DDDing with Akka Persistence

Konrad `@ktosopl` Malawski
Konrad `@ktosopl` Malawski
Konrad `@ktosopl` Malawski

typesafe.com
ggeecon.org
Java.pl / KrakowScala.pl
sckrk.com / meetup.com/Paper-Cup @ London
GDGKrakow.pl
meetup.com/Lambda-Lounge-Krakow
Show of hands!
Show of hands!
Show of hands!
Show of hands!
<table>
<thead>
<tr>
<th><strong>Command Sourcing</strong></th>
<th><strong>Event Sourcing</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>msg: <strong>DoThing</strong></td>
<td></td>
</tr>
<tr>
<td>msg persisted before receive</td>
<td></td>
</tr>
<tr>
<td>imperative, “do the thing”</td>
<td></td>
</tr>
<tr>
<td>business logic change, can be reflected in <em>reaction</em></td>
<td></td>
</tr>
<tr>
<td>no validation before persisting</td>
<td></td>
</tr>
</tbody>
</table>
## Command Sourcing vs. Event Sourcing

<table>
<thead>
<tr>
<th>Command Sourcing</th>
<th>Event Sourcing</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg: <strong>DoThing</strong></td>
<td>msg: <strong>ThingDone</strong></td>
</tr>
<tr>
<td>msg persisted before receive</td>
<td>commands converted to events, must be manually persisted</td>
</tr>
<tr>
<td>imperative, “do the thing”</td>
<td>past tense, “happened”</td>
</tr>
<tr>
<td>business logic change, can be reflected in reaction</td>
<td>business logic change, won’t change previous events</td>
</tr>
<tr>
<td>Processor</td>
<td>validate, then maybe persist</td>
</tr>
</tbody>
</table>
Compared to “Good ol’ CRUD Model”

- **Mutable Record**
  - State
  - “Series of Events”

- **Series of Events**
  - State = apply(es)
  - “Mutable Record”
Actors
An **Actor** that keeps **count** of **messages** it processed

Let’s send 2 **messages** to it (it’s “commands”)
class Counter extends Actor {
    var count = 0

    def receive = {
        case _ => count += 1
    }
}
Actor

count: 0
Actor

count: 0
Actor

count: 1

crash!
Actor

Crash!
Actor

restart
Acto

count: 0

restart
Actor

count: 0

restarted
Actor

count: 1

restarted

wrong!
expected count == 2!
Consistency Boundary
Consistency Boundary equals Async Boundary
Boundaries

 aggregate

“eventual”

aggregate

A.K.A. async
Any rule that spans AGGREGATES will not be expected to be up-to-date at all times. Through event processing, batch processing, or other update mechanisms, other dependencies can be resolved within some specific time. [Evans, p. 128]
Let’s open the toolbox
PersistentActor
PersistentActor

Replaces:
Processor & Eventsourced Processor

in Akka 2.3.4+
super quick domain modelling!

Commands - what others “tell” us; not persisted

sealed trait Command
case class GiveMe(geeCoins: Int) extends Command
case class TakeMy(geeCoins: Int) extends Command

Events - reflect effects, past tense; persisted

sealed trait Event
case class BalanceChangedBy(geeCoins: Int) extends Event

State - reflection of a series of events

case class Wallet(geeCoins: Int) {
  def updated(diff: Int) = State(geeCoins + diff)
}
PersistentActor

var state = S0
persistenceId = “a”
PersistentActor

var state = S0
persistenceId = “a”
PersistentActor

```
var state = S0
persistenceId = "a"
```

Generate Events

Journal
PersistantActor

var state = S0
persistenceld = “a”

Journal

ACK “persisted”
PersistentActor

var state = S0
persistenceId = “a”

“Apply” event

Journal
PersistentActor

```java
var state = S0
persistenceId = "a"
```

Okey!
PersistentActor

```scala
var state = S0
persistenceId = "a"
```

Okey!
PersistentActor

Ok, he got my $.

