Lab 3.1 — Portfolio Analytics with the Flink DataStream API

Session 3.1 — Data Streaming at Scale
Guided lab
Tools: Scala CLI, Apache Flink 2.2, DataStream API

1. Objective

Build a streaming portfolio analytics pipeline using the Flink DataStream API — no SQL client. Each step introduces one API primitive, building on the previous one, until the final pipeline tracks net positions, windowed notional, live market values, and inactivity alerts simultaneously.

2. Setup

2.1. Prerequisites

- Scala CLI (`scala-cli`) installed and on `PATH`
- Apache Flink 2.2 distribution — start the local cluster:

```
./bin/start-cluster.sh
```

- Flink Web UI at `http://localhost:8081`

2.2. Data generators

Two generators produce JSONL files at 1–4 second intervals. About 20 % of records carry a backdated `ts` (up to 15 s late).

Run each in a separate terminal before launching the pipeline:

```
scala-cli generate-trades.scala
scala-cli generate-prices.scala
```

`generate-trades.scala` → `data/trades/`

```
{"symbol":"AAPL","side":"buy","quantity":350,"price":182.14,"ts":"2025-01-15T10:23:45Z"}
```

`generate-prices.scala` → `data/prices/` (and `data/prices-baseline.json`)

```
{"symbol":"NVDA","price":876.32,"ts":"2025-01-15T10:23:46Z"}
```

2.3. Lab file

The complete pipeline is in `lab.scala`. Run it with:

```
scala-cli run . --main-class lab
```

The `project.scala` file in the same directory declares all Flink 2.2 dependencies — no `build.sbt` or Maven needed. Work through the steps below by reading, running, and modifying `lab.scala`.

3. Walkthrough

3.1. Step 1 — Sources and watermarks

Goal: connect Flink to the generator directories and declare event-time semantics.

The `fileSource` helper in `lab.scala` creates a `FileSource` that watches a directory and picks up new files every 2 seconds:

```
def fileSource(dir: String): FileSource[String] =
  FileSource
    .forRecordStreamFormat(new TextLineInputFormat(), new Path(new File(dir).toURI))
    .monitorContinuously(Duration.ofSeconds(2))
    .build()
```

Each line is parsed from JSON and assigned a watermark strategy:

```
val trades = rawTrades
  .flatMap(parseTrade(_).toList.asJava)
  .assignTimestampsAndWatermarks(
    WatermarkStrategy
      .forBoundedOutOfOrderness[Trade](Duration.ofSeconds(15))
      .withTimestampAssigner((t, _) => t.epochMillis)
  )
```

The 15 s bound matches the maximum late-arrival delay in the generators. Run the pipeline and open the Flink Web UI — find the running job, click the source operator, and watch the watermark metric advance.

Observe: the watermark does not advance continuously. It advances in steps as new events arrive and push `max(seen_ts)` forward.

3.2. Step 2 — Running net position (ValueState)

Goal: maintain a per-symbol running net quantity using `KeyedProcessFunction` and `ValueState`.

```
class NetPositionTracker extends KeyedProcessFunction[String, Trade, String]:
  lazy val position: ValueState[Long] = getRuntimeContext.getState(
    new ValueStateDescriptor("position", classOf[Long])
  )

  override def processElement(t: Trade, ctx: Context, out: Collector[String]): Unit =
    val prev = Option(position.value()).getOrElse(0L)
    val next = prev + t.delta          // +qty for buy, -qty for sell
    position.update(next)
    out.collect(s"${t.symbol}  net_qty=${next}")
```

Wired into the pipeline with:

```
trades.keyBy(_.symbol).process(new NetPositionTracker()).print()
```

Observe: every trade emits one output record immediately — there is no buffering. The state for each symbol lives on exactly one sub-task (check the parallelism in the Web UI).

Experiment: add `ListState[Trade]` to buffer individual trades and replay them. When would you need this?

3.3. Step 3 — Windowed notional (tumbling windows + allowed lateness + side output)

Goal: aggregate notional per symbol over 30-second event-time windows, with an explicit policy for late arrivals.

```
val lateTag = new OutputTag[Trade]("late-trades") {}

val windowedResult = trades
  .keyBy(_.symbol)
  .window(TumblingEventTimeWindows.of(Duration.ofSeconds(30)))
  .allowedLateness(Duration.ofSeconds(15))
  .sideOutputLateData(lateTag)
  .aggregate(new NotionalAgg(), new WindowLabel())
```

`NotionalAgg` accumulates `netQty` and `notional` incrementally — Flink calls it once per record, not once per window. `WindowLabel` is a `ProcessWindowFunction` that adds window start/end metadata to the result.

