Functional API for defining type safe, reliable Akka actors

NOKIA Bell Labs

Scalæ by the Bay

2016-11-12

Daniel Urban

Research Engineer

daniel.urban@nokia.com

durban

isorecursive

urban.dani@gmail.com

36A8 2002 483A 4CBF A5F8
DF6F 48B2 9573 BF19 7B13

"… making the world a better place through a telco-optimized platform as a service solution." ;-)

"… making the world a better place through a telco-optimized platform as a service solution." ;-)

- Telco PaaS
  - making distributed systems easy
  - telco-specific optimizations, …

"… making the world a better place through a telco-optimized platform as a service solution." ;-)

- Telco PaaS
    - making distributed systems easy
    - telco-specific optimizations, …
- prototype implementation in Scala

"... making the world a better place through a telco-optimized platform as a service solution." ;-)

- Telco PaaS
  - making distributed systems easy
  - telco-specific optimizations, ...
- prototype implementation in Scala
- using Akka as a *distributed systems toolkit*

- Define an actor's behavior by overriding `receive`:

```scala
class MyActor extends Actor {
  private var pings = 0
  override def receive = {
    case "ping" ⇒
      pings += 1
      sender ! "pong"
    case _ ⇒ // ignore
  }
}
```

- Define an actor's behavior by overriding `receive`:

```
class MyActor extends Actor {
  private var pings = 0
  override def receive = {
    case "ping" ⇒
      pings += 1
      sender ! "pong"
    case _ ⇒ // ignore
  }
}
```

- `receive : PartialFunction[Any, Unit]`
  - we might receive anything

- Define an actor's behavior by overriding `receive`:

```
class MyActor extends Actor {
  private var pings = 0
  override def receive = {
    case "ping" ⇒
      pings += 1
      sender ! "pong"
    case _ ⇒ // ignore
  }
}
```

- `receive : PartialFunction[Any, Unit]`
  - we might receive anything
  - we do our job by performing side-effects

Akka Actors

- Define an actor's behavior by overriding `receive`:

```
class MyActor extends Actor {
  private var pings = 0
  override def receive = {
    case "ping" ⇒
      pings += 1
      sender ! "pong"
    case _ ⇒ // ignore
  }
}
```

- `receive : PartialFunction[Any, Unit]`
  - we might receive anything
  - we do our job by performing side-effects
  - we can send *anything* to *anybody* we have the "address" of

- Define an actor's behavior by overriding `receive`:

```
class MyActor extends Actor {
  private var pings = 0
  override def receive = {
    case "ping" ⇒
      pings += 1
      sender ! "pong"
    case _ ⇒ // ignore
  }
}
```

- `receive : PartialFunction[Any, Unit]`
  - we might receive anything
  - we do our job by performing side-effects
  - we can send *anything* to *anybody* we have the "address" of
- It'd be nice to have more static guarantees ...

Akka Typed

- Experimental module, exists since Akka 2.4.0 (2015-09-30)

- Experimental module, exists since Akka 2.4.0 (2015-09-30)
- Define an actor's behavior with a function, which computes the *next behavior* (this is how we can change our state):

```scala
case class MyMsg(s: String, replyTo: ActorRef[String])
def myBehavior(pings: Int = 0): Behavior[MyMsg] = {
  Total { // the message is always a `MyMsg`:
    case MyMsg("ping", r) ⇒
      r ! "pong" // we're allowed to send `String`s
      myBehavior(pings + 1)
    case MyMsg(_, _) ⇒ Same
  }
}
```

- Experimental module, exists since Akka 2.4.0 (2015-09-30)
- Define an actor's behavior with a function, which computes the *next behavior* (this is how we can change our state):

```
case class MyMsg(s: String, replyTo: ActorRef[String])
def myBehavior(pings: Int = 0): Behavior[MyMsg] = {
  Total { // the message is always a `MyMsg`:
    case MyMsg("ping", r) ⇒
      r ! "pong" // we're allowed to send `String`s
      myBehavior(pings + 1)
    case MyMsg(_, _) ⇒ Same
  }
}
```

- Experimental module, exists since Akka 2.4.0 (2015-09-30)
- Define an actor's behavior with a function, which computes the *next behavior* (this is how we can change our state):

```
case class MyMsg(s: String, replyTo: ActorRef[String])
def myBehavior(pings: Int = 0): Behavior[MyMsg] = {
  Total { // the message is always a `MyMsg`:
    case MyMsg("ping", r) ⇒
      r ! "pong" // we're allowed to send `String`s
      myBehavior(pings + 1)
    case MyMsg(_, _) ⇒ Same
  }
}
```

- `Behavior[M]` ≅ `(M ⇒ Behavior[M])`

Akka Typed

- Experimental module, exists since Akka 2.4.0 (2015-09-30)
- Define an actor's behavior with a function, which computes the *next behavior* (this is how we can change our state):

```scala
case class MyMsg(s: String, replyTo: ActorRef[String])
def myBehavior(pings: Int = 0): Behavior[MyMsg] = {
  Total { // the message is always a `MyMsg`:
    case MyMsg("ping", r) ⇒
      r ! "pong" // we're allowed to send `String`s
      myBehavior(pings + 1)
    case MyMsg(_, _) ⇒ Same
  }
}
```

- `Behavior[M]` ≅ `(M ⇒ Behavior[M])`
  - statically type checked messages (send and receive side)

- Experimental module, exists since Akka 2.4.0 (2015-09-30)
- Define an actor's behavior with a function, which computes the *next behavior* (this is how we can change our state):

```
case class MyMsg(s: String, replyTo: ActorRef[String])
def myBehavior(pings: Int = 0): Behavior[MyMsg] = {
  Total { // the message is always a `MyMsg`:
    case MyMsg("ping", r) ⇒
      r ! "pong" // we're allowed to send `String`s
      myBehavior(pings + 1)
    case MyMsg(_, _) ⇒ Same
  }
}
```

- `Behavior[M]` $\cong$ `(M ⇒ Behavior[M])`
  - statically type checked messages (send and receive side)
  - fewer side-effects $\Rightarrow$ easier reasoning

Akka Typed Status

- Experimental Akka module (i.e., no stable API)
    - under active development (e.g., new actor system implementation in 2.4.11)

Akka Typed Status

- Experimental Akka module (i.e., no stable API)
  - under active development (e.g., new actor system implementation in 2.4.11)
- At the moment, it is *not* integrated with some of the Akka modules, for example:
  - persistence
  - cluster sharding
  - streams

Akka Typed Status

- Experimental Akka module (i.e., no stable API)
  - under active development (e.g., new actor system implementation in 2.4.11)
- At the moment, it is *not* integrated with some of the Akka modules, for example:
  - persistence
  - cluster sharding
  - streams
- But it's *so* much better than regular (untyped) Akka, that we're trying to use it when possible ...

