Axon Training

Module 5 – Snapshotting and Event Processors

Agenda

Week 1

1. DDD and CQRS Fundamentals
2. Command Model
3. Event Handling & Projections
4. Sagas and Deadlines

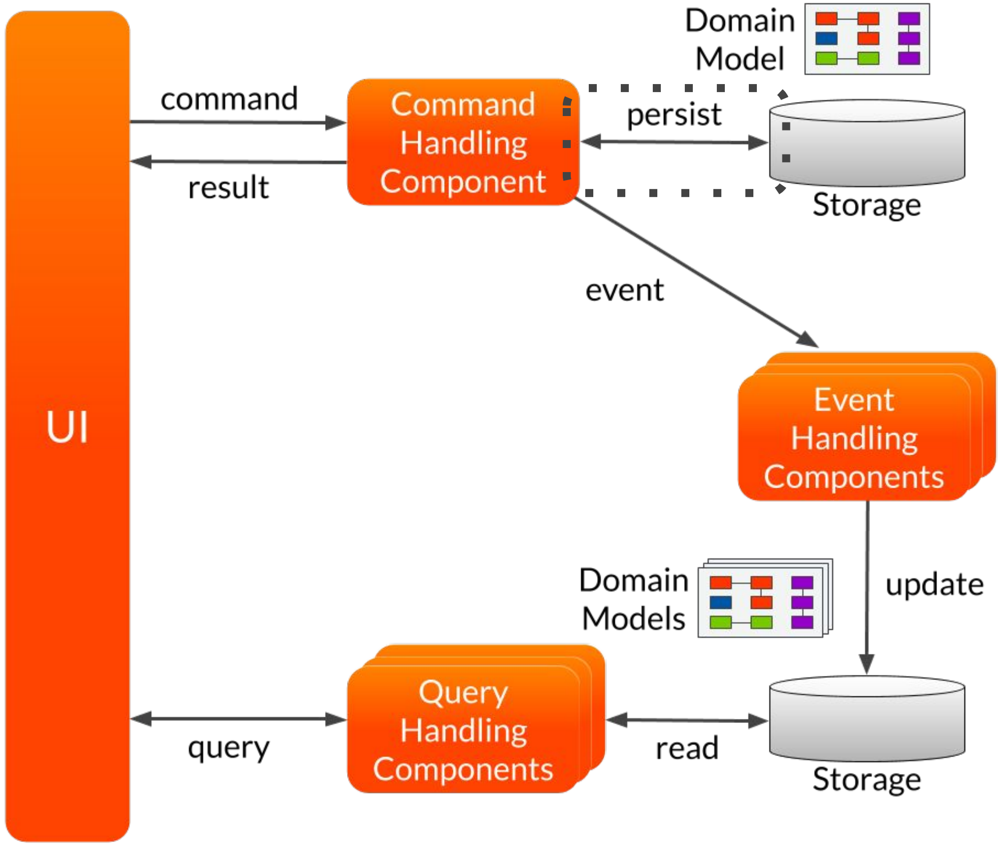
Week 2

1. **Snapshotting and Event Processors**
2. Preparing for Production
3. CQRS and Distributed Systems
4. Monitoring, Tracing, Advanced Tuning

