Real-Time Event Stream Processing System in Scala

Build Setup (SBT)

```
// build.sbt
ThisBuild / scalaVersion := "2.13.12"

libraryDependencies ++= Seq(
  "org.typelevel" %% "cats-core" % "2.10.0",
  "org.typelevel" %% "cats-effect" % "3.5.2",
  "co.fs2" %% "fs2-core" % "3.9.2",
  "io.circe" %% "circe-core" % "0.14.6",
  "io.circe" %% "circe-generic" % "0.14.6",
  "io.circe" %% "circe-parser" % "0.14.6"
)
```

Domain Model (ADTs)

```
import java.time.Instant

sealed trait EventType
case object Click extends EventType
case object View  extends EventType
case object Purchase extends EventType

final case class RawEvent(
  userId: String,
  timestamp: Instant,
  eventType: String,
  metadata: String
)

final case class EnrichedEvent(
  userId: String,
  timestamp: Instant,
  eventType: EventType,
  sessionId: String,
  geoLocation: Option[String]
)
```

EventParser: Type class to parse raw events

```
trait EventParser[F[_]] {
  def parse(raw: RawEvent): F[EnrichedEvent]
}
```

EventPersister: Abstract over any sink

```
trait EventPersister[F[_]] {
  def persist(event: EnrichedEvent): F[Unit]
}
```

Type Class Instances (using Cats and Circe)

```
import cats._
import cats.implicits._
import cats.effect._
import io.circe._, io.circe.generic.auto._, io.circe.parser._

class JsonEventParser[F[_]: Sync] extends EventParser[F] {
  override def parse(raw: RawEvent): F[EnrichedEvent] = Sync[F].delay {
    val eventType = raw.eventType match {
      case "click"    => Click
      case "view"     => View
      case "purchase" => Purchase
      case _          => throw new Exception("Unknown event type")
    }

    val sessionId = java.util.UUID.randomUUID().toString
    val geo = if (raw.metadata.contains("geo")) Some("USA") else None

    EnrichedEvent(
      userId = raw.userId,
      timestamp = raw.timestamp,
      eventType = eventType,
      sessionId = sessionId,
      geoLocation = geo
    )
  }
}
```

Concrete Persister (Console logger for simplicity)

```
class ConsoleEventPersister[F[_]: Sync] extends EventPersister[F] {
  override def persist(event: EnrichedEvent): F[Unit] =
```

```
    Sync[F].delay(println(s"[Persisted] $event"))
}
```

Streaming Pipeline (Using FS2)

```
import fs2._
import scala.concurrent.duration._

object EventStreamProcessor {

  def stream[F[_]: Temporal](
    parser: EventParser[F],
    persister: EventPersister[F]
  ): Stream[F, Unit] = {

    val simulatedKafka: Stream[F, RawEvent] = Stream.awakeEvery[F](1.second).map {
_ =>
      RawEvent(
        userId = java.util.UUID.randomUUID().toString,
        timestamp = Instant.now,
        eventType = "click",
        metadata = "geo:US"
      )
    }

    simulatedKafka
      .evalMap(parser.parse)
      .evalTap(e => Sync[F].delay(println(s"[Parsed] $e")))
      .evalMap(persister.persist)
  }
}
```

Tagless Final Program Entry Point

```
object MainApp extends IOApp {

  override def run(args: List[String]): IO[ExitCode] = {
    val parser = new JsonEventParser[IO]
    val persister = new ConsoleEventPersister[IO]

    EventStreamProcessor
      .stream[IO](parser, persister)
      .compile
```

```
      .drain
      .as(ExitCode.Success)
  }
}
```