Welcome to

4. Messaging, MQ systems

KEA System Integration F2020 10 ECTS

Henrik Lund Kramshøj hlk@zencurity.com @kramse

Slides are available as PDF, kramse@Github

4-Messaging-system-integration.tex in the repo security-courses

Plan for today

- Enterprise Integration Patterns: Messaging Systems and Messaging Channels
- Messaging, MQ systems
- Publish-Subscribe
- The Aggregator and Splitter EIPs
- The Routing Slip EIP
- The Dynamic Router EIP
- The Load Balancer EIP

Exercises

- Elasticsearch installation
- Nginx Load Balancer

Reading Summary

EIP ch 3-4, Camel ch 5
and read this

Browse / skim this:

Todays Agenda - approximate time plan

- 08:30 45m Book chapters 3 and 4 discussion
- 45m Messaging MQ systems: Apache ActiveMQ, Elasticsearch and Redis
- 10:00 Break 15m
- 10:15 45m Exercises with Elasticsearch
- 45m 2x 45m Camebook chapter 5: Enterprise integration patterns
  - contains aggregator, splitter, routing slip, dynamic router, load balancer
- 11:45 Lunch Break
- 12:30 Continued Camelbook chapter 5
- 45m Exercise: Run Nginx as a load balancer
- 14:00 Praktik intro! Not me

Goals for today

Todays goals:

- See more Enterprise Integration Patterns
- Look at a real example, documented in an article
- Work a bit with Ansible, Elasticsearch and Nginx

Photo by Thomas Galler on Unsplash

Enterprise Integration Patterns

EIP book chapters

Enterprise Integration Patterns: Messaging Systems

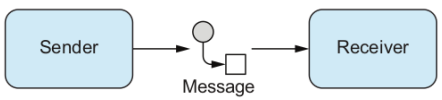


Figure 1.5 Messages are entities used to send data from one system to another.

Basic Messaging Concepts

- Channels Messaging applications transmit data through a Message channel, a virtual pipe that connects a sender to a receiver.
- Messages A message is an atomic packet of data that can be transmitted on a channel
- Pipes and filters can perform actions on the messages as they travel through the system, validation or transformed between formats
  We already saw that last time converting data with Camel
- Routing a message may have to go through several channels to reach its final destination
- Transformation, a message translator can convert from one format to another
- Endpoints enable applications to send and receive messages

3.1 Data transformation overview

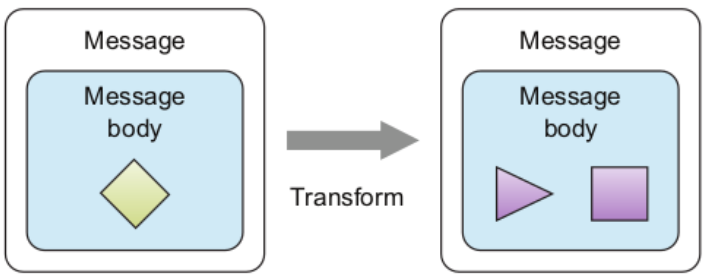


Figure 3.1 Camel offers many features for transforming data from one form to another.

- Messages consist of two parts: Header and Body
- See Internet email, picture formats, packet headers
- Data is envelopped many times during transmission!
- Data format transformation – The data format of the message body is transformed from one form to another. For example, a CSV record is formatted as XML.
- Data type transformation – The data type of the message body is transformed from one type to another. For example, java.lang.String is transformed into javax. jms.TextMessage .

SOAP Request

```
// HTTP here
POST / HTTP/1.0
Host: localhost:8080
User-agent: SOAPpy xxx (pywebsvcs.sf.net)
Content-type: text/xml; charset=UTF-8
Content-length: 340
SOAPAction: "hello"
// SOAP start here
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
>
<SOAP-ENV:Body>
<hello SOAP-ENC:root="1">
</hello>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP Response

```
// HTTP here
HTTP/1.0 200 OK
Server: <a href="http://pywebsvcs.sf.net">SOAPpy xxx</a> (Python 2.7.16)
Date: Mon, 24 Feb 2020 13:29:44 GMT
Content-type: text/xml; charset=UTF-8
Content-length: 510
// SOAP start here
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema"
>
<SOAP-ENV:Body>
<helloResponse SOAP-ENC:root="1">
<Result xsi:type="xsd:string">Hello World</Result>
</helloResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Routing with Camel