The three zones for a record with event timestamp `ts`:

Zone	What happens
`ts` within watermark lag	Included in window before it closes — normal path
`ts` within allowed lateness	Window re-opens; updated result re-emitted downstream
`ts` beyond both	Routed to `lateTag` side output — never silently dropped

Observe: window results appear in bursts, not one per record. Each burst corresponds to the watermark crossing a 30-second boundary. The `[LATE]` prefix in the console output marks records reaching the side output.

Experiment: set allowed lateness to 0. How does the output change? Set the watermark bound to 0. What happens to on-time records that the generator backdates?

3.4. Step 4 — Live market value (KeyedBroadcastProcessFunction)

Goal: enrich each trade with the latest market price and compute its market value, using the prices stream as a broadcast source.

The prices stream carries price ticks for all symbols. Rather than joining two unbounded streams (which buffers everything), we broadcast the low-volume price stream to every sub-task and let each sub-task enrich the trades it owns:

```
val broadcastPrices = prices.broadcast(priceStateDesc)

trades
  .keyBy(_.symbol)
  .connect(broadcastPrices)
  .process(new PortfolioEnricher())
  .print()
```

Inside `PortfolioEnricher`:

```
// Called for every price tick — all sub-tasks receive every tick
override def processBroadcastElement(tick, ctx, out) =
  ctx.getBroadcastState(priceStateDesc).put(tick.symbol, tick.price)

// Called for every trade — reads the latest price for this symbol
override def processElement(trade, ctx, out) =
```

```
  val marketPrice = Option(ctx.getBroadcastState(priceStateDesc).get(trade.symbol))
    .getOrElse(trade.price)
  out.collect(s"${trade.symbol} Δvalue=${trade.delta * marketPrice}")
```

Observe: the first few trades may use the trade price as fallback (no price tick seen yet). As the price generator runs, market prices update and diverge from trade prices.

Compare: the state footprint is one `Double` per symbol per sub-task — constant. An unbounded stream–stream join would grow linearly with the number of trades buffered.

Note: in `processBroadcastElement`, keyed state is read-only. All mutations to broadcast state happen here; keyed state is mutated in `processElement` only.

3.5. Step 5 — Inactivity alert (event-time timers)

Goal: emit an alert when no trade has been seen for a symbol within a 20-second event-time window.

```
override def processElement(t: Trade, ctx: Context, out: Collector[String]): Unit =
  lastSeen.update(ctx.timestamp())
  ctx.timerService().registerEventTimeTimer(ctx.timestamp() + ALERT_HORIZON_MS)

override def onTimer(ts: Long, ctx: OnTimerContext, out: Collector[String]): Unit =
  if ts >= lastSeen.value() + ALERT_HORIZON_MS then
    out.collect(s"No trades for ${ctx.getCurrentKey} in the last 20s of event time")
```

`onTimer` fires when the watermark advances past the registered timestamp — not when wall-clock time advances. Because timers are checkpointed alongside state, they survive failures.

Observe: alerts fire for symbols that the generator has not produced recently. When the generator finishes (after 20–50 records), alerts will fire for all symbols as the watermark advances past their last-seen timestamp.

3.6. Step 6 — Fault tolerance

Goal: verify that all five pipelines recover from a TaskManager failure without losing state.

Checkpointing is already enabled in `lab.scala`:

```
env.enableCheckpointing(10_000)  // every 10 s
```

While the pipeline is running:

```
jps | grep TaskManager       # find the pid
kill <pid>                   # simulate a crash
```

Watch the Web UI:

- the job enters **RESTARTING**
- it recovers to the last checkpoint
- net positions, window accumulators, broadcast price state, and timer registrations are all restored
- `FileSource` offsets are part of the checkpoint — already-processed files are not re-read

Discuss: the `print()` sink is not transactional. On recovery, some lines may be printed twice. What would you need to add to guarantee exactly-once output?

4. Key Takeaways

- `KeyedProcessFunction` + `ValueState` is the foundation: any aggregation that SQL can express can be implemented here, plus anything SQL cannot

- Tumbling windows with `allowedLateness` give you a two-phase commit: a first result when the watermark closes the window, then corrections within the grace period
- The side output is the production answer to "what do I do with truly late records?" — route them, not drop them
- Broadcast state is the efficient alternative to stream–stream join when one side is low-volume and shared across all keys
- Event-time timers fire on watermark progress, not wall-clock time — they are reproducible, checkpointed, and safe to use in stateful operators