Axon Training

Module 6 – Preparing for production

AxonIQ

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

Cross cutting concerns

Message Interceptors

Dispatch Interceptor

- Invoked in the thread that dispatches the Message
- Active Unit of Work is that of incoming message (if any)
- Allows transformation of Message or force failure

Handler Interceptor

- Invoked in thread that handles Message
- Active Unit of Work is that of intercepted message
- Can force early return / failure

Message Intercepting - Use cases

- Dispatch Interceptors
  - Structural validation
  - Attach node-id for distributed tracing
  - Attach security meta-data

- Handler Interceptors
  - Attach (database) transaction
  - Validate security meta data

Dispatch Interceptor - API

```
public interface MessageDispatchInterceptor<T extends Message<?>> {

    BiFunction<Integer, T, T> handle(List<? extends T> messages);

}
```

Handler Interceptor - API

```
public interface MessageHandlerInterceptor<T extends Message<?>> {

    Object handle(UnitOfWork<? extends T> unitOfWork,
                  InterceptorChain interceptorChain) throws Exception;

}
```

Registering Interceptors

```
// Once the configuration has been build from the Configurer
Configuration configuration = ...

// and the interceptors defined,
CustomMessageHandlerInterceptor myCustomHandlerInterceptor = ...
CustomDispatchInterceptor myCustomDispatchInterceptor = ...

// they can be registered to the right message dispatching and handling components
CommandBus commandBus = configuration.commandBus();
commandBus.registerDispatchInterceptor(myCustomDispatchInterceptor);
commandBus.registerHandlerInterceptor(myCustomHandlerInterceptor);
```

Registering Interceptors - Spring

```
// With Spring, you can auto wire the bean to register the interceptor for
@Autowired
public void configureInterceptorFor(EventBus eventBus,
                                    CustomDispatchInterceptor myCustomDispatchInterceptor) {
    eventBus.registerDispatchInterceptor(myCustomDispatchInterceptor);
}

@Autowired
public void configureInterceptorFor(EventProcessingConfigurer eventProcessingConfigurer,
                                    CustomMessageHandlerInterceptor myCustomHandlerInterceptor) {
    eventProcessingConfigurer.registerDefaultHandlerInterceptor(
        (config, processorName) -> myCustomHandlerInterceptor
    );
}
```

Mind the contract…

Serializers

XStream Serializer

- Serializes to/from XML
- Default serializer in Axon
  - Can serialize everything

- Aliases
  - Package aliases
  - Class name aliased
  - Etc.

- Lenient serialization
  - `XStream.ignoreUnknownElements()`
  - `XStreamSerializer.builder().lenientDeserialization().build()`

Jackson Serializer

- Serializes to/from JSON
- Cleaner output

- Has requirements on objects to serialize
  - Annotations
  - Getters/Setters
- Mainly suitable for commands, **events** and queries

- Lenient serialization
  - `@JsonIgnoreProperties(ignoreUnknown = true)`
  - `objectMapper.disable(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES);`
  - `JacksonSerializer.builder().lenientDeserialization().build()`

Tuning Serialization

- Content type converters
  - Serializer-specific
  - Generic

- Serializer levels
  - Generic → used for everything, unless...
  - Message → used for all types of messages, unless...
  - Event → only used for events

- Custom serializer
  - Wrapper to serialize specific events

Configuring Serializers

```
// When the Configurer is still in process,
Configurer configurer = ...;

// you can create your desired serializer
XStreamSerializer genericSerializer = XStreamSerializer.defaultSerializer();
JacksonSerializer messageSerializer = JacksonSerializer.defaultSerializer();

// and configure it at the right levels.
configurer.configureSerializer(configuration -> genericSerializer);
configurer.configureMessageSerializer(configuration -> messageSerializer);
configurer.configureEventSerializer(configuration -> messageSerializer);
```

Configuring Serializers - Spring

```
// With Spring, provide a bean with a qualifier to automatically configure it at the right level.
@Bean
@Qualifier("messageSerializer")
public Serializer messageSerializer() {
     return JacksonSerializer.defaultSerializer();
}

// In absence of a qualifier, it is expected to be the generic instance
@Bean
public Serializer genericSerializer() {
     return XStreamSerializer.defaultSerializer();
}
```

Beyond the 1.0

Application Evolution

Schemas and Message Versioning

- Your Command, Event and Query format are a contract
  - Implicit vs explicit schema

- Axon supports “schema revisions”
  - Maven version
  - Sequential revision
  - Any arbitrary (String) value with `@Revision(...)`

Naïve approach - class per version

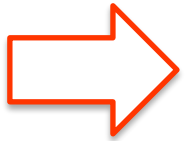
```
class FlightDelayedEvent_v0 {
    private String flightId;
    private Instant arrivalTime;
    private String start;
    private String destination;
}
```

```
class FlightDelayedEvent_v1 {
    private String flightId;
    private Instant arrivalTime;
    private Leg leg;
}

class Leg {
    private String from;
    private String to;
}
```