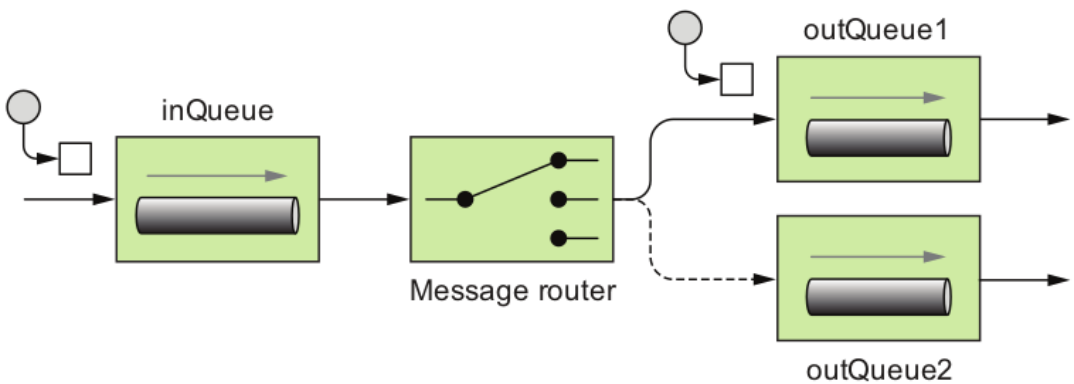


Figure 2.1 A message router consumes messages from an input channel and, depending on a set of conditions, sends the message to one of a set of output channels.

We already tried using Camel which supports routing messages and data

3.3 Transforming XML

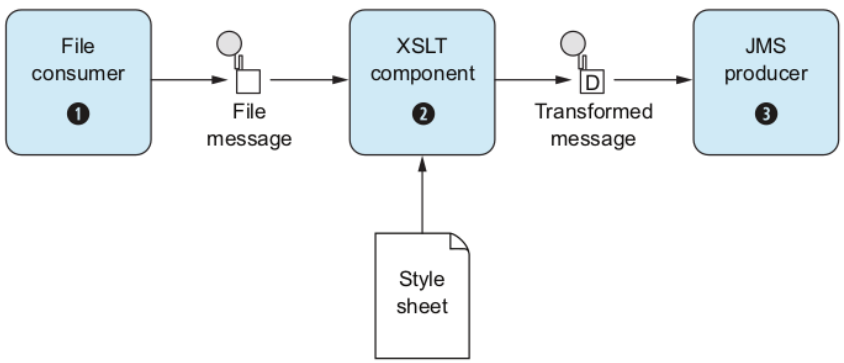
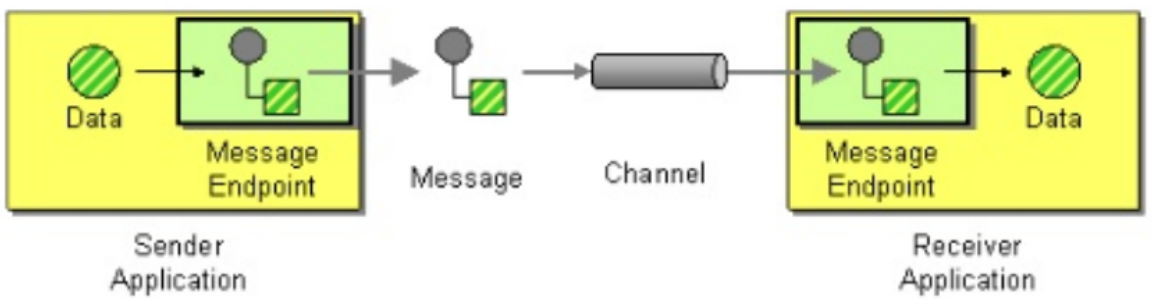


Figure 3.5 A Camel route using an XSLT component to transform an XML document before it's sent to a JMS queue

- Transforming XML with XSLT
- Transforming XML with object marshaling

Endpoints



- Endpoints are built-in or added to applications
- Interface between internal representation and code, to the external message format

Enterprise Integration Patterns: Messaging Channels

Messaging Channel Themes

> Fixed set of channels — One theme in this chapter is that the set of Message Channels available to an application tends to be static. When designing an application, a developer has to know where to put what types of data to share that data with other applications, and likewise where to look for what types of data coming from other applications. These paths of communication cannot be dynamically created and discovered at runtime; they need to be agreed upon at design time so that the application knows where its data is coming from and where the data is going to. (While it is true that most channels must be staticly defined, there are exceptions to this theme, cases where dynamic channels are practical and useful. One exception is the reply channel in Request-Reply. The requestor can create or obtain a new channel the replier knows nothing about, specify it as the Return Address of a request message, and then the replier can make use of it. Another exception is messaging system implementations that support hierarchical channels. A receiver can subscribe to a parent in the hierarchy, then a sender can publish to a new child channel the receiver knows nothing about, and the subscriber will still receive the message. These relatively unusual cases notwithstanding, channels are usually defined at deployment-time and applications are designed around a known set of channels.)

