Easy Scalability with Akka

Distribute your domain
Who?

● BoldRadius Solutions
  ○ boldradius.com
  ○ Typesafe Partner
  ○ Scala, Akka and Play specialists
  ○ Ottawa, Saskatoon, San Francisco, Boston, Chicago, Montreal, New York, Toronto

● Michael Nash, VP Capabilities
  ○ @MichaelPNash
  ○ michael.nash@boldradius.com
What?

What will I know in the next 45 minutes?

- Distributed Domain-Driven Design
  - What is it, and how does it apply to project?
  - What is CQRS and ES, and how do they relate?
- Akka
  - How to do DDDD and CQRS/ES with Akka
  - What is different in a DDDDD Akka project, and why?
  - How does this kind of project scale compared to a "normal" project, even with Akka?
Scalability

- Scalability is not performance
  - Performance increasing reduces response time
  - Scalability increasing improves capacity in relation to added resources

- Scalability can be...
  - Vertical (Scaling “up”): Bigger box (or boxes)
  - Horizontal (Scaling “out”): More boxes
The Actor model for the JVM
- A share-nothing distributed message-driven compute model
- Scala and Java APIs

Like Erlang/Elixir processes, only better
- Distributed by design, local is an optimization
CQRS and ES

- **Command Query Responsibility Segregation**
  - Separate the read and write paths of the problem

- **Event-Sourcing**
  - State is built from a journal of events

- One doesn’t need the other, but they can be combined very effectively
CQRS

Write Side

Command → Command → Command → Write Model

Read Side

Read Model → Event → Event → Event
Domain

- Your application’s *domain* is the part that relates to the *problem*
- The rest of your application is *infrastructure*, basically
- If you have a Payroll app, then *Employees, Benefits, Pay Stubs* are in your domain
Domain-Driven Design (DDD) is not only for Object-Oriented Systems.

Object-Oriented and Functional are not an either-or choice, or at odds with each other.

Distributed Domain Driven Design is a natural fit with the actor model.
Traditional Domain Instances

- Blocking Operations
- Reads
- Domain Object Instance
- Writes
- Database
Non-Distributed Domain

- **Basic CRUD** (Create, Read, Update, Delete)
- **Writes and reads to same database**
  - Always consistent
- **Scaled via multiple identical replicas**
  - and load balancing on HTTP
- **Bottlenecks on contention**
  - reads and writes interfere with each other
When we scale the traditional model...

- We add multiple servers
  - so there’s more than one copy of each domain instance (potentially)
- Each must read state every time
  - from the shared resource (the database)
- Each must write (fully) to the same shared resource to avoid conflict
- Scalability is limited
With Akka and DDDD

- Wouldn’t it be great if you could just keep your domain instances *in memory*?
- But how to recover from its volatile nature: an event journal!
But I have too many instances!

- Only *active* domain instances are in memory
- Instances *activate* and *passivate* as needed
Activation and Passivation

- **Activate**
  - No -> Restore State
  - Yes -> Active?
    - Yes -> Process Message
    - No -> Done

- **Message Received**
  - Active?
    - Yes -> Passivate
    - No -> Done

- **Idle Timeout Received**
  - Active?
    - Yes -> Passivate
    - No -> Done
Ingredients

- Akka - Scala API
  - with Clustering and Persistence modules
- Cassandra
  - And the DataStax driver
- Typesafe Activator
  - For our template
What’s Different?

- Domain object instances are *transient*, and memory-resident when in use
- Reads and writes don’t contend for resources
- Cluster can be grown until domain instances per node = 1
What do you get?

- **Scalability**
  - Add nodes to handle higher load on the fly

- **Failover**
  - No single point of failure or contention
  - Instances can be created anywhere as needed

- **Simplicity**
  - Build your app without having to worry about scalability later
Consider your own domain

- Consider writes independently from reads
  - Are they really the same even now?
- Deployment structure is critical to get all benefits safely
Example Applications

Online Auctions: Two implementations

- *Users* place *bids* for online *auctions*
- Most code shared
- Identical APIs
- Identical deploy structure, same machines, same number of nodes
- One CRUD, One CQRS/DDDD
CRUD Actor