Compacting the event stream

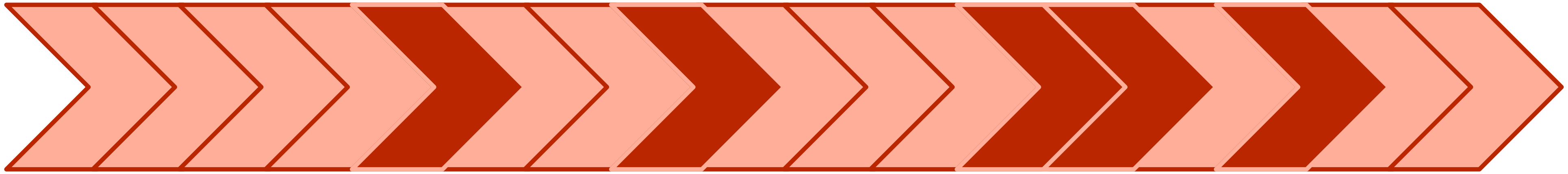
Snapshotting

Snapshotting

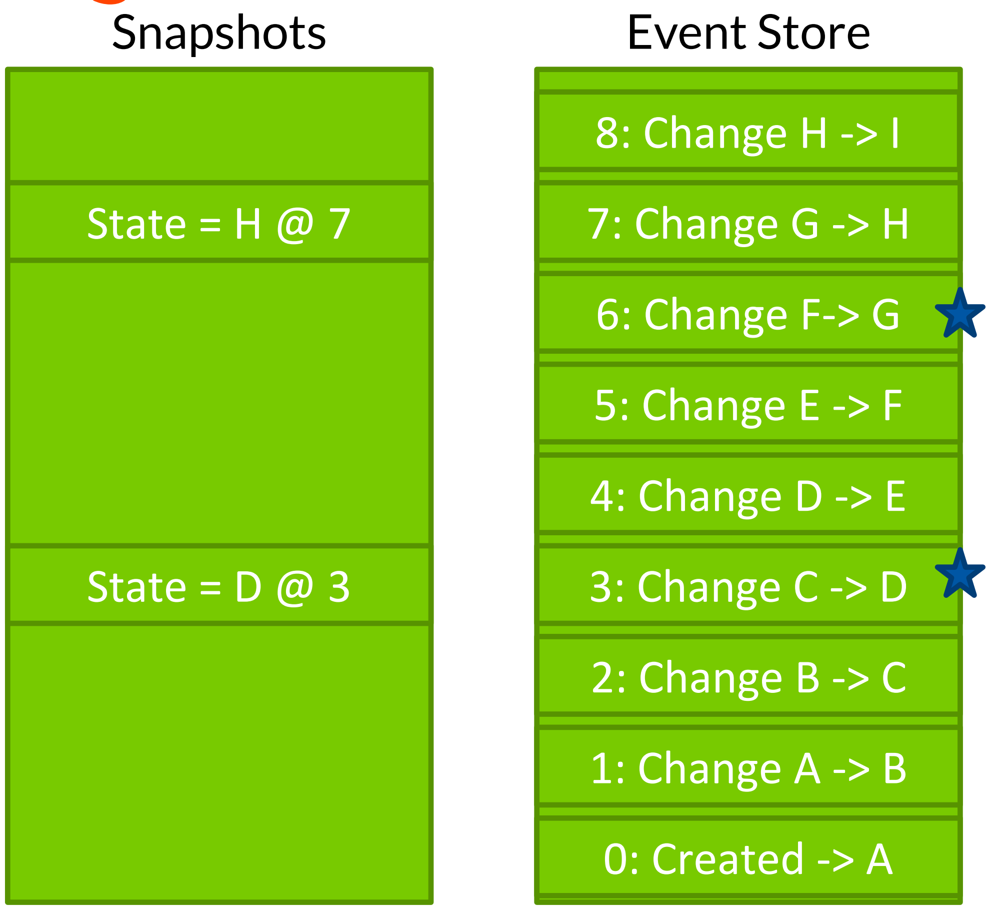


Event Store operations

- Read aggregate’s events



Snapshotting



Snapshots

State = H @ 7

State = D @ 3

Event Store

8: Change H -> I

7: Change G -> H

6: Change F-> G

5: Change E -> F

4: Change D -> E

3: Change C -> D

2: Change B -> C

1: Change A -> B

0: Created -> A

Snapshotting

- Snapshots are a (temporary) replacement for a set of historical events

- Snapshotting may be an asynchronous process
    - Regular intervals
    - After x events
    - When loading takes >= x ms

Snapshot Event

- Snapshot Events contain all relevant information needed to reconstruct an Aggregate's state at that point in time.