Source: more or less same as the book

`https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessagingChannelsIntro.html`

Messaging Channel Themes

> Determining the set of channels — A related issue is: Who decides what Message Channels are available, the messaging system or the applications? That is to say: Does the messaging system define certain channels and require the applications to make due with those? Or do the applications determine what channels they need and require the messaging system to provide them? There is no simple answer; designing the needed set of channels is iterative. First the applications determine the channels the messaging system will need to provide. Subsequent applications will try to design their communication around the channels that are available, but when this is not practical, they will require that additional channels be added. When a set of applications already use a certain set of channels, and new applications wish to join in, they too will use the existing set of channels. When existing applications add new functionality, they may require new channels.

Source: more or less same as the book

`https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessagingChannelsIntro.html`

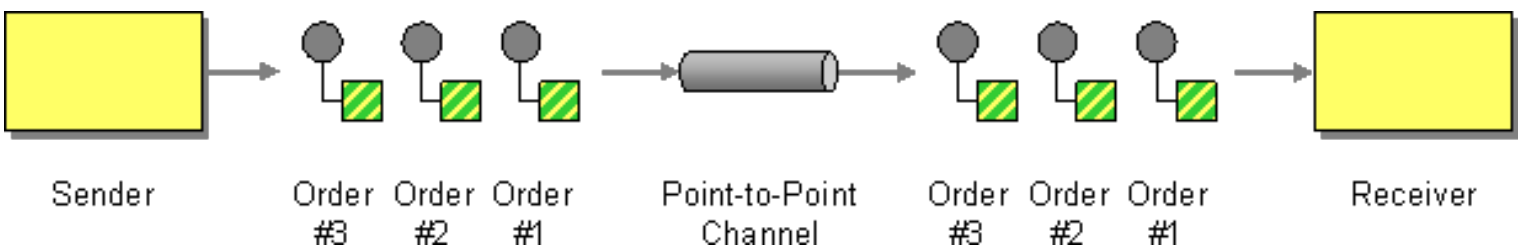
Messaging Channel Themes

> Unidirectional channels — Another common source of confusion is whether a Message Channel is unidirectional or bidirectional. Technically, it's neither; a channel is more like a bucket that some applications add data to and other applications take data from (albeit a bucket that is distributed across multiple computers in some coordinated fashion). But because the data is in messages that travel from one application to another, that gives the channel direction, making it unidirectional. If a channel were bidirectional, that would mean that an application would both send messages to and receive messages from the same channel, which—while technically possible—makes little sense because the application would tend to keep consuming its own messages, the messages it's supposed to be sending to other applications. So for all practical purposes, channels are unidirectional. As a consequence, for two applications to have a two-way conversation, they will need two channels, one in each direction (see Request-Reply in the next chapter).

Source: more or less same as the book

`https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessagingChannelsIntro.html`

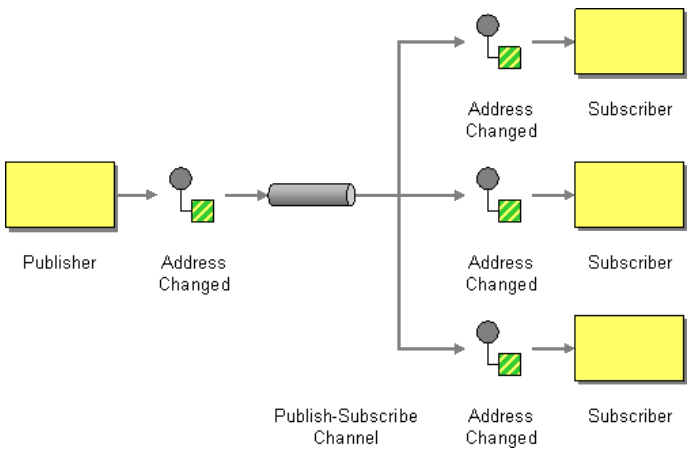
Point-to-Point Channels



A Point-to-Point Channel ensures that only one receiver consumes any given message. If the channel has multiple receivers, only one of them can successfully consume a particular message. If multiple receivers try to consume a single message, the channel ensures that only one of them succeeds, so the receivers do not have to coordinate with each other. The channel can still have multiple receivers to consume multiple messages concurrently, but only a single receiver consumes any one message.

