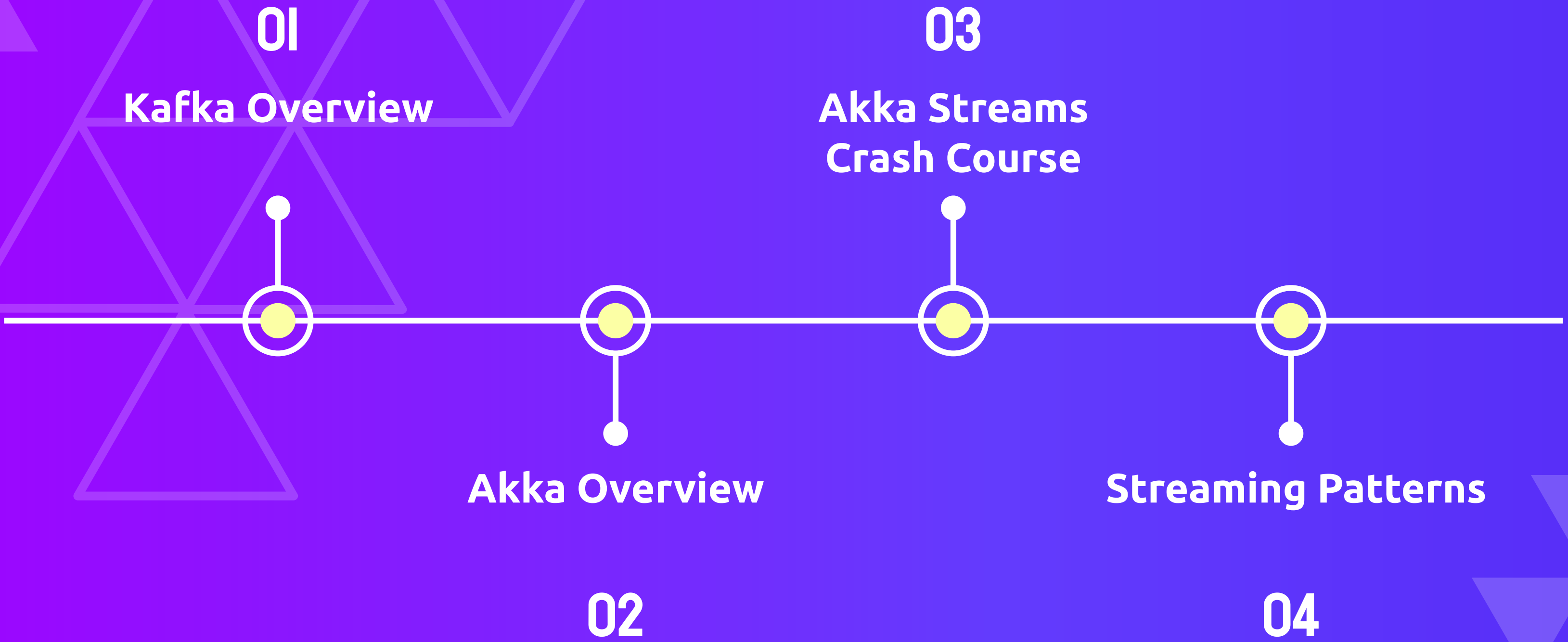
Microservices Patterns with Akka and Kafka

Talk Flow



01

Kafka Overview

02

Akka Overview

03

Akka Streams Crash Course

04

Streaming Patterns

About Me

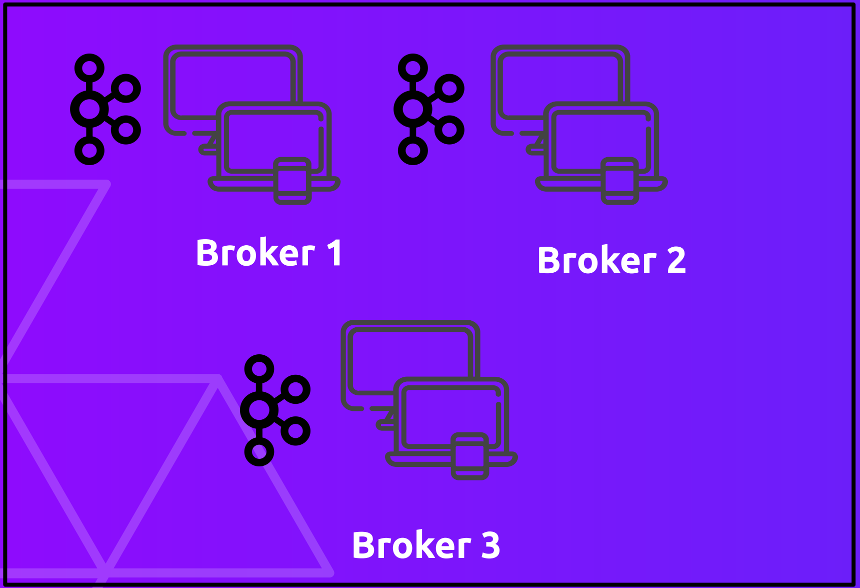
Aleksandar
Software Engineer @ SmartCat

Github: aleksandarskrbic
Twitter: skrbic_a
Blog: aleksandarskrbic.github.io
Mail: skrbic.alexa@gmail.com

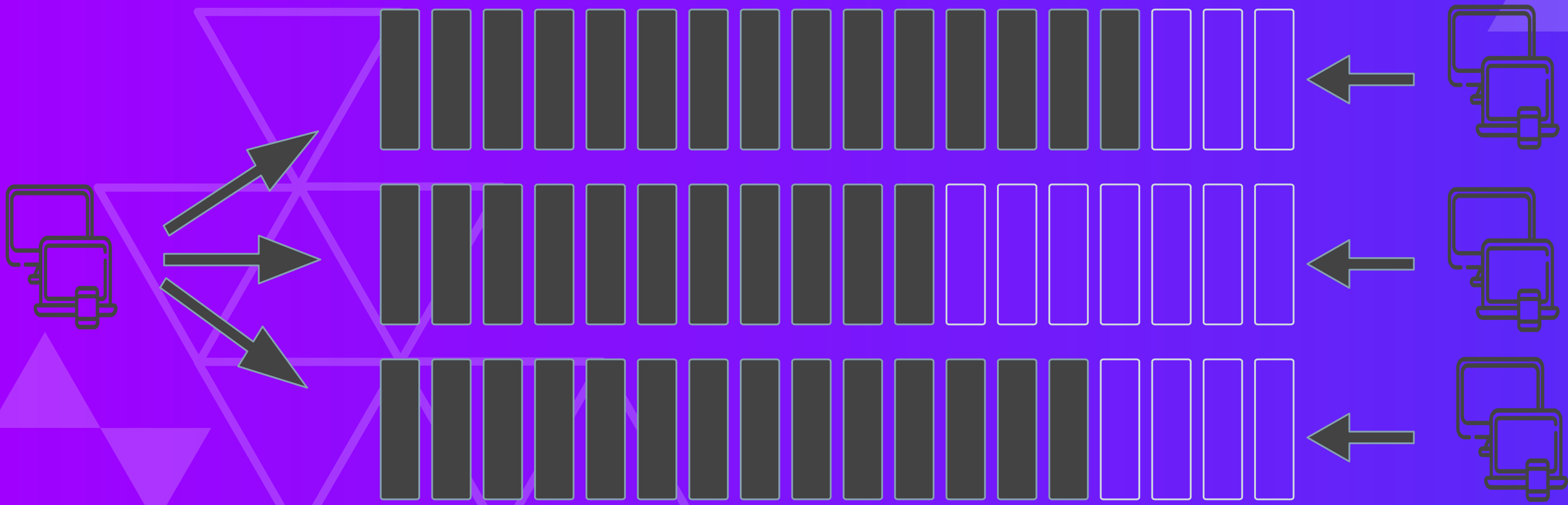
Kafka Overview

Apache Kafka is a distributed streaming platform

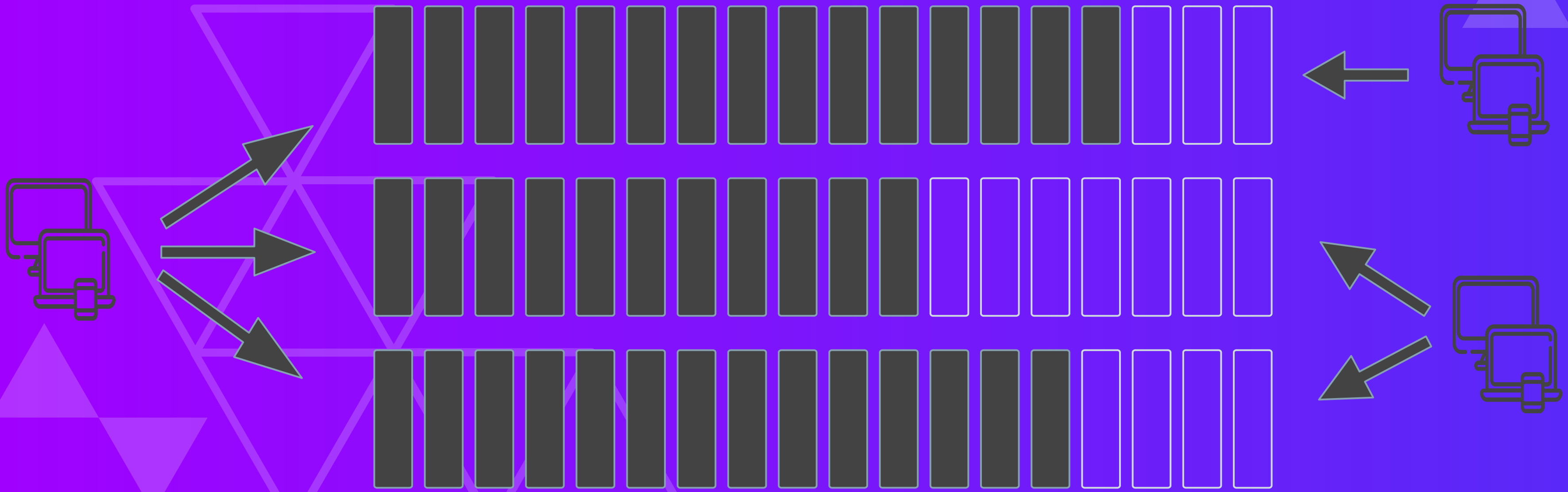
Kafka Architecture



Kafka Workflow (1/2)



Kafka Workflow (2/2)



Kafka Summary

- Message Broker
- Core API:
  - Producer
  - Consumer
  - Kafka Streams
  - Kafka Connect
- Common use cases:
  - Interservice communication
  - Real-time data pipelines
  - Stream processing

What is Akka?