- Experimental Akka module (i.e., no stable API)
    - under active development (e.g., new actor system implementation in 2.4.11)
- At the moment, it is *not* integrated with some of the Akka modules, for example:
    - persistence
    - cluster sharding
    - streams
- But it's *so* much better than regular (untyped) Akka, that we're trying to use it when possible …
    - … so we needed to integrate it with these modules.

Akka Typed Status

- Experimental Akka module (i.e., no stable API)
  - under active development (e.g., new actor system implementation in 2.4.11)
- At the moment, it is *not* integrated with some of the Akka modules, for example:
  - persistence
  - cluster sharding
  - streams
- But it's *so* much better than regular (untyped) Akka, that we're trying to use it when possible …
  - … so we needed to integrate it with these modules.
  - Let's take a look at how we did it with Akka Persistence.

Persistence API

- Event sourcing:

Untyped Persistence API

- Event sourcing:
    - persist: append *event* to storage

Untyped Persistence API

- Event sourcing:
  - persist: append *event* to storage
  - recover: replay the events

Untyped Persistence API

- Event sourcing:
  - persist: append *event* to storage
  - recover: replay the events
- Akka (untyped) event sourcing API: inheritance + overriding

```scala
class MyActor extends PersistentActor {
  private var pings = 0
  override def receiveCommand = {
    case "ping" ⇒ this.persist(Increment(1)) { ev ⇒ // callback
      pings += ev.amount // side-effect
      sender ! s"${pings} pings so far" }
    case _ ⇒
  }
  override def receiveRecover: PartialFunction[Any, Unit] = {
    case Increment(amount) ⇒ pings += amount
  }
}
```

Untyped Persistence API

- Event sourcing:
  - persist: append *event* to storage
  - recover: replay the events
- Akka (untyped) event sourcing API: inheritance + overriding

```scala
class MyActor extends PersistentActor {
  private var pings = 0
  override def receiveCommand = {
    case "ping" ⇒ this.persist(Increment(1)) { ev ⇒ // callback
      pings += ev.amount // side-effect
      sender ! s"${pings} pings so far" }
    case _ ⇒
  }
  override def receiveRecover: PartialFunction[Any, Unit] = {
    case Increment(amount) ⇒ pings += amount
  }
}
```

Untyped Persistence API

- Event sourcing:
  - persist: append *event* to storage
  - recover: replay the events
- Akka (untyped) event sourcing API: inheritance + overriding

```scala
class MyActor extends PersistentActor {
  private var pings = 0
  override def receiveCommand = {
    case "ping" ⇒ this.persist(Increment(1)) { ev ⇒ // callback
      pings += ev.amount // side-effect
      sender ! s"${pings} pings so far" }
    case _ ⇒
  }
  override def receiveRecover: PartialFunction[Any, Unit] = {
    case Increment(amount) ⇒ pings += amount
  }
}
```

Untyped Persistence API

- Event sourcing:
  - persist: append *event* to storage
  - recover: replay the events
- Akka (untyped) event sourcing API: inheritance + overriding

```
class MyActor extends PersistentActor {
  private var pings = 0
  override def receiveCommand = {
    case "ping" ⇒ this.persist(Increment(1)) { ev ⇒ // callback
      pings += ev.amount // side-effect
      sender ! s"${pings} pings so far" }
    case _ ⇒
  }
  override def receiveRecover: PartialFunction[Any, Unit] = {
    case Increment(amount) ⇒ pings += amount
  }
}
```

Untyped Persistence API

- Event sourcing:
  - persist: append *event* to storage
  - recover: replay the events
- Akka (untyped) event sourcing API: inheritance + overriding

```scala
class MyActor extends PersistentActor {
  private var pings = 0
  override def receiveCommand = {
    case "ping" ⇒ this.persist(Increment(1)) { ev ⇒ // callback
      pings += ev.amount // side-effect
      sender ! s"${pings} pings so far" }
    case _ ⇒
  }
  override def receiveRecover: PartialFunction[Any, Unit] = {
    case Increment(amount) ⇒ pings += amount
  }
}
```

Untyped Persistence API

- Event sourcing:
  - persist: append *event* to storage
  - recover: replay the events
- Akka (untyped) event sourcing API: inheritance + overriding

```scala
class MyActor extends PersistentActor {
  private var pings = 0
  override def receiveCommand = {
    case "ping" ⇒ this.persist(Increment(1)) { ev ⇒ // callback
      pings += ev.amount // side-effect
      sender ! s"${pings} pings so far" }
    case _ ⇒
  }
  override def receiveRecover: PartialFunction[Any, Unit] = {
    case Increment(amount) ⇒ pings += amount
  }
}
```

Problem Statement

We'd like a persistence API for Akka Typed, which

- is statically type safe

Problem Statement

We'd like a persistence API for Akka Typed, which

- is statically type safe
- can be used in a `Behavior`

Problem Statement

We'd like a persistence API for Akka Typed, which

- is statically type safe
- can be used in a `Behavior`
- doesn't rely on side-effects

Problem Statement

We'd like a persistence API for Akka Typed, which

- is statically type safe
- can be used in a `Behavior`
- doesn't rely on side-effects

Original idea:

- described in an Akka issue on GitHub [4]

We'd like a persistence API for Akka Typed, which

- is statically type safe
- can be used in a `Behavior`
- doesn't rely on side-effects

Original idea:

- described in an Akka issue on GitHub [4]
- *persistence is an "actor effect"*

Problem Statement

We'd like a persistence API for Akka Typed, which

- is statically type safe
- can be used in a **`Behavior`**
- doesn't rely on side-effects

Original idea:

- described in an Akka issue on GitHub [4]
- *persistence is an "actor effect"*
- our implementation is based on the same general idea
  (although possibly not in the way it was originally meant)

Our Persistence API

- The same example, with our typed API:

```
case class MyMsg(s: String, replyTo: ActorRef[String])
val myBehavior = Persistent[MyMsg, Increment, Int](0) { state ⇒ ctx ⇒ {
  case MyMsg("ping", r) ⇒ for {
    state ← ctx.apply(Increment(1))
  } yield {
    r ! s"${state} pings so far"
    state
  }
  case MyMsg(_, _) ⇒ ctx.same
}}
```

- Note:
  - static typing of events
  - `for`-comprehension (no callbacks, no side-effects)

- Persistence is an effect $\Rightarrow$ we can use, e.g., monads to handle it

API Design

- Persistence is an effect $\Rightarrow$ we can use, e.g., monads to handle it
- We need a generic type: let's call it `Proc[_]`
  - short for "actor (persistence) *process*"

- Persistence is an effect $\Rightarrow$ we can use, e.g., monads to handle it
- We need a generic type: let's call it `Proc[_]`
  - short for "actor (persistence) *process*"
  - type parameter:
    - the result of handling a message, i.e. the next actor *state*

