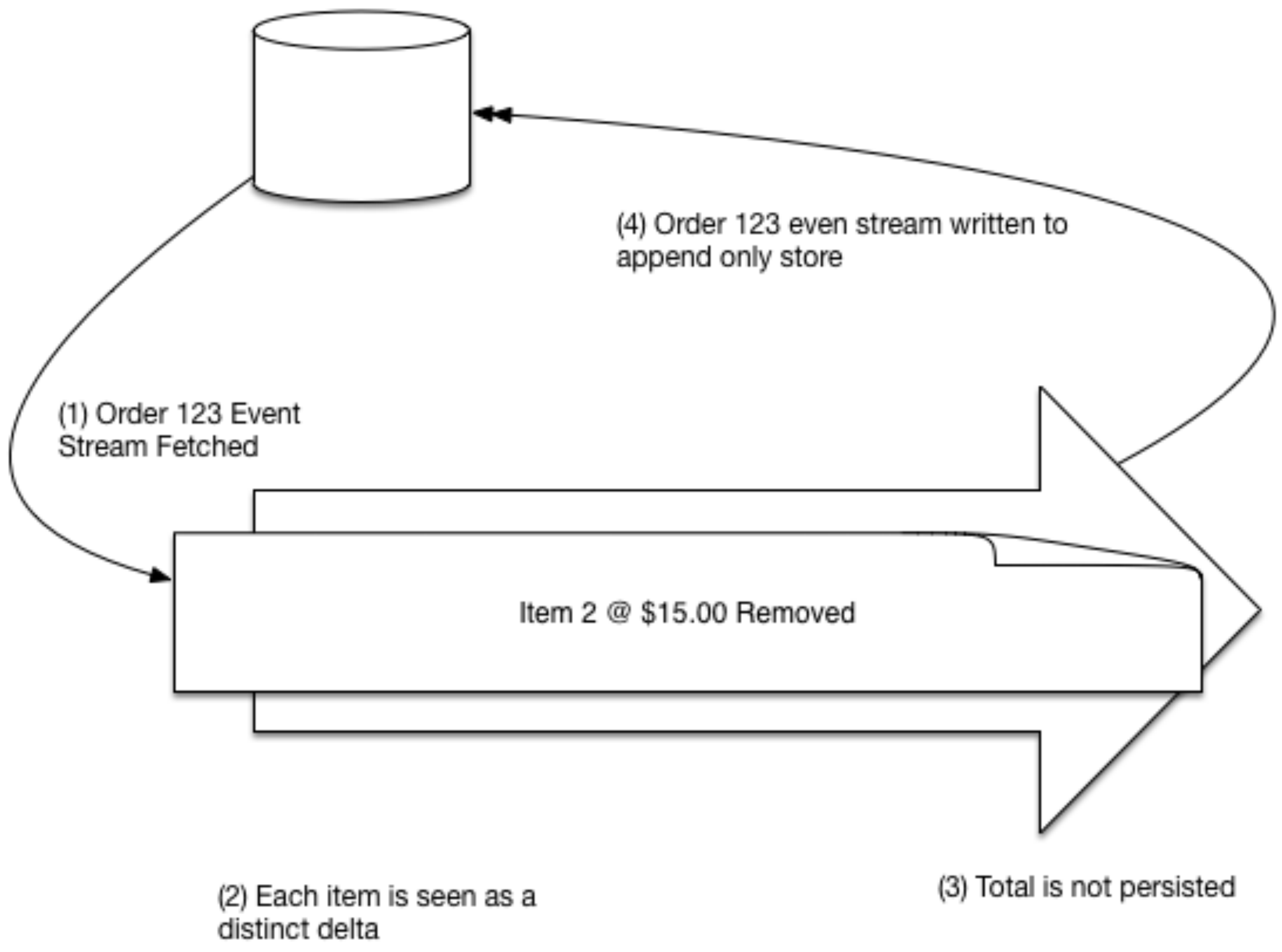


Reactive Shopping Cart

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

CQRS

What is CQRS?