- Axon's default: Use the actual Aggregate's state as snapshot

- Note: Snapshot Events are ***never*** published

Configuring Snapshots

```
// in the configuration of the aggregate
AggregateConfigurer<Flight> flightConfig = AggregateConfigurer.defaultConfiguration(Flight.class);

// we define the trigger
flightConfig.configureSnapshotTrigger(
      c -> new EventCountSnapshotTriggerDefinition(c.getComponent(Snapshotter.class), 100)
);

// in our main configuration, we provide an AggregateSnapshotter that we can reuse
Configurer config = DefaultConfigurer.defaultConfiguration()
       .registerComponent(Snapshotter.class, c -> AggregateSnapshotter.builder()
                                .aggregateFactories(new GenericAggregateFactory<>(Flight.class))
                                .eventStore(c.eventStore())
                                .transactionManager(c.getComponent(TransactionManager.class))
                                .build());
```

Configuring Snapshots - Spring

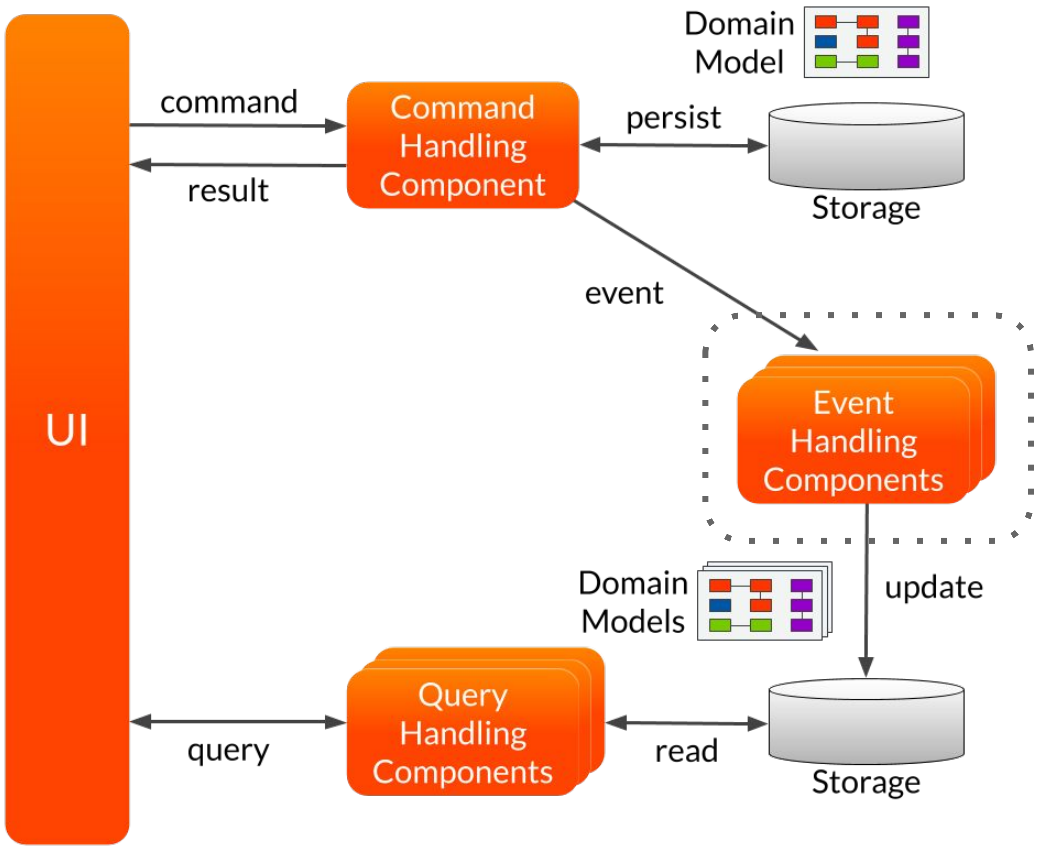
```
// in the configuration of the aggregate
@Aggregate(snapshotTriggerDefinition = "myTriggerDefinition")
public class Flight {
   ...
}

// we define the trigger. The snapshotter is automatically configured
@Bean
public SnapshotTriggerDefinition myTriggerDefinition(Snapshotter snapshotter) {
   return new EventCountSnapshotTriggerDefinition(snapshotter, 100);
}
```

