Acting Lessons for Scala engineers with AKKA and ZIO

By Salar Rahmanian

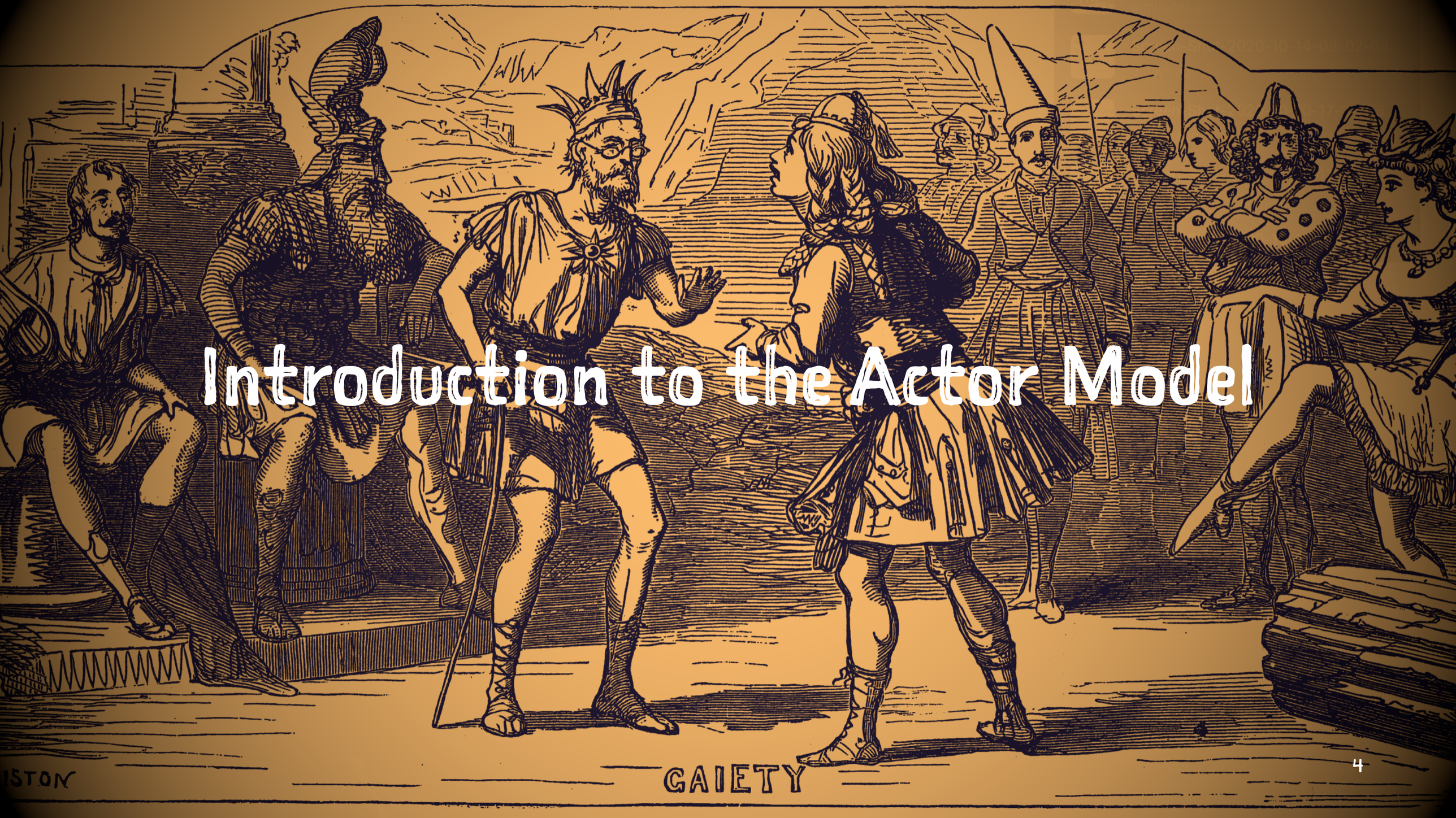
Follow me on Twitter: @SalarRahmanian - Blog: https://www.softinio.com

Agenda

- About Salar Rahmanian
- Introduction to the Actor Model
- Akka
- ZIO-Actors
- Future is Bright ☀️☀️☀️

