Akka Persistence Typed

Renato Cavalcanti

Lightbend

@renatocaval

History

- Started as an Akka Extension called Eventsourced by Martin Krasser
- Brought into Akka as Akka Persistence (untyped) in 2.4.0
- Akka Persistence Query in 2.5.0
- Akka Persistence Typed (upcoming 2.6.0)

Akka Typed - protocol

When defining an Actor, we start by defining it's protocol

```
sealed trait Message
final case class SayHello(name: String) extends Message
final case class ChangeGreeting(greet: String) extends Message
```

Akka Typed - behavior

In Akka Typed, an Actor is just a Behavior function

```
def behavior(greeting: String): Behavior[Message] =
  Behaviors.receiveMessage {
    case SayHello(name) ⇒
      println(s"$greeting $name!")
      Behaviors.same
    case ChangeGreeting(greet) ⇒
      behavior(greet)
  }
```

Akka Typed - ActorContext

```
def behaviorCtx(greeting: String): Behavior[Message] =
  Behaviors.setup { ctx: ActorContext[Message] ⇒
    Behaviors.receiveMessage {
      case SayHello(name) ⇒
        ctx.log.info(s"$greeting $name!")
        Behaviors.same
      case ChangeGreeting(greet) ⇒
        behavior(greet)
    }
  }
```

Akka Typed - ask pattern

There is no 'sender'. If you need to respond to an 'ask', your incoming message must have an `ActorRef[R]` that you can use to respond.

```
final case class Hello(msg: String)
final case class SayHello(name: String, replyTo: ActorRef[Hello])

def behavior(greeting: String): Behavior[SayHello] =
  Behaviors.receiveMessage {
    case SayHello(name, replyTo) ⇒
      replyTo ! Hello(s"$greeting $name!")
      Behaviors.same
  }
```

Akka Typed

```
final case class Hello(msg: String)
final case class SayHello(name: String, replyTo: ActorRef[Hello])

def behavior(greeting: String): Behavior[SayHello] =
  Behaviors.receiveMessage {
    case SayHello(name, replyTo) ⇒
      replyTo ! Hello(s"$greeting $name!")
      Behaviors.same
  }

val greeter: ActorSystem[SayHello] = ActorSystem(behavior("Hello"), "HelloAkka")
val res: Future[Hello] =
  greeter.ask(replyTo ⇒ SayHello("Akka Typed", replyTo))
```

Akka Persistence Typed - Highlights

- Protocol is defined in terms of `Command`, `Event` and `State`
- `EventSourcedBehavior` instead of `Behavior`
- Tagging function
- Better controlled snapshotting (number of events and/or predicate)
- Enforced Replies

- Old plugins are still compatible
- Akka Persistence Query untouched, already typed and based on Akka Streams

Commands, Events and State

```
sealed trait AccountCommand
final case class Deposit(amount: Double) extends AccountCommand
final case class Withdraw(amount: Double) extends AccountCommand
case class GetBalance(replyTo: ActorRef[Balance]) extends AccountCommand

sealed trait AccountEvent
final case class Deposited(amount: Double) extends AccountEvent
final case class Withdrawn(amount: Double) extends AccountEvent

case class Account(balance: Double)
```

Command Handler

```
// (State, Command) ⇒  Effect
case class Account(balance: Double) {
  def applyCommand(cmd: AccountCommand): Effect[AccountEvent, Account] =
    cmd match {
      case Deposit(amount) ⇒
        Effect.persist(Deposited(amount))

      // other cases intentionally omitted
    }
}
```

Event Handler

```
// (State, Event) ⇒  State
case class Account(balance: Double) {
  def applyEvent(evt: AccountEvent): Account = {
      evt match {
        case Deposited(amount) ⇒ copy(balance = balance + amount)
        case Withdrawn(amount) ⇒ copy(balance = balance - amount)
      }
    }
}
```

EventSourcedBehavior

```
def behavior(id: String): EventSourcedBehavior[AccountCommand, AccountEvent, Account] = {

  EventSourcedBehavior[AccountCommand, AccountEvent, Account](
    persistenceId = PersistenceId(id),
    emptyState = Account(balance = 0),
    // command handler: (State, Command) ⇒ Effect
    commandHandler = (account, cmd) ⇒ account.applyCommand(cmd),
    // event handler: (State, Event) ⇒ State
    eventHandler = (account, evt) ⇒ account.applyEvent(evt)
  )

}
```

Tagging

```
def behavior(id: String): EventSourcedBehavior[AccountCommand, AccountEvent, Account] = {

  EventSourcedBehavior[AccountCommand, AccountEvent, Account](
    persistenceId = PersistenceId(id),
    emptyState = Account(balance = 0),
    commandHandler = (account, cmd) ⇒ account.applyCommand(cmd),
    eventHandler = (account, evt) ⇒ account.applyEvent(evt)
  )
  .withTagger {
    // tagging events are useful for querying by tag
    case evt: Deposited ⇒ Set("account", "deposited")
    case evt: Withdrawn ⇒ Set("account", "withdrawn")
  }

}
```

Snapshots

```
def behavior(id: String): EventSourcedBehavior[AccountCommand, AccountEvent, Account] = {

  EventSourcedBehavior[AccountCommand, AccountEvent, Account](
    persistenceId = PersistenceId(id),
    emptyState = Account(balance = 0),
    commandHandler = (account, cmd) ⇒ account.applyCommand(cmd),
    eventHandler = (account, evt) ⇒ account.applyEvent(evt)
  )
  // save a snapshot on every 100 events and keep max 2
  .withRetention(RetentionCriteria.snapshotEvery(numberOfEvents = 100, keepNSnapshots = 2))

  // save a snapshot when a predicate holds
  .snapshotWhen {
    case (account, evt: Withdrawn, seqNr) ⇒ true
    case _                                ⇒ false
  }

}
```

Some live coding

Cluster Sharding and Persistence

- Manage state over different JVMs
- Knows where is your instance
- Entity Passivation
- Honours single writer principle for Persistence
- Rolling updates without downtime
- Commands must be serializable

Cluster Sharding - EntityContext

```
object Account {

  val typeKey = EntityTypeKey[AccountCommand]("Account")

  def behavior(entityContext: EntityContext):
    EventSourcedBehavior[AccountCommand, AccountEvent, Account] = {

      EventSourcedBehavior[AccountCommand, AccountEvent, Account](
        persistenceId = PersistenceId(entityContext.entityId),
        emptyState = Account(balance = 0),
        commandHandler = (account, cmd) ⇒ account.applyCommand(cmd),
        eventHandler = (account, evt) ⇒ account.applyEvent(evt)
      )
}
```

Cluster Sharding - Entity

```
clusterSharding.init(
    Entity(
      Account.typeKey,
      ctx: EntityContext ⇒ Account.behavior(ctx)
    )
  )
```

Cluster Sharding - EntityRef

```
val account: EntityRef[AccountCommand] =
  clusterSharding.entityRefFor(
    typeKey = Account.typeKey,
    entityId = "BE50 7314 3515 2919"
  )
```

Takeaways

- Declarative API
- Developer can concentrate on modelling
- Types everywhere
- And functions9
- `Any` ⇒ `Unit` is part of the past

- Event Sourcing opens the door for decoupling your services
- High throughput with append only journals
- Scalability with Cluster Sharding
- Rolling updates when clustered
- Distributed event consuming (included in Lagom, will be extracted)

Thanks for listening

In GitHub:
renatocaval/akka-persistence-typed-talk

Twitter: @renatocaval