Command Query Responsibility Segregation

Origins from CQS

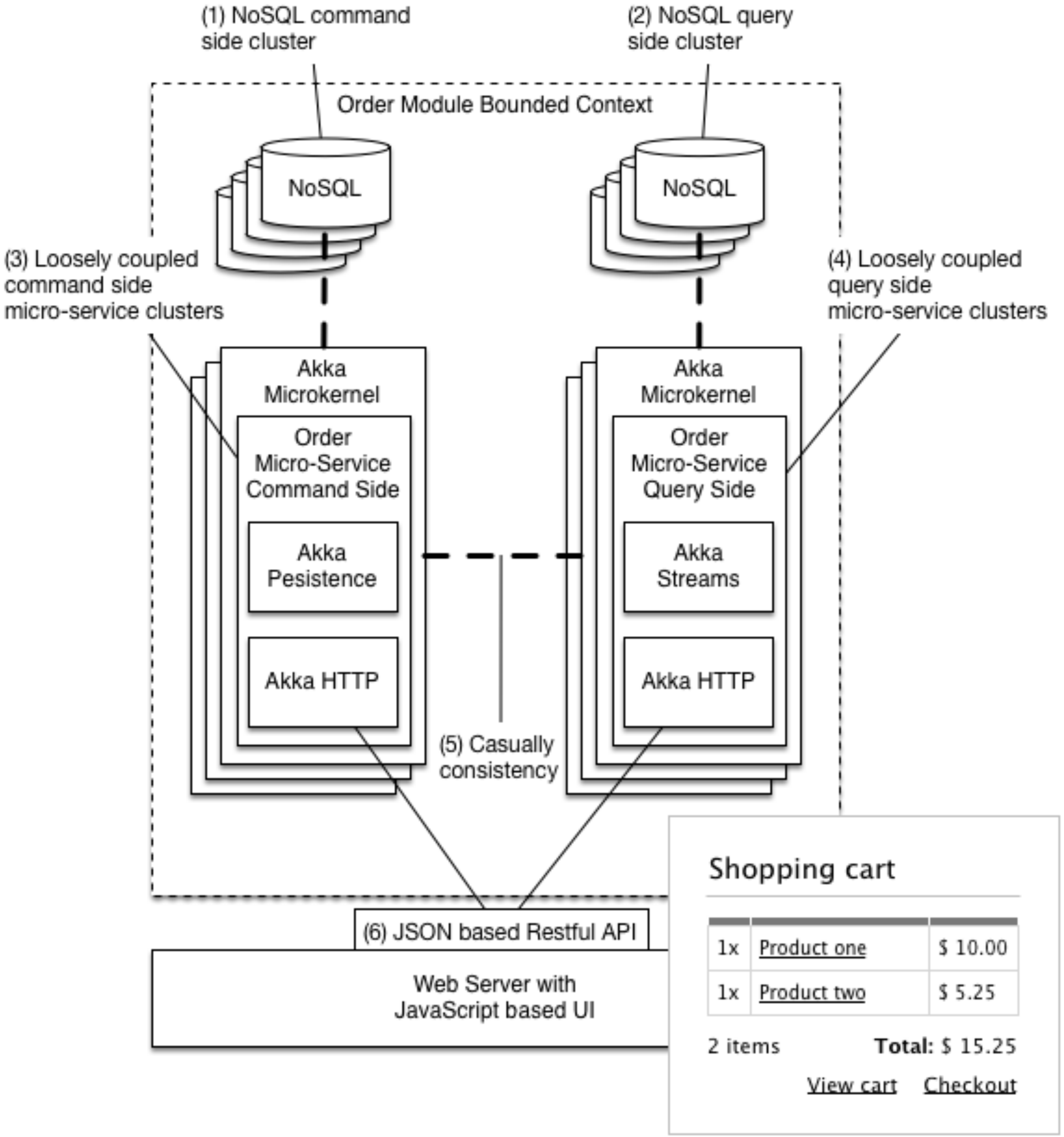
- Command Query Separation
- *Object Oriented Software Construction* by Bertrand Meyer.
- Methods should be either **commands** or **queries**.
- A query **returns** data, does **not** alter the state.
- A command **changes** the state, does **not** return data.
- Becomes clear what **does** and **does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

Akka Persistence

What is Akka Persistence?

