Introduction to Event Sourcing using Akka Persistence

Juris Krikis
Evolution Gaming

Juris Krikis

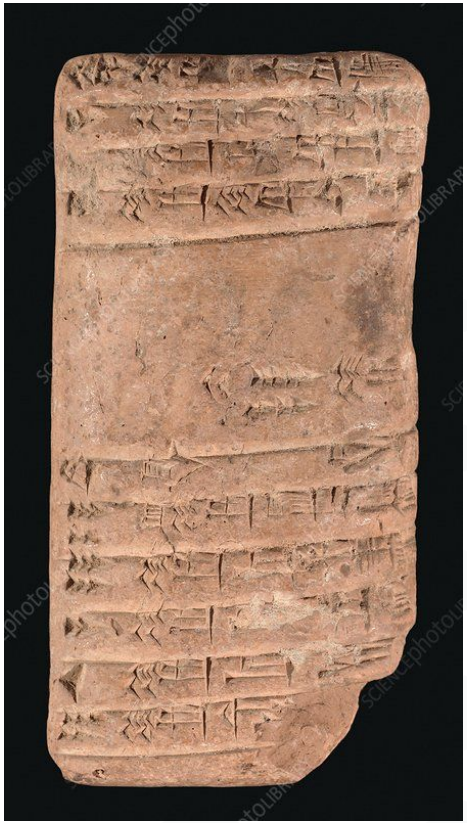
- At Evolution Gaming since 2014
- Head of Scala Department, Riga
- Writing code since 1997
  - Lately Scala
  - Java & JavaScript before that

Agenda

- What is Event Sourcing?
- A simple example using Akka Persistence
- Benefits
- Challenges & Considerations

What is Event Sourcing?

- A design approach that persists all state changes to the system
- Don’t persist the current state
- Persist the changes that lead to this state
  - Atomic
  - Replayable
  - Individually called “events” and collectively - “the event journal”
- Event journal is the principal source of truth
- Time-proven - ledgers used for 7000 years



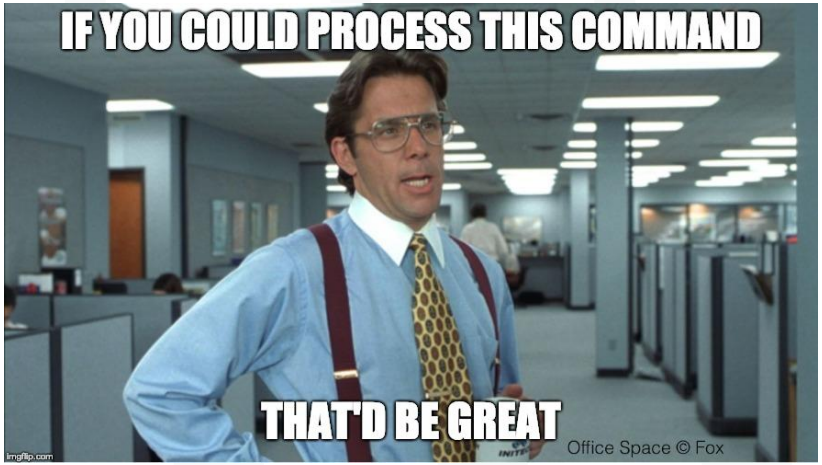
“Account” Event Sourcing example

Command	Event or Error	State
		balance == 0
A. Add 4	#1. Added 4	balance == 4
B. Subtract 5	Error - cannot subtract	balance == 4
C. Subtract 3	#2. Subtracted 3	balance == 1
… system restart or passivation ...		
D. Add 2	#3. Added 2	balance == 3

- Assume a constraint that balance has to be > 0
- Before processing command D, events #1 and #2 are replayed to restore the state

Commands vs Events

- A command is asking to perform something
- A command applied to state either returns
  - … a list of Events (for this presentation we will assume it is always one Event), or
  - … an Error
- An event is a fact that has happened, in the past



State in Event Sourcing

- Current state is a “cache”
  - A function of the event journal
  - Can be rebuilt from the journal
  - Stored so we don’t have to rebuild every time
- Can be discarded
  - System restarts
  - To save memory

```
val state = journal.foldLeft(initialState)((acc, event) => acc.withEvent(event))
```

Snapshots

- Replaying long event journals can be slow
- Snapshots often used as a performance optimisation
  - Restoring state becomes a function of “snapshot + event tail after snapshot”

Command	Event or Error	State	Snapshot
A. Add 4	#1. Added 4	balance == 4	
B. Subtract 3	#2. Subtracted 3	balance == 1	balance == 1, saved after event #2
C. Add 2	#3. Added 2	balance == 3	
… system restart or passivation ...			
Recovering would replay event #3 on top of snapshot saved after event #2			

Akka Persistence

- Event Sourcing implementation within the Akka ecosystem
- Scala & Java APIs
- Event journal and snapshot persistence plugins, e.g.:
  - Development using LevelDB / local file system
  - https://github.com/akka/akka-persistence-cassandra
  - https://github.com/dnvriend/akka-persistence-jdbc
  - https://github.com/evolution-gaming/kafka-journal - discussed in next presentation
- Cluster Sharding
  - Distribute persistent actors across Akka Cluster nodes