var state = S0
persistenceId = “a”
class BitCoinWallet extends PersistentActor {

    var state = Wallet(coins = 0)

    def updateState(e: Event): State = {
        case BalanceChangedBy(coins) => state.updatedWith(coins)
    }

    // API:

    def receiveCommand = ??? // TODO

    def receiveRecover = ??? // TODO
}
class BitCoinWallet extends PersistentActor {

   var state = Wallet(coins = 0)

   def updateState(e: Event): State = {
      case BalanceChangedBy(coins) => state.updatedWith(coins)
   }

   // API:

   def receiveCommand = ??? // TODO

   def receiveRecover = ??? // TODO

}
persist(e) { e => }

def receiveCommand = {

    case TakeMy(coins) =>
        persist(BalanceChangedBy(coins)) { changed =>
            state = updateState(changed)
        }

}
def receiveCommand = {

  case GiveMe(coins) if coins <= state.coins =>
    persist(BalanceChangedBy(-coins)) {
      changed =>
        state = updateState(changed)
        sender() ! TakeMy(coins)
    }

  }
PersistentActor

def receiveCommand = {

  case GiveMe(coins) if coins <= state.coins =>
    persist(BalanceChangedBy(-coins)) { changed =>
      state = updateState(changed)
      sender() ! TakeMy(coins)
    }
}

Safe to access sender here
persist(){} – Ordering guarantees

var state = S0
persistenceId = "a"

Journal
Persist(){} – Ordering guarantees

Commands get “stashed” until processing C1’s events are acted upon.

```javascript
var state = S0
persistenceId = “a”
```
var state = S0
persistenceId = “a”

events get applied in-order
var state = S0
persistenceId = “a”

and the cycle repeats

persist(){} – Ordering guarantees
persistAsync(e) { e => }
persistAsync(e) { e => }
+
defer(e) { e => }
def receiveCommand = {

  case Mark(id) =>
    sender() ! InitMarking
    persistAsync(Marker) { m =>
      // update state...
    }

}

PersistingActor: persistAsync{}

will NOT force stashing of commands
PersistentActor: persistAsync(){}
PersistAsync(){} – Ordering guarantees

var state = S0
persistenceId = "a"

Journal
persistAsync(){} – Ordering guarantees

var state = S1
persistenceId = “a”
persistAsync(){} – Ordering guarantees

var state = S2
persistenceId = “a”
`persistAsync()` – Ordering guarantees

```
var state = S2
persistenceId = "a"
```
persistAsync(){} – Ordering guarantees

var state = S2
persistenceId = “a”
`persistAsync()` – Ordering guarantees

```
var state = S3
persistenceld = “a”
```
persistAsync{{}} – Ordering guarantees

defered handlers triggered

var state = S3
persistenceId = “a”
Recovery
Eventsourced, recovery

```scala
/** MUST NOT SIDE-EFFECT! */
def receiveRecover = {
  case replayedEvent: Event =>
    state = updateState(replayedEvent)
}
```

re-using `updateState`, as seen in `receiveCommand`

Akka Persistence ScalaDays
Snapshots
(in SnapshotStore)
/** MUST NOT SIDE-EFFECT! */
def receiveRecover = {
  case SnapshotOffer(meta, snapshot: State) =>
    this.state = state

  case replayedEvent: Event =>
    updateState(replayedEvent)
}

def receiveCommand = {
  case command: Command =>
    saveSnapshot(state) // async!
}

snapshot!? how?
Snapshots

…sum of states…

Journal

E1  E2  E3  E4
E5  E6  E7  E8
Snapshots

State until [E8]

Snapshot Store

E1  E2  E3  E4
E5  E6  E7  E8

Journal

Snapshot!
Snapshots

Journal

state until [E8]

S8

crash!