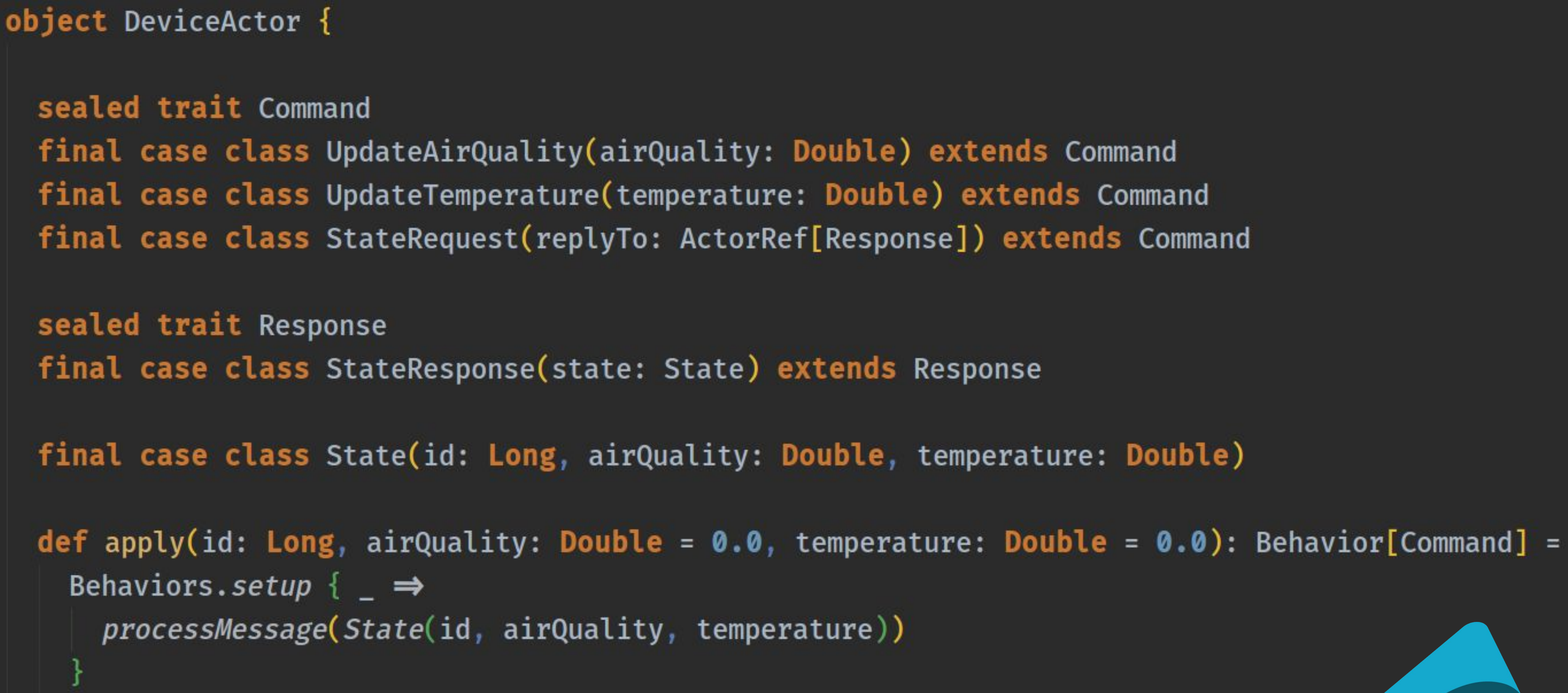
Akka is a toolkit for building highly concurrent, distributed, and resilient message-driven applications for Java and Scala

What is Akka offers?

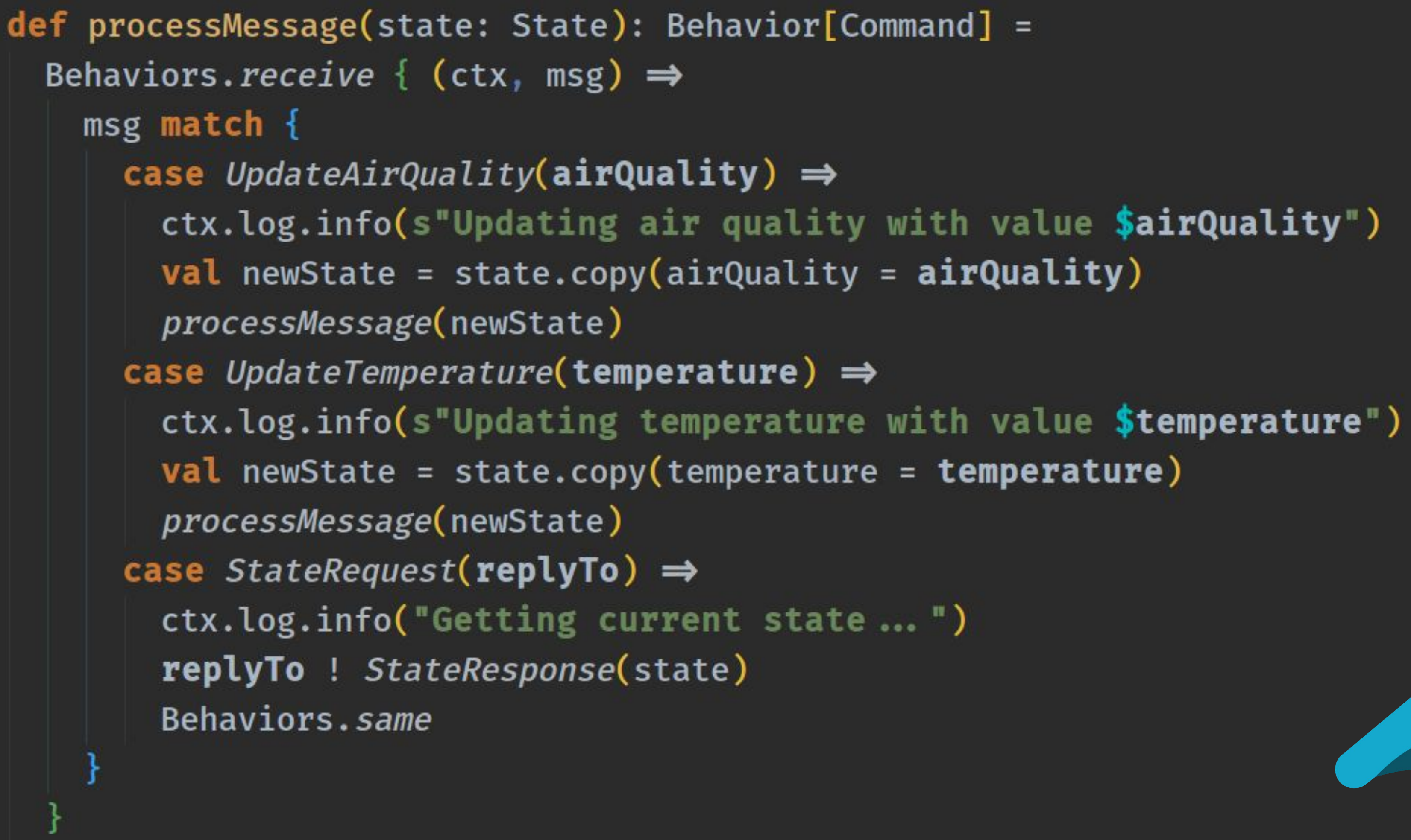
- Simpler concurrent programming
  - No need for locking or synchronization mechanism
- Simpler distributed computing
  - Distributed by default (no difference between local or remote actor)
- Great fault tolerance mechanism
  - Supervision hierarchy for error handling

The Actor Model

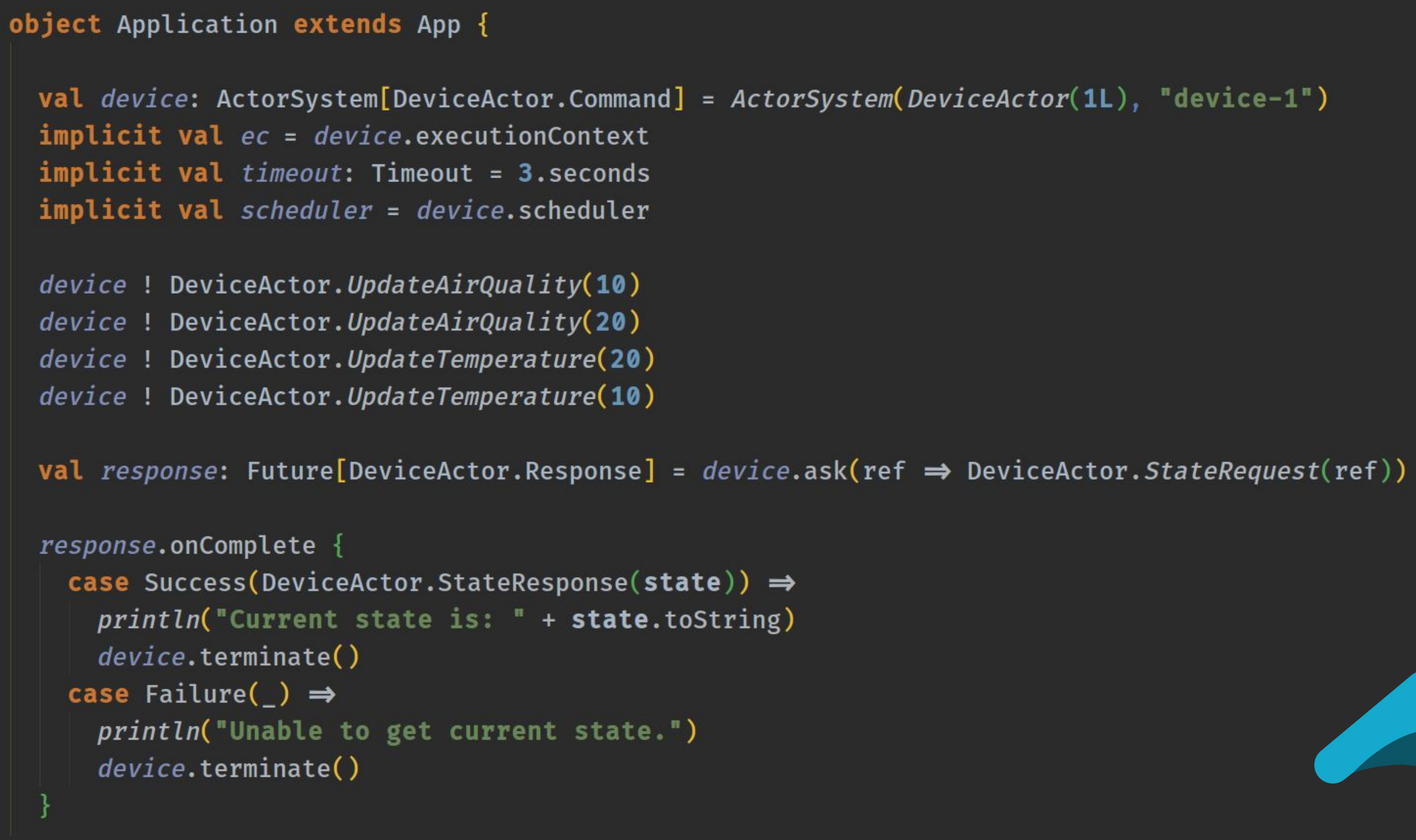
- Everything is an actor
- Every actor has an address
- When actor is handling a message it can:
  - Create new (child) actors
  - Send message to another actor
  - Mutate local state
  - Change the behaviour for handling next message



```scala
object DeviceActor {

  sealed trait Command
  final case class UpdateAirQuality(airQuality: Double) extends Command
  final case class UpdateTemperature(temperature: Double) extends Command
  final case class StateRequest(replyTo: ActorRef[Response]) extends Command

  sealed trait Response
  final case class StateResponse(state: State) extends Response

  final case class State(id: Long, airQuality: Double, temperature: Double)

  def apply(id: Long, airQuality: Double = 0.0, temperature: Double = 0.0): Behavior[Command] =
    Behaviors.setup { _ ⇒
      processMessage(State(id, airQuality, temperature))
    }
```