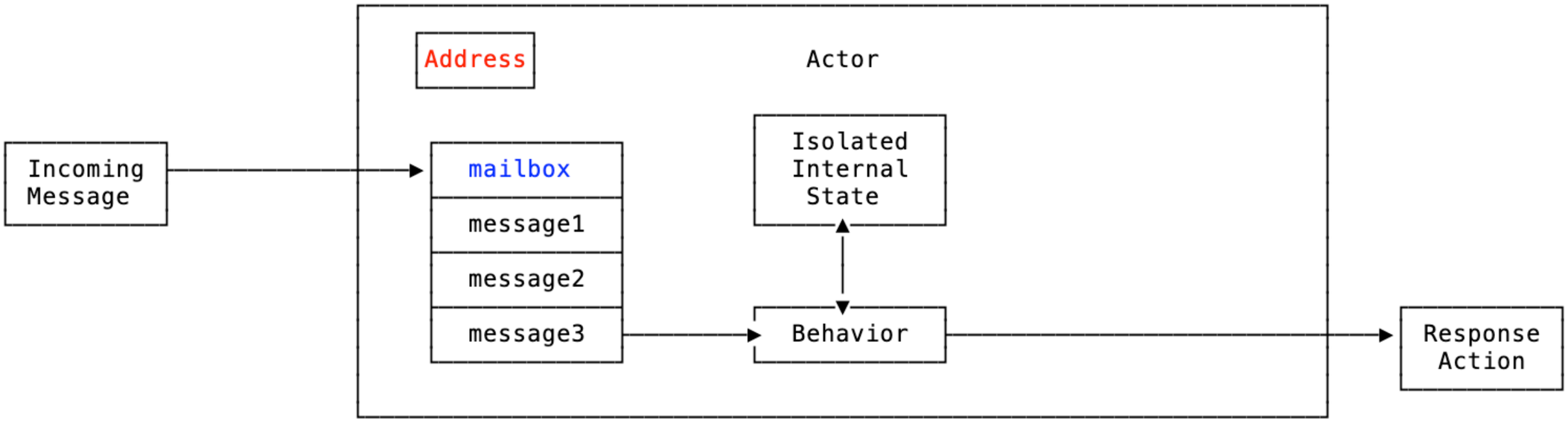
About Salar Rahmanian

- Also known as **softinio**
- Blog: **www.softinio.com**
- Twitter: **@SalarRahmanian**
- Stream on Twitch: **@softinio**
- Scala Engineer
- Trying to learn & get better at Functional Programming!



