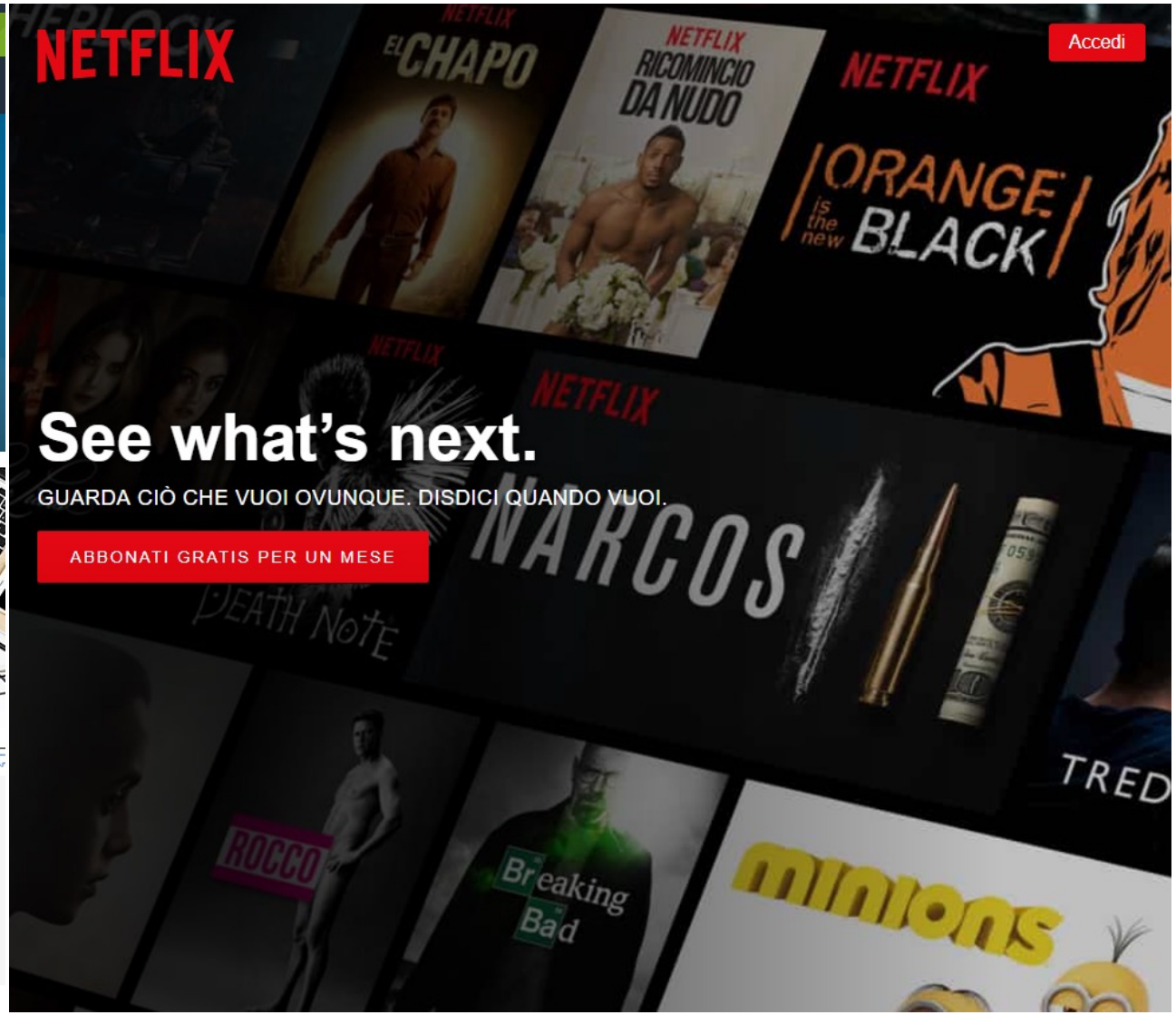
Enterprise Application Integration

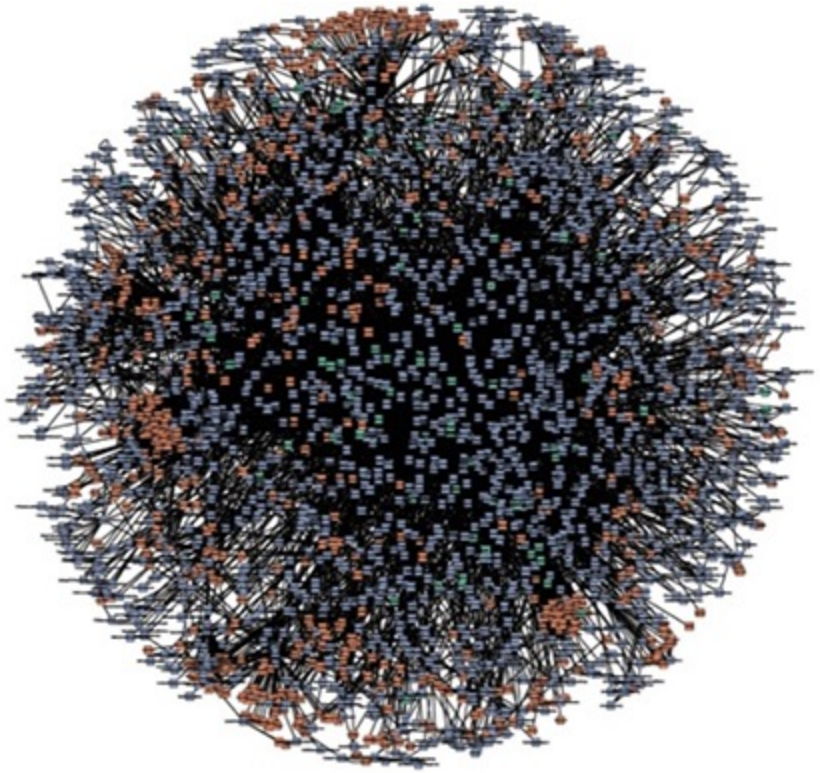
Advanced Software Engineering – Guest Lecture

Jacopo Soldani
Dipartimento di Informatica, Università of Pisa
http://pages.di.unipi.it/soldani

Amazon & Netflix: Two widely-used enterprise applications



Amazon & Netflix: What's behind the scenes



amazon.com

NETFLIX

Enterprise Applications

- Various heterogeneous **services**
  - *sources* and *sinks*
  - some you are in control of, others not
- Various heterogeneous **data types**
  - different representations
  - even for analogous data
- Multiple different **participants**
  - different organizations agreeing to share data
- All interconnected via **network**

In short, enterprise applications are
complex distributed multi-service applications