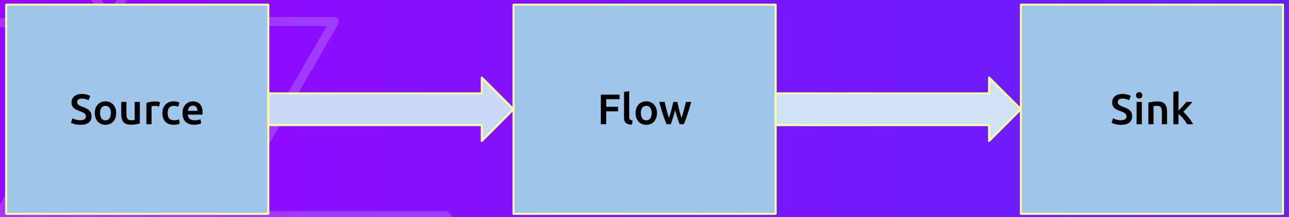


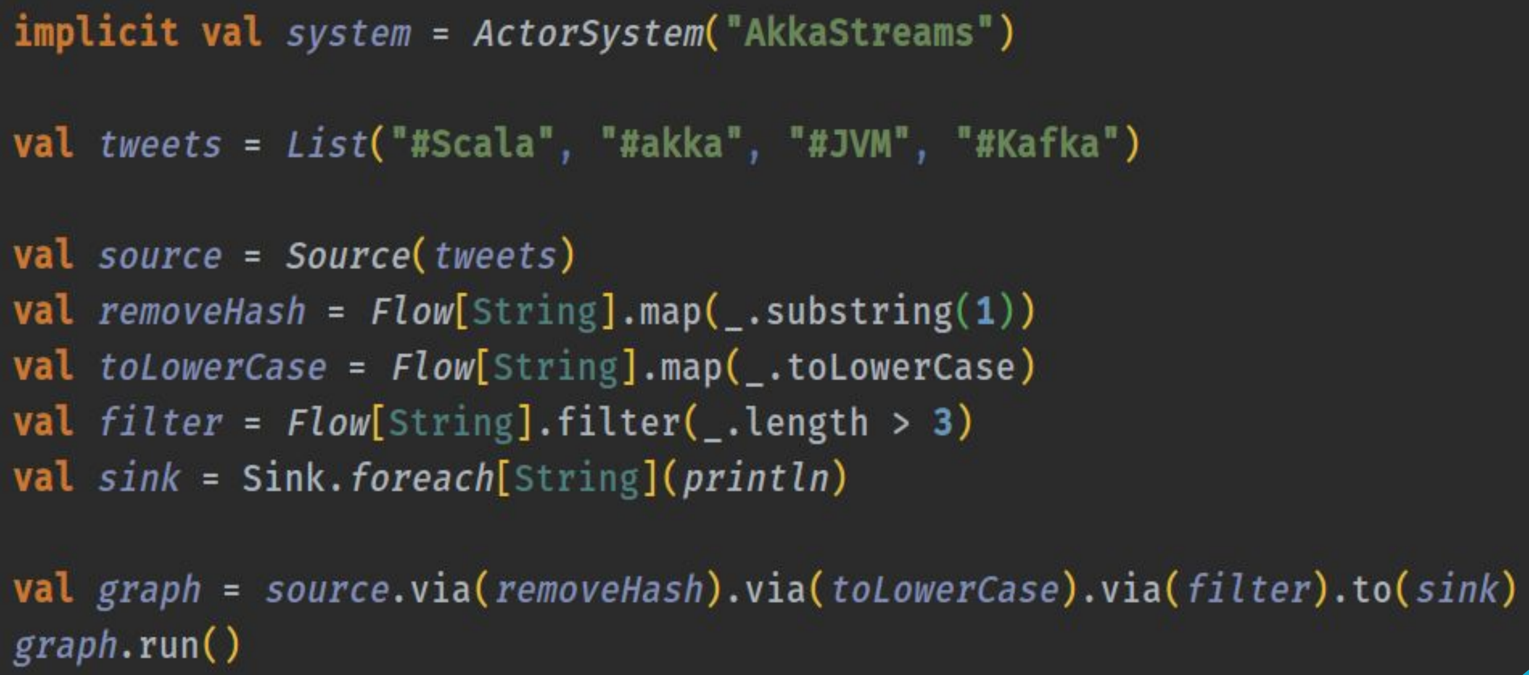
```scala
def processMessage(state: State): Behavior[Command] =
  Behaviors.receive { (ctx, msg) =>
    msg match {
      case UpdateAirQuality(airQuality) =>
        ctx.log.info(s"Updating air quality with value $airQuality")
        val newState = state.copy(airQuality = airQuality)
        processMessage(newState)
      case UpdateTemperature(temperature) =>
        ctx.log.info(s"Updating temperature with value $temperature")
        val newState = state.copy(temperature = temperature)
        processMessage(newState)
      case StateRequest(replyTo) =>
        ctx.log.info("Getting current state ... ")
        replyTo ! StateResponse(state)
        Behaviors.same
    }
  }
```



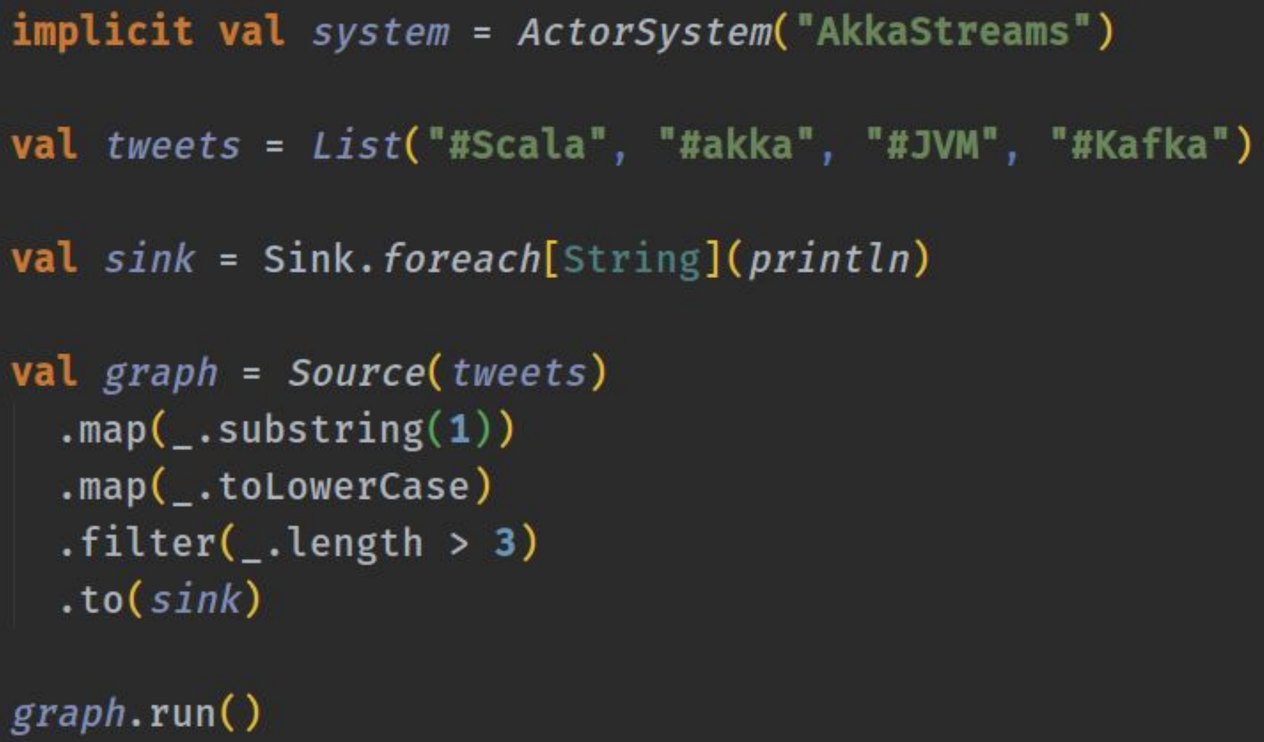
```scala
object Application extends App {

  val device: ActorSystem[DeviceActor.Command] = ActorSystem(DeviceActor(1L), "device-1")
  implicit val ec = device.executionContext
  implicit val timeout: Timeout = 3.seconds
  implicit val scheduler = device.scheduler

  device ! DeviceActor.UpdateAirQuality(10)
  device ! DeviceActor.UpdateAirQuality(20)
  device ! DeviceActor.UpdateTemperature(20)
  device ! DeviceActor.UpdateTemperature(10)

  val response: Future[DeviceActor.Response] = device.ask(ref ⇒ DeviceActor.StateRequest(ref))

  response.onComplete {
    case Success(DeviceActor.StateResponse(state)) ⇒
      println("Current state is: " + state.toString)
      device.terminate()
    case Failure(_) ⇒
      println("Unable to get current state.")
      device.terminate()
  }
```

Akka Streams

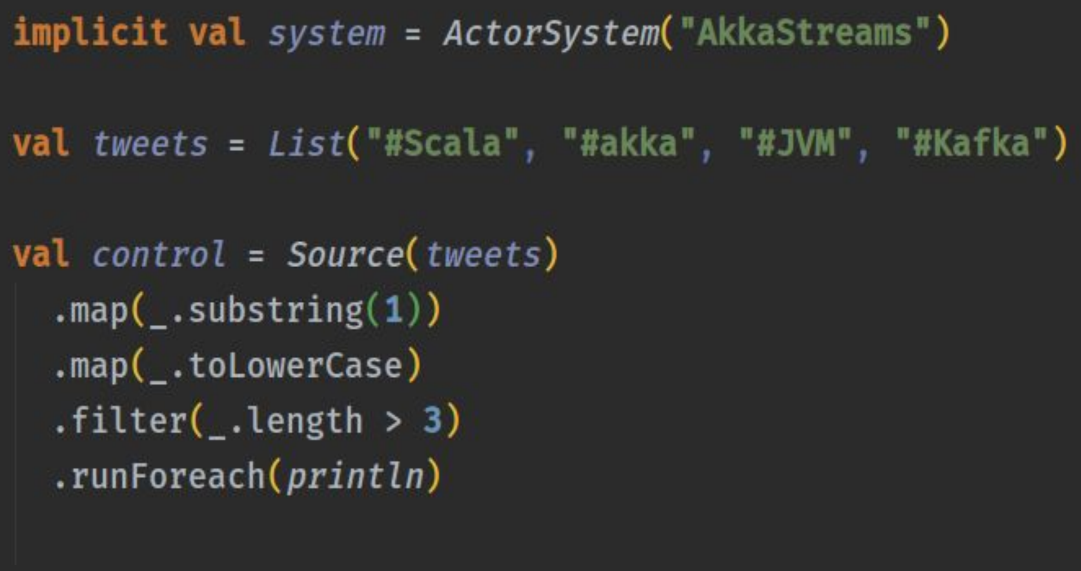




```scala
implicit val system = ActorSystem("AkkaStreams")

val tweets = List("#Scala", "#akka", "#JVM", "#Kafka")

val source = Source(tweets)
val removeHash = Flow[String].map(_.substring(1))
val toLowerCase = Flow[String].map(_.toLowerCase)
val filter = Flow[String].filter(_.length > 3)
val sink = Sink.foreach[String](println)

val graph = source.via(removeHash).via(toLowerCase).via(filter).to(sink)
graph.run()
```



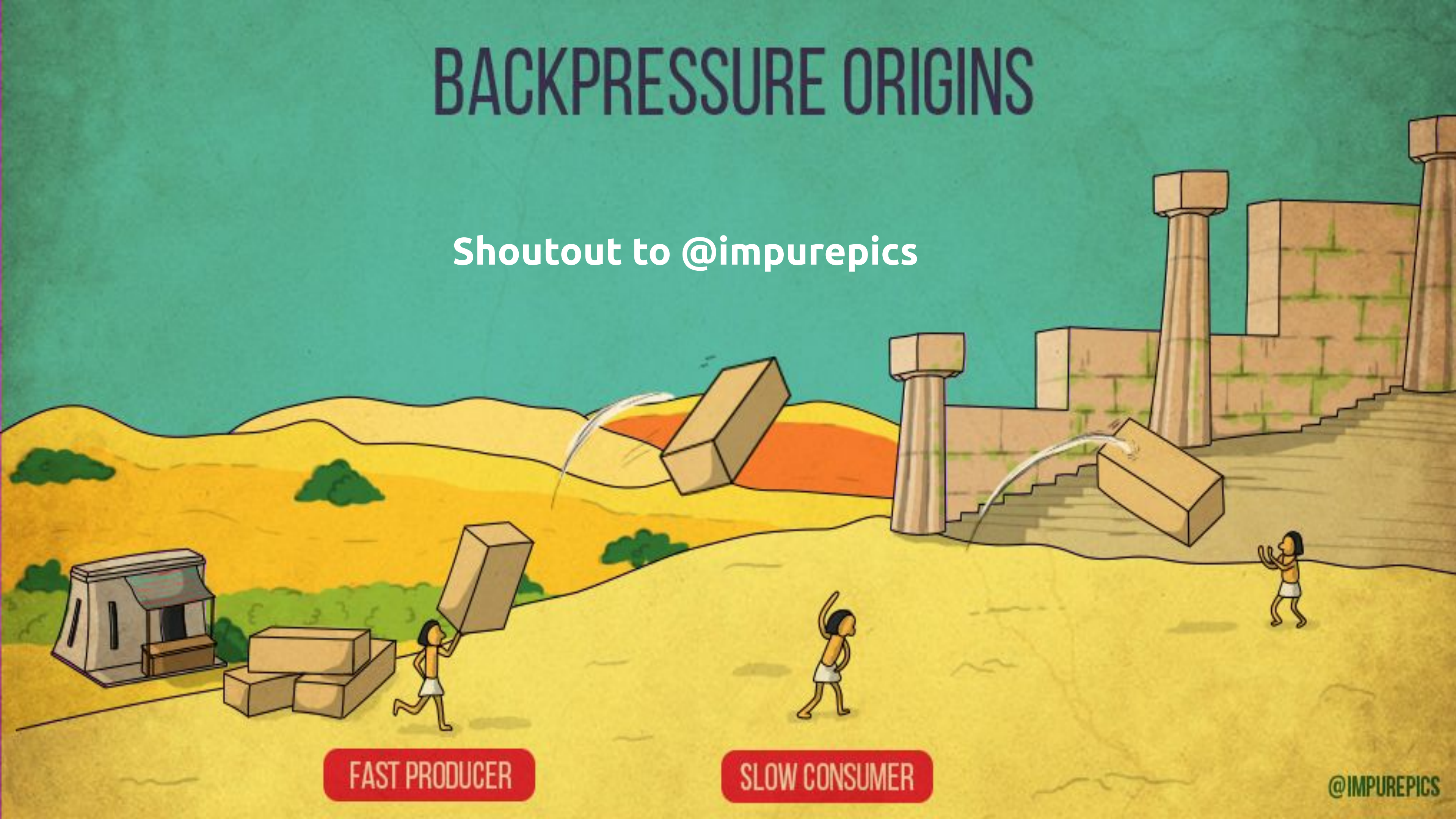
```scala
implicit val system = ActorSystem("AkkaStreams")

val tweets = List("#Scala", "#akka", "#JVM", "#Kafka")

val sink = Sink.foreach[String](println)

val graph = Source(tweets)
  .map(_.substring(1))
  .map(_.toLowerCase)
  .filter(_.length > 3)
  .to(sink)

graph.run()
```