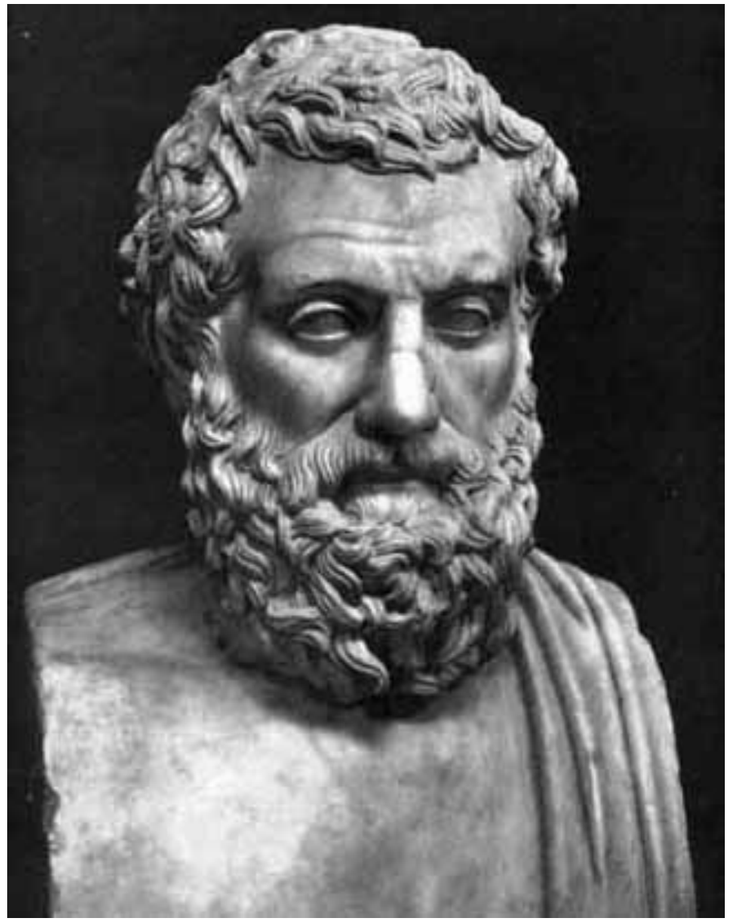
Introduction to the Actor Model

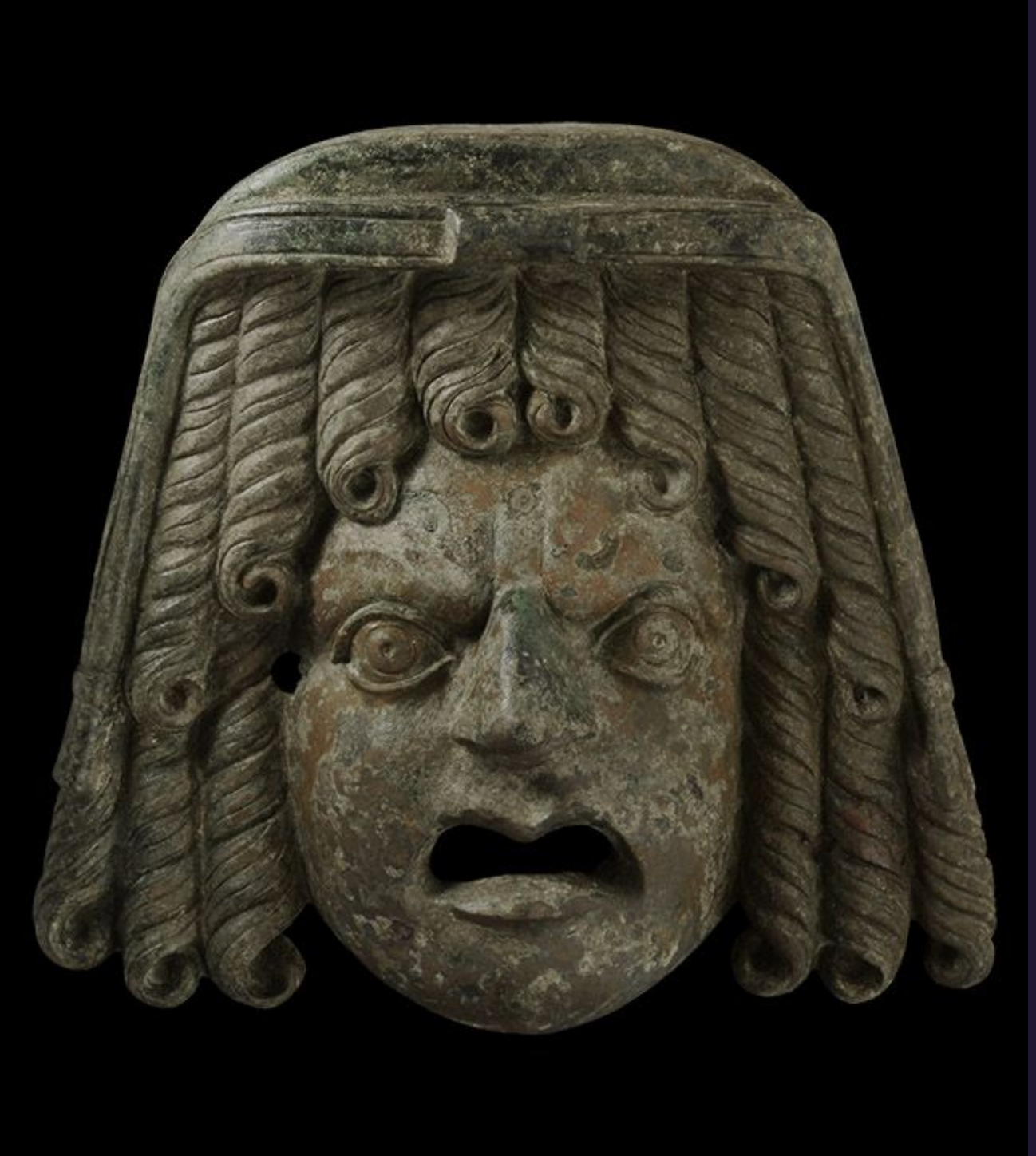
What is an Actor ?



Characteristics of an Actor

- Actors persist
- Actors have an isolated state
- 🦸‍♀️🦸 Super powers for building concurrent applications 🦹‍♀️🦹



