Axon Training

Module 8 – Monitoring, Tracing and Advanced Tuning

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**

Big Brother is watching you…

Monitoring

Messaging infrastructure

- Message flow provides valuable information about component health
- Cause-and-effect flow gives insight in what's happening

Message Monitors

- Axon infrastructure components allow for `MessageMonitors`
  - Invoked on ingest and after processing of message
  - Measure throughput, response times, utilization, etc.

Message Monitors - Implementations

- Generic monitors
  - `NoOpMessageMonitor`
  - `MultiMessageMonitor`

- Dropwizard Metrics / Micrometer monitors
  - `MessageTimerMonitor`
  - `CapacityMonitor`
  - `MessageCountingMonitor`
  - `EventProcessorLatencyMonitor`
  - `PayloadTypeMessageMonitorWrapper`

Configuring Message Monitors

```
// With Axon's Configuration API in place...
Configurer configurer = DefaultConfigurer.defaultConfiguration();

// a registry (micrometer in this sample) can be build
MeterRegistry meterRegistry = ...;

// GlobalMetricRegistry is a quick handle to configure Axon's default set of MessageMonitors
GlobalMetricRegistry metricRegistry = new GlobalMetricRegistry(meterRegistry);

// and by registering it with the Configurer, you are done.
metricRegistry.registerWithConfigurer(configurer);



// For custom MessageMonitors the Configurer should be called
CustomMessageMonitor customMessageMonitor ...
configurer.configureMessageMonitor(CommandBus.class, configuration -> customMessageMonitor);
```

A dependency on `axon-metrics` or `axon-micrometer` is required

Configuring Message Monitors - Spring

```
// With Spring Boot, just this dependency is sufficient for the default MessageMonitors

// Maven
<dependency>
   <groupId>org.axonframework</groupId>
   <artifactId>axon-micrometer</artifactId>
   <version>{axon-version}</version>
</dependency>


// Gradle
implementation 'org.axonframework:axon-micrometer:{axon-version}'
```

Tracking Event Processor Status

Tracking Event Processor status is an important health-indicator as well

- Overall state
  - `isRunning()`
  - `isError()`
- Thread counts
  - `availableProcessorThreads()`
  - `activeProcessorThreads()`