API Design

- Persistence is an effect $\Rightarrow$ we can use, e.g., monads to handle it
- We need a generic type: let's call it `Proc[_]`
  - short for "actor (persistence) *process*"
  - type parameter:
    - the result of handling a message, i.e. the next actor *state*
    - we'll have `M => Proc[S]` (instead of `M => Behavior[M]`)

API Design

- Persistence is an effect ⇒ we can use, e.g., monads to handle it
- We need a generic type: let's call it `Proc[_]`
    - short for "actor (persistence) *process*"
    - type parameter:
        - the result of handling a message, i.e. the next actor *state*
        - we'll have `M => Proc[S]` (instead of `M => Behavior[M]`)
        - design decision: explicit state (will help with snapshots)

API Design

- Persistence is an effect ⇒ we can use, e.g., monads to handle it
- We need a generic type: let's call it `Proc[_]`
    - short for "actor (persistence) *process*"
    - type parameter:
        - the result of handling a message, i.e. the next actor *state*
        - we'll have `M => Proc[S]` (instead of `M => Behavior[M]`)
        - design decision: explicit state (will help with snapshots)
- We'd like to have the following operations:

API Design

- Persistence is an effect ⇒ we can use, e.g., monads to handle it
- We need a generic type: let's call it `Proc[_]`
    - short for "actor (persistence) *process*"
    - type parameter:
        - the result of handling a message, i.e. the next actor *state*
        - we'll have `M => Proc[S]` (instead of `M => Behavior[M]`)
        - design decision: explicit state (will help with snapshots)
- We'd like to have the following operations:
    - *persist* an event, and *apply* it to the current state

API Design

- Persistence is an effect ⇒ we can use, e.g., monads to handle it
- We need a generic type: let's call it `Proc[_]`
  - short for "actor (persistence) *process*"
  - type parameter:
    - the result of handling a message, i.e. the next actor *state*
    - we'll have `M => Proc[S]` (instead of `M => Behavior[M]`)
    - design decision: explicit state (will help with snapshots)
- We'd like to have the following operations:
  - *persist* an event, and *apply* it to the current state
  - save a *snapshot* of the current state

API Design

- Persistence is an effect ⇒ we can use, e.g., monads to handle it
- We need a generic type: let's call it `Proc[_]`
    - short for "actor (persistence) *process*"
    - type parameter:
        - the result of handling a message, i.e. the next actor *state*
        - we'll have `M ⇒ Proc[S]` (instead of `M ⇒ Behavior[M]`)
        - design decision: explicit state (will help with snapshots)
- We'd like to have the following operations:
    - *persist* an event, and *apply* it to the current state
    - save a *snapshot* of the current state
    - keep the *same* state (no-op)

API Design

- Persistence is an effect ⇒ we can use, e.g., monads to handle it
- We need a generic type: let's call it `Proc[_]`
  - short for "actor (persistence) *process*"
  - type parameter:
    - the result of handling a message, i.e. the next actor *state*
    - we'll have `M => Proc[S]` (instead of `M => Behavior[M]`)
    - design decision: explicit state (will help with snapshots)
- We'd like to have the following operations:
  - *persist* an event, and *apply* it to the current state
  - save a *snapshot* of the current state
  - keep the *same* state (no-op)
  - *stop* the actor

API Design

- Persistence is an effect ⇒ we can use, e.g., monads to handle it
- We need a generic type: let's call it `Proc[_]`
  - short for "actor (persistence) *process*"
  - type parameter:
    - the result of handling a message, i.e. the next actor *state*
    - we'll have `M => Proc[S]` (instead of `M => Behavior[M]`)
    - design decision: explicit state (will help with snapshots)
- We'd like to have the following operations:
  - *persist* an event, and *apply* it to the current state
  - save a *snapshot* of the current state
  - keep the *same* state (no-op)
  - *stop* the actor
  - *handle* persistence errors

API Design

- Persistence is an effect ⇒ we can use, e.g., monads to handle it
- We need a generic type: let's call it `Proc[_]`
  - short for "actor (persistence) *process*"
  - type parameter:
    - the result of handling a message, i.e. the next actor *state*
    - we'll have `M => Proc[S]` (instead of `M => Behavior[M]`)
    - design decision: explicit state (will help with snapshots)
- We'd like to have the following operations:
  - *persist* an event, and *apply* it to the current state
  - save a *snapshot* of the current state
  - keep the *same* state (no-op)
  - *stop* the actor
  - *handle* persistence errors
- We'll need 3 types: `msg: M, event: E, state: S`

So we'd like to have something like this:

```scala
sealed trait Proc[A] {
  def map[B](f: A ⇒ B): Proc[B]
  def flatMap[B](f: A ⇒ Proc[B]): Proc[B]
}
```

The Proc Monad

So we'd like to have something like this:

```scala
sealed trait Proc[A] {
  def map[B](f: A ⇒ B): Proc[B]
  def flatMap[B](f: A ⇒ Proc[B]): Proc[B]
}

def pure[A](a: A): Proc[A] // lift a value ⤳ the value
```

So we'd like to have something like this:

```
sealed trait Proc[A] {
  def map[B](f: A ⇒ B): Proc[B]
  def flatMap[B](f: A ⇒ Proc[B]): Proc[B]
}

def pure[A](a: A): Proc[A] // lift a value ⤳ the value
def apply(event: E): Proc[S]
  // persist and apply event ⤳ updated state
```

The Proc Monad

So we'd like to have something like this:

```scala
sealed trait Proc[A] {
  def map[B](f: A ⇒ B): Proc[B]
  def flatMap[B](f: A ⇒ Proc[B]): Proc[B]
}

def pure[A](a: A): Proc[A] // lift a value ⤳ the value
def apply(event: E): Proc[S]
  // persist and apply event ⤳ updated state
def snapshot: Proc[S] // take a snapshot
```

The Proc Monad

So we'd like to have something like this:

```scala
sealed trait Proc[A] {
  def map[B](f: A ⇒ B): Proc[B]
  def flatMap[B](f: A ⇒ Proc[B]): Proc[B]
}
```

```scala
def pure[A](a: A): Proc[A] // lift a value ⇝ the value
def apply(event: E): Proc[S]
  // persist and apply event ⇝ updated state
def snapshot: Proc[S] // take a snapshot
def same: Proc[S] // no-op (for convenience)
```

So we'd like to have something like this:

```
sealed trait Proc[A] {
  def map[B](f: A ⇒ B): Proc[B]
  def flatMap[B](f: A ⇒ Proc[B]): Proc[B]
}

def pure[A](a: A): Proc[A] // lift a value ⤳ the value
def apply(event: E): Proc[S]
  // persist and apply event ⤳ updated state
def snapshot: Proc[S] // take a snapshot
def same: Proc[S] // no-op (for convenience)
def stop: Proc[S] // stop the actor
```