Source: `https://www.enterpriseintegrationpatterns.com/patterns/messaging/PointToPointChannel.html`

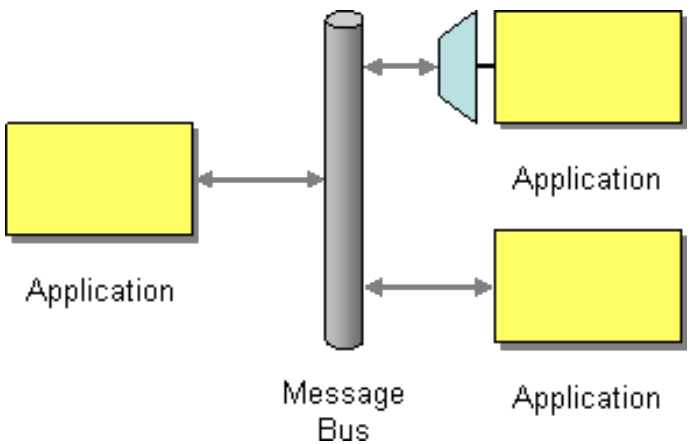
Publish-Subscribe Channel



A Publish-Subscribe Channel works like this: It has one input channel that splits into multiple output channels, one for each subscriber. When an event is published into the channel, the Publish-Subscribe Channel delivers a copy of the message to each of the output channels. Each output channel has only one subscriber, which is only allowed to consume a message once. In this way, each subscriber only gets the message once and consumed copies disappear from their channels.

Source: `https://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChann`
`html`

Message Bus



A Message Bus is a combination of a common data model, a common command set, and a messaging infrastructure to allow different systems to communicate through a shared set of interfaces. This is analogous to a communications bus in a computer system, which serves as the focal point for communication between the CPU, main memory, and peripherals. Just as in the hardware analogy, there are a number of pieces that come together to form the message bus:

Source: `https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageBus.html`

Java Message Service (JMS) Elements

- JMS provider An implementation of the JMS interface for message-oriented middleware (MOM). Providers are implemented as either a Java JMS implementation or an adapter to a non-Java MOM.
- JMS client An application or process that produces and/or receives messages. JMS producer/publisher
- A JMS client that creates and sends messages. JMS consumer/subscriber
- A JMS client that receives messages. JMS message An object that contains the data being transferred between JMS clients.
- JMS queue A staging area that contains messages that have been sent and are waiting to be read (by only one consumer). As the name queue suggests, the messages are delivered in the order sent. A JMS queue guarantees that each message is processed only once.
- JMS topic A distribution mechanism for publishing messages that are delivered to multiple subscribers.

Source: `https://en.wikipedia.org/wiki/Java_Message_Service`

Publish-Subscribe

The JMS API supports two distinct models:

- Point-to-point Under the point-to-point messaging system, messages are routed to individual consumers who maintain queues of incoming messages.
- Publish-and-subscribe The publish-and-subscribe model supports publishing messages to a particular message "topic". Subscribers may register interest in receiving messages published on a particular message topic. In this model, neither the publisher nor the subscriber knows about each other. A good analogy for this is an anonymous bulletin board.

Source: `https://en.wikipedia.org/wiki/Java_Message_Service`

Messaging, MQ systems

There are a couple of systems we can use for message queuing:

- Apache ActiveMQ - Java-based messaging server
- Elasticsearch a search engine and document store
- Redis Remote Dictionary Server

Apache ActiveMQ

Apache ActiveMQ™ is the most popular open source, multi-protocol, Java-based messaging server. It supports industry standard protocols so users get the benefits of client choices across a broad range of languages and platforms. Connectivity from C, C++, Python, .Net, and more is available. Integrate your multi-platform applications using the ubiquitous AMQP protocol. Exchange messages between your web applications using STOMP over websockets. Manage your IoT devices using MQTT. Support your existing JMS infrastructure and beyond. ActiveMQ offers the power and flexibility to support any messaging use-case.

`https://activemq.apache.org/`

Elasticsearch

> Elasticsearch is a search engine based on the Lucene library. It provides a distributed, multitenant-capable full-text search engine with an HTTP web interface and schema-free JSON documents. Elasticsearch is developed in Java.

Source: `https://en.wikipedia.org/wiki/Elasticsearch`

- Open core means parts of the software are licensed under various open-source licenses (mostly the Apache License)
- Various browser tools and plugins for ES exist, to make life easier
- I often use ES for storing Log Messages and Events from multiple systems, a SIEM Security information and event management.

Redis

Redis (/ r d s/;[7][8] Remote Dictionary Server)[7] is an in-memory data structure project implementing a distributed, in-memory key-value database with optional durability.