- Event Sourcing is not confined to the “Akka Persistence world”
  - “Exploring CQRS and Event Sourcing” book by Microsoft
  - https://www.microsoft.com/en-us/download/details.aspx?id=34774

PersistentActor

```
class Account extends PersistentActor {
 /** Recovery handler that receives persisted events during recovery. */
 override def receiveRecover: Receive = ???

 /**
   * Command handler. Typically validates commands against current state (and/or by
   * communication with other actors). On successful validation, one or more events
   * are derived from a command and these events are then persisted by calling
   * `persist`.
   */
 override def receiveCommand: Receive = ???

 /** Id of the persistent entity for which messages should be replayed. */
 override def persistenceId: String = ???
}
```

State, Command, Event

```
case class State(balance: BigDecimal) {
  def withEvent(event: Event): State = event match {
     case Added(amount: BigDecimal)       =>  copy(balance = balance + amount)
     case Subtracted(amount: BigDecimal)  =>  copy(balance = balance - amount)
   }
 }

 sealed trait Command
 case class Add(amount: BigDecimal) extends Command
 case class Subtract(amount: BigDecimal) extends Command

 sealed trait Event
 case class Added(amount: BigDecimal) extends Event
 case class Subtracted(amount: BigDecimal) extends Event
```

State variable & persistenceId

```
class Account(id: UUID) extends PersistentActor {
  private var state: State = State(0)

  override def persistenceId: String = s"account-$id"

  // ...
}
```

receiveCommand

```
override def receiveCommand: Receive = {
 case cmd: Command   =>
   val snd = sender()

   validateCommand(state, cmd) match {
     case Left(error)      =>
       snd ! Nack(error, cmd)

     case Right(event)     =>
       persist(event) { event =>
         updateState(event)

         snd ! Ack(cmd)
       }
   }
}
```

Command validation - “Behavior”

```
def validateCommand(state: State, cmd: Command): Either[Error, Event] = {
 def isPositive(x: BigDecimal) = Either.cond(x > 0, x, s"Amount is negative: $x")

 cmd match {
     case Add(x: BigDecimal)      =>
     for {
       x <- isPositive(x)
     } yield Added(x)

   case Subtract(x: BigDecimal)  =>
     def isSufficient(x: BigDecimal) =
       Either.cond(x <= state.balance, x, s"Balance ${state.balance} too low")

     for {
       y <- isPositive(x)
       z <- isSufficient(y)
     } yield Subtracted(z)
 }
```

receiveRecover

```
private def updateState(evt: Event): Unit = {
 state = state.withEvent(evt)
}

override def receiveRecover: Receive = {
 case evt: Event  =>   updateState(evt)
}
```

Test using Akka TestKit

```
val id = UUID.randomUUID()
val actorRef = system.actorOf(Account.props(id))

actorRef ! Subtract(2)
expectMsg(Nack(s"Balance 0 is too low", Subtract(2)))

actorRef ! Add(4)
expectMsg(Ack(Add(4)))

actorRef ! Subtract(5)
expectMsg(Nack(s"Balance 4 is too low", Subtract(5)))

actorRef ! Subtract(3)
expectMsg(Ack(Subtract(3)))

actorRef ! GetState
expectMsg(State(1))
```

Scala Event Sourcing libraries

- Safe Akka by Evolution Gaming
  - https://github.com/evolution-gaming/safe-akka
- Aecor
  - https://github.com/notxcain/aecor
- Fun.CQRS
  - https://github.com/fun-cqrs/fun-cqrs
- Lagom
  - https://www.lagomframework.com/documentation/1.4.x/scala/PersistentEntity.html
  - https://www.lagomframework.com/documentation/1.4.x/scala/ES_CQRS.html

Scalability

- Appends scale better than updates
  - Especially for databases optimised for such workloads
- Horizontally scalable using Akka Cluster Sharding
  - Evolution has developed custom strategies
  - https://github.com/evolution-gaming/sharding-strategy

Avoids Object-Relational Mapping

- Object-relational impedance mismatch endemic to ORM-ed CRUD applications
  - Mapping objects to tables gets hard for non-trivial models

Other Benefits

- Audit log
  - Enforce append-only on database level
  - Always accurate as transactional data in the event journal is the audit log
- Time travel
  - Easy temporal queries
  - What was state of the system at a previous point in time?