The Proc Monad

So we'd like to have something like this:

```
sealed trait Proc[A] {
  def map[B](f: A ⇒ B): Proc[B]
  def flatMap[B](f: A ⇒ Proc[B]): Proc[B]
}

def pure[A](a: A): Proc[A] // lift a value ⇝ the value
def apply(event: E): Proc[S]
  // persist and apply event ⇝ updated state
def snapshot: Proc[S] // take a snapshot
def same: Proc[S] // no-op (for convenience)
def stop: Proc[S] // stop the actor
def attempt[A](p: Proc[A]): Proc[Try[A]] // "catch"
```

So we'd like to have something like this:

```scala
sealed trait Proc[A] {
  def map[B](f: A ⇒ B): Proc[B]
  def flatMap[B](f: A ⇒ Proc[B]): Proc[B]
}

sealed trait PersistenceContext[E, S] {
  def pure[A](a: A): Proc[A]
  def apply(event: E): Proc[S]
  def snapshot: Proc[S]
  def same: Proc[S]
  def stop: Proc[S]
  def attempt[A](p: Proc[A]): Proc[Try[A]]
}
```

Actor Definition API

- We'll use this to define an actor (behavior):

```scala
def Persistent[M, E, S](initialState: S)(
  f: S ⇒ PersistenceContext[E, S] ⇒ M ⇒ Proc[S]
): PersistentBehavior[M, E, S]
```

Actor Definition API

- We'll use this to define an actor (behavior):

```scala
def Persistent[M, E, S](initialState: S)(
  f: S ⇒ PersistenceContext[E, S] ⇒ M ⇒ Proc[S]
): PersistentBehavior[M, E, S]
```

- As we've seen before:

```scala
val myBehavior = Persistent[MyMsg, Increment, Int](0) { state ⇒ ctx ⇒ {
  case MyMsg("ping", r) ⇒ for {
    state ← ctx.apply(Increment(1))
  } yield {
    r ! s"${state} pings so far"
    state
  }
  case MyMsg(_, _) ⇒ ctx.same
}}
```

Actor Definition API

- We'll use this to define an actor (behavior):

```
def Persistent[M, E, S](initialState: S)(
  f: S ⇒ PersistenceContext[E, S] ⇒ M ⇒ Proc[S]
): PersistentBehavior[M, E, S]
```

- As we've seen before:

```
val myBehavior = Persistent[MyMsg, Increment, Int](0) { state ⇒ ctx ⇒ {
  case MyMsg("ping", r) ⇒ for {
    state ← ctx.apply(Increment(1))
  } yield {
    r ! s"${state} pings so far"
    state
  }
  case MyMsg(_, _) ⇒ ctx.same
}}
```

Actor Definition API

- We'll use this to define an actor (behavior):

```scala
def Persistent[M, E, S](initialState: S)(
  f: S ⇒ PersistenceContext[E, S] ⇒ M ⇒ Proc[S]
): PersistentBehavior[M, E, S]
```

- As we've seen before:

```scala
val myBehavior = Persistent[MyMsg, Increment, Int](0) { state ⇒ ctx ⇒ {
  case MyMsg("ping", r) ⇒ for {
    state ← ctx.apply(Increment(1))
  } yield {
    r ! s"${state} pings so far"
    state
  }
  case MyMsg(_, _) ⇒ ctx.same
}}
```

Actor Definition API

- We'll use this to define an actor (behavior):

```
def Persistent[M, E, S](initialState: S)(
  f: S ⇒ PersistenceContext[E, S] ⇒ M ⇒ Proc[S]
): PersistentBehavior[M, E, S]
```

- As we've seen before:

```
val myBehavior = Persistent[MyMsg, Increment, Int](0) { state ⇒ ctx ⇒ {
  case MyMsg("ping", r) ⇒ for {
    state ← ctx.apply(Increment(1))
  } yield {
    r ! s"${state} pings so far"
    state
  }
  case MyMsg(_, _) ⇒ ctx.same
}}
```

Actor Definition API

- We'll use this to define an actor (behavior):

```
def Persistent[M, E, S](initialState: S)(
  f: S ⇒ PersistenceContext[E, S] ⇒ M ⇒ Proc[S]
): PersistentBehavior[M, E, S]
```

- As we've seen before:

```
val myBehavior = Persistent[MyMsg, Increment, Int](0) { state ⇒ ctx ⇒ {
  case MyMsg("ping", r) ⇒ for {
    state ← ctx.apply(Increment(1))
  } yield {
    r ! s"${state} pings so far"
    state
  }
  case MyMsg(_, _) ⇒ ctx.same
}}
```

Updating the State

- Event sourcing:
  - creating an *event* from the message (and the current state)

Updating the State

- Event sourcing:
    - creating an *event* from the message (and the current state)
    - computing the *new state* from the event (and the current state)

Updating the State

- Event sourcing:
  - creating an *event* from the message (and the current state)
  - computing the *new state* from the event (and the current state)
  - i.e., we'll need a function of type `(S, E) ⇒ S`

Updating the State

- Event sourcing:
  - creating an *event* from the message (and the current state)
  - computing the *new state* from the event (and the current state)
  - i.e., we'll need a function of type `(S, E) ⇒ S`
  - we could add this as another argument of the `Persistent` method …

Updating the State

- Event sourcing:
  - creating an *event* from the message (and the current state)
  - computing the *new state* from the event (and the current state)
  - i.e., we'll need a function of type `(S, E) => S`
  - we could add this as another argument of the `Persistent` method ...
- But we can use a type class instead:

Updating the State

- Event sourcing:
  - creating an *event* from the message (and the current state)
  - computing the *new state* from the event (and the current state)
  - i.e., we'll need a function of type `(S, E) => S`
  - we could add this as another argument of the `Persistent` method ...
- But we can use a type class instead:

```
trait Update[S, E] {
  def update(state: S, event: E): S
}
```

Updating the State

- Event sourcing:
  - creating an *event* from the message (and the current state)
  - computing the *new state* from the event (and the current state)
  - i.e., we'll need a function of type `(S, E) ⇒ S`
  - we could add this as another argument of the `Persistent` method ...
- But we can use a type class instead:

```scala
trait Update[S, E] {
  def update(state: S, event: E): S
}
```

  - we require it on `PersistenceContext.apply`:

```scala
def apply(event: E)(implicit u: Update[S, E]): Proc[S]
```

Updating the State

- Event sourcing:
  - creating an *event* from the message (and the current state)
  - computing the *new state* from the event (and the current state)
  - i.e., we'll need a function of type `(S, E) => S`
  - we could add this as another argument of the `Persistent` method …
- But we can use a type class instead:

```
trait Update[S, E] {
  def update(state: S, event: E): S
}
```

  - we require it on `PersistenceContext.apply`:

```
def apply(event: E)(implicit u: Update[S, E]): Proc[S]
```

  - so we'll have to implement it for our example S and E:

```
implicit val myUpdater: Update[Int, Increment] =
  Update.instance(_ + _.amount)
```

Recovery

Recovery

- Untyped API had `receiveRecover: PartialFunction[Any, Unit]`
  - updates the state using the replayed event

Recovery

- Untyped API had `receiveRecover: PartialFunction[Any, Unit]`
  - updates the state using the replayed event
- `Update[S, E]` ≅ `(S, E) ⇒ S`

Recovery

- Untyped API had `receiveRecover: PartialFunction[Any, Unit]`
  - updates the state using the replayed event
- `Update[S, E]` ≅ `(S, E) => S`
  - we can use our `Update` instance for recovery too!