- Often used as a queue, a kind of buffer between systems
- `https://en.wikipedia.org/wiki/Redis`
- `https://redislabs.com`

Real Example using Elasticsearch

We will read about a real example:

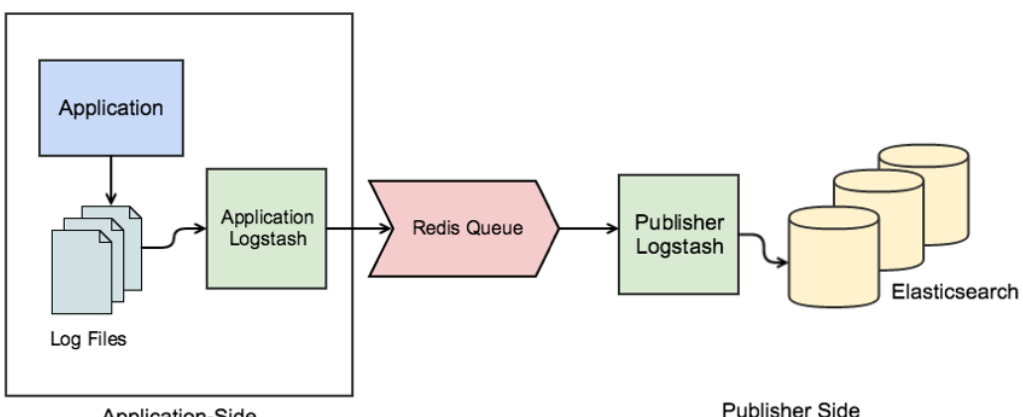
> Scaling the Elastic Stack in a Microservices Architecture @ Rightmove
> Rightmove is the UK's #1 Property Portal. In the process of helping people find the places they want to live, we serve 55 million requests a day and use Elasticsearch to power our searches and provide our teams with useful analytics to help support our applications. ... Fast forward to 2017 and we have over 50 microservices all sending their logs to our Elasticsearch cluster. In doing so, we needed a way to scale our configuration on both the hardware and application side of things. So how did we achieve this and what did we learn along the way?

Source:

`https://www.elastic.co/blog/scaling-the-elastic-stack-in-a-microservices-architecture-rightmove`

Exercise: Read about Elasticsearch

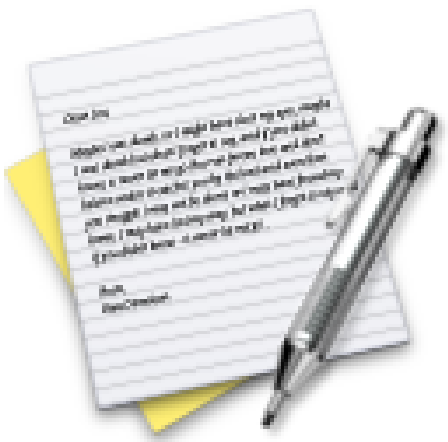
Look at the example Elasticsearch



Source:

`https://www.elastic.co/blog/scaling-the-elastic-stack-in-a-microservices-architecture-rightmove`

Exercise



Now lets do the exercise

Use Ansible to install Elastic Stack

which is number **5** in the exercise PDF.

Chapter 5

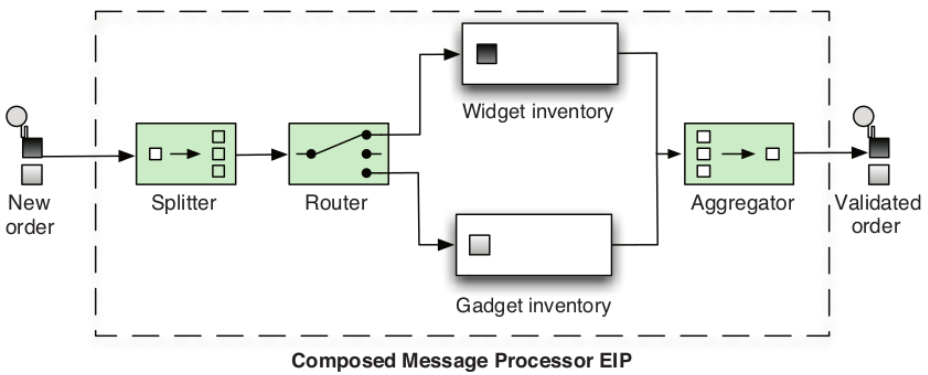


Figure 5.1 The Composed Message Processor EIP splits up the message, routes the submessages to the appropriate destinations, and reaggregates the response back into a single message.

- 5.1 Introducing enterprise integration patterns
- The Aggregator and Splitter EIPs
- The Routing Slip and Dynamic Router EIPs
- The Load Balancer EIP

5.2 The Aggregator EIP

Figure 5.2 The Aggregator stores incoming messages until it receives a complete set of related messages. Then the Aggregator publishes a single message distilled from the individual messages.