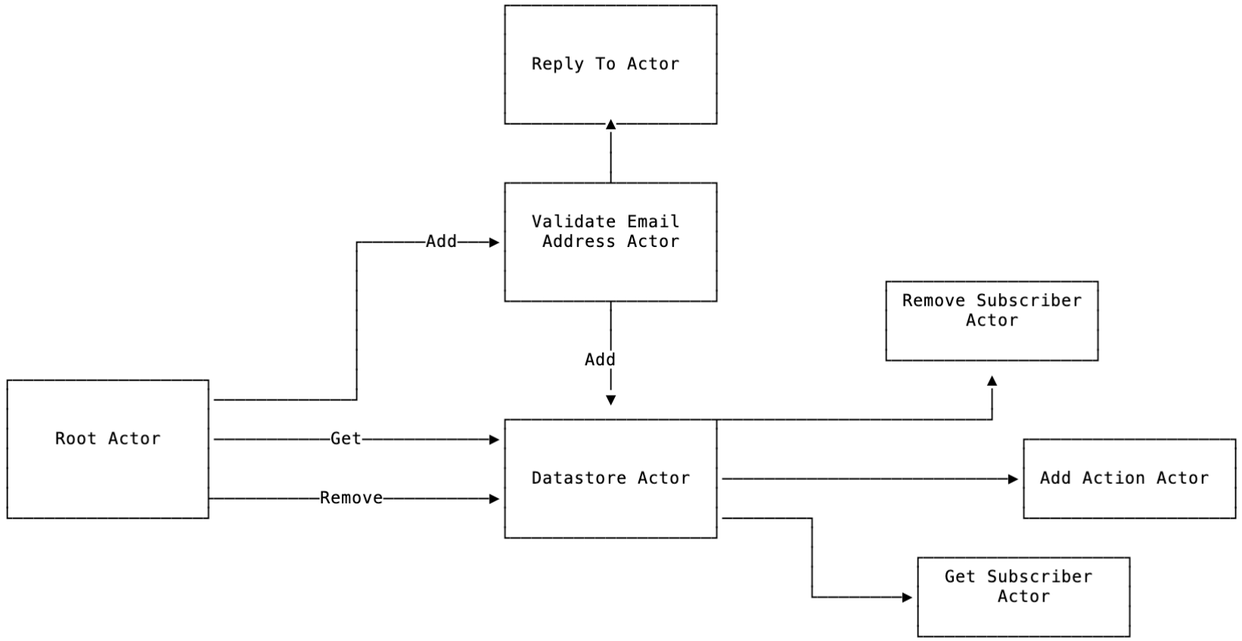


Handling Failure with Actors

akka

- Akka is a toolkit for building highly concurrent, distributed, and resilient message-driven applications for Java and Scala
- Implements the Actor Model
- Inspiration from Erlang
- Recently introduced Typed Actors (now the default)

Example Application using akka and Scala



Message Types used in our akka app

Command

Our actors will be sending 3 types of commands to determine whether we are adding, removing or fetching a person

```
sealed trait Command
final case object Add extends Command
final case object Remove extends Command
final case object Get extends Command
```

Message Types used in our akka app

Customer

Message type received by root actor for adding a person

```
final case class Customer(
    firstName: String,
    lastName: String,
    emailAddress: String
)
```

Message Types used in our akka app

Message

Message sent by root actor to add a new subscriber

```
final case class Message(
    firstName: String,
    lastName: String,
    emailAddress: String,
    command: Command,
    db: ActorRef[Message],
    replyTo: ActorRef[SubscribedMessage]
) {
  def isValid: Boolean = EmailValidator.getInstance().isValid(emailAddress)
}
```

Message Types used in our akka app

Subscribed Message

Message type sent to replyTo actor to confirm we are storing the customer.

```
final case class SubscribedMessage(
    subscriberId: Long,
    from: ActorRef[Message]
)
```

Validate Email Address Actor (akka)

```
object Subscriber {
  def apply(): Behavior[Message] = Behaviors.receive { (context, message) =>
    context.log.info(s"Validating ${message.firstName} ${message.lastName} with email ${message.emailAddress}!")
    if (message.isValid) {
      message.replyTo ! SubscribedMessage(1L, context.self)
      message.db ! message
    } else {
      context.log.info(s"Received an invalid message $message.emailAddress")
    }
    Behaviors.same
  }
}
```

Datastore Actor (akka)

```
object Datastore {
  def apply(): Behavior[Message] = Behaviors.receive { (context, message) =>
    context.log.info(s"Adding ${message.firstName} ${message.lastName} with email ${message.emailAddress}!")
    message.command match {
      case Add => println(s"Adding message with email: ${message.emailAddress}") // Send message to Add Action Actor
      case Remove => println(s"Removing message with email: ${message.emailAddress}") // Send message to Remove Subscriber Actor
      case Get => println(s"Getting message with email: ${message.emailAddress}") // Send message to Get Subscriber Actor
    }
    Behaviors.same
  }
}
```

Root Actor (akka)