whose services must play together, being them suitably **integrated**

How to integrate?

The **architectural** question is how to integrate multiple different services to realize enterprise applications that are

- coherent,
- extensible,
- maintainable, and
- (reasonably) simple to understand

This is really what **enterprise application integration** was conceived for

- complexity management
- change management
- **pattern-based**

What is a pattern?

Pattern = high-level abstraction of accepted, reusable solutions to recurring problems

Typically, patterns are given in terms of

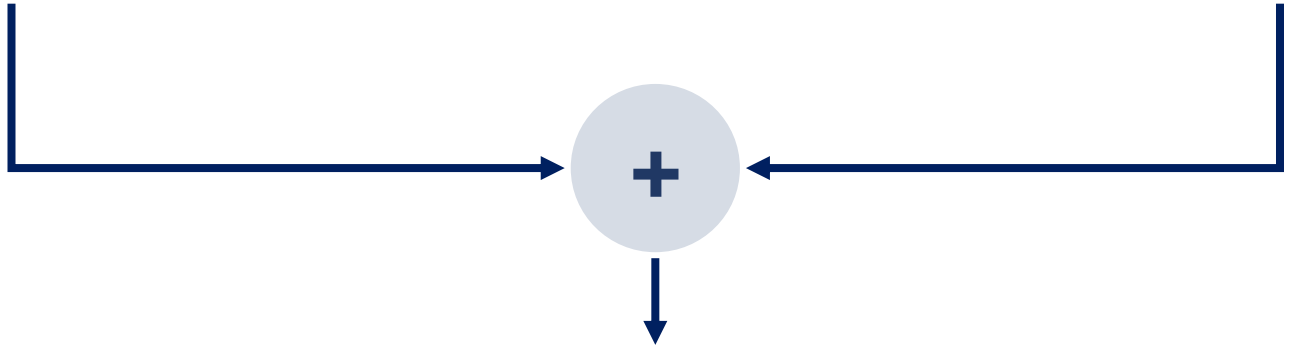
- problem statement // including involved software components
- context // including involved actors
- forces // clarifying the problem rationale and importance
- solution // given abstractly, and independent of its actual // implementations