- Using the Aggregator EIP
- Completion conditions for the Aggregator
- Using persistence with the Aggregator
- Using recovery with the Aggregator

5.2.1 Using the Aggregator EIP

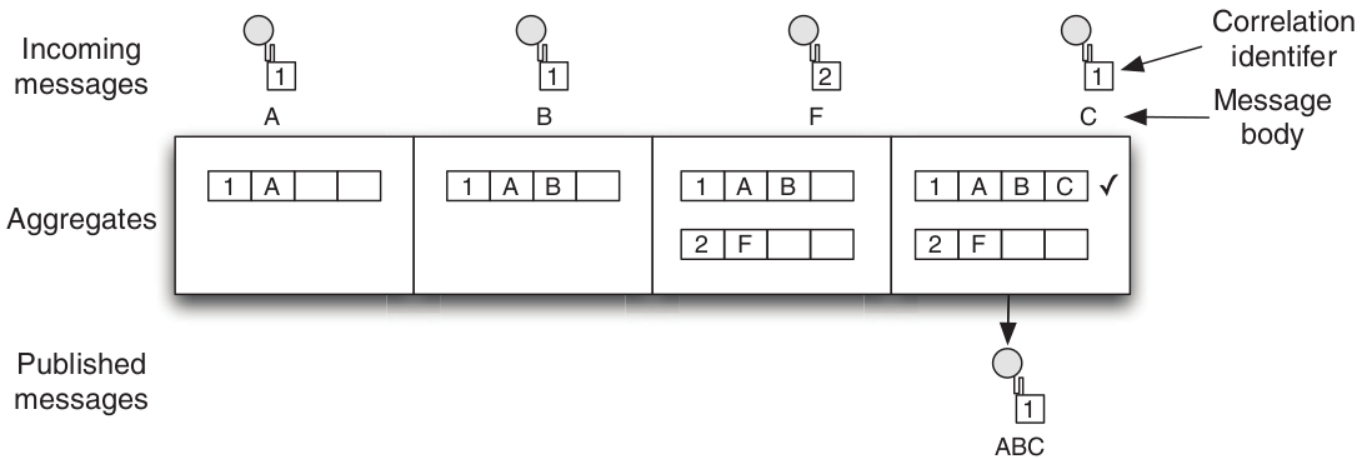


Figure 5.3 The Aggregator EIP in action, with partial aggregated messages updated with arriving messages

5.2.4 Using recovery with the Aggregator

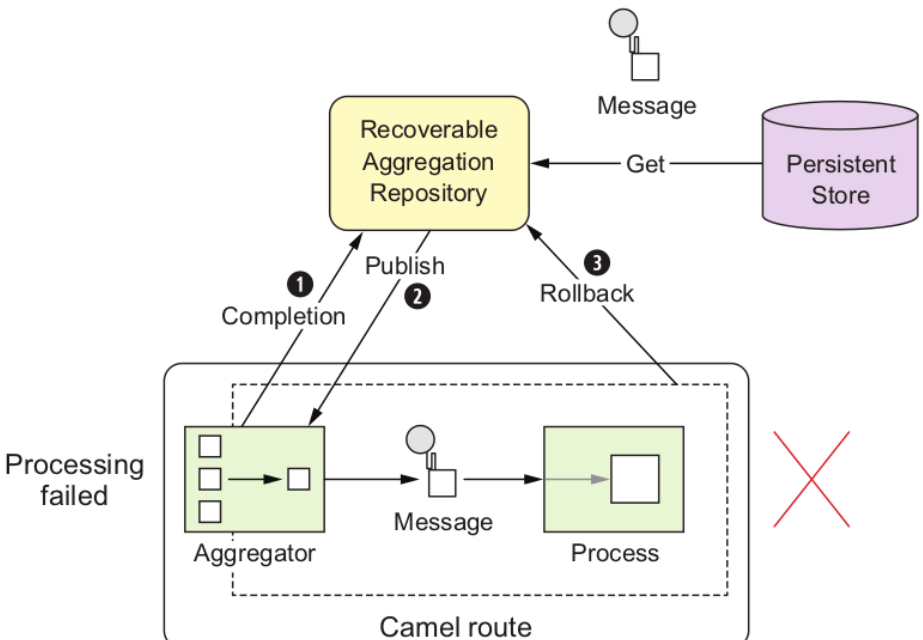


Figure 5.4 An aggregated message is completed ❶, it's published from the Aggregator ❷, and processing fails ❸, so the message is rolled back.