Recovery

- Untyped API had `receiveRecover: PartialFunction[Any, Unit]`
  - updates the state using the replayed event
- `Update[S, E]` $\cong$ `(S, E) ⇒ S`
  - we can use our `Update` instance for recovery too!
  - the actor definition API (`Persistent`) will simply require an instance

Recovery

- Untyped API had `receiveRecover: PartialFunction[Any, Unit]`
  - updates the state using the replayed event
- `Update[S, E]` ≅ `(S, E) ⇒ S`
  - we can use our `Update` instance for recovery too!
  - the actor definition API (`Persistent`) will simply require an instance
- Our final actor definition API:

```scala
def Persistent[M, E, S](initialState: S)(
  f: S ⇒ PersistenceContext[E, S] ⇒ M ⇒ Proc[S]
)(implicit m: Update[S, E]): PersistentBehavior[M, E, S]
```

Recovery

- Untyped API had `receiveRecover: PartialFunction[Any, Unit]`
  - updates the state using the replayed event
- `Update[S, E]` ≅ `(S, E) ⇒ S`
  - we can use our `Update` instance for recovery too!
  - the actor definition API (`Persistent`) will simply require an instance
- Our final actor definition API:

```scala
def Persistent[M, E, S](initialState: S)(
  f: S ⇒ PersistenceContext[E, S] ⇒ M ⇒ Proc[S]
)(implicit m: Update[S, E]): PersistentBehavior[M, E, S]
```

- Summary:

Recovery

- Untyped API had `receiveRecover: PartialFunction[Any, Unit]`
  - updates the state using the replayed event
- `Update[S, E]` ≅ `(S, E) ⇒ S`
  - we can use our `Update` instance for recovery too!
  - the actor definition API (`Persistent`) will simply require an instance
- Our final actor definition API:

```
def Persistent[M, E, S](initialState: S)(
  f: S ⇒ PersistenceContext[E, S] ⇒ M ⇒ Proc[S]
)(implicit m: Update[S, E]): PersistentBehavior[M, E, S]
```

- Summary:
  - no boilerplate in the actor for changing state

Recovery

- Untyped API had `receiveRecover: PartialFunction[Any, Unit]`
  - updates the state using the replayed event
- `Update[S, E]` ≅ `(S, E) ⇒ S`
  - we can use our `Update` instance for recovery too!
  - the actor definition API (`Persistent`) will simply require an instance
- Our final actor definition API:

```
def Persistent[M, E, S](initialState: S)(
  f: S ⇒ PersistenceContext[E, S] ⇒ M ⇒ Proc[S]
)(implicit m: Update[S, E]): PersistentBehavior[M, E, S]
```

- Summary:
  - no boilerplate in the actor for changing state
  - simple recovery API

Recovery

- Untyped API had `receiveRecover: PartialFunction[Any, Unit]`
  - updates the state using the replayed event
- `Update[S, E]` ≅ `(S, E) ⇒ S`
  - we can use our `Update` instance for recovery too!
  - the actor definition API (`Persistent`) will simply require an instance
- Our final actor definition API:

```
def Persistent[M, E, S](initialState: S)(
  f: S ⇒ PersistenceContext[E, S] ⇒ M ⇒ Proc[S]
)(implicit m: Update[S, E]): PersistentBehavior[M, E, S]
```

- Summary:
  - no boilerplate in the actor for changing state
  - simple recovery API
  - extensible for user-defined state and event types

Implementation (sketch)

Outline

We'll have to implement:

Outline

We'll have to implement:

- The `Proc[_]` monad

Outline

We'll have to implement:

- The `Proc[_]` monad
    - the necessary methods: `pure`, `flatMap`, (`map`)

Outline

We'll have to implement:

- The `Proc[_]` monad
  - the necessary methods: `pure`, `flatMap`, (`map`)
  - other discussed factory methods and combinators: `apply`, `snapshot`, `stop`, `attempt`, ... (i.e., the `PersistenceContext` trait)

We'll have to implement:

- The `Proc[_]` monad
  - the necessary methods: `pure`, `flatMap`, (`map`)
  - other discussed factory methods and combinators: `apply`, `snapshot`, `stop`, `attempt`, ... (i.e., the `PersistenceContext` trait)
- The actor definition API

Outline

We'll have to implement:

- The `Proc[_]` monad
  - the necessary methods: `pure`, `flatMap`, (`map`)
  - other discussed factory methods and combinators: `apply`, `snapshot`, `stop`, `attempt`, … (i.e., the `PersistenceContext` trait)
- The actor definition API
  - the `Persistent` factory method

Outline

We'll have to implement:

- The `Proc[_]` monad
  - the necessary methods: `pure`, `flatMap`, (`map`)
  - other discussed factory methods and combinators: `apply`, `snapshot`, `stop`, `attempt`, ... (i.e., the `PersistenceContext` trait)
- The actor definition API
  - the `Persistent` factory method
  - and its return type, the `PersistentBehavior` trait

Outline

We'll have to implement:

- The `Proc[_]` monad
  - the necessary methods: `pure`, `flatMap`, (`map`)
  - other discussed factory methods and combinators: `apply`, `snapshot`, `stop`, `attempt`, ... (i.e., the `PersistenceContext` trait)
- The actor definition API
  - the `Persistent` factory method
  - and its return type, the `PersistentBehavior` trait
    - it needs to be very similar to `akka.typed.Behavior`

We'll have to implement:

- The `Proc[_]` monad
    - the necessary methods: `pure`, `flatMap`, (`map`)
    - other discussed factory methods and combinators: `apply`, `snapshot`, `stop`, `attempt`, ... (i.e., the `PersistenceContext` trait)
- The actor definition API
    - the `Persistent` factory method
    - and its return type, the `PersistentBehavior` trait
        - it needs to be very similar to `akka.typed.Behavior`
    - and something which runs the behavior, and executes the `Proc`

Outline

We'll have to implement:

- The `Proc[_]` monad
  - the necessary methods: `pure`, `flatMap`, (`map`)
  - other discussed factory methods and combinators: `apply`, `snapshot`, `stop`, `attempt`, ... (i.e., the `PersistenceContext` trait)
- The actor definition API
  - the `Persistent` factory method
  - and its return type, the `PersistentBehavior` trait
    - it needs to be very similar to `akka.typed.Behavior`
  - and something which runs the behavior, and executes the `Proc`
    - i.e., it will have to be able to actually persist events