Easy to Develop, Test & Debug

- Updates on write side are sequentially consistent due to the actor model
  - “Single Writer Principle”
  - Easy to reason about & thus develop
  - Avoids common update anomalies due to concurrent updates
- Restore system state how it was at any point in time
  - Replay the event log
  - Issue in production can be recreated in development environment
    - Similar to how frameworks like Redux can do on the client side

Conceptual Benefits

- Affinity with Command Query Responsibility Segregation
  - Separate the command/event processing and query/reporting layers
  - Easier composability into micro-services
  - Can add more views later without affecting write layer
- Affinity with Domain Driven Design

Challenges & Key Considerations

- Illustrate what to consider
  - Most points can warrant a longer discussion

$$state(now) = \int_{t=0}^{now} stream(t)\ \mathrm{d}t \qquad stream(t) = \frac{\mathrm{d}\ state(t)}{\mathrm{d}t}$$

Choice of Aggregate Roots

- Aggregate - a cluster of associated objects that we treat as a unit for the purpose of data change
  - Each aggregate has its own event stream
- Too small leads to need for cross-aggregate transactions
  - Sagas
- Too large leads to large states and complicated event & state models
  - Hard to maintain & scale
- Regulatory & data protection considerations
  - GDPR “Forget a user” easier if the user is the aggregate root
- Choice of keys & looking up by synthetic keys
  - An aggregate has one persistent ID and lookup is most efficient by that
  - If clients want to use other IDs to look them up, it adds complexity

Read Layer is Eventually Consistent

- Stream / poll events & update the “read side”
- Don’t depend on read layer being strongly consistent
- Work on minimising the latency
- Can share state locally, through PubSub, distributed data
- Can poll persistent actors directly if performance allows
  - Be careful about not exposing mutable actor internal state
- Your database may also be eventually consistent

Dealing with side effects

- Understand upfront where you will initiate side effects
  - Command processors vs When saving events vs Read layer
  - Don’t re-initiate side effects upon replaying events
- Easier if side effects are idempotent
  - Same effect can be applied multiple times without changing the result
- Not always the case
  - Was the email you asked the mail server to send sent or not?
- Understand which direction you want to fail
  - Better to not send the email?
  - Better to send it multiple times?

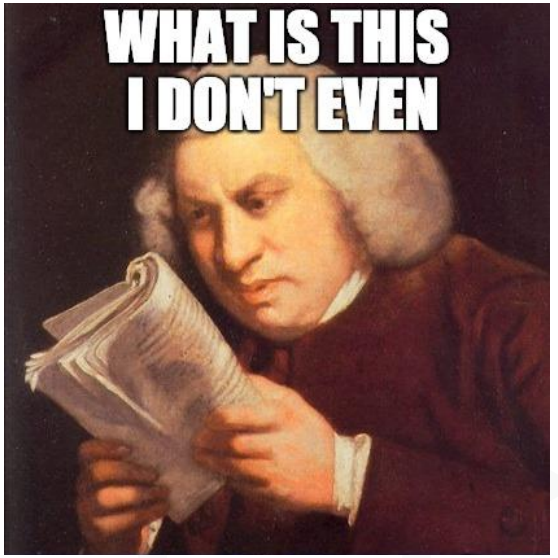
Fast growing aggregates & infinite event journals

- In theory, space is cheap
  - In practice, fast space isn't
- Deleting events should be avoided - even those before snapshots
  - We may need them
  - Don't delete, backup to long term storage
  - Can make read layers more complicated
  - May be required due to GDPR - cryptography can help
- Rebuilding read layers from big data sets
  - Test for this, avoid getting painted in the corner
  - Use "on demand" resources for spikes

Schema evolution

- Event serialisation format
  - Performance / size (e.g. ProtoBuf, Kryo, Avro) vs. human readability (e.g. JSON)
  - Understand how to deal with schema evolution
  - https://doc.akka.io/docs/akka/2.5/persistence-schema-evolution.html
- Event granularity
  - Smaller events easier to migrate between versions
- Backwards compatibility of old events - for how long?
  - Forever?
  - Next version?
- Migrate events in-place to newer versions usually is an anti-pattern
- What if we published the events to our clients or read layer and they stored them?

Unreadable events



- Events can be unreadable or unknown
- What should we do?
  - Crash vs stop vs ignore?
  - Alerting
  - Tests should cover compatibility
    - Event (de-)serialisation tests diligently maintained
    - Run new versions against existing event histories
- Are events your external contract...
  - … or do you have a separate contract?

Other Challenges & Considerations

- Frequency & content of snapshots
- Fixing history for business errors & code errors
  - Usually corrective events
  - Events are immutable and cannot be updated or deleted
- Aggregate multiple events from multiple roots reliably – can take much memory
- Avoiding cluster split brains & parallel event histories
- Choice of database for event & snapshot storage
- How to build a low-latency, reliable read layer

In conclusion...

- Easy enough to get a simple example working…
- Lots of things to consider
- Significant community support & existing knowledge
- Worth it for many use cases, e.g.:
  - Your domain is inherently event-driven
  - Accountability / traceability is important
  - Version control / undo features needed (e.g. Wiki-s)
  - You're using DDD and/or CQRS

- Evolution Gaming successfully using Event Sourcing with Akka Persistence since 2015

Thanks for listening!

- Yaroslav will continue with “Kafka Journal”
  - https://github.com/evolution-gaming/kafka-journal
- Suggestions for topics you want to hear on future meetups - tell us!