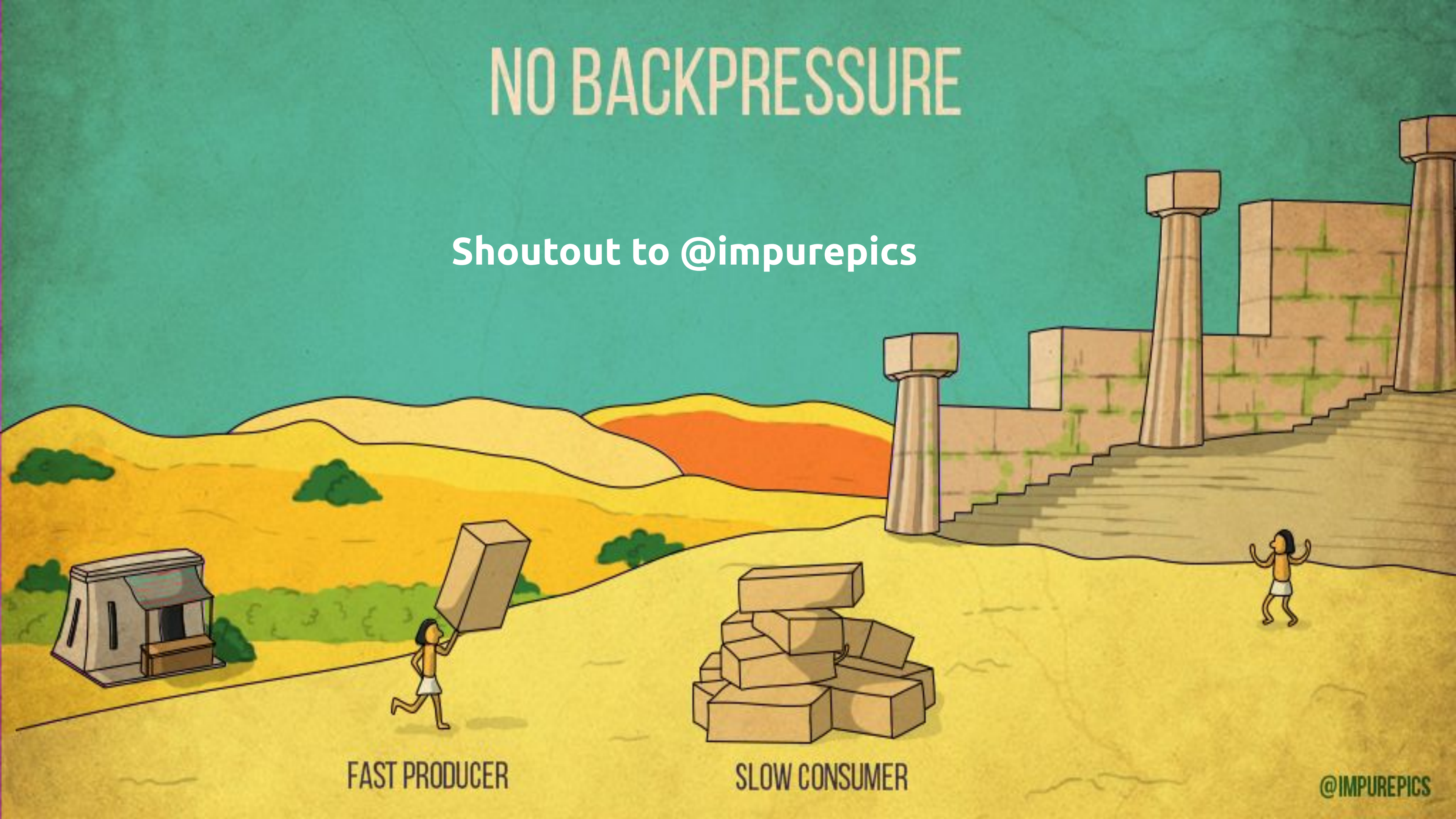
```scala
implicit val system = ActorSystem("AkkaStreams")

val tweets = List("#Scala", "#akka", "#JVM", "#Kafka")

val control = Source(tweets)
  .map(_.substring(1))
  .map(_.toLowerCase)
  .filter(_.length > 3)
  .runForeach(println)
```

Why Akka Streams?

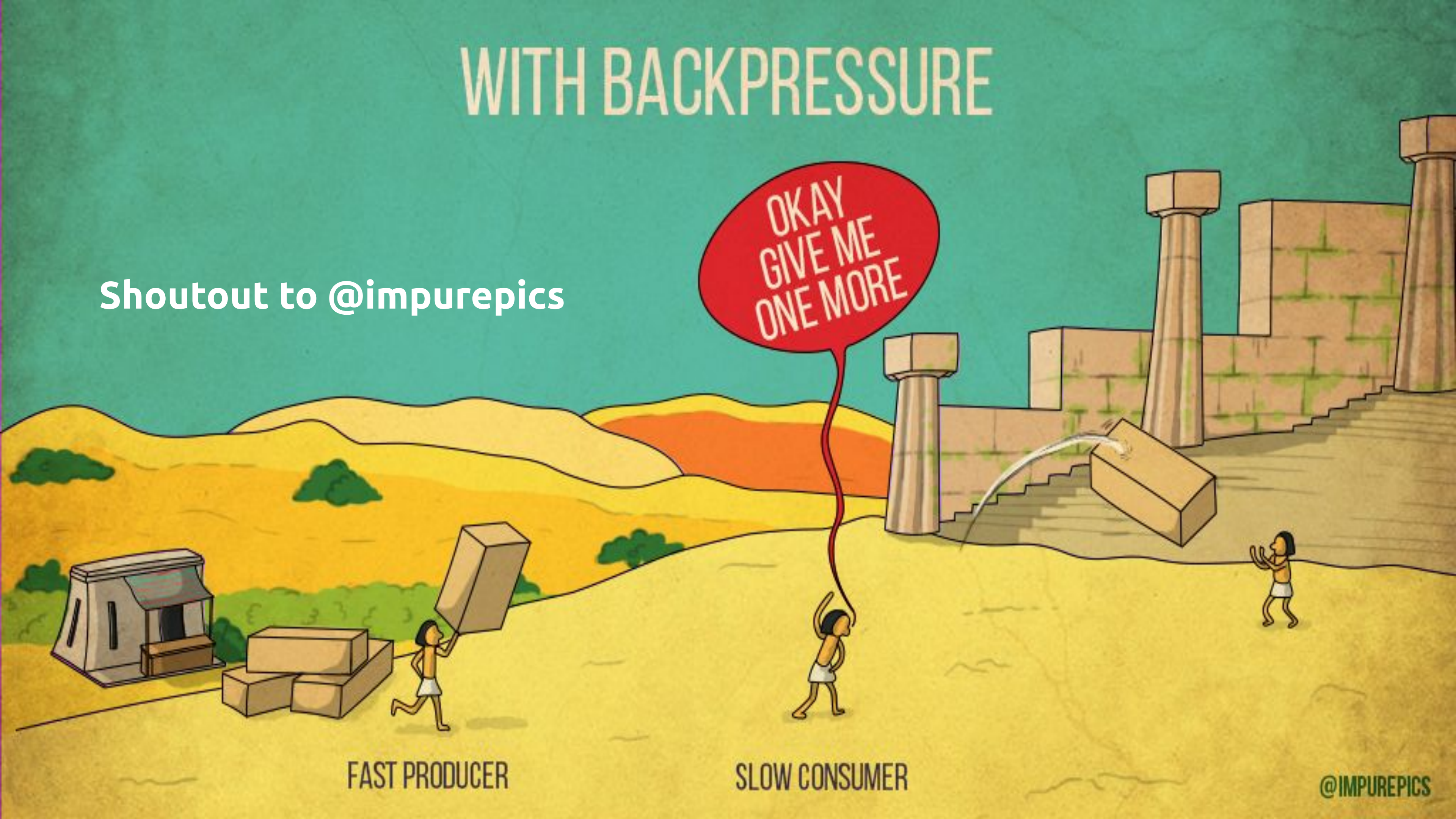
- Declarative API
- Nice integration with other systems
- Stream supervision
  - Recovery
  - Error handling
  - Retries
- Back-pressure aware by design
  - No need for explicit back-pressure handing code