"Akka persistence enables stateful actors to persist their internal state so that it can be recovered when an actor is started, restarted after a JVM crash or by a supervisor, or migrated in a cluster."

— *Akka Documentation*

But Wait, It's More!

- Provides the **C** in CQRS
- Provides the **E** in event sourcing
- Provides the **Q** in CQRS
- Provides the **R** in Resilience

Akka Persistence = ES + C

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:

- `CreateOrder`
- `AddItem`

Akka Persistence = ES + C

- 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

Akka Persistence = ES + C

- Conceptually **not** editing data, rather performing a task
- Can be thought of as serializable method calls
- Commands can be **REJECTED**
- Internal state not exposed
- Your repository layer is greatly simplified

Akka Persistence = ES + C&E

In Akka Persistence a `PersistentActor` processes commands & persist events.

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

Akka Persistence = ES + C&E

```scala
case class CreateShoppingCart(...) extends Command
case class ShoppingCartCreated(...) extends Event

final case class ShoppingCart(...) { // <--- immutable aggregate instance

private case class State(shoppingCart: ShoppingCart) {

  def update(e: Event): State = evt match {
    case ShoppingCartCreated(sc) => copy(shoppingCart = sc)
  ...
  }
}

class ShoppingCart extends PersistentActor with ActorLogging {
  override def persistenceId: String = self.path.parent.name + "-" + self.path.name
  var state = ShoppingCart(...)
  ...
}
```

Akka Persistence = ES + C&E

```scala
case class CreateShoppingCart(...) extends Command
case class ShoppingCartCreated(...) extends Event

final case class ShoppingCart(...) { // <--- immutable aggregate instance

private case class State(shoppingCart: ShoppingCart) {

  def update(e: Event): State = evt match {
    case ShoppingCartCreated(sc) => copy(shoppingCart = sc) // <--- immutable state object defensive copy
  ...
  }
}

class ShoppingCart extends PersistentActor with ActorLogging {
  override def persistenceId: String = self.path.parent.name + "-" + self.path.name
  var state = ShoppingCart(...)
  ...
}
```

Akka Persistence = ES + C&E

```
case class CreateShoppingCart(...) extends Command
case class ShoppingCartCreated(...) extends Event

final case class ShoppingCart(...) { // <--- immutable instance

private case class State(shoppingCart: ShoppingCart) { // <--- immutable state object defensive copy

  def update(e: Event): State = evt match {
    case ShoppingCartCreated(sc) => copy(shoppingCart = sc)
  ...
  }
}

class ShoppingCart extends PersistentActor with ActorLogging { // <--- aggregate root is an actor!
  override def persistenceId: String = self.path.parent.name + "-" + self.path.name
  var state = ShoppingCart(...)
  ...
}
```

Akka Persistence = ES + C&E

```scala
case class CreateShoppingCart(...) extends Command
case class ShoppingCartCreated(...) extends Event

final case class ShoppingCart(...) { // <--- immutable instance

private case class State(shoppingCart: ShoppingCart) { // <--- immutable state object defensive copy

  def update(e: Event): State = evt match {
    case ShoppingCartCreated(sc) => copy(shoppingCart = sc)
  ...
  }
}

class ShoppingCart extends PersistentActor with ActorLogging { // <--- aggregate root is an actor!
  override def persistenceId: String = self.path.parent.name + "-" + self.path.name
  var state = ShoppingCart(...) // <--- mutable state, but NOT shared = OK!
  ...
}
```

