Making Music With Akka Streams
Reactive Summit, Montreal QC, 2018 October 24
Agenda

- Quick overview of Akka Streams
- A Graph of Streams For Making Music
- Quick Demonstration
- Stream Construction Techniques
- But Wait! There’s more.
A Quick Review Of Akka Streams

- Streams support continuous processing of data in a pipeline
- Akka Streams provides a rich set of operators to compose streams
- Build a model of the stream and then run it as many times as needed.
- Three Element Kinds For Modeling a Stream: Source, Flow, Sink

```scala
object StreamExample {

  implicit val system: ActorSystem = ActorSystem("foo")
  implicit val materializer: Materializer = ActorMaterializer()

  def main(args: Array[String]): Unit = {
    val source: Source[Int, NotUsed] = Source(1 to 100)
    val flow: Flow[Int, Int, NotUsed] = Flow.fromFunction(x => x / 2)
    val sink: Sink[Int, Future[Done]] = Sink.foreach(println(_))
    val _ = source.via(flow).to(sink).run()
  }
}
```
Quick Demonstration
Stream Construction Techniques
Creating A Source With An Actor

```scala
private final val stream: Source[ScreenInputEvent,ActorRef] = Source.actorRef(bufferSize = 100, OverflowStrategy.dropNew)

private final val (eventActor, theSource) = stream.preMaterialize()
```

- Messages sent to an Actor become elements of the stream
- Must pay attention to buffering and overflow
- Back pressure is not possible with this kind of Source
- `Source.actorRef` materializes the Actor and the Source
- Messages sent to the Actor become elements in the Source
- This is how we transform Swing events to stream elements

```scala
protected def mouseWheelMoved(e : MouseWheelEvent) : Unit = {
  eventActor ! MouseWheelRotated(e.getWheelRotation)
}
```
Extracting Elements By Type

val mouseWheelRotatedSource: Source[MouseWheelRotated, NotUsed] = {
  source.collectType[MouseWheelRotated](classTag[MouseWheelRotated])
}

- Source.collectType[T] can be used to filter out all messages except those of a specific type.
- Shown above is how we get a source of MouseWheelRotated messages from a source of many types of events.
- Shown below is how we get a source of MouseClicked messages from a source of many types of events.
- Note the use of classTag[MouseClicked] from scala.reflect.

val mouseClickedSource: Source[MouseClicked, NotUsed] = {
  source.collectType[MouseClicked](classTag[MouseClicked])
}
Generating Elements At A Fixed Rate

- Use `Source.tick` to get a stream of elements at a fixed rate.
- `Source.tick` has three arguments: initialDelay, interval, and tick.
- Map the “tick” to the desired type.
- Here we’ve mapped a Unit value, (), to MouseRelocated with data that comes from the `MouseInfo.getPointerInfo.getLocation` function from Swing.

```scala
val mousePositionSource: Source[MouseRelocated, Cancellable] = {
  Source
    .tick(0.millis, config.sampleRate, ())
    .map { _: Unit =>
      import ... 
      val p: Point = MouseInfo.getPointerInfo.getLocation
      MouseRelocated(p.x, p.y)
    }
}
```
Transforming Elements With map

- Here we convert a `MouseWheelRotated` swing event into a simple `Int`

```scala
case class MouseWheelRotatedInstrumentDevice(
  input: ScreenInput
) extends InstrumentDevice {
  val maxValue: Int = 1000
  val minValue: Int = 0
  private var value: Int = 0

  override def dataSource: Source[Int, NotUsed] = {
    input.mouseWheelRotatedSource.map { rotation: MouseWheelRotated =>
      val newVal = value + rotation.rotation
      value = Math.min(maxValue, Math.max(minValue, newVal))
      value % 127
    }
  }
}
```
Transformation Of Elements

- Update paneData and convert to a pair of MusicMakerMessages

```kotlin
private val instruments: Source[MusicMakerMessage, NotUsed] = {
  instrumentDev
  .dataSource
    .via(Flow.fromFunction{ instrument: Int =>
      paneData = paneData.copy(instrument = instrument)
      instrument
    })
  .flatMapConcat { instrument =>
    Source.apply[MusicMakerMessage](
      Vector(
        ChangeInstrument(channel, instrument),
        ChangeInstrument(specialKeyChannel, instrument)
      )
    )
  }
}
```
Merge And Broadcast

- Merge a group of Sources together
- Materialize it with `BroadcastHub.sink` to obtain `RunnableGraph[Source[MusicMakerMessage,NotUse]]`
- Use the `RunnableGraph` to get multiple Sources for the broadcast.

```swift
Source
    .combine(pitchesAndVolumes, onoffs, keys, portamentos, instruments, specialKeys)
    { numSources: Int =>
        Merge(numSources, eagerComplete = false)
    }
    .toMat(BroadcastHub.sink)(Keep.right)
```
Combining Two Sources With A Graph

```scala
class DevicesToNotesGraph(
    channel: Int, volumes: Source[Int, NotUsed]
)(implicit mat: Materializer) extends GraphStage[FlowShape[Int, PlayNote]] {

  val pitchesPort: Inlet[Int] = Inlet[Int](name = "pitches.in")
  val musicMakerMessagesPort: Outlet[PlayNote] =
      Outlet[PlayNote](name = "musicMakerMessages.out")

  override val shape: FlowShape[Int, PlayNote] =
      FlowShape(pitchesPort, musicMakerMessagesPort)

  def createLogic(inheritedAttributes: Attributes): GraphStageLogic = {
    new GraphStageLogic(shape) {
      var lastVolume: Int = 0
      volumes.runForeach( v => lastVolume = v)
      setHandler(pitchesPort, new InHandler {
        def onPush(): Unit = {
          val pitch = grab(pitchesPort)
          push(musicMakerMessagesPort, PlayNote(channel, pitch, lastVolume))
        }
      })
      setHandler(musicMakerMessagesPort, new OutHandler {
        override def onPull(): Unit = {
          pull(pitchesPort)
        }
      })
    }
  }
}
```

- This defines a Flow
- The logic retains the value of volumes in lastVolume.
- Elements flow on the timing of the pitches (input)
- Output is PlayNote
Random Music Generation

3 Sources combined to set the instrument, play the notes and stop the channel.

Note use of limit(n)

Velocity (volume) is limited to a quieter but audible range.
Demonstrate All Features
But wait! There’s Prometheus Too!

[Diagram showing the integration of Music Maker and Prometheus with Screen Output and /metrics endpoints.]
Demonstrate Grafana/Prometheus
● Use “via” to process data through a Flow (next page)
● Sample and print every 1000th message.
Use:

`Flow.fromFunction` to do intermediate processing in a stream.

Each of the purple values is a metric defined with the Prometheus Java Client. This is how the metrics are collected.
And It Runs On Raspberry Pi 3+ (Demonstration)
Thank You!

reid.spencer@yoppworks.com

- **About YoppWorks Inc:**
  - Lightbend Platinum Partner
  - We Deliver Services & Products
  - Based In Ottawa, Canada
  - Lightbend Training, Consulting
    - GoSpaces, RobotHive, Fast Data

- **Presentation Code Is Here:**
  - https://github.com/YoppWorks/mmaws.git