Tracking Event Processor Status

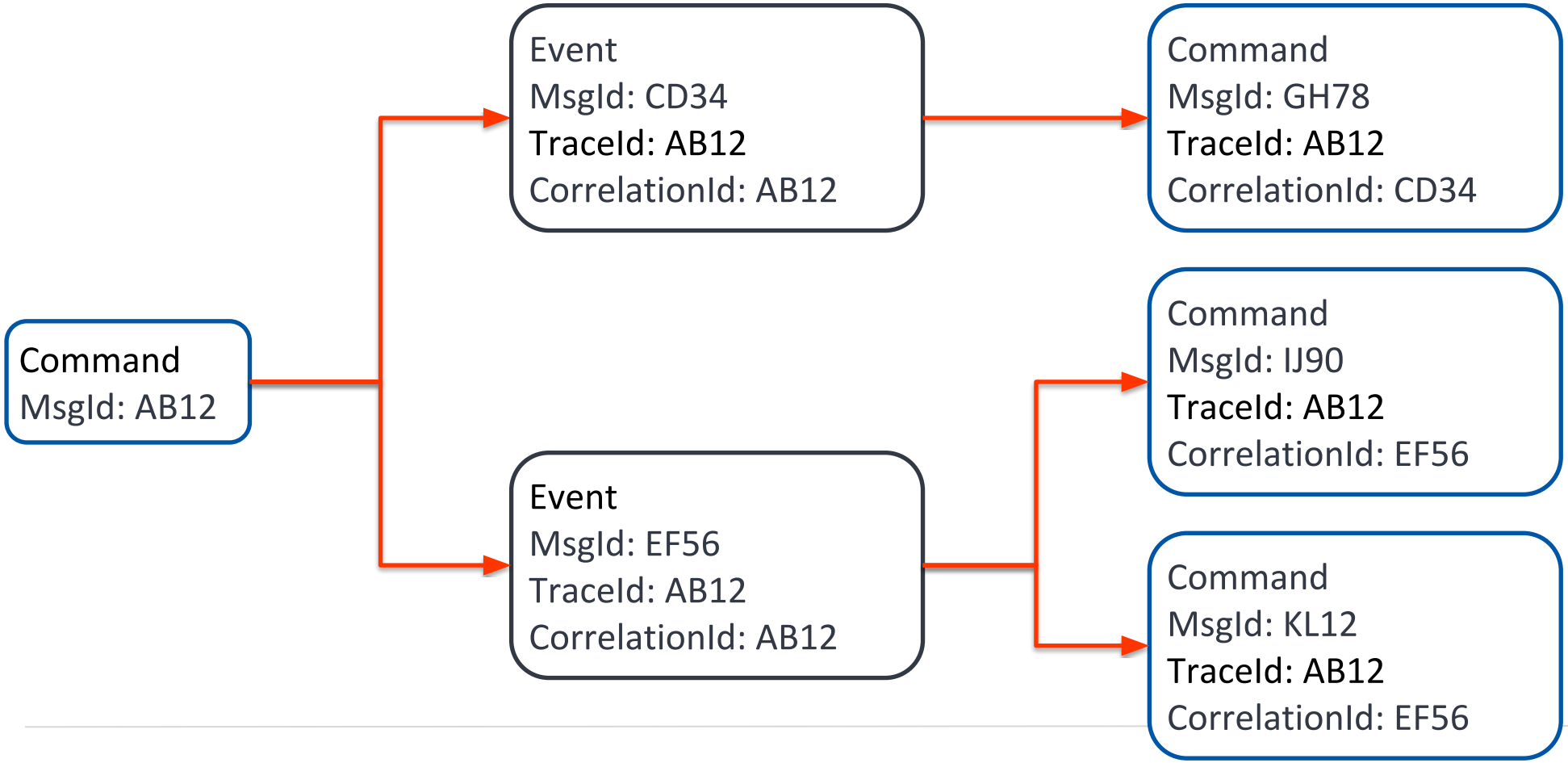
- Status provides information per active segment in a processor
  - `boolean isCaughtUp();`
  - `boolean isReplaying();`
  - `boolean isMerging();`
  - `TrackingToken getTrackingToken();`
  - `OptionalLong getCurrentPosition();`
  - `OptionalLong getResetPosition();`
- Overall state (`getState()`)
  - `NOT_STARTED`
  - `STARTED`
  - `PAUSED / SHUTDOWN`
  - `PAUSED_ERROR`

Correlation Data Providers

- Unit of Work attaches correlation data
  - Based on incoming message
  - Attached to all outgoing messages

- `CorrelationDataInterceptor`
- `CorrelationDataProvider`
  - `MessageOriginProvider (correlationId, traceId)`

MessageOriginProvider – traceId and correlationId



Dropping the breadcrumbs

Tracing

Distributed Tracing

- Tracing is a great tool for additional monitoring
  - Especially in a distributed application