Shoutout to @impurepics



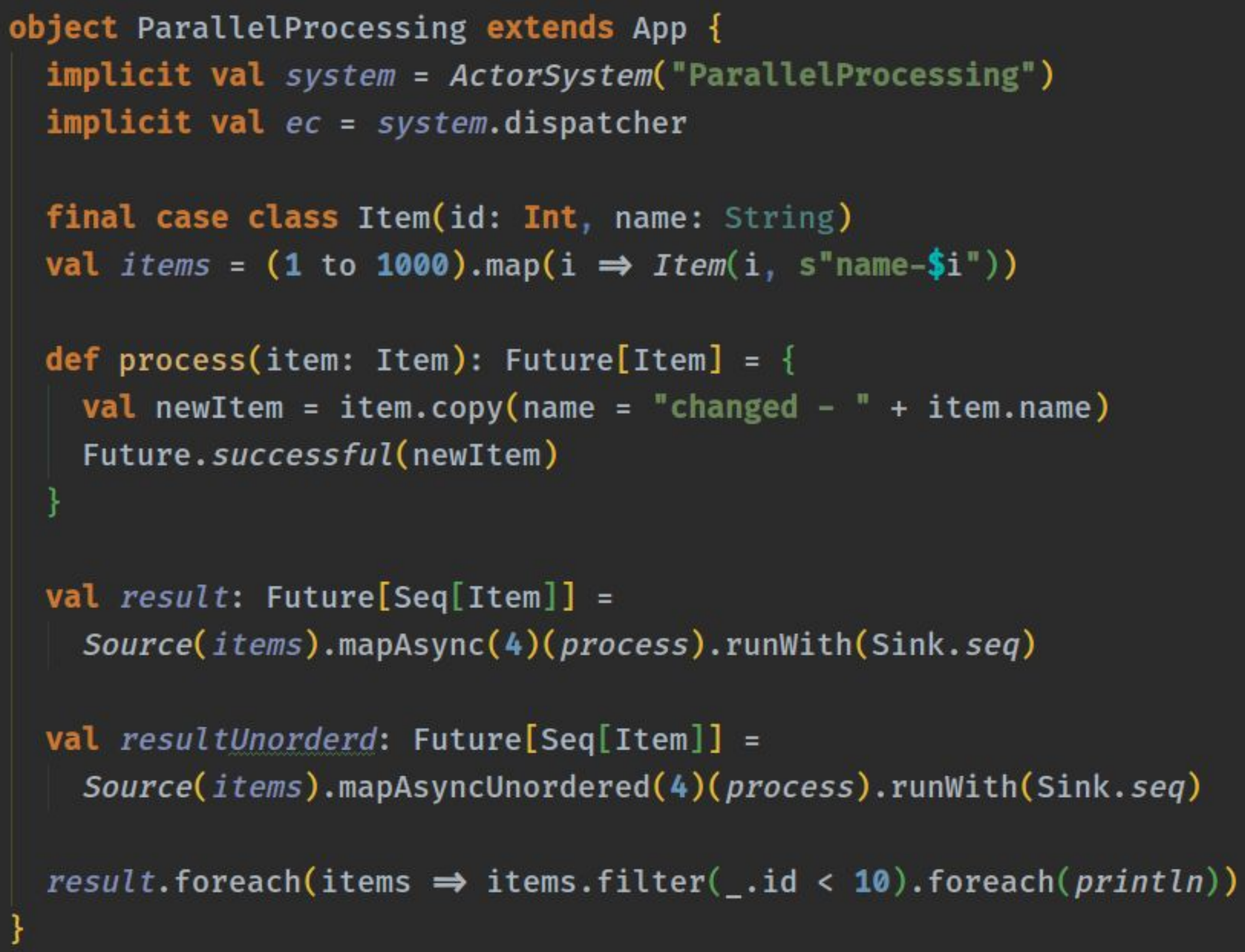
Shoutout to @impurepics



Shoutout to @impurepics

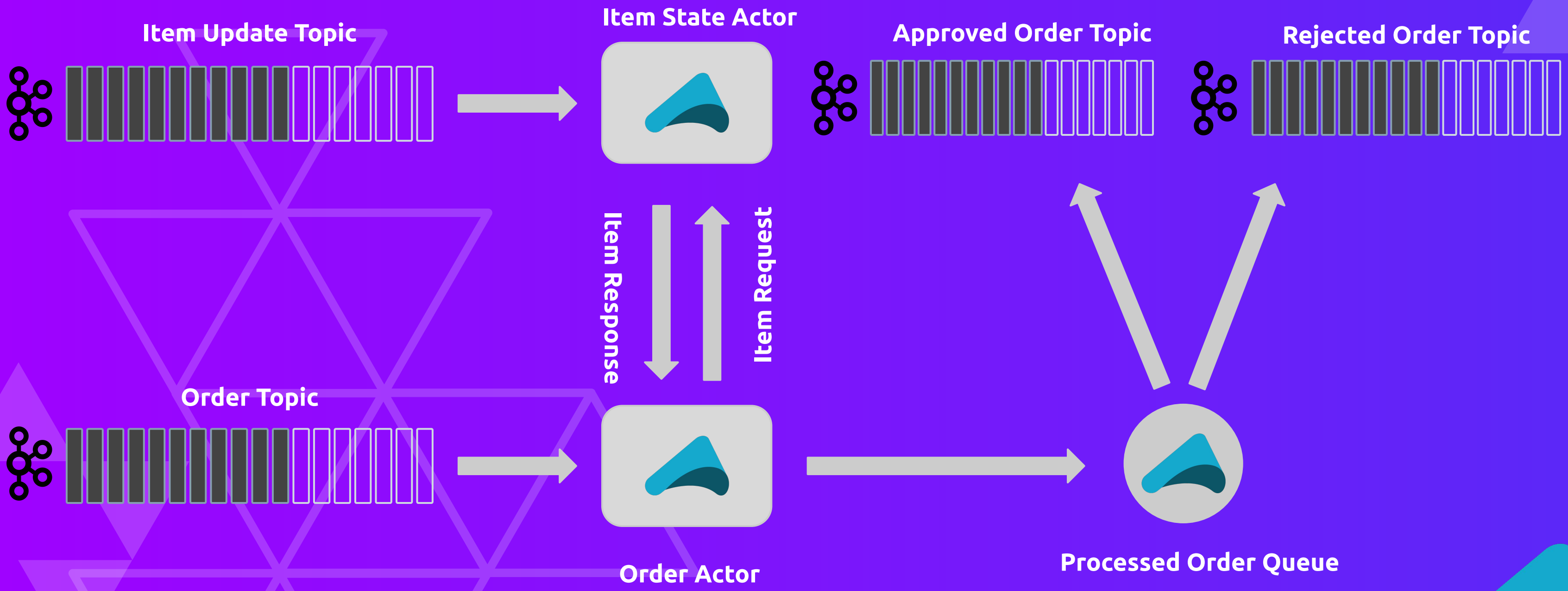
Streaming Patterns

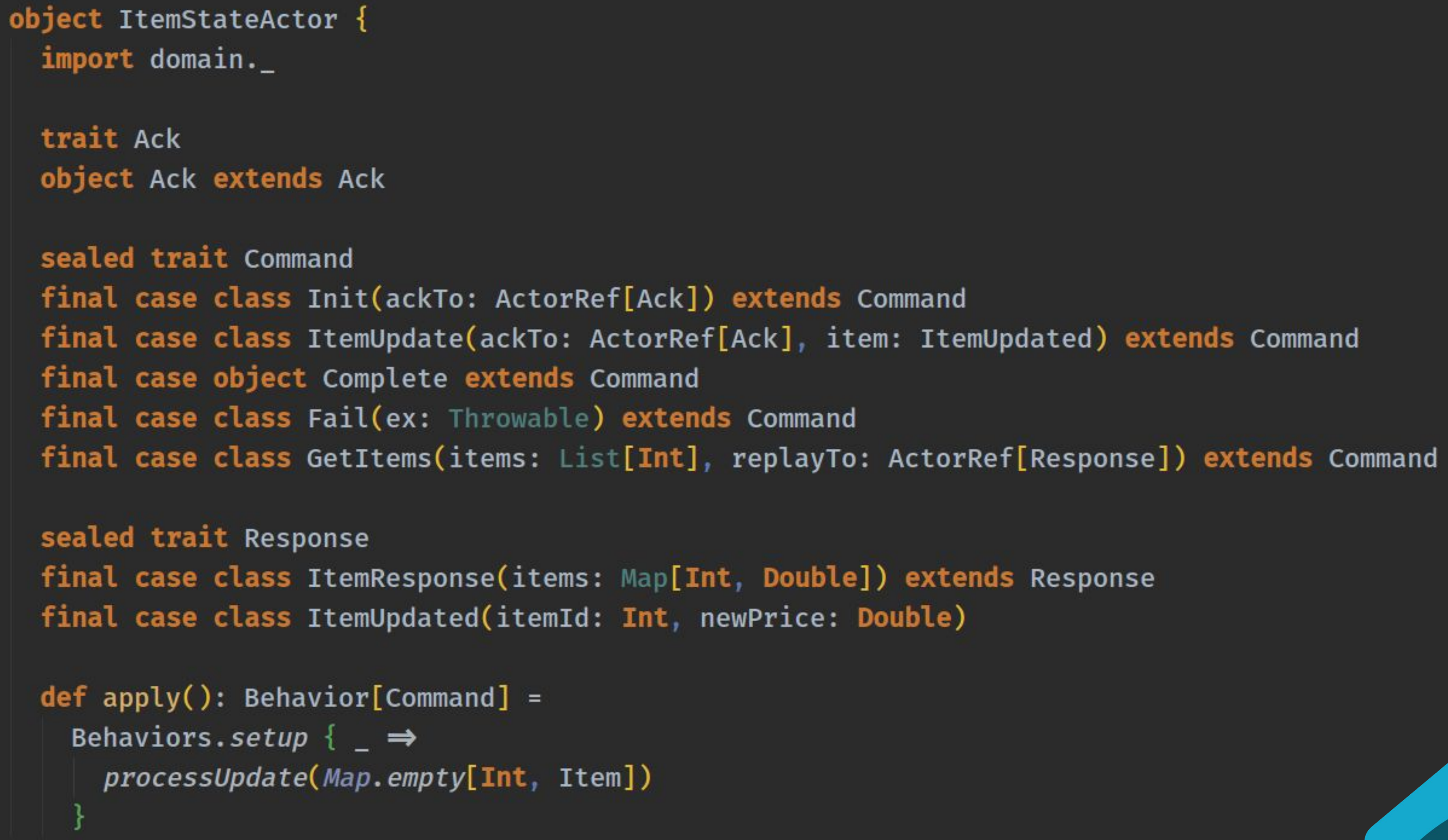
- Parallel Stream Processing
- Actors and Streams integration
  - Actor as a Sink
  - Actor as a Flow
- Handling WebSockets and HTTP with Akka Streams
  - WebSocket to Kafka Pipeline
  - Rate Limiter



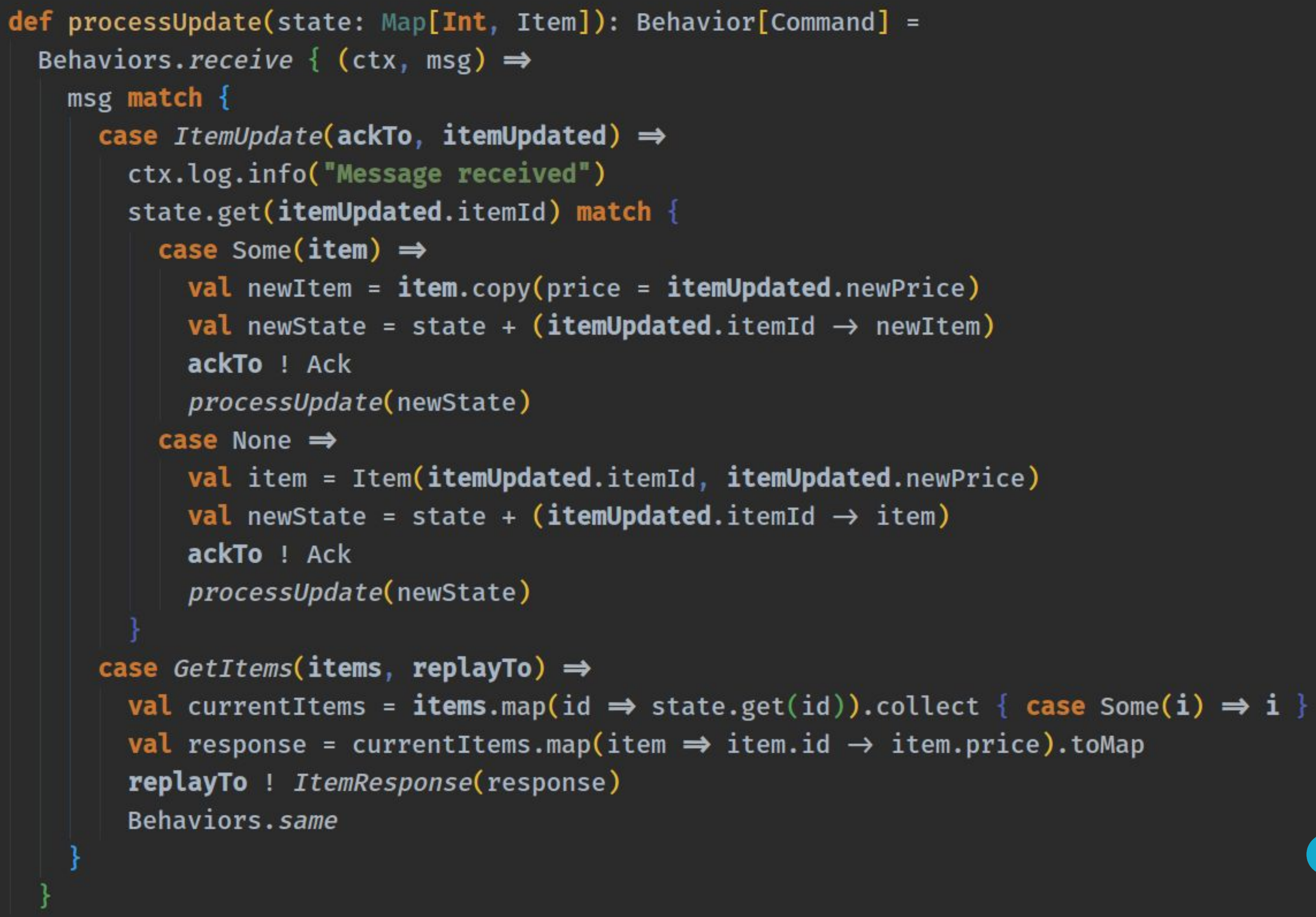
```scala
object ParallelProcessing extends App {
  implicit val system = ActorSystem("ParallelProcessing")
  implicit val ec = system.dispatcher

  final case class Item(id: Int, name: String)
  val items = (1 to 1000).map(i => Item(i, s"name-$i"))

  def process(item: Item): Future[Item] = {
    val newItem = item.copy(name = "changed - " + item.name)
    Future.successful(newItem)
  }

  val result: Future[Seq[Item]] =
    Source(items).mapAsync(4)(process).runWith(Sink.seq)

  val resultUnorderd: Future[Seq[Item]] =
    Source(items).mapAsyncUnordered(4)(process).runWith(Sink.seq)

  result.foreach(items => items.filter(_.id < 10).foreach(println))
}
```