Last representation only

```
class FlightDelayedEvent {
   private String flightId;
   private Instant arrivalTime;
   private String start;
   private String destination;
}
```



```
@Revision("1")
class FlightDelayedEvent {
   private String flightId;
   private Instant arrivalTime;
   private Leg leg;
}

class Leg {
   private String from;
   private String to;
}
```

Upcasters

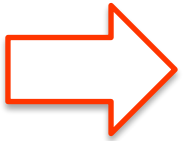
- Upcasters transform old event representations to the newer format
  - One format, one revision per upcaster
  - Chain upcasters into an “Upcaster Chain”

- Upcasters work on an intermediate representation
  - e.g. xml, json

Last representation only – Upcasters

FlightDelayedEvent, revision '0'

```
{
 "flightId": "123",
 "start": "AMS",
 "destination": "LAX",
 "arrivalTime": "10:30"
}
```



FlightDelayedEvent, revision '1'

```
{
 "flightId": "123",
 "leg": {
   "from": "AMS",
   "to": "LAX"
 },
 "arrivalTime": "10:30"
}
```

Upcaster - API

```
@FunctionalInterface
public interface Upcaster<T> {

    Stream<T> upcast(Stream<T> intermediateRepresentations);

}


@FunctionalInterface
public interface EventUpcaster extends Upcaster<IntermediateEventRepresentation> {

}
```

Upcaster - API

```
class FlightDelayedEvent0_to_1Upcaster extends SingleEventUpcaster {

    @Override
    protected boolean canUpcast(IntermediateEventRepresentation intermediateRep) {
        return ... // is this the type/version that should be upcast?
    }

    @Override
    protected IntermediateEventRepresentation doUpcast(
                            IntermediateEventRepresentation intermediateRepresentation) {
        return ... // create the upcasted version of the representation
    }
}
```

Upcaster - Example

```
class FlightDelayedEvent1_to_2Upcaster extends SingleEventUpcaster {
        @Override
        protected boolean canUpcast(IntermediateEventRepresentation intermediateRep) {
                SerializedType type = intermediateRep.getType();
                return type.getName().equals("com.example.FlightDelayedEvent") && type.getRevision().equals("1");
        }


        @Override
        protected IntermediateEventRepresentation doUpcast(IntermediateEventRepresentation intermediateRep) {
                return intermediateRep.upcastPayload(
                        new SimpleSerializedType("com.example.FlightDelayedEvent", "2"),
                        JsonNode.class,
                        event -> {
                                ((ObjectNode) event).put("reason", "unknown");
                                return event;
                        }
                );
        }
}
```

Allows to return an instance that lazily upcasts to the new version

The type of representation to work with

The actual modification of the intermediate representation

AxonIQ

Deployment strategy - Big Bang

- Easiest approach
- No need for concurrent versions
- Deploy upcaster with new application

But...

- Not a feasible solution for distributed systems
- Requires downtime

Deployment strategy - Blue-Green

- Bring new version “up to speed” in parallel
- No need for concurrent versions
- Deploy upcaster with new application

But...

- Not a feasible solution for large-scale distributed systems

Deployment strategy - Rolling upgrade

- Both versions run side-by-side for a ‘while’
- Old version needs to be able to understand new events
  - Forward and backward compatibility

But...

- Don’t publish both ‘old’ and ‘new’ events separately

Message Schema – Forward Compatibility

FlightDelayedEvent, revision '0'

```
{
 "flightId": "123",
 "start": "AMS",
 "destination": "LAX",
 "arrivalTime": "10:30"
}
```

FlightDelayedEvent, revision '2'

```
{
 "flightId": "123",
 "leg": {
   "from": "AMS",
   "to": "LAX"
 },
 "arrivalTime": "10:30"
}
```

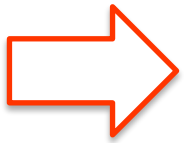
Message Schema – Compatibility Recommendations

- Never change semantic meaning of an event
  - That would mean it's a new event
- Never remove or change attributes
  - Deprecate attributes instead
- Only add (optional) attributes
- Use lenient deserialization
  - Use sensible defaults for missing attributes
  - Ignore unknown attributes

Message Schema – Forward Compatibility

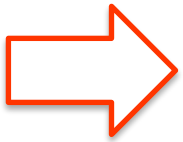
FlightDelayedEvent, revision '0'

```
{
 "flightId": "123",
 "start": "AMS",
 "destination": "LAX",
 "arrivalTime": "10:30"
}
```



FlightDelayedEvent, revision '1'

```
{
 "flightId": "123",
 "start": "AMS",
 "destination": "LAX",
 "leg": {
   "from": "AMS",
   "to": "LAX"
 },
 "arrivalTime": "10:30"
}
```



FlightDelayedEvent, revision '2'

```
{
 "flightId": "123",
 "leg": {
   "from": "AMS",
   "to": "LAX"
 },
 "arrivalTime": "10:30"
}
```

Whatever else you wanted to know...

Questions