Processing what happened…

Event Processors & Replays

Event Processing

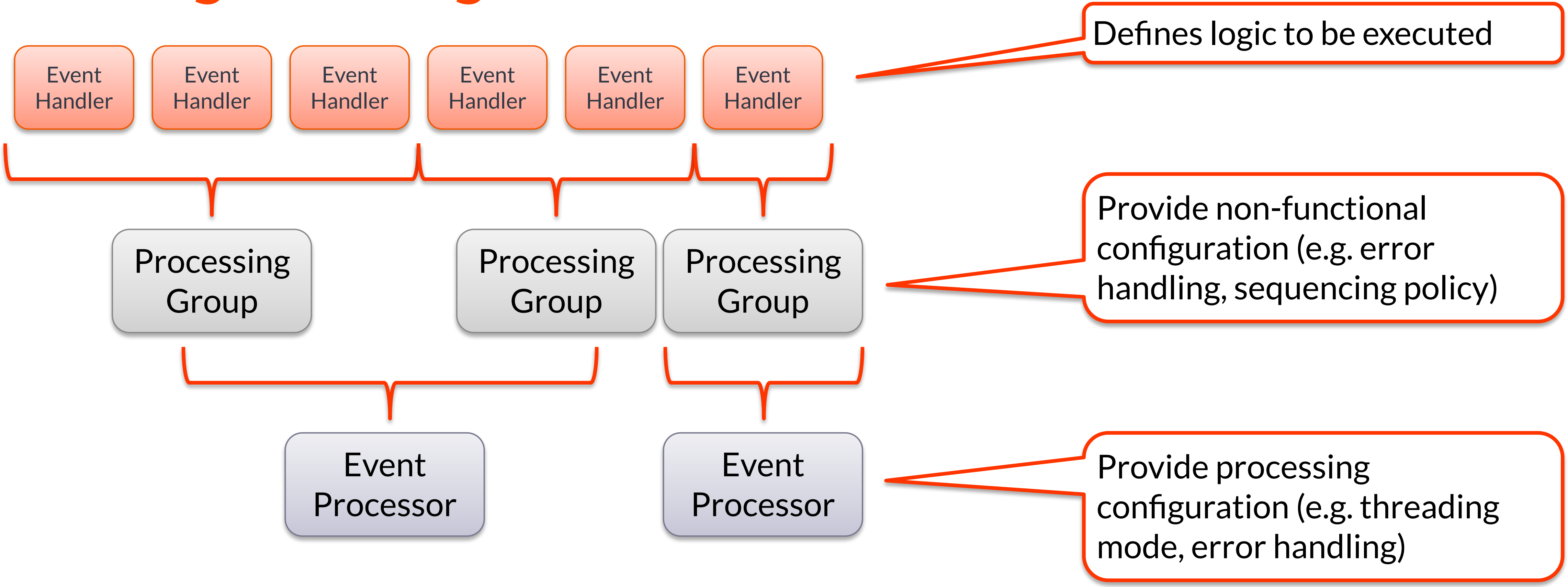


Organizing Event Handlers

- Event Processor
  - Responsible for managing the technical aspect of processing an Event
  - Starts and Commits 'Unit of Work'
  - Invokes handler methods

- Each handler is assigned to a single Processor
  - `@ProcessingGroup` on Event Handler class
  - Assignment rules in & `EventProcessingConfigurer` (part of Configuration API)

Organizing Event Handlers



Defines logic to be executed

Provide non-functional configuration (e.g. error handling, sequencing policy)