When facing a problem, considering existing patterns that are applicable to solve such problem **saves us from re-inventing the wheel and making the same mistakes as others**

Enterprise Integration Patterns (EIPs)

Enterprise applications are “big” applications comprising legacy and commercial software components, and owned and third-party services

Integration is the process for making various heterogeneous bits of software work together, seamlessly, to fullfill some business workflows

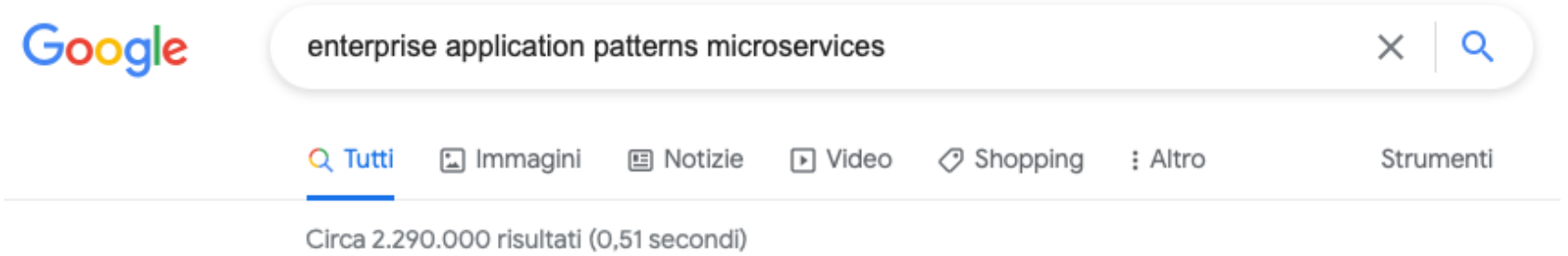


An EIP is a reusable abstraction of **proven solutions** to well-known problems raising while **integrating** the software components/services forming **enterprise applicatioms**

The sacred text of EIPs

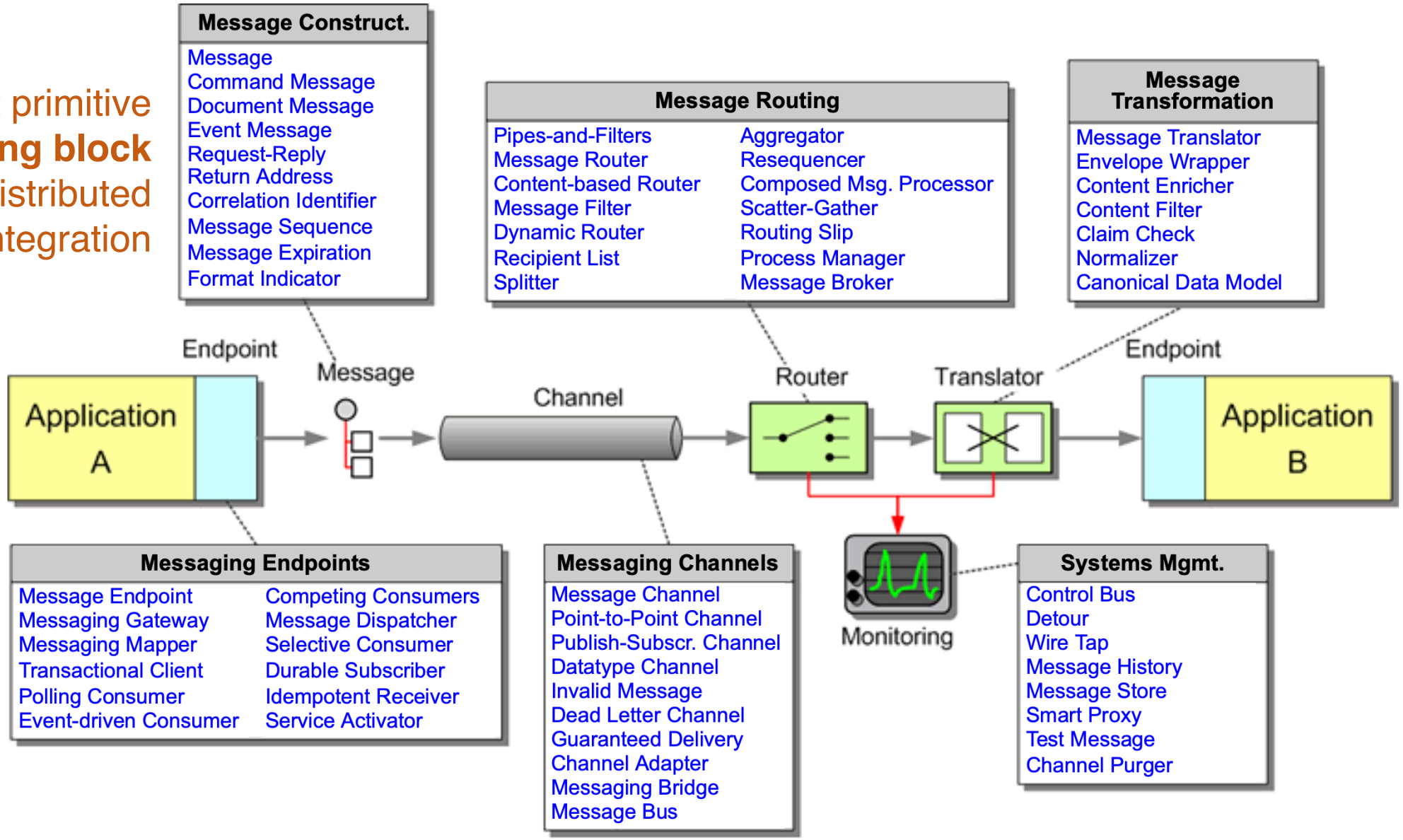
Gregor Hohpe, Bobby Woolf.
Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.
Addison-Wesley Professional, 1st Edition, 2003. → too old?