And with recovery

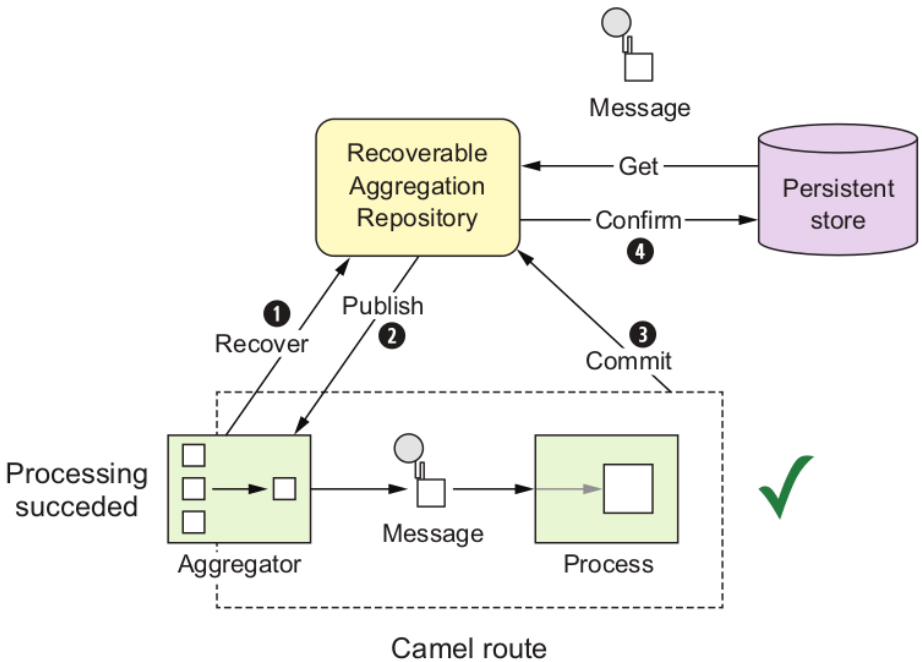


Figure 5.5 The Aggregator recovers failed messages ❶, which are published again ❷, and this time the messages completed ❸ successfully ❹.

5.3 The Splitter EIP

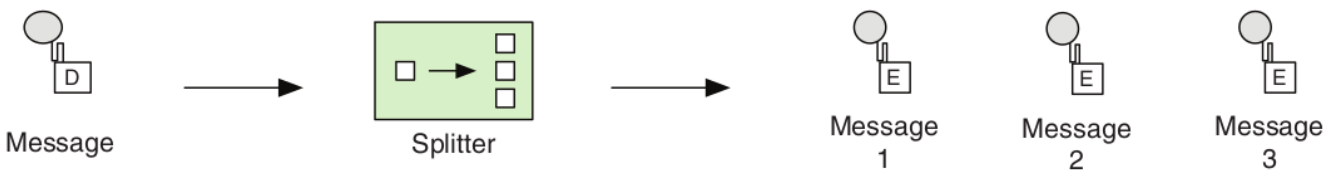
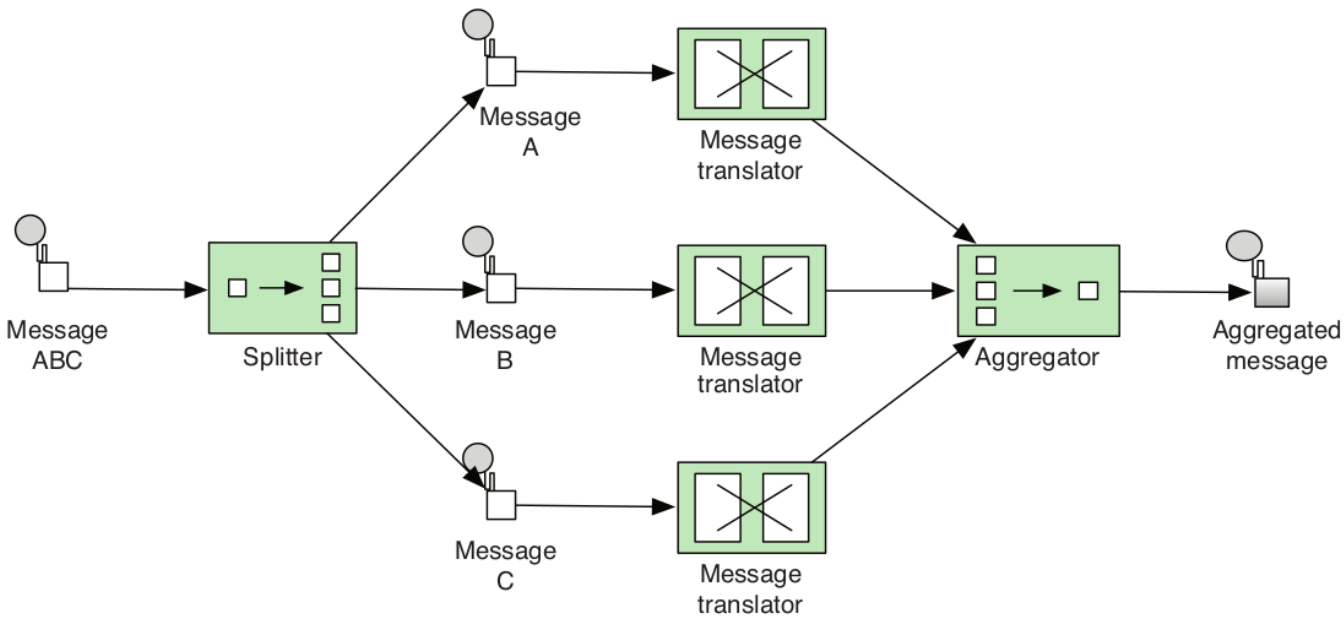


Figure 5.6 The Splitter breaks the incoming message into a series of individual messages.