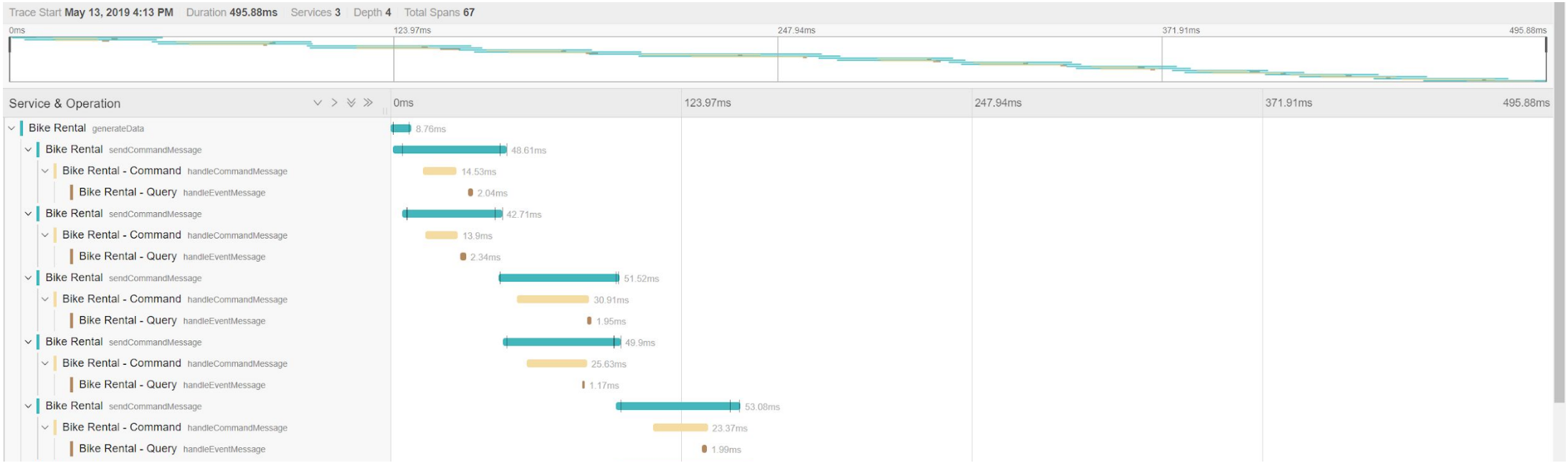
- Several API's out there
  - OpenTracing → opentracing.io
  - OpenCensus → opencensus.io
  - OpenTelemetry (merger of the former two) → opentelemetry.io

- Axon provides the Tracing Extension, based on OpenTracing

Distributed Tracing - Axon

- Tracing Extension exist out of several components:
  - `TracingCommandGateway`
  - `TracingQueryGateway`
  - `OpenTraceDispatchInterceptor`
  - `OpenTraceHandlerInterceptor`
  - `TracingProvider`

Distributed Tracing - Sample UI



Configuring Tracing

```
// Build the Tracer from the preferred implementation and the dispatch and handler interceptor,
Tracer tracer = ...;
OpenTraceDispatchInterceptor traceDispatchInterceptor = new OpenTraceDispatchInterceptor(tracer);
OpenTraceHandlerInterceptor traceHandlerInterceptor = new OpenTraceHandlerInterceptor(tracer);

// then configure the TracingProvider as a CorrelationDataProvider,
List<CorrelationDataProvider> correlationDataProviders = Collections.singletonList(new TracingProvider(tracer));
Configuration configuration = DefaultConfigurer.defaultConfiguration()
      .configureCorrelationDataProviders(config -> correlationDataProviders)...

// Ending with configuring the handler interceptor
configuration.commandBus().registerHandlerInterceptor(traceHandlerInterceptor);

// and building the dedicated gateways with the dispatch interceptor
TracingCommandGateway tracingCommandGateway = TracingCommandGateway.builder().tracer(tracer)
      .delegateCommandGateway(configuration.commandGateway()).build();
tracingCommandGateway.registerDispatchInterceptor(traceDispatchInterceptor);
```

Configuring Tracing - Spring

```
// With Spring Boot, just this dependency is sufficient

// Maven
<dependency>
   <groupId>org.axonframework.extensions.tracing</groupId>
   <artifactId>axon-tracing-spring-boot-starter</artifactId>
   <version>{extension-version}</version>
</dependency>


// Gradle
implementation
'org.axonframework.extensions.tracing:axon-tracing-spring-boot-starter:{extension-version}'
```

But wait, there's more!