Implementing Proc

Implementing Proc

Let's use the free monad! (because it's easier, and we can have a separate interpreter)

Implementing Proc

Let's use the free monad! (because it's easier, and we can have a separate interpreter)

```scala
sealed trait ProcOp[A] // ← our "operations" ADT
object ProcOp {
  case class Apply[E, S](event: E) extends ProcOp[S]
  case class Stop[S]() extends ProcOp[S]
  case class Attempt[A](proc: Proc[A]) extends ProcOp[Try[A]]
  // ..., all possible operations
}
```

Let's use the free monad! (because it's easier, and we can have a separate interpreter)

```
sealed trait ProcOp[A] // ← our "operations" ADT
object ProcOp {
  case class Apply[E, S](event: E) extends ProcOp[S]
  case class Stop[S]() extends ProcOp[S]
  case class Attempt[A](proc: Proc[A]) extends ProcOp[Try[A]]
  // ..., all possible operations
}
```

Implementing Proc

Let's use the free monad! (because it's easier, and we can have a separate interpreter)

```
sealed trait ProcOp[A] // ← our "operations" ADT
object ProcOp {
  case class Apply[E, S](event: E) extends ProcOp[S]
  case class Stop[S]() extends ProcOp[S]
  case class Attempt[A](proc: Proc[A]) extends ProcOp[Try[A]]
  // ..., all possible operations
}
```

Implementing Proc

Let's use the free monad! (because it's easier, and we can have a separate interpreter)

```
sealed trait ProcOp[A] // ← our "operations" ADT
object ProcOp {
  case class Apply[E, S](event: E) extends ProcOp[S]
  case class Stop[S]() extends ProcOp[S]
  case class Attempt[A](proc: Proc[A]) extends ProcOp[Try[A]]
  // ..., all possible operations
}
```

Implementing Proc

Let's use the free monad! (because it's easier, and we can have a separate interpreter)

```scala
sealed trait ProcOp[A] // ← our "operations" ADT
object ProcOp {
  case class Apply[E, S](event: E) extends ProcOp[S]
  case class Stop[S]() extends ProcOp[S]
  case class Attempt[A](proc: Proc[A]) extends ProcOp[Try[A]]
  // ..., all possible operations
}
```

Implementing Proc

Let's use the free monad! (because it's easier, and we can have a separate interpreter)

```scala
sealed trait ProcOp[A] // ← our "operations" ADT
object ProcOp {
  case class Apply[E, S](event: E) extends ProcOp[S]
  case class Stop[S]() extends ProcOp[S]
  case class Attempt[A](proc: Proc[A]) extends ProcOp[Try[A]]
  // ..., all possible operations
}

type Proc[A] = Free[ProcOp, A] // ← our monadic type
```

Implementing Proc

Let's use the free monad! (because it's easier, and we can have a separate interpreter)

```
sealed trait ProcOp[A] // ← our "operations" ADT
object ProcOp {
  case class Apply[E, S](event: E) extends ProcOp[S]
  case class Stop[S]() extends ProcOp[S]
  case class Attempt[A](proc: Proc[A]) extends ProcOp[Try[A]]
  // ..., all possible operations
}

type Proc[A] = Free[ProcOp, A] // ← our monadic type

class PersistenceContextImpl[E, S]() extends PersistenceContext[E, S] {
  def apply(event: E): Proc[S] =
    Free.liftF(ProcOp.Apply(event))
  def stop: Proc[S] =
    Free.liftF(ProcOp.Stop())
  // ..., smart constructors to all operations
}
```

Implementing Proc

Let's use the free monad! (because it's easier, and we can have a separate interpreter)

```
sealed trait ProcOp[A] // ← our "operations" ADT
object ProcOp {
  case class Apply[E, S](event: E) extends ProcOp[S]
  case class Stop[S]() extends ProcOp[S]
  case class Attempt[A](proc: Proc[A]) extends ProcOp[Try[A]]
  // ..., all possible operations
}

type Proc[A] = Free[ProcOp, A] // ← our monadic type

class PersistenceContextImpl[E, S]() extends PersistenceContext[E, S] {
  def apply(event: E): Proc[S] =
    Free.liftF(ProcOp.Apply(event))
  def stop: Proc[S] =
    Free.liftF(ProcOp.Stop())
  // ..., smart constructors to all operations
}
```

Implementing Proc

Let's use the free monad! (because it's easier, and we can have a separate interpreter)

```
sealed trait ProcOp[A] // ← our "operations" ADT
object ProcOp {
  // (as before)
}

type Proc[A] = Free[ProcOp, A] // ← our monadic type

class PersistenceContextImpl[E, S]() extends PersistenceContext[E, S] {
  // (as before)
}
```

Implementing Proc

Let's use the free monad! (because it's easier, and we can have a separate interpreter)

```
sealed trait ProcOp[A] // ← our "operations" ADT
object ProcOp {
  // (as before)
}

type Proc[A] = Free[ProcOp, A] // ← our monadic type

class PersistenceContextImpl[E, S]() extends PersistenceContext[E, S] {
  // (as before)
}

// we have a monad (for free) ...
val myProc: Proc[Int] = for {
  _ ← ctx.apply(Increment(1))
  s ← ctx.stop
} yield s
```

Implementing Proc

Let's use the free monad! (because it's easier, and we can have a separate interpreter)

```
sealed trait ProcOp[A] // ← our "operations" ADT
object ProcOp {
  // (as before)
}

type Proc[A] = Free[ProcOp, A] // ← our monadic type

class PersistenceContextImpl[E, S]() extends PersistenceContext[E, S] {
  // (as before)
}

// we have a monad (for free) ...
val myProc: Proc[Int] = for {
  _ ← ctx.apply(Increment(1))
  s ← ctx.stop
} yield s

// ... but we still need an interpreter
```

How to Run a `Behavior`?

- For executing a `Proc[A]`, we need:

How to Run a `Behavior`?

- For executing a `Proc[A]`, we need:
  - an interpreter, which can actually persist events, access actor state, ...

How to Run a `Behavior`?

- For executing a `Proc[A]`, we need:
  - an interpreter, which can actually persist events, access actor state, ...
  - i.e., it needs access to a real (untyped) `PersistentActor`

How to Run a Behavior?

- For executing a `Proc[A]`, we need:
  - an interpreter, which can actually persist events, access actor state, ...
  - i.e., it needs access to a real (untyped) `PersistentActor`
- In Akka Typed, a `Behavior` is executed by an `ActorAdapter` (pseudocode):

```
class ActorAdapter[A](initialBehavior: Behavior[A]) extends Actor {
  var currentBehavior: Behavior[A] = initialBehavior
  override def receive = {
    case msg: A ⇒
      // Behavior.message(ActorContext[A], A): Behavior[A]
      currentBehavior = currentBehavior.message(this.ctx, msg)
  }
}
```

How to Run a Behavior?