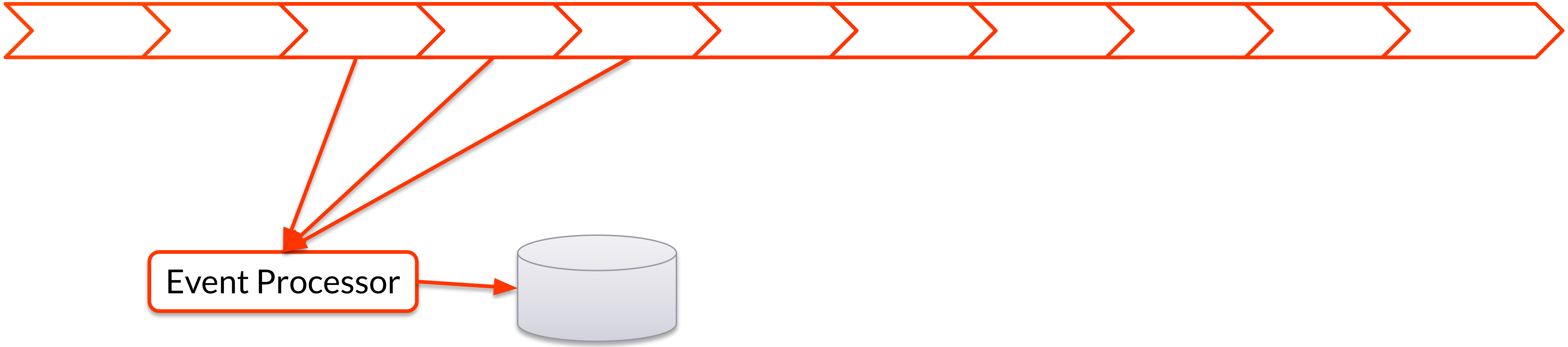
Provide processing configuration (e.g. threading mode, error handling)

Event Processors

- SubscribingEventProcessor
  - Receives messages as they are published, in the thread that publishes the messages
  - Requires a Subscribable Message Source

- TrackingEventProcessor (default *)
  - Uses its own thread(s) to read EventMessages from a Stream
  - Requires a Streamable Message Source
  - Saves progress using TrackingToken

Tracking Token



Event Processor Configuration

```
public void configure(EventProcessingConfigurer configurer) {
    StreamableMessageSource<TrackedEventMessage<?>> source = …;
    configurer.registerTrackingEventProcessor("com.example.viewmodel", c -> source);
}
```

The name of the processor to explicitly register.
It is only created when handlers are actually assigned to it.

A function returning the source to read from, given Configuration c.

Or in Spring Boot with `application.properties`:

```
axon.eventhandling.processors.processor-name.source=message-source-bean-name
axon.eventhandling.processors.processor-name.mode=tracking
```

Tracking Event Processor Configuration

- **Batch Size -** The number of events that are processed in a single transaction
- **Initial Token -** The position at which a processor must start when initializing
- **Initial Segment Count -** The number of segments to create when initializing
- **Thread Count** – The maximum number of Threads the processor may start
- **Event Availability Timeout** – Time to wait for events before updating the claim
- **Token Claim Interval** – How long to wait between attempts to claim a segment

Error Handling

- Exceptions thrown while handling an Event
  - `ListenerInvocationErrorHandler`
  - Defined on Processing Group
  - Default: log error and proceed
  - Rethrow to trigger ErrorHandler

- Exceptions that fail the transaction
  - `ErrorHandler`
  - Defined on Event Processor
  - Default: rollback, release segment claim, and retry (with incremental back-off)

Thread Count and Segmentation

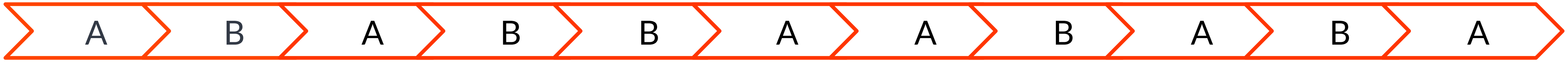
- A segment can only be processed by a single thread at a time
- A single thread will process a single segment

- At any time: `total thread count ≥ segment count`
  - Otherwise: unclaimed segment / partial processing