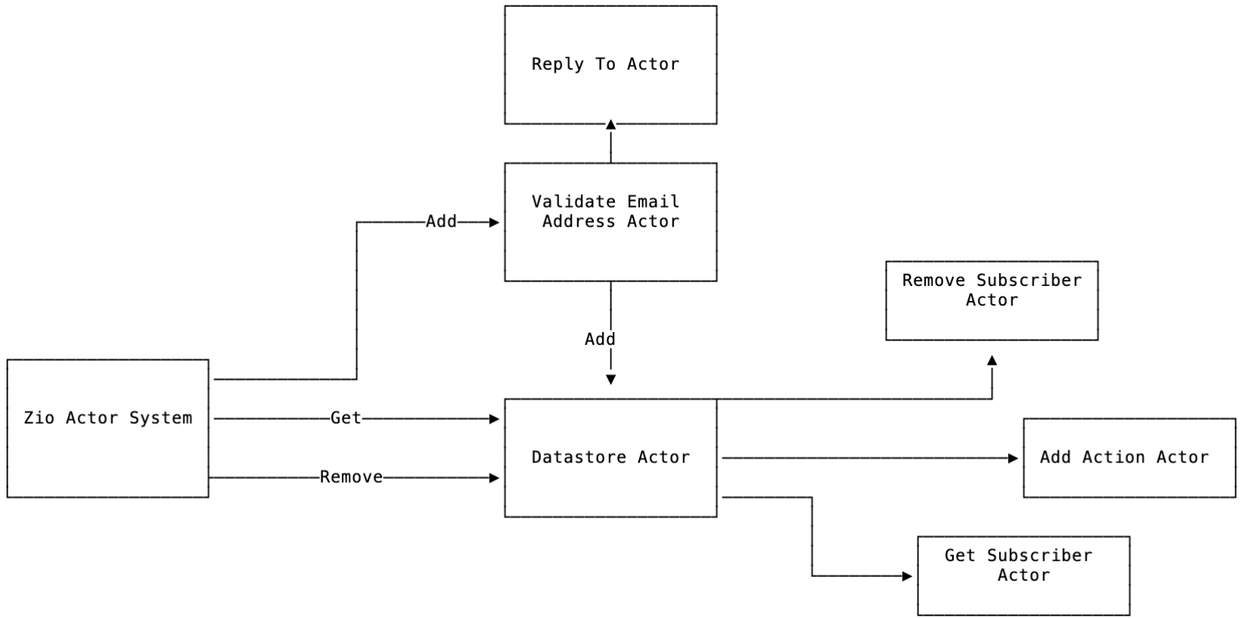
```
object ActorsMain {
  def apply(): Behavior[Customer] =
    Behaviors.setup { context =>
      val subscriber = context.spawn(Subscriber(), "subscriber")
      val db = context.spawn(Datastore(), "db")

      Behaviors.receiveMessage { message =>
        val replyTo = context.spawn(Reply(), "reply")
        subscriber ! Message(message.firstName,message.lastName,message.emailAddress,Add,db,replyTo)
        Behaviors.same
      }
    }
}

object Pat extends App {
  val actorsMain: ActorSystem[Customer] = ActorSystem(ActorsMain(), "PatSystem")
  actorsMain ! Customer("Salar", "Rahmanian", "code@softinio.com")
}
```

```scala
object ActorsMain {
  def apply(): Behavior[Customer] =
    Behaviors.setup { context =>
      val subscriber = context.spawn(Subscriber(), "subscriber")
      val db = context.spawn(Datastore(), "db")

      Behaviors.receiveMessage { message =>
        val replyTo = context.spawn(Reply(), "reply")
        subscriber ! Message(message.firstName,message.lastName,message.emailAddress,Add,db,replyTo)
        Behaviors.same
      }
    }
}

object Pat extends App {
  val actorsMain: ActorSystem[Customer] = ActorSystem(ActorsMain(), "PatSystem")
  actorsMain ! Customer("Salar", "Rahmanian", "code@softinio.com")
}
```

ZIO Actors

- a high-performance, purely-functional library for building, composing, and supervising typed actors
- backed by ZIO and Scala
- Models actor's communication without side effects
- Everything Typed

Example App using ZIO Actors & Scala



Message Types used in our ZIO app

Command

Our actors will be sending 3 types of commands to determine whether we are adding, removing or fetching a person

```
sealed trait Command
final case object Add extends Command
final case object Remove extends Command
final case object Get extends Command
```

Message Types used in our ZIO app

Protocol

```
sealed trait Protocol[+A]
```

Message Types used in our ZIO app

Customer

```
final case class Customer(
    firstName: String,
    lastName: String,
    emailAddress: String
) extends Protocol[Unit]
```

Message Types used in our ZIO app

Message

```
final case class Message(
    firstName: String,
    lastName: String,
    emailAddress: String,
    command: Command,
    db: ActorRef[Protocol],
    replyTo: ActorRef[Protocol]
) extends Protocol[Unit] {
  def isValid: UIO[Boolean] =
    UIO(EmailValidator.getInstance().isValid(emailAddress))
}
```

Message Types used in our ZIO app

SubscribedMessage

```
final case class SubscribedMessage(
    subscriberId: Long, from: ActorRef[Protocol]
) extends Protocol[Unit]
```

Message Types used in our ZIO app

InvalidEmailException

```
case class InvalidEmailException(
    msg: String
) extends Throwable
```

ZIO

`ZIO[R, E, A]`

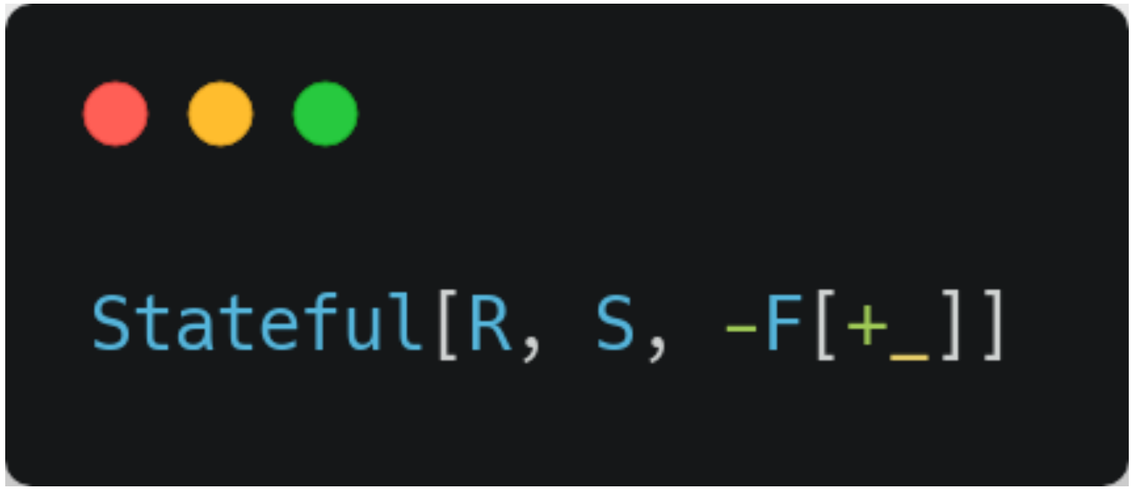
- R - Environment type
- E - Failure type
- A - Success