https://www.enterpriseintegrationpatterns.com



EIPs, in short

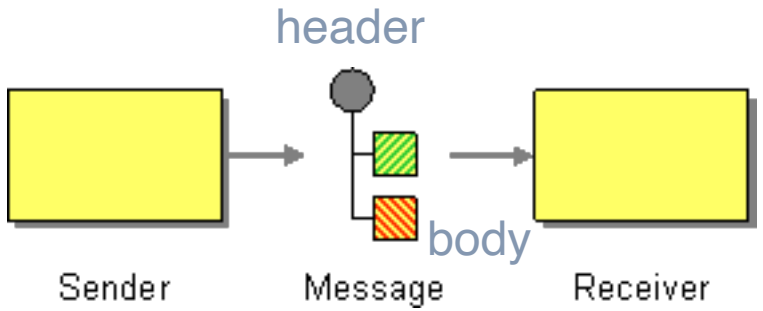
most primitive **building block** for distributed integration



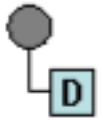
Messages

A **message** is a discrete **piece of data** sent from a service to another

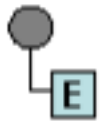
- typically, structured into **header** (metadata) and **body** (payload)



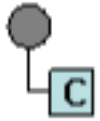
Concrete **examples** of messages:



document message
("pure data")



event message
(something has happened)



command message
(do something)