Snapshots S8

E1 E2 E3 E4

E5 E6 E7 E8
Snapshots

Journal

S8

crash!
Snapshots

Journal

“bring me up-to-date!”

restart!

replay!
Snapshots

"bring me up-to-date!"

Journal

S8

S8

restart!

replay!
Snapshots

State until [E8]

Journal

Snapshots
Snapshots

We *could* delete these!

Journal

state until [E8]
trait MySummer extends Processor {
  var nums: List[Int]
  var total: Int

  def receive = {
    case "snap" => saveSnapshot(total)
    case SaveSnapshotSuccess(metadata) => // ...
    case SaveSnapshotFailure(metadata, reason) => // ...
  }
}
trait MySummer extends Processor {
  var nums: List[Int]
  var total: Int

  def receive = {
    case "snap" => saveSnapshot(total)
    case SaveSnapshotSuccess(metadata) => // ...
    case SaveSnapshotFailure(metadata, reason) => // ...
  }
}

final case class SnapshotMetadata(
  processorId: String, sequenceNr: Long,
  timestamp: Long)
trait MySummer extends Processor {
  var nums: List[Int]
  var total: Int

  def receive = {
    case "snap" => saveSnapshot(total)
    case SaveSnapshotSuccess(metadata) => // ...
    case SaveSnapshotFailure(metadata, reason) => // ...
  }
}
class Counter extends Processor {
    var total = 0

    def receive = {
        case SnapshotOffer(metadata, snap: Int) =>
            total = snap

        case Persistent(payload, sequenceNr) => // ...
    }
}
Snapshots

Upsides

• Simple!

• Faster recovery (!)

• Allows to delete “older” events

• “known state at point in time”
Snapshots

Downsides

• More logic to write

• Maybe not needed if events replay “fast enough”

• Possibly “yet another database” to pick

• snapshots are different than events, may be big!
Views
Views

Journal (DB)

Persistent Actor
persistenceId = "a"

Persistent View
persistenceId = "a"

polling
Views

Persistent Actor
persistenceId = “a”

different ActorPath, same processorId

polling

Journal (DB)

Persistent View
persistenceId = “a”

Persistent View
persistenceId = “a”
class DoublingCounterProcessor extends View {
  var state = 0

  override val processorId = "counter"

  def receive = {
    case Persistent(payload, seqNr) =>
      // "state += 2 * payload"
  }
}
Views, as Reactive Streams
View, as ReactiveStream

// Imports ...

import org.reactivestreams.Publisher

import akka.stream._
import akka.stream.scaladsl.Flow

import akka.persistence._
import akka.persistence.stream._

val materializer = FlowMaterializer(MaterializerSettings())
// 1 producer and 2 consumers:
val p1: Publisher[Persistent] = PersistentFlow.
    fromPersistenceId("p-1").
toPublisher(materializer)

Flow(p1).
    foreach(p => println(s"subs-1: \${p.payload}"))
    consume(materializer)

Flow(p1).
    foreach(p => println(s"subs-2: \${p.payload}"))
    consume(materializer)
View, as ReactiveStream

// 2 producers (merged) and 1 consumer:
val p2: Publisher[Persistent] = PersistentFlow.
  fromPersistenceId("p-2").
  toPublisher(materializer)

val p3: Publisher[Persistent] = PersistentFlow.
  fromPersistenceId("p-3").
  toPublisher(materializer)

Flow(p2).merge(p3). // triggers on "either"
  foreach { p => println(s"subs-3: \${p.payload}" ) }.
  consume(materializer)
Persistence + Cluster
Usage in a Cluster

- **distributed journal** ([http://akka.io/community/](http://akka.io/community/))
  - Cassandra
  - DynamoDB
  - HBase
  - Kafka
  - MongoDB
  - shared LevelDB journal for testing
Usage in a Cluster

