Camel 3.0 - Message Store

Work in progress

The page intends to collect all ideas and proposals around the idea of a Message Store as a architectural concept in Camel. No implementation has started yet. You can participate by sharing your input here or post it to the dev mailing list.

For the initial brainstorming on the dev mailing list, please check this thread.

From the Camel 3.0 ideas page

(+1: hadrian, claus)
We should make this EIP easier to use for end users, but offering a better public API. And also have a pluggable message store, with filters that can filter what should be stored. As well pluggable marshallers so people can marshal data from Exchange into a format the message store can store (BLOB, XML, JSon etc.).
A Message Store could provide transparent persistence to various EIP patterns (or used directly). Implementations would handle the mapping to the underlying database or file system or NoSQL or memory or whatever.
Message Store implementations are already there in various places, using different approaches, like in Stream Caching (only file system), AggregationRepository or IdempotentRepository. A Message Store is requested for in other places like Reliable stream resequencing (CAMEL-949) or Persistent Dead Letter Queue (CAMEL-4575).

Summary

In Camel 2.x, a **Message Store** is not a "first class citizen" in the sense of a architectural concept, which can be applied consistently whereever needed. Camel 3.0 should introduce a unified, generic, pluggable Message Store that consolidates the different approaches and allows to similarly parameterize persistence to various EIP patterns.
It should be usable independently of EIP patterns as well.
Implementations would handle the mapping to the underlying persistence layer, which can be InMemory, RDBMS, NoSQL-based etc. and can be chosen based on the requirements towards reliability (survive system shutdowns with losing messages) and scalability (reduce memory consumption when processing many/big messages).

Side note: Spring Integration provides a corresponding concept.

List of proposed features

Work in progress

Feel free to add, edit, comment.... At some time we probably need to assign priorities as to what is indispensable for Camel 3.0 and what could be delivered in a later patch release

- Generically, entries in a Message Store can be created, updated, read and deleted.
- Ability to temporarily store exchanges for the following EIPs:
  - Aggregator, Multicast, RecipientList, Splitter : alternative to AggregationRepository, making it eventually obsolete
  - Streaming Resequencer (CAMEL-949)
  - Stream Caching
  - Claim check
- Ability to store exchanges for a defined period of time
  - Idempotent Consumer
  - Dead Letter Queue (CAMEL-4575)
  - Destination for the Tracer
- Ability to permanently store exchanges (e.g. for audit trails)
- Provide a certain level of manual retry. That is to get the original message from the store and feed it back in the originating route.
- Flexibility to specify what part of an exchange should be stored (e.g. what exchange properties and message headers) and in which format (e.g. object serialization, JSON, using encryption)
- Possibility to provide a filter condition to determine which exchanges should be stored (e.g. only failed exchanges, only with a certain message header)
- Polling Consumer to randomly access a message store
- Producer to write an exchange into a message store

- There is a default message store defined for the Camel Context. This can be overridden by a route-specific message store. This again can be overridden by a specific EIP processor.

Message Store Data

In order to disambiguate stored exchanges and make them retrievable again, message store entries must carry attributes in addition to the marshaled exchange.

Item	Description/Reason
ID	Generated unique ID of the entry
Exchange	exchange (or parts thereof), usually in some marshalled form (except in-memory stores)
CamelContext	A message store may be used by several CamelContext instances

Correlation	Correlation ID to be able to aggregate related exchanges
Source	Identifier that uniquely describes the point in the route from which an exchange was stored. E.g. there might be more than one aggregator processor in a route. Could also be used to manually or automatically refeed exchanges back into the route
Status	Status of an exchange (e.g. PROCESSING, DONE, FAILED)
Creation Time	Timestamp when the entry was created
ExpirationTime	Timestamp when the entry can be considered as expired and picked up by some cleaning process
...	... more?

Code examples

Work in progress

Sometimes it is easier to express thoughts by providing a fictional piece of code along with some comments....

This section intends to demonstrate the usage of a Message Store by providing hypothetical code snippets, e.g.

AggregatorExample.java

```
...
from(...)
   .aggregate()
       .correlationExpression(header(id))
       .aggregationStrategy(myStrategy)
       .completionTimeout(10000)
       .messageStore(myStore)
...
```

Claim check

THe claim check pattern temporarily reduces the data volume of the message by storing content in a message store in exchange for a claim check token. The content is retrieved later on before it's needed again.

Claim Check EIP store

```
// Optionally: override default store from context
// (ohr:) IMHO I don't think that this configuration level is really necessary
defaultMessageStore(myStore);

// 1) Store body.
// 2) Set body to null.
// 3) Set Exchange.CLAIM_CHECK header to unique claim id.
from(...)
   .checkIn()  // store body in default store
   // .checkIn(header('bigHeader'), customStore) : store header in custom store
   .to(...);
```

Claim Check EIP read

```
// 1) Lookup for the Exchange.CLAIM_CHECK header value.
// 2) Read the message.
// 3) Set body to the value fetched from the store.
from(...)
   // .setHeader(Exchange.CLAIM_CHECK, const("id")) : header should still contain the claim id
   .reclaim() // read body from default store
   // .reclaim().aggregationStrategy(myStrategy) : more generically using a aggregation strategy
   // .reclaim(customStore).aggregationStrategy(myStrategy) : reclaim from custom store using a aggregation
strategy
   .to(...);
```

Open issues:

- exception handling if there's no data available for a specific token
- clean up of stale content that was never claimed back
- maybe return some kind of DataHandler instead of a token (cf. CXF MTOM attachments) and retrieve content transparently?