ZIO

Some ZIO Type aliases

```
UIO[A] is ZIO[Any, Nothing, A]
RIO[R,A] is ZIO[R, Throwable, A]
```

Implementing a behavior with ZIO Actors



- R represents the environment type (similar to R in `ZIO[R, E, A]`)
- S represents the state of the actor that gets updated after every message.
- F represents the message the actor will receive to process.

Validate Email Address Actor (ZIO)

```
val subscriber = new Stateful[Console, Unit, Protocol] {
  override def receive[A](
      state: Unit,
      protocol: Protocol[A],
      context: Context
  ): RIO[Console, (Unit, A)] =
    protocol match {
      case message: Message =>
        for {
          _ <- putStrLn(
            s"Validating ${message.firstName} ${message.lastName} with email ${message.emailAddress}!"
          )
          valid <- message.isValid
          self <- context.self[Protocol]
          _ <- message.replyTo ! SubscribedMessage(1L, self)
               if (valid)
          _ <- message.db ! message
               if (valid)
        } yield ((), ())
      case _ => IO.fail(InvalidEmailException("Failed"))
    }
}
```

```scala
val subscriber = new Stateful[Console, Unit, Protocol] {
  override def receive[A](
      state: Unit,
      protocol: Protocol[A],
      context: Context
  ): RIO[Console, (Unit, A)] =
    protocol match {
      case message: Message =>
        for {
          _ <- putStrLn(
            s"Validating ${message.firstName} ${message.lastName} with email ${message.emailAddress}!"
          )
          valid <- message.isValid
          self <- context.self[Protocol]
          _ <- message.replyTo ! SubscribedMessage(1L, self)
               if (valid)
          _ <- message.db ! message
               if (valid)
        } yield ((), ())
      case _ => IO.fail(InvalidEmailException("Failed"))
    }
}
```

```scala
val subscriber = new Stateful[Console, Unit, Protocol] {
  override def receive[A](
      state: Unit,
      protocol: Protocol[A],
      context: Context
  ): RIO[Console, (Unit, A)] =
    protocol match {
      case message: Message =>
        for {
          _ <- putStrLn(
            s"Validating ${message.firstName} ${message.lastName} with email ${message.emailAddress}!"
          )
          valid <- message.isValid
          self <- context.self[Protocol]
          _ <- message.replyTo ! SubscribedMessage(1L, self)
               if (valid)
          _ <- message.db ! message
               if (valid)
        } yield ((), ())
      case _ => IO.fail(InvalidEmailException("Failed"))
    }
}
```

```scala
val subscriber = new Stateful[Console, Unit, Protocol] {
  override def receive[A](
      state: Unit,
      protocol: Protocol[A],
      context: Context
  ): RIO[Console, (Unit, A)] =    <-- SAME AS ZIO[Console, Throwable, (Unit, A)]
    protocol match {
      case message: Message =>
        for {
          _ <- putStrLn(
            s"Validating ${message.firstName} ${message.lastName} with email ${message.emailAddress}!"
          )
          valid <- message.isValid
          self <- context.self[Protocol]
          _ <- message.replyTo ! SubscribedMessage(1L, self)
          if (valid)
          _ <- message.db ! message
          if (valid)
        } yield ((), ())
      case _ => IO.fail(InvalidEmailException("Failed"))
    }
}
```

```scala
val subscriber = new Stateful[Console, Unit, Protocol] {
  override def receive[A](
      state: Unit,
      protocol: Protocol[A],
      context: Context
  ): RIO[Console, (Unit, A)] =    <-- SAME AS ZIO[Console, Throwable, (Unit, A)]
    protocol match {
      case message: Message =>
        for {
          _ <- putStrLn(
            s"Validating ${message.firstName} ${message.lastName} with email ${message.emailAddress}!"
          )
          valid <- message.isValid
          self <- context.self[Protocol]
          _ <- message.replyTo ! SubscribedMessage(1L, self)
          if (valid)
          _ <- message.db ! message
          if (valid)
        } yield ((), ())
      case _ => IO.fail(InvalidEmailException("Failed"))
    }
}
```

```scala
val subscriber = new Stateful[Console, Unit, Protocol] {
  override def receive[A](
      state: Unit,
      protocol: Protocol[A],
      context: Context
  ): RIO[Console, (Unit, A)] =    <-- SAME AS ZIO[Console, Throwable, (Unit, A)]
    protocol match {
      case message: Message =>
        for {
          _ <- putStrLn(
            s"Validating ${message.firstName} ${message.lastName} with email ${message.emailAddress}!"
          )
          valid <- message.isValid
          self <- context.self[Protocol]
          _ <- message.replyTo ! SubscribedMessage(1L, self)
          if (valid)
          _ <- message.db ! message
          if (valid)
        } yield ((), ())
      case _ => IO.fail(InvalidEmailException("Failed"))
    }
}
```