- Uses Akka
- Receives request
- Writes to datastore directly
class SimpleAuctionBackend(implicit val storageAdapter: StorageAdapter) extends Actor with ActorLogging {

  import context.dispatcher

  override def postStop() = {
    storageAdapter.close()
  }

  def receive = {
    case StartAuction(auctionId, start, end, initialPrice, productId) =>
      storageAdapter.addAuctionAsync(auctionId, start, end, productId, initialPrice).map(_ match {
        case Success(a) => StartedAuctionAck(auctionId)
        case Failure(f) => InvalidAuctionAck(auctionId, f.getMessage())
      }) pipeTo sender()
    case PlaceBid(aid, bidBuyer, bidPrice) =>
      val currentTimestamp = StorageAdapter.timestamp
      val resF = for {
        auctionT <- storageAdapter.auctionAsync(aid)
        addT <- {
          auctionT match {
            case Success(a) if a.start > currentTimestamp => Future(Failure(new Exception("The auction has not started")))
            case Success(a) if a.end <= currentTimestamp => Future(Failure(new Exception("The auction has ended")))
            case Success(a) if a.initialPrice > bidPrice => Future(Failure(new Exception("The action initial price is higher than bidPrice")))
            case Success(a) => storageAdapter.addAuctionBidAsync(aid, bidPrice, bidBuyer)
            case Failure(f) => Future(Failure(f))
          }
        }
        highBidT <- addT match {
          case Success(_):= storageAdapter.highestBidAsync(aid)
          case Failure(f):= Future(Failure(f))
        }
      } yield {
        highBidT match {
          case Success(highBid) if highBid.buyer == bidBuyer && highBid.price == bidPrice =>
            PlacedBidAck(aid, bidBuyer, bidPrice, highBid.timeStamp)
          case Success(highBid) => RefusedBidAck(aid, bidBuyer, bidPrice, highBid.price)
          case Failure(f) => FailedBidAck(aid, bidBuyer, bidPrice, f.getMessage())
        }
      }
    resF pipeTo sender()
    case GetWinningBid(auctionId) =>
      storageAdapter.highestBidAsync(auctionId).map(_ match {
        case Success(bid) => winningBidPriceResponse(auctionId, bid.price)
        case Failure(f) => InvalidBidResponseResponse(auctionId, f.getMessage())
      }) pipeTo sender()
    case GetAuctionHistory(auctionId) =>
      storageAdapter.auctionHistoryAsync(auctionId, 1000000).map(_ match {
        case Success(bids) => BidHistoryResponse(auctionId, bids)
        case Failure(f) => InvalidBidQueryResponse(auctionId, f.getMessage())
      }) pipeTo sender()
  }
}
CQRS/DDDD Actor

- In-memory state
- Recovers state from journal on startup
- Uses become to change states
- Passivates on timeout
class BidProcessor(readRegion: ActorRef) extends PersistentActor with AuctionLogging {

import BidProcessor._

override def persistenceId: String = self.path.parent.name + "-" + self.path.name

val tickTask = context.system.scheduler.schedule(1.seconds, 1.seconds, self, Tick)

private var auctionStateMaybe: Option[AuctionBidState] = None

private def startMaybeState(auctionId: String, startTime: Long, endTime: Long, initialPrice: Double): Option[AuctionBidState] = Some(AuctionBidState(auctionId, startTime, endTime, initialPrice, Nil, Nil, false))

private def updateMaybeState(auctionId: String, f: AuctionBidState => AuctionBidState): Option[AuctionBidState] = auctionStateMaybe.flatMap(state => Some(f(state)))

private def updateState(evt: AuctionEvt): Unit = {
  auctionStateMaybe = evt match {
    case AuctionStartedEvt(auctionId, timeStart, end, initialPrice, prodId) => startMaybeState(auctionId, timeStart, end, initialPrice)
    case AuctionEndedEvt(auctionId, timeStart) => updateMaybeState(auctionId, a => a.copy(ended = true))
    case BidPlacedEvt(auctionId, buyer: String, bidPrice: Double, timeStart) => updateMaybeState(auctionId, a => a.copy(acceptedBids = Bid(bidPrice, buyer, timeStart) :: a.acceptedBids))
    case BidRefusedEvt(auctionId, buyer: String, bidPrice: Double, timeStart) => updateMaybeState(auctionId, a => a.copy(refusedBids = Bid(bidPrice, buyer, timeStart) :: a.refusedBids))
    case BidFailedEvt(auctionId, buyer: String, bidPrice: Double, timeStart, error: String) => updateMaybeState(auctionId, a => a.copy(refusedBids = Bid(bidPrice, buyer, timeStart) :: a.refusedBids))
  }
}

private def getCurrentBid(state: AuctionBidState): Double = state.acceptedBids match {
  case Bid(_, _, _) :: tail => tail
  case _ => state.initialPrice
}

override def receiveCommand: Receive = initial

def initial: Receive = {
  case @StartAuctionCmd(id, start, end, initialPrice, prodId) =>
    val currentTime = System.currentTimeMillis()
    if (currentTime >= end) sender ! InvalidAuctionAck(id, "This auction is already over")
    else {
      val event = AuctionStartedEvt(id, start, end, initialPrice, prodId).logDebug("AuctionStartedEvt " + _.toString)
      persistAsync(event) { evt =>
        readRegion ! Update(await = true)
        auctionStateMaybe = startMaybeState(id, start, end, initialPrice)
        context.become(takingBids(id, start, end))
      }
    }
}
case ReceiveTimeout => context.parent ! Passivate(stopMessage = PoisonPill).logDebug("ReceiveTimeout + _.toString")