• **single writer**
  • cluster singleton
  • cluster sharding
Cluster Singleton
Cluster Singleton

role: backend-1

A
B

role: backend-1

role: backend-2

C
D

role: backend-2
Cluster Sharding

A

B

C

D
Cluster Sharding
Cluster Sharding
Cluster Sharding

sender

region node-1

region node-2

id:17

id:17

17 -> node2

17 -> node2

region node-3

coordinator

17 -> node2
Cluster Sharding

sender

region node-1

region node-2

17 -> node2

region node-3

coordinator 17 -> node2

id:17

17
Cluster Sharding

sender

region node-1

17 -> node2

region node-2

17 -> node2

id:17

region node-3

coordinator

17 -> node2
Cluster Sharding

```
val idExtractor: ShardRegion.IdExtractor = {
  case cmd: Command => (cmd.postId, cmd)
}

val shardResolver: ShardRegion.ShardResolver = msg => msg match {
  case cmd: Command => (math.abs(cmd.postId.hashCode) % 100).toString
}

ClusterSharding(system).start(
  typeName = BlogPost.shardName,
  entryProps = Some(BlogPost.props()),
  idExtractor = BlogPost.idExtractor,
  shardResolver = BlogPost.shardResolver)

val blogPostRegion: ActorRef =
  ClusterSharding(context.system).shardRegion(BlogPost.shardName)

val postId = UUID.randomUUID().toString
blogPostRegion ! BlogPost.AddPost(postId, author, title)
```
Lost messages

sender

$\times$

destination
At-least-once delivery – duplicates
At-least-once delivery - unordered
At-least-once delivery – crash

1. Sent M1
2. Sent M2
3. Sent M3
4. M1 Confirmed
5. M2 Confirmed
6. M3 Confirmed
PersistentActor with AtLeastOnceDelivery

case class Msg(deliveryId: Long, s: String)
case class Confirm(deliveryId: Long)
sealed trait Evt
case class MsgSent(s: String) extends Evt
case class MsgConfirmed(deliveryId: Long) extends Evt

class Sender(destination: ActorPath)
  extends PersistentActor with AtLeastOnceDelivery {

  def receiveCommand: Receive = {
    case s: String => persist(MsgSent(s))(updateState)
    case Confirm(deliveryId) => persist(MsgConfirmed(deliveryId))(updateState)
  }

  def receiveRecover: Receive = { case evt: Evt => updateState(evt) }

  def updateState(evt: Evt): Unit = evt match {
    case MsgSent(s) =>
      deliver(destination, deliveryId => Msg(deliveryId, s))
    case MsgConfirmed(deliveryId) => confirmDelivery(deliveryId)
  }
}
Try it now
Try it now() // !

typesafe.com/activator

akka-sample-persistence-scala

https://github.com/hseeberger/akkamazing
Next step

- Documentation
  - http://doc.akka.io/docs/akka/2.3.3/scala/persistence.html
  - http://doc.akka.io/docs/akka/2.3.3/java/persistence.html
  - http://doc.akka.io/docs/akka/2.3.3/contrib/cluster-sharding.html

- Typesafe Activator
  - https://typesafe.com/activator/template/akka-sample-persistence-scala
  - https://typesafe.com/activator/template/akka-sample-persistence-java

- Mailing list
  - http://groups.google.com/group/akka-user

- Migration guide from Eventsourced
  - http://doc.akka.io/docs/akka/2.3.3/project/migration-guide-eventsourced-2.3.x.html
Links

- Official docs: http://doc.akka.io/docs/akka/2.3.0/scala/persistence.html
- Patrik’s Slides & Webinar: http://www.slideshare.net/patriknw/akka-persistence-webinar
- Papers:
- Pics:
Mailing List

groups.google.com/forum/#!forum/akka-user
Links

Eric Evans: "The DDD book"
Talk: "Acknowledging CAP at the Root"

Vaughn Vernon’s Book and Talk
Dzięki!
Thanks!
ありがとう！

ktoso @ typesafe.com
twitter: ktosopl
github: ktoso
blog: project13.pl
team blog: letitcrash.com