Magical Functional Programming

```
ZIO[Console, Throwable, (Unit, A)]
```

ZIO will force you down the right path !

Datastore Actor (ZIO)

```
val datastore = new Stateful[Console, Unit, Protocol] {
  override def receive[A](
      state: Unit,
      protocol: Protocol[A],
      context: Context
  ): RIO[Console, (Unit, A)] =
    protocol match {
      case message: Message =>
        for {
          _ <- putStrLn(s"Processing Command")
          _ <- message.command match {
            case Add =>
              putStrLn(s"Adding message with email: ${message.emailAddress}")
            case Remove =>
              putStrLn(
                s"Removing message with email: ${message.emailAddress}"
              )
            case Get =>
              putStrLn(s"Getting message with email: ${message.emailAddress}")
          }
        } yield ((), ())
      case _ => IO.fail(InvalidEmailException("Failed"))
    }
}
```

```scala
val datastore = new Stateful[Console, Unit, Protocol] {
  override def receive[A](
      state: Unit,
      protocol: Protocol[A],
      context: Context
  ): RIO[Console, (Unit, A)] =
    protocol match {
      case message: Message =>
        for {
          _ <- putStrLn(s"Processing Command")
          _ <- message.command match {
            case Add =>
              putStrLn(s"Adding message with email: ${message.emailAddress}")
            case Remove =>
              putStrLn(
                s"Removing message with email: ${message.emailAddress}"
              )
            case Get =>
              putStrLn(s"Getting message with email: ${message.emailAddress}")
          }
        } yield ((), ())
      case _ => IO.fail(InvalidEmailException("Failed"))
    }
}
```

ZIO Actor System

```
val program = for {
  actorSystemRoot <- ActorSystem("salarTestActorSystem")
  subscriberActor <- actorSystemRoot.make("subscriberActor", Supervisor.none, (), subscriber)
  datastoreActor  <- actorSystemRoot.make("datastoreActor", Supervisor.none, (), datastore)
  replyActor      <- actorSystemRoot.make("replyActor", Supervisor.none, (), reply)
  _               <- subscriberActor ! Message(
                       "Salar",
                       "Rahmanian",
                       "code@softinio.com",
                       Add,
                       datastoreActor,
                       replyActor
                     )
  _               <- zio.clock.sleep(Duration.Infinity)
} yield ()
```

```scala
val program = for {
  actorSystemRoot <- ActorSystem("salarTestActorSystem")
  subscriberActor <- actorSystemRoot.make("subscriberActor", Supervisor.none, (), subscriber)
  datastoreActor  <- actorSystemRoot.make("datastoreActor", Supervisor.none, (), datastore)
  replyActor      <- actorSystemRoot.make("replyActor", Supervisor.none, (), reply)
  _               <- subscriberActor ! Message(
                       "Salar",
                       "Rahmanian",
                       "code@softinio.com",
                       Add,
                       datastoreActor,
                       replyActor
                     )
  _               <- zio.clock.sleep(Duration.Infinity)
} yield ()
```

```scala
val program = for {
  actorSystemRoot <- ActorSystem("salarTestActorSystem")
  subscriberActor <- actorSystemRoot.make("subscriberActor", Supervisor.none, (), subscriber)
  datastoreActor  <- actorSystemRoot.make("datastoreActor", Supervisor.none, (), datastore)
  replyActor      <- actorSystemRoot.make("replyActor", Supervisor.none, (), reply)
  _               <- subscriberActor ! Message(
                       "Salar",
                       "Rahmanian",
                       "code@softinio.com",
                       Add,
                       datastoreActor,
                       replyActor
                     )
  _               <- zio.clock.sleep(Duration.Infinity)
} yield ()
```

akka and ZIO Actors feature comparison

Feature	akka	ZIO Actors
Typed	✅	✅
No side effects	❌	✅
Supervisor	✅	✅
Persistance	✅	✅
Remoting	✅	✅
Diagnostics/Monitoring/Metrics	Lightbend Telemetry	zio-zmx + Promestheus
Receptionist	✅	WIP
Clustering/Distributed/Sharding	✅	WIP

akka and ZIO Actors feature comparison

Feature	akka	ZIO Actors
Java 8	✅	✅
Java 11	✅	✅
Scala 2.11	✅	✅
Scala 2.12	✅	✅
Scala 2.13	✅	✅
Scala 3	?	✅
scala.js	❌	✅ (next release)
scala native	❌	Evaluating

☀️☀️☀️ Future is Bright for the Actor model with Scala ☀️☀️☀️

- ZIO Actors has Interop with akka 🎊🎉🍾
- **YES YES** this means ability to send and receive messages between zio actors and akka typed actors!!!