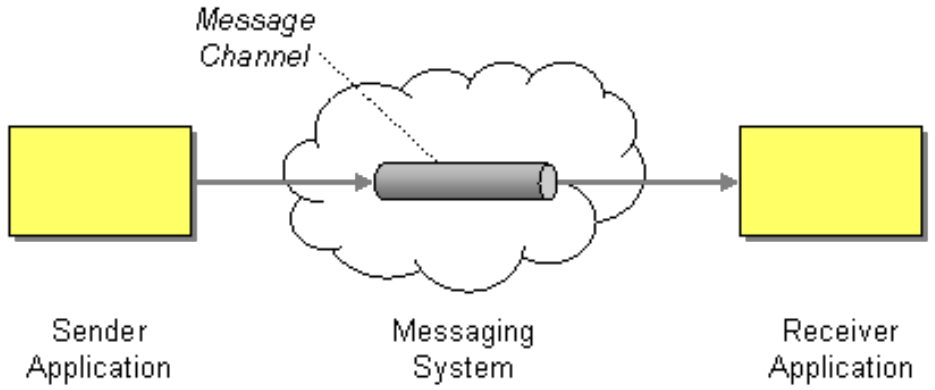
Sending messages, through *channels*

Message-based communication to enable **loose coupling**

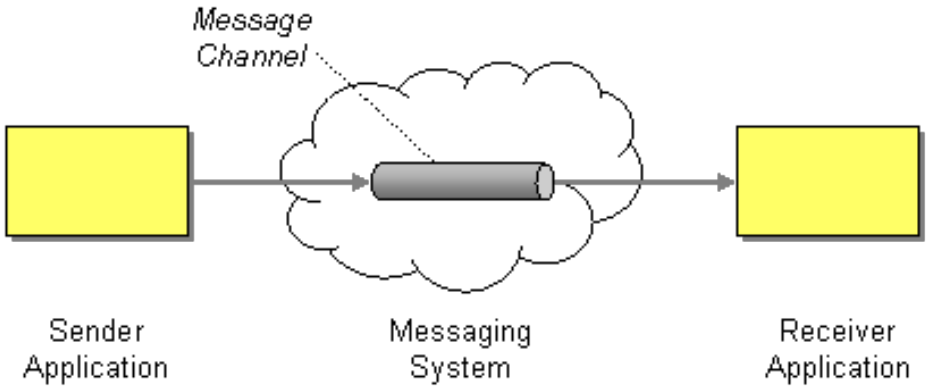
- enabling complexity management/change/growth
- based on simplest exchange pattern (one-way)
- messages' format/metadata independent of the integrated services

Realized with **channels**

- **abstraction** for components **sending messages** from a source to a destination
- could be implemented in many ways (RPC, HTTP, TCP, etc.)



Sending messages, through *channels* (2)

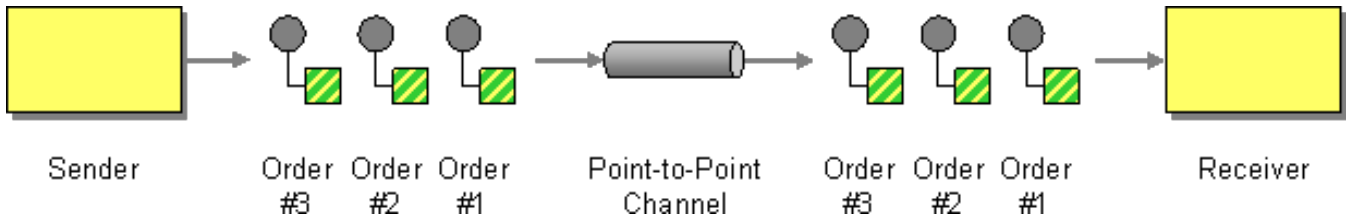


Channels are **one-way**

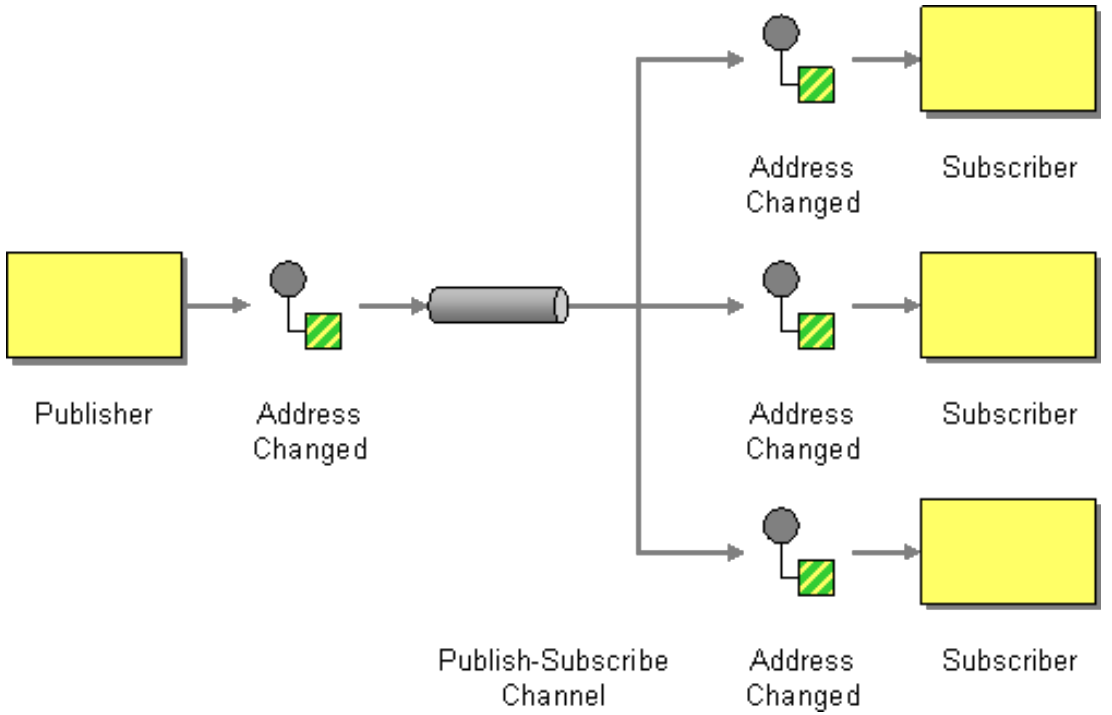
- communications are hence natively **asynchronous**
- synchronous (requests/response) communications use two channels