Actors and Streams

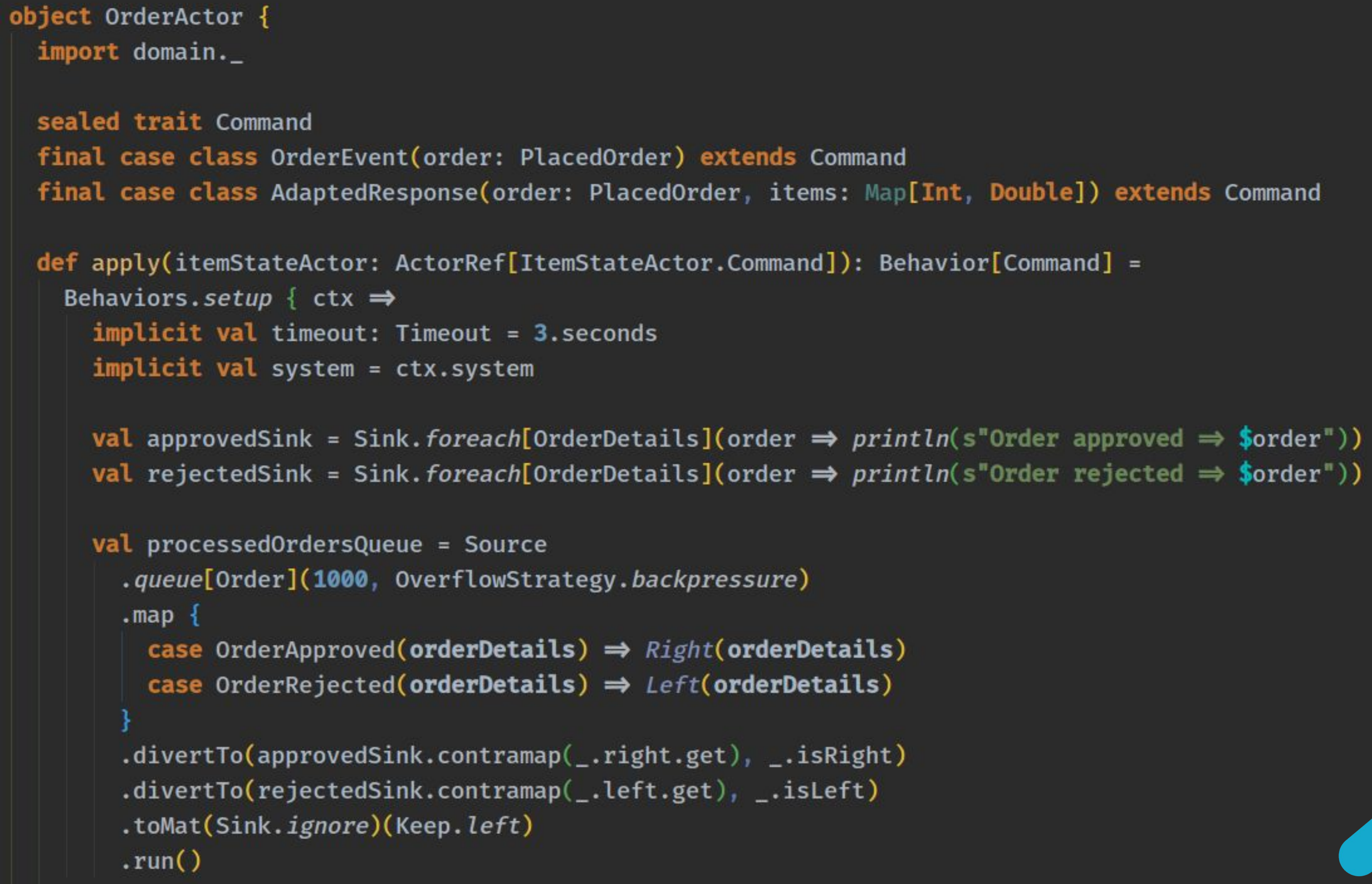




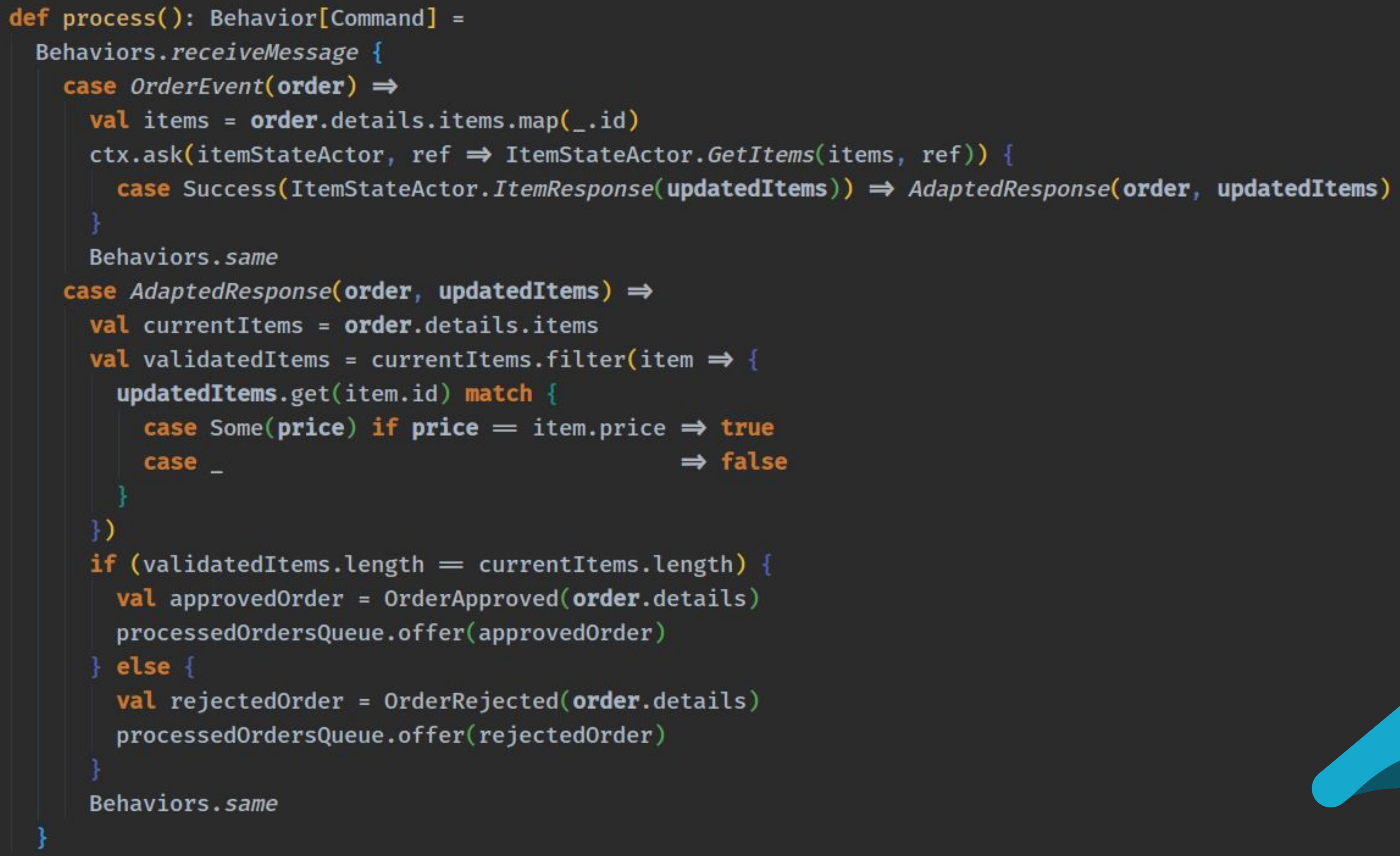
```scala
object ItemStateActor {
  import domain._

  trait Ack
  object Ack extends Ack

  sealed trait Command
  final case class Init(ackTo: ActorRef[Ack]) extends Command
  final case class ItemUpdate(ackTo: ActorRef[Ack], item: ItemUpdated) extends Command
  final case object Complete extends Command
  final case class Fail(ex: Throwable) extends Command
  final case class GetItems(items: List[Int], replayTo: ActorRef[Response]) extends Command

  sealed trait Response
  final case class ItemResponse(items: Map[Int, Double]) extends Response
  final case class ItemUpdated(itemId: Int, newPrice: Double)

  def apply(): Behavior[Command] =
    Behaviors.setup { _ =>
      processUpdate(Map.empty[Int, Item])
    }
```



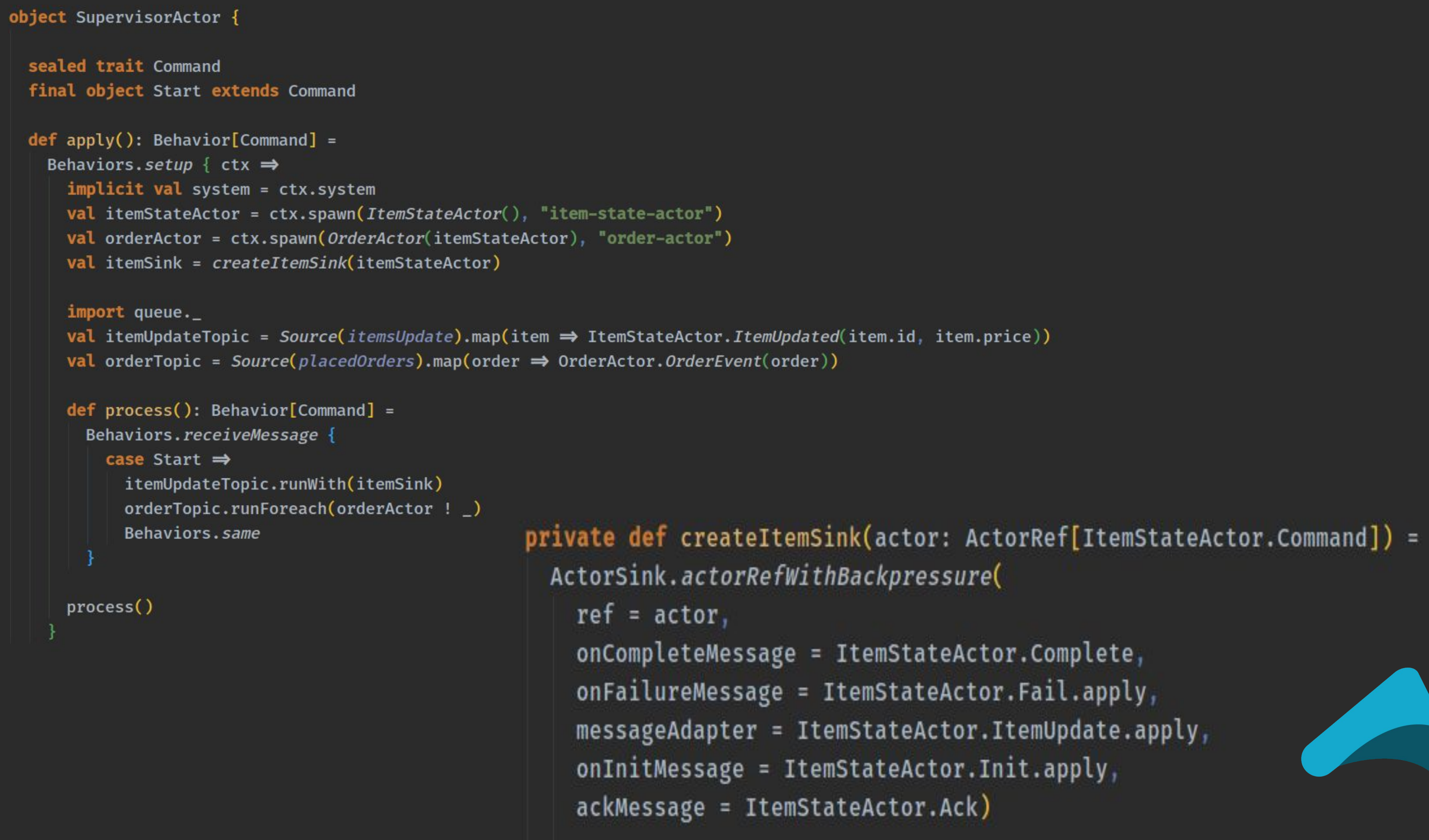
```scala
def processUpdate(state: Map[Int, Item]): Behavior[Command] =
  Behaviors.receive { (ctx, msg) =>
    msg match {
      case ItemUpdate(ackTo, itemUpdated) =>
        ctx.log.info("Message received")
        state.get(itemUpdated.itemId) match {
          case Some(item) =>
            val newItem = item.copy(price = itemUpdated.newPrice)
            val newState = state + (itemUpdated.itemId -> newItem)
            ackTo ! Ack
            processUpdate(newState)
          case None =>
            val item = Item(itemUpdated.itemId, itemUpdated.newPrice)
            val newState = state + (itemUpdated.itemId -> item)
            ackTo ! Ack
            processUpdate(newState)
        }
      case GetItems(items, replayTo) =>
        val currentItems = items.map(id => state.get(id)).collect { case Some(i) => i }
        val response = currentItems.map(item => item.id -> item.price).toMap
        replayTo ! ItemResponse(response)
        Behaviors.same
    }
  }
```



```scala
object OrderActor {
  import domain._

  sealed trait Command
  final case class OrderEvent(order: PlacedOrder) extends Command
  final case class AdaptedResponse(order: PlacedOrder, items: Map[Int, Double]) extends Command

  def apply(itemStateActor: ActorRef[ItemStateActor.Command]): Behavior[Command] =
    Behaviors.setup { ctx =>
      implicit val timeout: Timeout = 3.seconds
      implicit val system = ctx.system

      val approvedSink = Sink.foreach[OrderDetails](order => println(s"Order approved => $order"))
      val rejectedSink = Sink.foreach[OrderDetails](order => println(s"Order rejected => $order"))

      val processedOrdersQueue = Source
        .queue[Order](1000, OverflowStrategy.backpressure)
        .map {
          case OrderApproved(orderDetails) => Right(orderDetails)
          case OrderRejected(orderDetails) => Left(orderDetails)
        }
        .divertTo(approvedSink.contramap(_.right.get), _.isRight)
        .divertTo(rejectedSink.contramap(_.left.get), _.isLeft)
        .toMat(Sink.ignore)(Keep.left)
        .run()
```