Resources

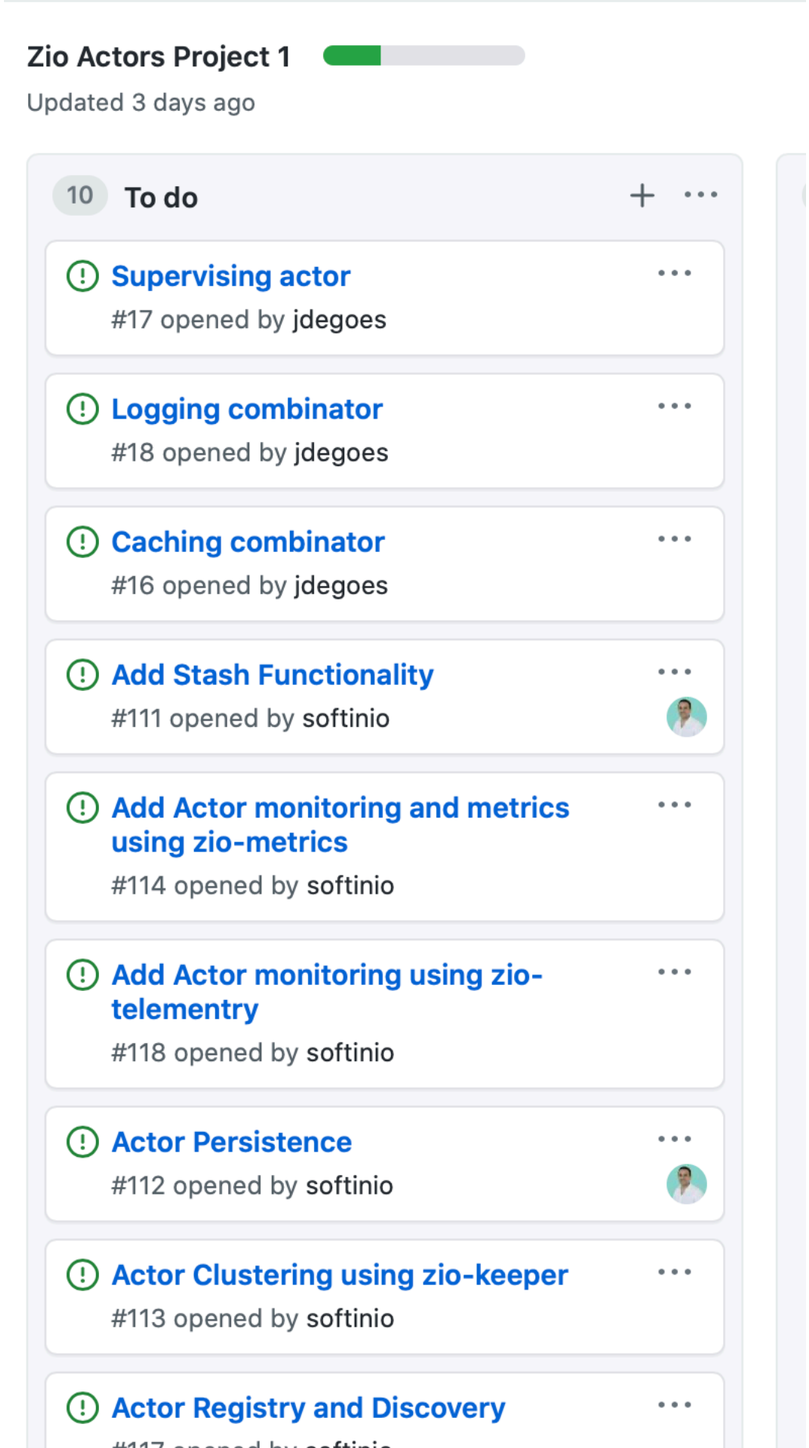
- My Sample Application used in this post: https://github.com/softinio/pat
- My Blog: https://www.softinio.com
- Akka Documentation: https://akka.io/docs/
- ZIO Actors Documentation https://zio.github.io/zio-actors/
- ZIO Documentation: https://zio.dev
- ZIO / ZIO Actors Discord: http://sca.la/ziodiscord

Contribute to ZIO Actors

- Repo: https://github.com/zio/zio-actors
- #zio-actors channel on ZIO Discord:

http://sca.la/ziodiscord

- First time contributors welcome and we will help!
- Look at Issues tab on GitHub



Thank you

Follow me on Twitter: @SalarRahmanian - Blog: https://www.softinio.com

Follow my stream on Twitch: https://www.twitch.tv/softinio