Advanced Tuning

There’s a time and place for every task

Unit of Work

Unit of Work

- Records Message and Execution Result
- Coordinate lifecycle of message handling
  - start → prepare commit → commit → after commit → cleanup
    → rollback → cleanup

- Register for resources used during processing
  - e.g. Database connections
- Correlation data management
  - Correlation data automatically attached to generated messages

Unit of Work

- To access the current Unit of Work
  - Parameter on Message Handler method
  - `CurrentUnitOfWork.get();`

- Unit of Work is created by all components processing Messages
- Only one Unit of Work can be active at any time
  - Nesting is supported

UoW – Message and Execution Result

- `T getMessage();`
- `ExecutionResult getExecutionResult()`

- `boolean isRolledBack()`

- ```
  transformMessage(
      Function<T, ? extends Message<?>> transformOperator
  )
  ```
  ```

UoW – Hooking into the lifecycle

- `phase()`

- `onPrepareCommit(Consumer<UnitOfWork<T>> handler)`
- `onCommit(Consumer<UnitOfWork<T>> handler)`
- `afterCommit(Consumer<UnitOfWork<T>> handler)`
- `onRollback(Consumer<UnitOfWork<T>> handler)`
- `onCleanup(Consumer<UnitOfWork<T>> handler)`

UoW – Registering Resources

- `Map<String, Object> resources()`
- `R getResource(String name)`
- `R getOrDefaultResource(String key, R defaultValue)`
- `R getOrComputeResource(`
  `String key,`
  `Function<? super String, R> mappingFunction`
  `)`

Register resources that should be reused with the `root()` Unit of Work.

UoW – Correlation Data Management

- `registerCorrelationDataProvider( CorrelationDataProvider correlationDataProvider )`
- `MetaData getCorrelationData()`

Used by constructors of all Message implementations to initialize MetaData
(except copy-constructors)

```
interface CorrelationDataProvider {
   Map<String, ?> correlationDataFor(Message<?> message);
}
```

Injecting dependencies…

Parameter Resolvers

Parameter Resolvers

- Resolves parameters of `@MessageHandler` methods
  - Based on incoming Message
  - Resolves single parameter value

- Explicitly configured on components
- Located using `ServiceLoader`
  - `META-INF/services/org.axonframework.messaging.annotation.ParameterResolverFactory`
- As Spring bean in the `ApplicationContext`

Parameter Resolvers - API

```
@FunctionalInterface
public interface ParameterResolverFactory {
   ParameterResolver createInstance(
      Executable executable, Parameter[] parameters, int parameterIndex
   );
}
```

```
public interface ParameterResolver<T> {
   T resolveParameterValue(Message<?> message);

   boolean matches(Message<?> message);
}
```

Enhance the handling experience

Handler Enhancers

Handler Enhancers

- All message handlers are (meta-)annotated with `@MessageHandler`
- Type specific logic is implemented as Handler Enhancers
  - Wrap handler method
  - Provide additional information about handler (e.g. routing keys)
  - Add additional behavior to handler (e.g. end Saga lifecycle)

- Explicitly configured on components
- Configure using ServiceLoader
  - `META-INF/services/org.axonframework.messaging.annotation.HandlerEnhancerDefinition`
- As Spring bean in the `ApplicationContext`

Handler Enhancers - API

```
public interface HandlerEnhancerDefinition {
    <T> MessageHandlingMember<T> wrapHandler(
        MessageHandlingMember<T> original
    );
}
```

- For convenience, return a `WrappedMessageHandlingMember` instance
- Return “original” to reject “enhancement”

Handler Enhancers - Use Cases

- More specific alternative to Handler Interceptor
  - Handler Enhancers have more information about handling type / instance
- React to (additional) annotations on Handler methods
  - e.g. Security annotations
- Detailed logging / tracing

Whatever else you wanted to know...

Questions

That’s all folks!

Survey - https://lp.axoniq.io/survey-axoniq

Thanks for attending!