- For executing a `Proc[A]`, we need:
  - an interpreter, which can actually persist events, access actor state, ...
  - i.e., it needs access to a real (untyped) `PersistentActor`
- In Akka Typed, a `Behavior` is executed by an `ActorAdapter` (pseudocode):

```scala
class ActorAdapter[A](initialBehavior: Behavior[A]) extends Actor {
  var currentBehavior: Behavior[A] = initialBehavior
  override def receive = {
    case msg: A ⇒
      // Behavior.message(ActorContext[A], A): Behavior[A]
      currentBehavior = currentBehavior.message(this.ctx, msg)
  }
}
```

How to Run a Behavior?

- For executing a `Proc[A]`, we need:
  - an interpreter, which can actually persist events, access actor state, ...
  - i.e., it needs access to a real (untyped) `PersistentActor`
- In Akka Typed, a `Behavior` is executed by an `ActorAdapter` (pseudocode):

```
class ActorAdapter[A](initialBehavior: Behavior[A]) extends Actor {
  var currentBehavior: Behavior[A] = initialBehavior
  override def receive = {
    case msg: A ⇒
      // Behavior.message(ActorContext[A], A): Behavior[A]
      currentBehavior = currentBehavior.message(this.ctx, msg)
  }
}
```

How to Run a Behavior?

- For executing a **Proc[A]**, we need:
  - an interpreter, which can actually persist events, access actor state, ...
  - i.e., it needs access to a real (untyped) **PersistentActor**
- In Akka Typed, a **Behavior** is executed by an **ActorAdapter** (pseudocode):

```scala
class ActorAdapter[A](initialBehavior: Behavior[A]) extends Actor {
  var currentBehavior: Behavior[A] = initialBehavior
  override def receive = {
    case msg: A ⇒
      // Behavior.message(ActorContext[A], A): Behavior[A]
      currentBehavior = currentBehavior.message(this.ctx, msg)
  }
}
```

- This is almost good, but we need it to

How to Run a Behavior?

- For executing a **Proc[A]**, we need:
  - an interpreter, which can actually persist events, access actor state, ...
  - i.e., it needs access to a real (untyped) **PersistentActor**
- In Akka Typed, a **Behavior** is executed by an **ActorAdapter** (pseudocode):

```
class ActorAdapter[A](initialBehavior: Behavior[A]) extends Actor {
  var currentBehavior: Behavior[A] = initialBehavior
  override def receive = {
    case msg: A ⇒
      // Behavior.message(ActorContext[A], A): Behavior[A]
      currentBehavior = currentBehavior.message(this.ctx, msg)
  }
}
```

- This is almost good, but we need it to
  - work with our **PersistentBehavior** (⇒ it will be a subtype of **Behavior**)

How to Run a Behavior?

- For executing a Proc[A], we need:
  - an interpreter, which can actually persist events, access actor state, ...
  - i.e., it needs access to a real (untyped) PersistentActor
- In Akka Typed, a Behavior is executed by an ActorAdapter (pseudocode):

```
class ActorAdapter[A](initialBehavior: Behavior[A]) extends Actor {
  var currentBehavior: Behavior[A] = initialBehavior
  override def receive = {
    case msg: A ⇒
      // Behavior.message(ActorContext[A], A): Behavior[A]
      currentBehavior = currentBehavior.message(this.ctx, msg)
  }
}
```

- This is almost good, but we need it to
  - work with our PersistentBehavior (⇒ it will be a subtype of Behavior)
  - be a PersistentActor (⇒ let's mix the two ...)

PersistentActorAdapter (sketch)

```scala
class PersistentActorAdapter[M, E, S](initial: PersistentBehavior[M, E, S])
                                     (implicit u: Update[S, E])
  extends ActorAdapter[M](initial) with PersistentActor { actor ⇒

  val interpreter = new (ProcOp ~> Task) {
    def apply[A](op: ProcOp[A]): Task[A] = op match {
      case op: ProcOp.Apply[E, S] ⇒
        Task.unforkedAsync[S] { callback ⇒
          actor.persist(op.event) { ev ⇒
            val newState: S = u.update(actor.currentState, ev)
            actor.changeState(newState)
            callback(Right(newState))
          }
        }
      // ... other operations
    }
  }
```

PersistentActorAdapter (sketch)

```
class PersistentActorAdapter[M, E, S](initial: PersistentBehavior[M, E, S])
                                     (implicit u: Update[S, E])
  extends ActorAdapter[M](initial) with PersistentActor { actor ⇒

  val interpreter = new (ProcOp ~> Task) {
    def apply[A](op: ProcOp[A]): Task[A] = op match {
      case op: ProcOp.Apply[E, S] ⇒
        Task.unforkedAsync[S] { callback ⇒
          actor.persist(op.event) { ev ⇒
            val newState: S = u.update(actor.currentState, ev)
            actor.changeState(newState)
            callback(Right(newState))
          }
        }
      // ... other operations
    }
  }
```

PersistentActorAdapter (sketch)

```scala
class PersistentActorAdapter[M, E, S](initial: PersistentBehavior[M, E, S])
                                     (implicit u: Update[S, E])
  extends ActorAdapter[M](initial) with PersistentActor { actor ⇒

  val interpreter = new (ProcOp ~> Task) {
    def apply[A](op: ProcOp[A]): Task[A] = op match {
      case op: ProcOp.Apply[E, S] ⇒
        Task.unforkedAsync[S] { callback ⇒
          actor.persist(op.event) { ev ⇒
            val newState: S = u.update(actor.currentState, ev)
            actor.changeState(newState)
            callback(Right(newState))
          }
        }
      // ... other operations
    }
  }
```

PersistentActorAdapter (sketch)

```
class PersistentActorAdapter[M, E, S](initial: PersistentBehavior[M, E, S])
                                     (implicit u: Update[S, E])
  extends ActorAdapter[M](initial) with PersistentActor { actor ⇒

  val interpreter = new (ProcOp ~> Task) {
    def apply[A](op: ProcOp[A]): Task[A] = op match {
      case op: ProcOp.Apply[E, S] ⇒
        Task.unforkedAsync[S] { callback ⇒
          actor.persist(op.event) { ev ⇒
            val newState: S = u.update(actor.currentState, ev)
            actor.changeState(newState)
            callback(Right(newState))
          }
        }
      // ... other operations
    }
  }
```