- Initial Segment Count *only* works when initializing a processor
- At runtime, use Split and Merge to increase/decrease segment count

Tracking Processor Segments

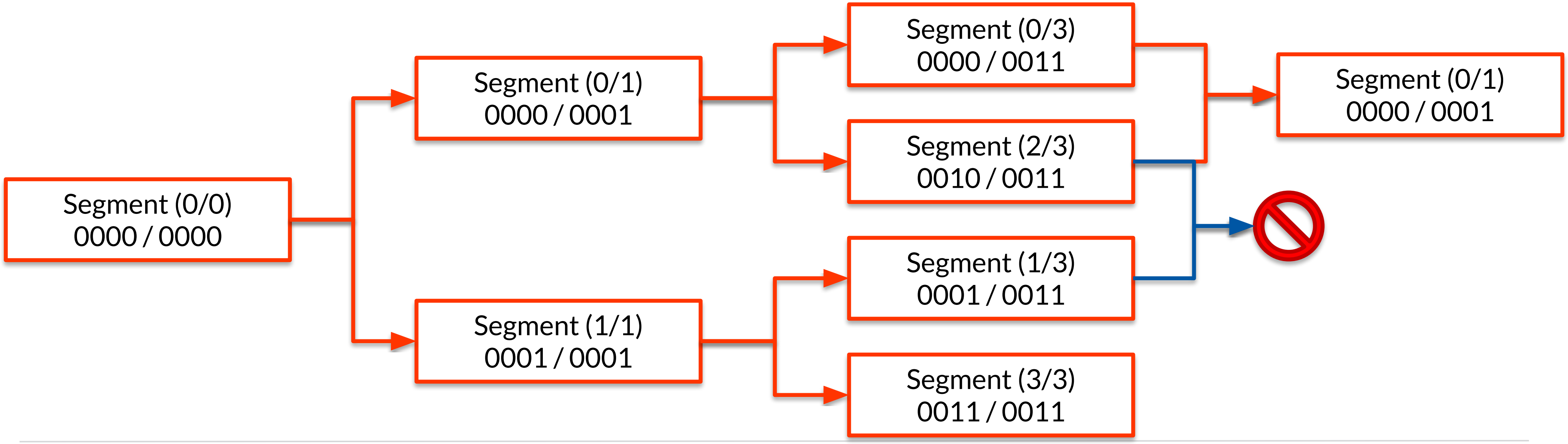
- Multi-threading and/or multi-node
- Each thread “claims” a segment in `TokenStore`
- `SequencingPolicy` defines segment
  - the same value for two messages means they ‘belong’ to same segment
  - Message in same segment are always handles sequentially
  - E.g. `SequentialPerAggregatePolicy`



A B A B B A A B A B A

Segmentation – Split and Merge

- Segmentation is dynamic
  - Split splits a claimed segment into 2 segments
  - Merge merges two segments into their original combined form



Replays

- Tracking Processors can be “reset”
  - Clean up any state their handlers have
  - Reset all tokens for that processor

- Tracking Processor replay status
  - `@AllowReplay` / `@DisallowReplay` – *indicates whether components can deal with replays*
  - `ReplayStatus` – *handler parameter to add conditional logic*
  - `@ResetHandler` – *handler invoked when a replay is triggered*

