EXAMPLE 04

Add processing to an existing application

You can evaluate generated event processing inside an application you already operate. This example uses a small Java host that owns the processor lifecycle and reads each decision without a separate Mongoose service.

4 valid orders checked · Invalid input refused · Closed host refuses events

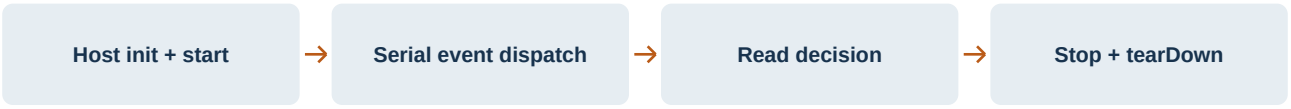


Illustration of the demonstrated workflow.

Observed result

The host receives REJECTED, ACCEPTED and REJECTED for three successive orders. A zero-quantity order is refused; a later equality case is accepted. After the host closes, another submission is refused. The run uses the included generated processor and runtime JAR, with no Mongoose service or generation request.

Case or component	Recorded result
Sequential orders	REJECTED / ACCEPTED / REJECTED
Zero quantity	Refused before dispatch
Next valid boundary order	ACCEPTED
Submission after close	Refused by host

Synthetic demonstration with recorded results, not a customer case study.

FOLLOW THE DEMONSTRATION

4-stage walkthrough

1. Start under host ownership

A plain Java application creates, initializes and starts the generated processor.

2. Submit an order

The host serializes access and supplies an explicit quantity, price and limit snapshot.

3. Read and check the outcome

The host receives rejected, accepted and rejected results, refuses an invalid input, then accepts a valid boundary case.

4. Close the boundary

The host stops and tears down the processor. A later submission is refused.

Why this matters

This fixture shows a concrete incremental adoption boundary: an existing Java host owns lifecycle and receives synchronous decisions. The demonstration works without a separate Mongoose service; the right production operating model can be assessed in a pilot.

Apply it to your process

Choose an existing event boundary and define a small integration slice. Evaluate lifecycle ownership, backpressure, error handling, state and observability in the real host before choosing whether embedded operation or Mongoose is the better fit.

Discuss a five-stage pilot

TECHNICAL APPENDIX

Reproduce this result

Unzip the host reproduction package, then run from its embedded-host directory. The sibling order-check tree supplies the processor and host class:

```
python3 run.py \
  --runtime /path/to/fluxtion-runtime-1.1.3.jar \
  --out results
```

The runner verifies the runtime hash, compiles the supplied processor and host, then checks the host lifecycle and sequential decisions. It captures the summary, portable commands and source hashes. No generation, model or Mongoose service is called.

Source and reproduction package for this example

This runner builds the released generated source. Upstream describes compiler 1.0.83; this example does not require a generation call.

Runtime prerequisite: com.telamin.fluxtion:fluxtion-runtime:1.1.3. Download the JAR without an account; the runner checks its SHA-256. The source ZIP includes RUNTIME-PREREQUISITES.txt with the complete download command.

Download runtime JAR | Published POM: declares GNU AGPL version 3, separate from Apache 2.0 demo code.

SHA-256: 2bd45eb62409403422d5a676a7ad177f1984aa80dc399c20ceb6b467eab4b775

Detailed evidence

summary.txt

commands.json

manifest.json

Scope and interpretation

A minimal Java host and synthetic inputs, not integration into a customer application.

No latency, throughput, memory, crash-recovery or GraalVM-native result is reported. The capture suite and embedded-host suite are separate runs.

Generation entitlements, runtime licensing and production or redistribution rights remain separate; technical independence from the hosted compiler does not determine those rights.

Demo code is Apache 2.0 under owner authorization dated 1 November 2026. Runtime, third-party and general customer-output terms remain separate. Source and artifact hashes compare bytes; they do not authenticate a sender or certify correctness.

Deployment and data flows | Applicable product rights