PersistentActorAdapter (sketch)

```scala
class PersistentActorAdapter[M, E, S](initial: PersistentBehavior[M, E, S])
                                     (implicit u: Update[S, E])
    extends ActorAdapter[M](initial) with PersistentActor { actor ⇒

  val interpreter = new (ProcOp ~> Task) // (... as  before)

  override def receiveCommand: PartialFunction[Any, Unit] = {
    case msg: M ⇒
      val proc: Proc[S] = currentBehavior.message(this.ctx, msg)
      val task: Task[S] = proc.foldMap(this.interpreter)
      task.unsafeRunAsync {
        case Right(newState) ⇒
          this.changeState(newState)
      }
  }
}
```

PersistentActorAdapter (sketch)

```scala
class PersistentActorAdapter[M, E, S](initial: PersistentBehavior[M, E, S])
                                       (implicit u: Update[S, E])
    extends ActorAdapter[M](initial) with PersistentActor { actor ⇒

  val interpreter = new (ProcOp ~> Task) // (... as  before)

  override def receiveCommand: PartialFunction[Any, Unit] = {
    case msg: M ⇒
      val proc: Proc[S] = currentBehavior.message(this.ctx, msg)
      val task: Task[S] = proc.foldMap(this.interpreter)
      task.unsafeRunAsync {
        case Right(newState) ⇒
          this.changeState(newState)
      }
  }
}
```

PersistentActorAdapter (sketch)

```
class PersistentActorAdapter[M, E, S](initial: PersistentBehavior[M, E, S])
                                     (implicit u: Update[S, E])
  extends ActorAdapter[M](initial) with PersistentActor { actor ⇒

 val interpreter = new (ProcOp ~> Task) // (... as  before)

 override def receiveCommand: PartialFunction[Any, Unit] = {
   case msg: M ⇒
     val proc: Proc[S] = currentBehavior.message(this.ctx, msg)
     val task: Task[S] = proc.foldMap(this.interpreter)
     task.unsafeRunAsync {
       case Right(newState) ⇒
         this.changeState(newState)
     }
 }
}
```

PersistentActorAdapter (sketch)

```scala
class PersistentActorAdapter[M, E, S](initial: PersistentBehavior[M, E, S])
                                     (implicit u: Update[S, E])
    extends ActorAdapter[M](initial) with PersistentActor { actor ⇒

  val interpreter = new (ProcOp ~> Task) // (... as  before)

  override def receiveCommand: PartialFunction[Any, Unit] = {
    case msg: M ⇒
      val proc: Proc[S] = currentBehavior.message(this.ctx, msg)
      val task: Task[S] = proc.foldMap(this.interpreter)
      task.unsafeRunAsync {
        case Right(newState) ⇒
          this.changeState(newState)
      }
  }
}
```

Summary

- We created an actor definition API, which

Summary

- We created an actor definition API, which
    - is statically type safe

Summary

- We created an actor definition API, which
    - is statically type safe
    - supports persistence (with event sourcing and snapshots)

Summary

- We created an actor definition API, which
  - is statically type safe
  - supports persistence (with event sourcing and snapshots)
  - doesn't rely on side-effects (for persistence, but see later)

Summary

- We created an actor definition API, which
  - is statically type safe
  - supports persistence (with event sourcing and snapshots)
  - doesn’t rely on side-effects (for persistence, but see later)
  - doesn’t require working with callbacks

Summary

- We created an actor definition API, which
  - is statically type safe
  - supports persistence (with event sourcing and snapshots)
  - doesn't rely on side-effects (for persistence, but see later)
  - doesn't require working with callbacks
  - is integrated with the existing Akka Typed API

Summary

- We created an actor definition API, which
  - is statically type safe
  - supports persistence (with event sourcing and snapshots)
  - doesn't rely on side-effects (for persistence, but see later)
  - doesn't require working with callbacks
  - is integrated with the existing Akka Typed API
- We implemented this API on top of Akka

Summary

- We created an actor definition API, which
  - is statically type safe
  - supports persistence (with event sourcing and snapshots)
  - doesn't rely on side-effects (for persistence, but see later)
  - doesn't require working with callbacks
  - is integrated with the existing Akka Typed API
- We implemented this API on top of Akka
  - by extending both `ActorAdapter` (`akka-typed`) and `PersistentActor` (`akka-persistence`),

Summary

- We created an actor definition API, which
  - is statically type safe
  - supports persistence (with event sourcing and snapshots)
  - doesn't rely on side-effects (for persistence, but see later)
  - doesn't require working with callbacks
  - is integrated with the existing Akka Typed API
- We implemented this API on top of Akka
  - by extending both `ActorAdapter` (`akka-typed`) and `PersistentActor` (`akka-persistence`),
  - and creating an interpreter for the `Proc` (free) monad.

Summary

- We created an actor definition API, which
  - is statically type safe
  - supports persistence (with event sourcing and snapshots)
  - doesn't rely on side-effects (for persistence, but see later)
  - doesn't require working with callbacks
  - is integrated with the existing Akka Typed API
- We implemented this API on top of Akka
  - by extending both `ActorAdapter` (`akka-typed`) and `PersistentActor` (`akka-persistence`),
  - and creating an interpreter for the `Proc` (free) monad.
- This way, we can create type safe persistent actors

Related Work

We also

We also

- created another interpreter for *testing* a persistent actor (without asynchrony or a database)

We also

- created another interpreter for *testing* a persistent actor (without asynchrony or a database)
- integrated our API with
  - Akka Cluster Sharding
  - Akka Streams

Open Sourcing

Coming soon ...

`https://github.com/nokia`

Future Work

Possible future work:

Future Work

Possible future work:

- The API is not *entirely* side-effect free:

Future Work

Possible future work:

- The API is not *entirely* side-effect free:
    - e.g., sending messages and spawning child actors are still side-effects (as they are in Akka Typed)

Future Work

Possible future work:

- The API is not *entirely* side-effect free:
  - e.g., sending messages and spawning child actors are still side-effects (as they are in Akka Typed)
  - we could include them in the **`Proc`** monad ...

Future Work

Possible future work:

- The API is not *entirely* side-effect free:
  - e.g., sending messages and spawning child actors are still side-effects (as they are in Akka Typed)
  - we could include them in the `Proc` monad ...
- The test interpreter is not entirely complete (e.g., no special support for testing the side-effects above)

?

References

- *Akka*. URL: http://akka.io.
- *Akka Typed*. URL: http://doc.akka.io/docs/akka/2.4.11/scala/typed.html.
- *Akka Persistence*. URL: http://doc.akka.io/docs/akka/2.4.11/scala/persistence.html.
- Roland Kuhn. *Issue comment about persistence*. URL: https://github.com/akka/akka-meta/issues/7#issuecomment-105602702.
- *Free Monad*. URL: http://typelevel.org/cats/datatypes/freemonad.html.
- *FS2 Task*. URL: https://github.com/functional-streams-for-scala/fs2/blob/v0.9.1/core/shared/src/main/scala/fs2/Task.scala.
- *Nokia Bell Labs*. URL: https://www.bell-labs.com.
- *Nokia on GitHub*. URL: https://github.com/nokia.