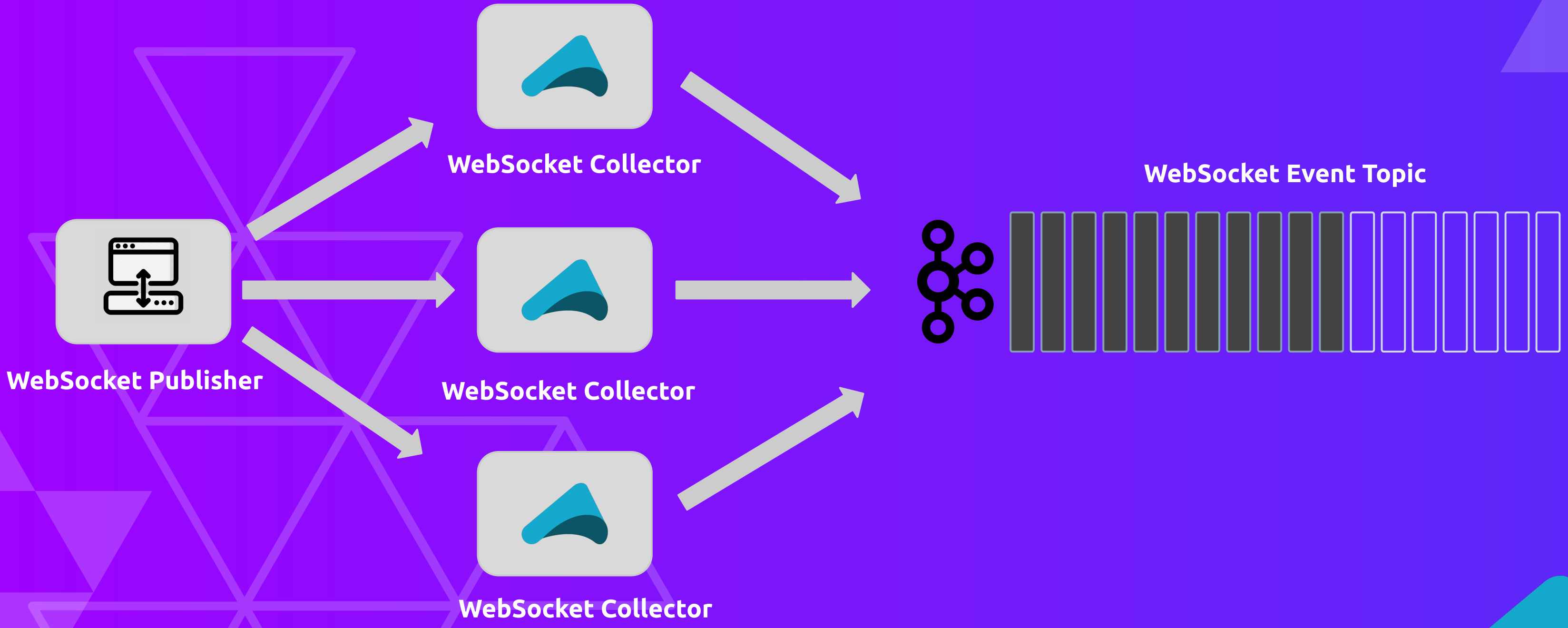
```scala
def process(): Behavior[Command] =
  Behaviors.receiveMessage {
    case OrderEvent(order) ⇒
      val items = order.details.items.map(_.id)
      ctx.ask(itemStateActor, ref ⇒ ItemStateActor.GetItems(items, ref)) {
        case Success(ItemStateActor.ItemResponse(updatedItems)) ⇒ AdaptedResponse(order, updatedItems)
      }
      Behaviors.same
    case AdaptedResponse(order, updatedItems) ⇒
      val currentItems = order.details.items
      val validatedItems = currentItems.filter(item ⇒ {
        updatedItems.get(item.id) match {
          case Some(price) if price == item.price ⇒ true
          case _                                  ⇒ false
        }
      })
      if (validatedItems.length == currentItems.length) {
        val approvedOrder = OrderApproved(order.details)
        processedOrdersQueue.offer(approvedOrder)
      } else {
        val rejectedOrder = OrderRejected(order.details)
        processedOrdersQueue.offer(rejectedOrder)
      }
      Behaviors.same
  }
```

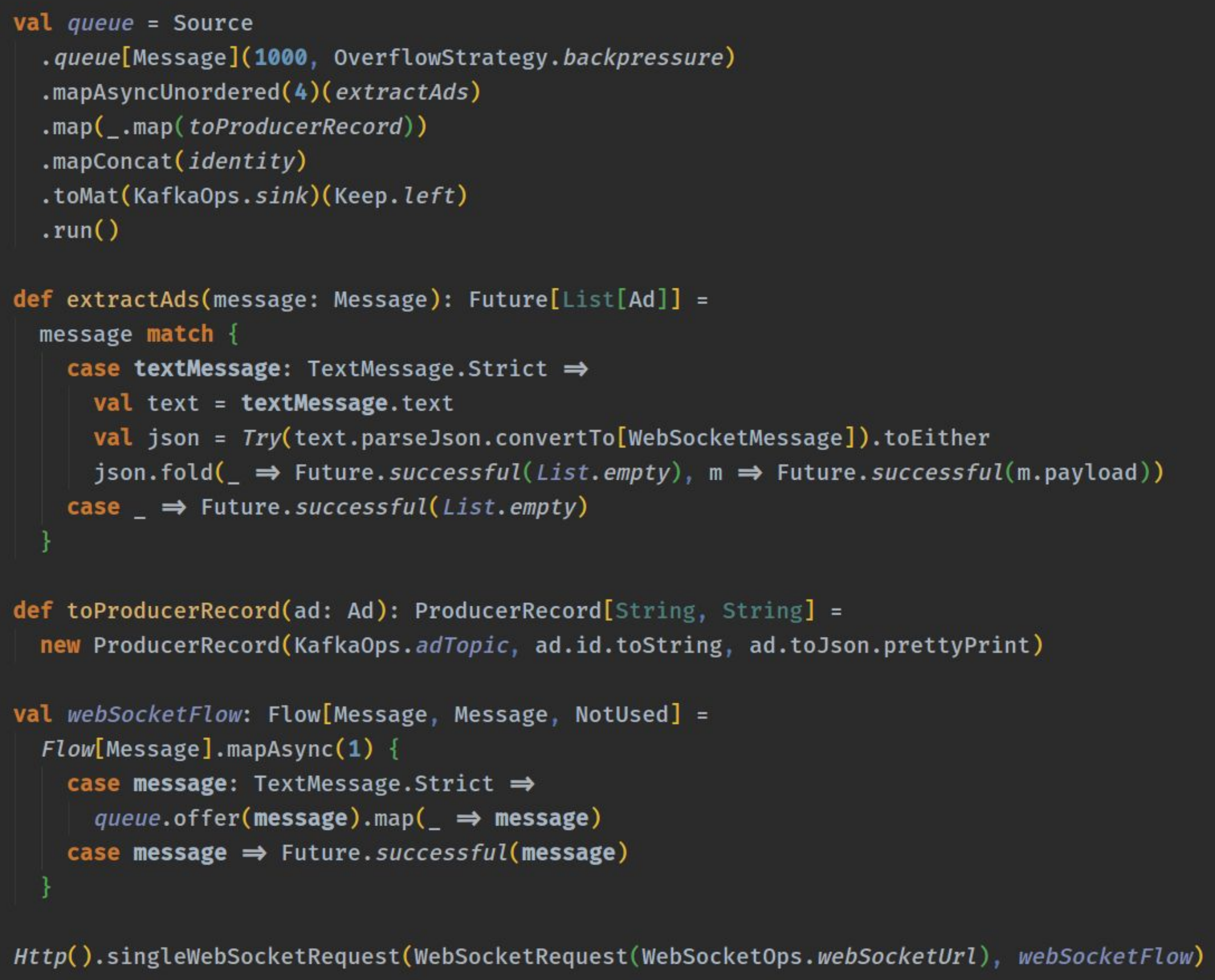


```scala
object SupervisorActor {

  sealed trait Command
  final object Start extends Command

  def apply(): Behavior[Command] =
    Behaviors.setup { ctx =>
      implicit val system = ctx.system
      val itemStateActor = ctx.spawn(ItemStateActor(), "item-state-actor")
      val orderActor = ctx.spawn(OrderActor(itemStateActor), "order-actor")
      val itemSink = createItemSink(itemStateActor)

      import queue._
      val itemUpdateTopic = Source(itemsUpdate).map(item => ItemStateActor.ItemUpdated(item.id, item.price))
      val orderTopic = Source(placedOrders).map(order => OrderActor.OrderEvent(order))

      def process(): Behavior[Command] =
        Behaviors.receiveMessage {
          case Start =>
            itemUpdateTopic.runWith(itemSink)
            orderTopic.runForeach(orderActor ! _)
            Behaviors.same
        }

      process()
    }
```

```scala
private def createItemSink(actor: ActorRef[ItemStateActor.Command]) =
  ActorSink.actorRefWithBackpressure(
    ref = actor,
    onCompleteMessage = ItemStateActor.Complete,
    onFailureMessage = ItemStateActor.Fail.apply,
    messageAdapter = ItemStateActor.ItemUpdate.apply,
    onInitMessage = ItemStateActor.Init.apply,
    ackMessage = ItemStateActor.Ack)
```