Akka Persistence = ES + C&E

```
class ShoppingCart extends PersistentActor with ActorLogging {
  ...
  override def receiveCommand: Receive = initial

  def initial: Receive = {
    case cmd: CreateShoppingCart(...) => create(cmd) fold ( // <--- command received and validated
      f => sender ! s"Error $f occurred on $cmd",
      s => persist(ShoppingCartCreated(...)) { evt =>
        state = state.updated(evt)
        context.become(created)
    })
  }
}
```

Akka Persistence = ES + C&E

```scala
class ShoppingCart extends PersistentActor with ActorLogging {
  ...
  override def receiveCommand: Receive = initial

  def initial: Receive = {
    case cmd: CreateShoppingCart(...) => create(cmd) fold ( // <--- command received and validated
      f => sender ! s"Error $f occurred on $cmd", // <--- on failure return to sender
      s => persist(ShoppingCartCreated(...)) { evt =>
        state = state.updated(evt)
        context.become(created)
    })
  }
}
```

Akka Persistence = ES + C&E

```
class ShoppingCart extends PersistentActor with ActorLogging {
  ...
  override def receiveCommand: Receive = initial

  def initial: Receive = {
    case cmd: CreateShoppingCart(...) => create(cmd) fold ( // <--- command received and validated
      f => sender ! s"Error $f occurred on $cmd", // <--- on failure return to sender
      s => persist(ShoppingCartCreated(...)) { evt => // <--- on success event persisted
        state = state.updated(evt)
        context.become(created)
    })
  }
}
```

Akka Persistence = ES + C&E

```scala
class ShoppingCart extends PersistentActor with ActorLogging {
  ...
  override def receiveCommand: Receive = initial

  def initial: Receive = {
    case cmd: CreateShoppingCart(...) => create(cmd) fold ( // <--- command received and validated
      f => sender ! s"Error $f occurred on $cmd", // <--- on failure return to sender
      s => persist(ShoppingCartCreated(...)) { evt => // <--- on success event persisted
        state = state.updated(evt) // <--- partial function updates state
        context.become(created)
    })
  }
}
```

Akka Persistence = ES + C&E

```scala
class ShoppingCart extends PersistentActor with ActorLogging {
  ...
  override def receiveCommand: Receive = initial

  def initial: Receive = {
    case cmd: CreateShoppingCart(...) => create(cmd) fold ( // <--- command received and validated
      f => sender ! s"Error $f occurred on $cmd", // <--- on failure return to sender
      s => persist(ShoppingCartCreated(...)) { evt => // <--- on success event persisted
        state = state.updated(evt) // <--- partial function updates state
        context.become(created) // <--- actor's contect becomes created
    })
  }
}
```

Akka Persistence = ES + Q

In Akka Persistence the `PersistentView` provides a means to manage eventual consistency

- Associated with a `PersistentActor`
- The `PersistentActor` does have to be running.
- The `PersistentView` reads form the journal directly.
- Update intervals for the view are configurable
- Supports recovery & snapshots

Akka Persistence = ES + Q

```scala
class ShoppingCartView extends PersistentView {

  override def persistenceId: String = "some-persistent-actor-id" // <--- ties the view to the persistent actor
  override def viewId = "some-persistent-actor-view-id"

  def receive = {
    case "snap" =>
      saveSnapshot(...)
    case SnapshotOffer(metadata, snapshot: Int) =>
      ...
    case payload if isPersistent =>
      payload match {
        case evt: ShoppingCartCreated =>
          ...
      }
    case payload =>
      ...
  }
}
```

