ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Akka Typed Persistence

Programming Reactive Systems

Roland Kuhn

Akka Typed Persistence

Persistent actors take note of changes:

- incoming commands may result in events
- events are persisted in the journal
- persisted events are applied as state changes

Akka Typed Persistence

Persistent actors take note of changes:

- incoming commands may result in events
- events are persisted in the journal
- persisted events are applied as state changes

This pattern lends itself well to a type-safe expression:

- one function turns commands into events
- one function codifies the effect of an event on the state

Persistence example: the transfer saga

The bank account example from week 2 revisited:

- Alice and Bob have accounts, held in a ledger
- the transfer is performed in two steps by a transfer actor
- when the credit to Bob fails, Alice gets a refund (rollback)

Persistence example: the transfer saga

The bank account example from week 2 revisited:

- Alice and Bob have accounts, held in a ledger
- the transfer is performed in two steps by a transfer actor
- when the credit to Bob fails, Alice gets a refund (rollback)

Assume a simple ledger service:

```scala
sealed trait Ledger
case class Debit (account: String, amount: BigDecimal,
                  replyTo: ActorRef[Result]) extends Ledger
case class Credit(account: String, amount: BigDecimal,
                  replyTo: ActorRef[Result]) extends Ledger
```

Persistence example: the transfer saga

```
ActorSystem(Behaviors.setup[Result] { ctx =>
    val ledger = ctx.spawn(Ledger.initial, ”ledger”)
    val config = TransferConfig(ledger, ctx.self, 1000.00, ”Alice”, ”Bob”)
    val transfer = ctx.spawn(PersistentBehaviors.receive(
        persistenceId = ”transfer-1”,
        emptyState = AwaitingDebit(config),
        commandHandler = commandHandler,
        eventHandler = eventHandler
    ), ”transfer”)

    Behaviors.receiveMessage(_ => Behaviors.stopped)
}, ”Persistence”)
```

The saga's input commands

```
sealed trait Command
case object DebitSuccess extends Command
case object DebitFailure extends Command
case object CreditSuccess extends Command
case object CreditFailure extends Command
case object Stop extends Command
```

The saga's events

```scala
sealed trait Event
case object Aborted extends Event
case object DebitDone extends Event
case object CreditDone extends Event
case object RollbackStarted extends Event
case object RollbackFailed extends Event
case object RollbackFinished extends Event
```

The stateful command handler

```scala
sealed trait State
case class AwaitingDebit(config: TransferConfig) extends State
case class AwaitingCredit(config: TransferConfig) extends State
case class AwaitingRollback(config: TransferConfig) extends State
case class Finished(result: ActorRef[Result]) extends State
case class Failed(result: ActorRef[Result]) extends State
```

The stateful command handler

```scala
sealed trait State
case class AwaitingDebit(config: TransferConfig) extends State
case class AwaitingCredit(config: TransferConfig) extends State
case class AwaitingRollback(config: TransferConfig) extends State
case class Finished(result: ActorRef[Result]) extends State
case class Failed(result: ActorRef[Result]) extends State

val commandHandler: CommandHandler[Command, Event, State] =
    CommandHandler.byState {
        case _: AwaitingDebit => awaitingDebit
        case _: AwaitingCredit => awaitingCredit
        case _: AwaitingRollback => awaitingRollback
        case _ => (_, _, _) => Effect.stop
    }
```

Tangent: a single-use adapter

Sometimes a single response is enough, follow-ups shall be dropped.

An actor’s lifecycle makes this easy to express.

Tangent: a single-use adapter

Sometimes a single response is enough, follow-ups shall be dropped.

An actor's lifecycle makes this easy to express.

```scala
def adapter[T](ctx: ActorContext[Command], f: T => Command): ActorRef[T] =
    ctx.spawnAnonymous(Behaviors.receiveMessage[T] { msg =>
        ctx.self ! f(msg)
        Behaviors.stopped
    })
```

Handling a saga step

```
val awaitingDebit: CommandHandler[Command, Event, State] = {
    case (ctx, AwaitingDebit(tc), DebitSuccess) =>
        Effect.persist(DebitDone).andThen { state =>
            tc.ledger ! Credit(tc.to, tc.amount, adapter(ctx, {
                case Success => CreditSuccess
                case Failure => CreditFailure
            }))
        }
    case (ctx, AwaitingDebit(tc), DebitFailure) =>
        Effect.persist(Aborted)
            .andThen((state: State) => tc.result ! Failure)
            .andThenStop
    case x => throw new IllegalStateException(x.toString)
}
```

Handling a saga event

```scala
val eventHandler: (State, Event) => State = { (state, event) =>
    (state, event) match {
        case (AwaitingDebit(tc), DebitDone) => AwaitingCredit(tc)
        case (AwaitingDebit(tc), Aborted) => Failed(tc.result)
        case (AwaitingCredit(tc), CreditDone) => Finished(tc.result)
        case (AwaitingCredit(tc), RollbackStarted) => AwaitingRollback(tc)
        case (AwaitingRollback(tc), RollbackFinished) => Failed(tc.result)
        case (AwaitingRollback(tc), RollbackFailed) => Failed(tc.result)
        case x => throw new IllegalStateException(x.toString)
    }
}
```

Taking the saga back up after recovery

When waking up after a crash, the saga may find itself in any state:

- needs to take up the transfer again while still successful
- needs to take up the rollback again if already failed
- needs to signal completion if already terminated

Taking the saga back up after recovery

When waking up after a crash, the saga may find itself in any state:

- needs to take up the transfer again while still successful
- needs to take up the rollback again if already failed
- needs to signal completion if already terminated

```
PersistentBehaviors.receive(
  ...
).onRecoveryCompleted {
    case (ctx, AwaitingDebit(tc)) =>
        ledger ! Debit(tc.from, tc.amount, adapter(ctx, {
            case Success => DebitSuccess
            case Failure => DebitFailure
        }))
    ...
}
```

Do not forget to stop the saga after recovery!

```
PersistentBehaviors.receive(
  ...
).onRecoveryCompleted {
    ...
    case (ctx, Finished(result)) =>
        println(”still finished”)
        ctx.self ! CreditSuccess // will effectively stop this actor
        result ! Success
    case (ctx, Failed(result)) =>
        println(”still failed”)
        ctx.self ! CreditSuccess // will effectively stop this actor
        result ! Failure
}
```

Summary

In this video we have seen:

- type-safe persistent actor state
- using the Saga pattern to model transactions