def takingBids(auctionId: String, startTime: Long, closeTime: Long): Receive = {
  case ReceiveTimeout => context.parent ! Passivate(stopMessage = PoisonPill)
  case Tick => {
    val currentTime = System.currentTimeMillis()
    if (currentTime >= closeTime) {
      persistAsync(AuctionEndedEvt(auctionId, currentTime)) { evt =>
        readRegion ! Update(await = true)
        updateState(evt)
      }
      context.become(auctionClosed(auctionId, currentTime))
    }

    case a@PlaceBidCmd(id, buyer, bidPrice) => {
      val timestamp = System.currentTimeMillis()
      auctionStateMaybe.map(state => {
        if (timestamp < closeTime && timestamp >= startTime) {
          val currentPrice = getCurrentBid(state)
          if (bidPrice > currentPrice) {
            val event = BidPlacedEvt(id, buyer, bidPrice, timestamp).logInfo(s"${auctionId} BidPlacedEvt " + _.toString)
            readRegion ! Update(await = true)
            updateState(evt)
            sender() ! PlacedBidAck(id, buyer, bidPrice, timestamp)
          } else {
            val event = BidRefusedEvt(id, buyer, bidPrice, timestamp)
            readRegion ! Update(await = true)
            updateState(evt)
            sender() ! RefusedBidAck(id, buyer, bidPrice, currentPrice)
          }
        } else {
          if (timestamp < closeTime) sender() ! AuctionEndedAck(id)
        } else sender() ! AuctionNotYetStartedAck(id)
      })
    }

    case other => sender() ! InvalidAuctionAck(auctionId, "This auction is already started, only taking bids").logDebug(s"${auctionId} ${other} " + _.toString)
  }
}

def auctionClosed(auctionId: String, closeTime: Long): Receive = {
  case ReceiveTimeout => context.parent ! Passivate(stopMessage = PoisonPill)
  case _ => sender() ! AuctionEndedAck(auctionId)
}
```scala
def receiveRecover: Receive = {
  case evt: AuctionEvt => currentState(evt.logInfo("receiveRecover" + _.toString))

  case RecoveryCompleted => {
    auctionStateMaybe.fold[Unit]({}) { s =>
      if (s.ended) context.become(auctionClosed(s.auctionId, s.endTime))
      else context.become(takingBids(s.auctionId, s.startTime, s.endTime))
    }.logInfo("receiveRecover RecoveryCompleted auctionStateMaybe: " + _.toString)
  }

  case SnapshotOffer(_, snapshot) =>
    auctionStateMaybe = snapshot/asInstanceOf[Option[AuctionBidState]].logInfo("recovery from snapshot auctionStateMaybe:" + _.toString)
  }
```
Test Setup

3 Cassandra Nodes (Clustered)
3 Akka processing Nodes (Clustered)
1 Front-end (Part of the Akka cluster)
All Small AWS Instances
100 simultaneous users
100 auctions
Two minutes of bidding
class Auctions extends Simulation {
  val httpProtocol = http
  .baseURL(scala.io.Source.fromFile("urls.txt").getLines().toList.head)
  .inferHtmlResources()
  .acceptHeader("text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8")
  .acceptEncodingHeader("gzip, deflate")
  .acceptLanguageHeader("en-US,en;q=0.5")
  .connection("keep-alive")
  .userAgentHeader("Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:33.0) Gecko/20100101 Firefox/33.0")

  lazy val auctionIds = Seq.fill(100)(UUID.randomUUID().toString())
  lazy val buyerIds = Seq.fill(100)(UUID.randomUUID().toString())
  lazy val bids = Range(2,100).toList

  val create = scenario("Start auctions").exec(Auctions.startAuctions(auctionIds))
  val bid = scenario("Bid auctions").exec(Auctions.bidAuctions(auctionIds,bids,buyerIds))

  setUp(create.inject(atOnceUsers(1),nothingFor(4 seconds)), bid.inject(nothingFor(4 seconds),atOnceUsers(100))).maxDuration(2 minutes).protocols(httpProtocol)
Response Time Distribution

**CRUD**

**CQRS**

Scale is different!
Response Time Percentiles: CRUD
Response Time Percentiles: CQRS

Note that the scale is different!
Requests per Second: CRUD
Requests per Second: CQRS

Scale is different!
Total Requests/Responses in 2 mins

CRUD: 75,991
CQRS: 199,357
Adding more nodes...

- **CRUD version**
  - Contention increases
  - Diminishing benefit to adding more nodes
  - Tuning doesn’t help much
- **CQRS version**
  - No contention
  - Near-linear benefit until number of nodes = number of auctions
  - Tuning is very important
Activator Template

https://github.com/boldradius/akka-dddd-template

Add your own domain!
Q & A
michael.nash@boldradius.com

Any Questions?
Further Reading