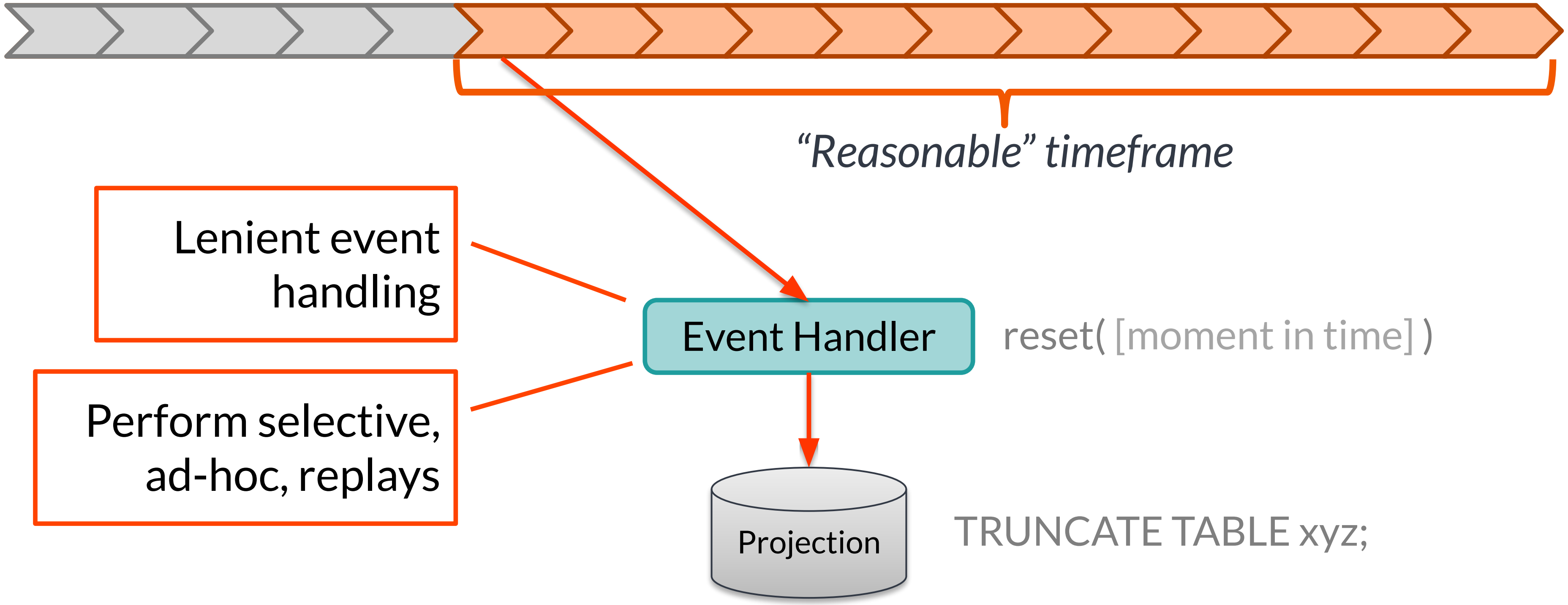
Replays - API

```
class FlightStatusProjection {
   @DisallowReplay
   @EventHandler
   public void on(FlightDelayed event) {
       // This handler is not invoked when replaying
   }

   @EventHandler
   public void on(ArrivalTimeChanged event, ReplayStatus replayStatus) {
       if (replayStatus != ReplayStatus.REPLAY) {
           // This block is not invoked when replaying
       }
       // ...
   }

   @ResetHandler
   public void reset() {
       // Invoked when replay is triggered
       // e.g. to clear out the view's database
   }
}
```

Partial replays



"Reasonable" timeframe

Lenient event handling

Perform selective, ad-hoc, replays

Event Handler

reset([moment in time])

Projection

TRUNCATE TABLE xyz;

Triggering a reset

A reset requires that a single processor can update ***all*** tokens simultaneously. This is only possible when the processor is stopped

1. Stop all processors
2. Ask a single processor to “reset” all tokens
3. Start all processors

In a distributed environment, the AxonServer API/UI can be used to stop all instances of a processor. AxonServer ***does not*** trigger resets.

Triggering a reset - API

```
// Get access to the processors with the configuration
EventProcessingConfiguration eventProcessingConfiguration = ...;

// and if you know the name of the processor to reset,
String processorName = ...;

// then you can execute a reset
eventProcessingConfiguration.eventProcessor(processorName, TrackingEventProcessor.class)
                           .ifPresent(trackingEventProcessor -> {
                               trackingEventProcessor.shutDown();
                               trackingEventProcessor.resetTokens();
                               trackingEventProcessor.start();
                           });
```

Whatever else you wanted to know…

Questions