Akka Persistence = ES + Q

```scala
class ShoppingCartView extends PersistentView {

  override def persistenceId: String = "some-persistent-actor-id" // <--- ties the view to the persistent actor
  override def viewId = "some-persistent-actor-view-id"

  def receive = {
    case "snap" => // <--- supports snapshots
      saveSnapshot(...)
    case SnapshotOffer(metadata, snapshot: Int) =>
      ...
    case payload if isPersistent =>
      payload match {
        case evt: ShoppingCartCreated =>
          ...
      }
    case payload =>
      ...
  }
}
```

Akka Persistence = ES + Q

```
class ShoppingCartView extends PersistentView {

  override def persistenceId: String = "some-persistent-actor-id" // <--- ties the view to the persistent actor
  override def viewId = "some-persistent-actor-view-id"

  def receive = {
    case "snap" => // <--- supports snapshots
      saveSnapshot(...)
    case SnapshotOffer(metadata, snapshot: Int) => // <--- recovers from snapshot
      ...
    case payload if isPersistent =>
      payload match {
        case evt: ShoppingCartCreated =>
          ...
      }
    case payload =>
      ...
  }
}
```

Akka Persistence = ES + Q

```
class ShoppingCartView extends PersistentView {

  override def persistenceId: String = "some-persistent-actor-id" // <--- ties the view to the persistent actor
  override def viewId = "some-persistent-actor-view-id"

  def receive = {
    case "snap" => // <--- supports snapshots
      saveSnapshot(...)
    case SnapshotOffer(metadata, snapshot: Int) => // <--- recovers from snapshot
      ...
    case payload if isPersistent => // <--- update query side store from journal
      payload match {
        case evt: ShoppingCartCreated =>
          ...
      }
    case payload =>
      ...
  }
}
```

Akka Persistence = ES + Q

```
class ShoppingCartView extends PersistentView {

  override def persistenceId: String = "some-persistent-actor-id" // <--- ties the view to the persistent actor
  override def viewId = "some-persistent-actor-view-id"

  def receive = {
    case "snap" => // <--- supports snapshots
      saveSnapshot(...)
    case SnapshotOffer(metadata, snapshot: Int) => // <--- recovers from snapshot
      ...
    case payload if isPersistent => // <--- update query side store from journal
      payload match {
        case evt: ShoppingCartCreated =>
          ...
      }
    case payload => // <--- process non-journaled messages
      ...
  }
}
```

Akka Persistence = ES + Q

```
class ShoppingCartView extends PersistentView {

  override def persistenceId: String = "some-persistent-actor-id" // <--- ties the view to the persistent actor
  override def viewId = "some-persistent-actor-view-id"

  def receive = {
    case "snap" => // <--- supports snapshots
      saveSnapshot(...)
    case SnapshotOffer(metadata, snapshot: Int) => // <--- recovers from snapshot
      ...
    case payload if isPersistent => // <--- update query side store from journal
      payload match {
        case evt: ShoppingCartCreated =>
          ...
      }
    case payload => // <--- process non-journaled messages
      ...
  }
}

akka.persistence.view.auto-update-interval = 5s // <--- eventual consistency configurable
```

Akka Persistence = Resilience

- Isolates failure
- Failover via clustering
- State is recoverable
- On both sides of the CQRS fence!

Akka Persistence = Resilience

```scala
class ShoppingCart extends PersistentActor with ActorLogging {
  ...
  override def receiveRecover: Receive = { // <--- process persisted events on bootstrap
    case evt: ShoppingCartCreated =>
      context.become(created)
      state = state.updated(evt)
    case SnapshotOffer(_, snapshot: ExampleState) => state = snapshot
  ...
}
```

Akka Persistence = Resilience

```
class ShoppingCart extends PersistentActor with ActorLogging {
  ...
  override def receiveRecover: Receive = { // <--- process persisted events on bootstrap
    case evt: ShoppingCartCreated =>
      context.become(created)
      state = state.updated(evt)
    case SnapshotOffer(_, snapshot: ExampleState) => state = snapshot // <--- snapshots supported!
  ...
}
```

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

The End

Thank You!