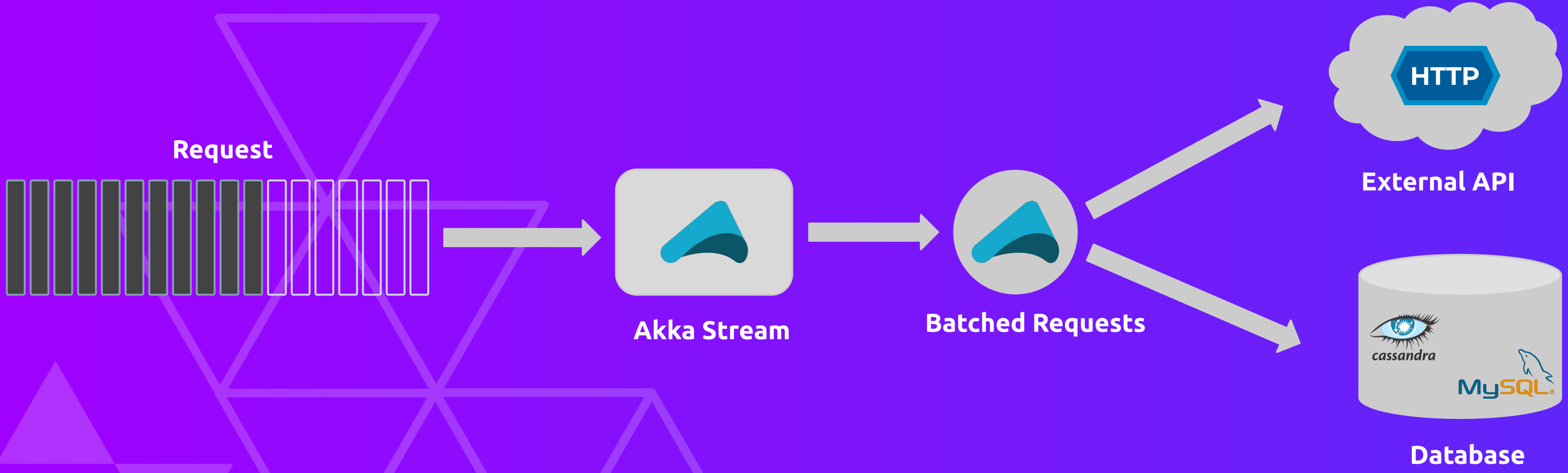
WebSocket to Kafka Pipeline

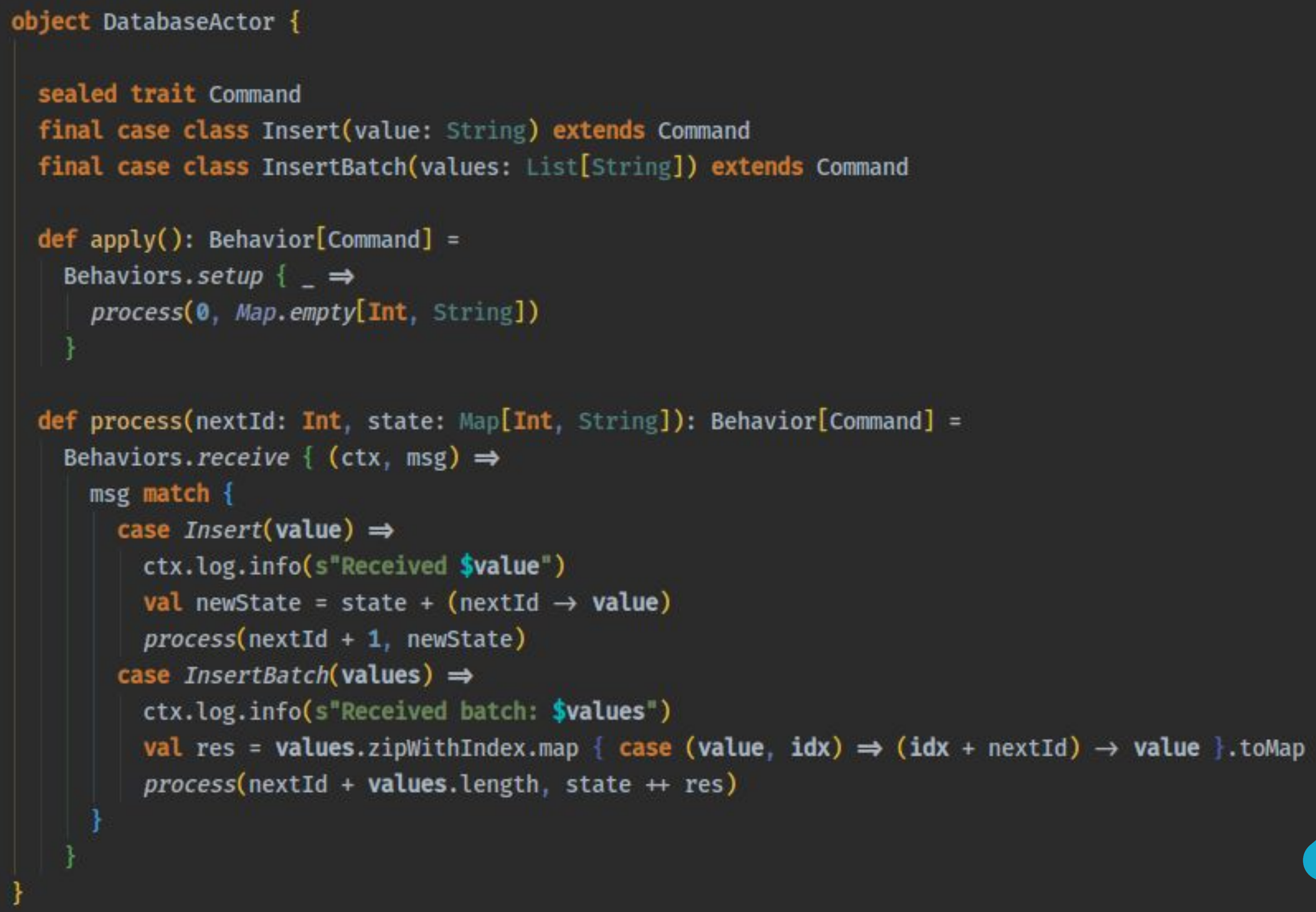




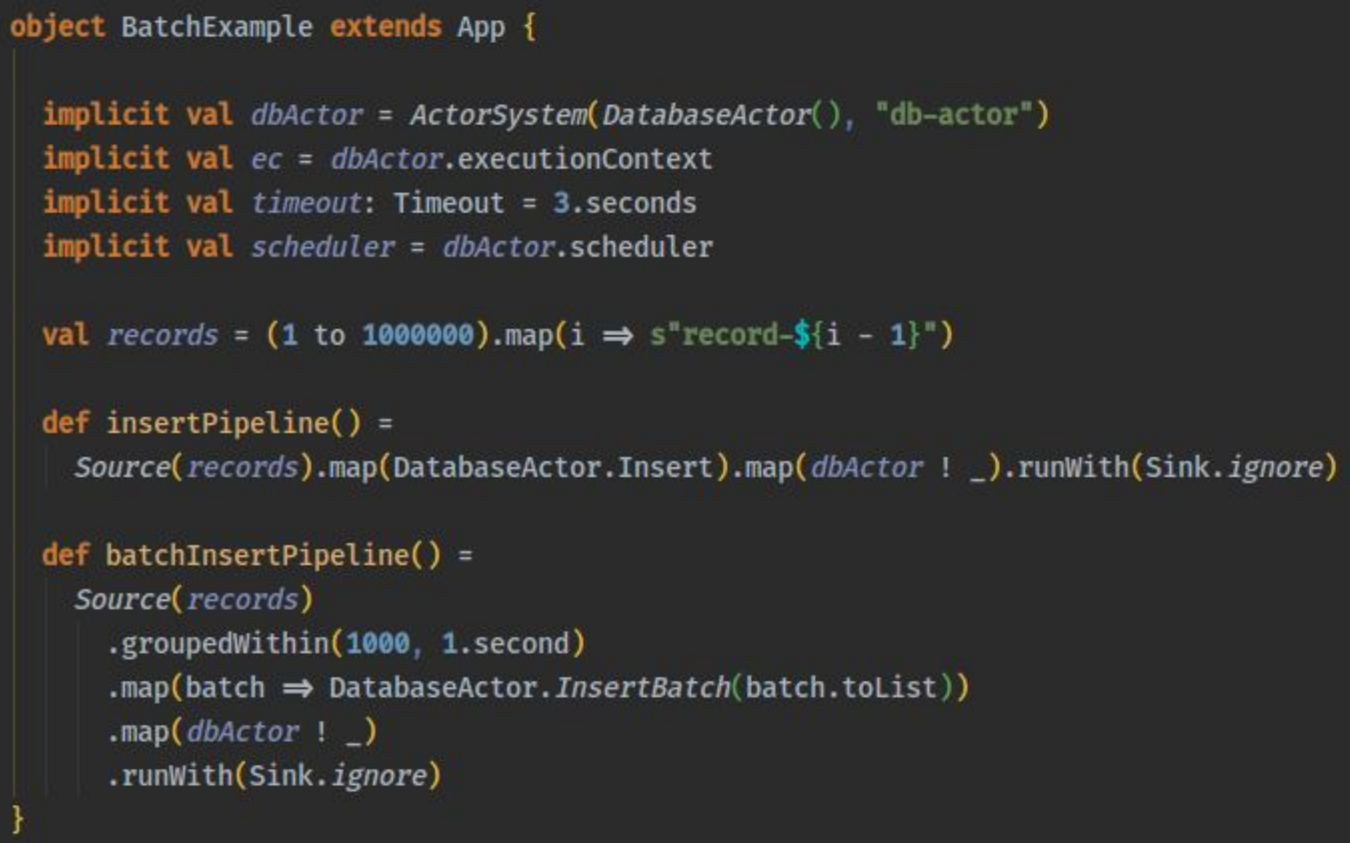
```scala
val queue = Source
  .queue[Message](1000, OverflowStrategy.backpressure)
  .mapAsyncUnordered(4)(extractAds)
  .map(_.map(toProducerRecord))
  .mapConcat(identity)
  .toMat(KafkaOps.sink)(Keep.left)
  .run()

def extractAds(message: Message): Future[List[Ad]] =
  message match {
    case textMessage: TextMessage.Strict =>
      val text = textMessage.text
      val json = Try(text.parseJson.convertTo[WebSocketMessage]).toEither
      json.fold(_ => Future.successful(List.empty), m => Future.successful(m.payload))
    case _ => Future.successful(List.empty)
  }

def toProducerRecord(ad: Ad): ProducerRecord[String, String] =
  new ProducerRecord(KafkaOps.adTopic, ad.id.toString, ad.toJson.prettyPrint)

val webSocketFlow: Flow[Message, Message, NotUsed] =
  Flow[Message].mapAsync(1) {
    case message: TextMessage.Strict =>
      queue.offer(message).map(_ => message)
    case message => Future.successful(message)
  }

Http().singleWebSocketRequest(WebSocketRequest(WebSocketOps.webSocketUrl), webSocketFlow)
```

Rate Limiter





```scala
object DatabaseActor {

  sealed trait Command
  final case class Insert(value: String) extends Command
  final case class InsertBatch(values: List[String]) extends Command

  def apply(): Behavior[Command] =
    Behaviors.setup { _ =>
      process(0, Map.empty[Int, String])
    }

  def process(nextId: Int, state: Map[Int, String]): Behavior[Command] =
    Behaviors.receive { (ctx, msg) =>
      msg match {
        case Insert(value) =>
          ctx.log.info(s"Received $value")
          val newState = state + (nextId -> value)
          process(nextId + 1, newState)
        case InsertBatch(values) =>
          ctx.log.info(s"Received batch: $values")
          val res = values.zipWithIndex.map { case (value, idx) => (idx + nextId) -> value }.toMap
          process(nextId + values.length, state ++ res)
      }
    }
}
```



```scala
object BatchExample extends App {

  implicit val dbActor = ActorSystem(DatabaseActor(), "db-actor")
  implicit val ec = dbActor.executionContext
  implicit val timeout: Timeout = 3.seconds
  implicit val scheduler = dbActor.scheduler

  val records = (1 to 1000000).map(i ⇒ s"record-${i - 1}")

  def insertPipeline() =
    Source(records).map(DatabaseActor.Insert).map(dbActor ! _).runWith(Sink.ignore)

  def batchInsertPipeline() =
    Source(records)
      .groupedWithin(1000, 1.second)
      .map(batch ⇒ DatabaseActor.InsertBatch(batch.toList))
      .map(dbActor ! _)
      .runWith(Sink.ignore)
}
```

THANKS!

Do you have any questions?

CREDITS: This presentation template was created by Slidesgo, including icons by Flaticon, and infographics & images by Freepik