- Using the Splitter, Using beans for splitting, Splitting big messages, Aggregating split messages, When errors occur during splitting

Aggregating split messages



There can be multiple reasons for doing the split, to process messages and then aggregate them.

5.4 The Routing Slip EIP

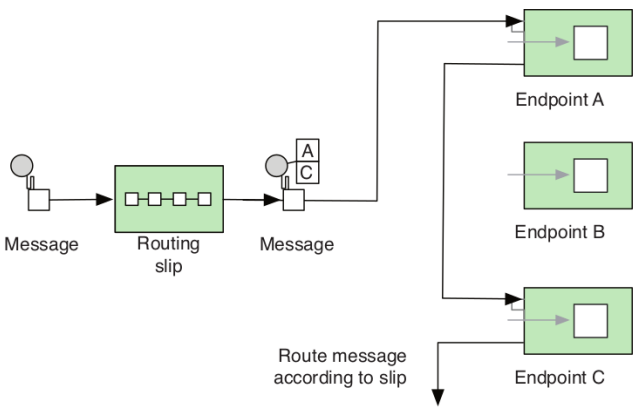


Figure 5.11 The incoming message has a slip attached that specifies the sequence of the processing steps. The Routing Slip EIP reads the slip and routes the message to the next endpoint in the list.

- Using the Routing Slip EIP, Using a bean to compute the routing slip header, Using an Expression as the routing slip, Using @RoutingSlip annotation

5.5 The Dynamic Router EIP

- Using the Dynamic Router, Using the @DynamicRouter annotation

5.6 The Load Balancer EIP

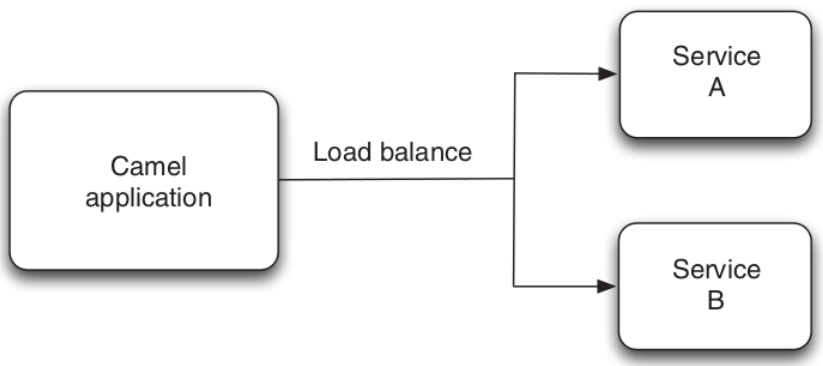


Figure 5.12 A Camel application load-balances across two services.

- Introducing the Load Balancer EIP, Using load-balancing strategies, Using the failover load balancer, Using a custom load balancer
- The EIP book doesn't list the Load Balancer, which is a pattern implemented in Camel.
- Nginx is a very popular HTTP routing engine and load balancer for HTTP(S)

Nginx Load Balancer

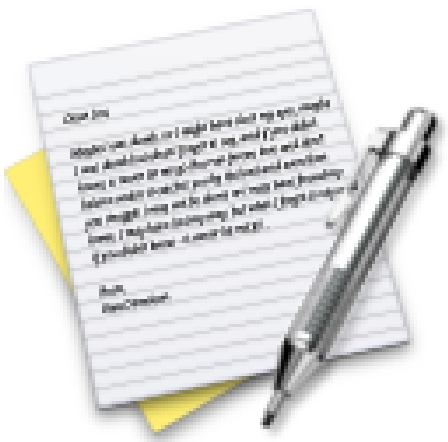
```
http {
    upstream myapp1 {
        least_conn;
        server srv1.example.com;
        server srv2.example.com;
        server srv3.example.com;
    }

    server {
        listen 80;

        location / {
            proxy_pass http://myapp1;
        }
    }
}
```

Example from: `http://nginx.org/en/docs/http/load_balancing.html`

Exercise

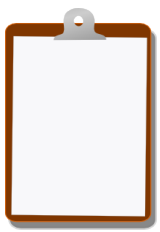


Now lets do the exercise

Run Nginx as a load balancer

which is number **6** in the exercise PDF.

For Next Time



Think about the subjects from this time, write down questions

Check the plan for chapters to read in the books

Visit web sites and download papers if needed

Retry the exercises to get more confident using the tools