Different types of channels

Point-to-point channels ensure that only one receiver will receive a particular message



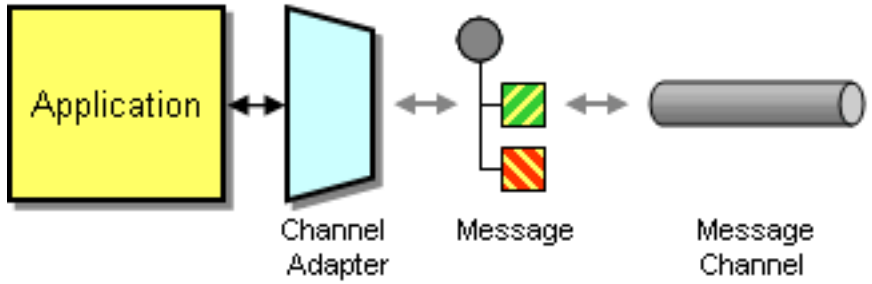
Publish-subscribe channels deliver a copy of incoming messages to each receiver



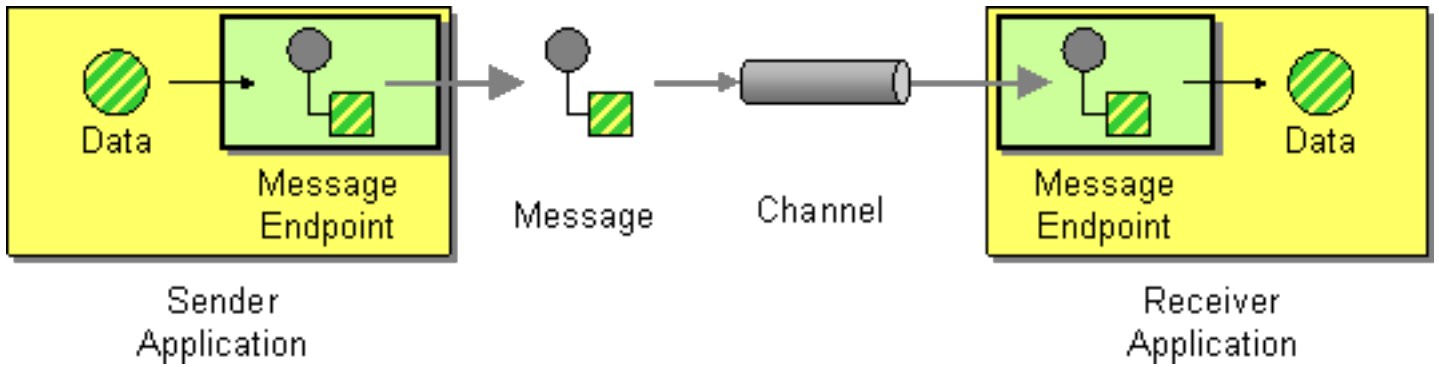
Binding messages to enterprise applications

Application services are typically independent of the messaging systems

- **adapters** enable application-specific data to be sent to channels



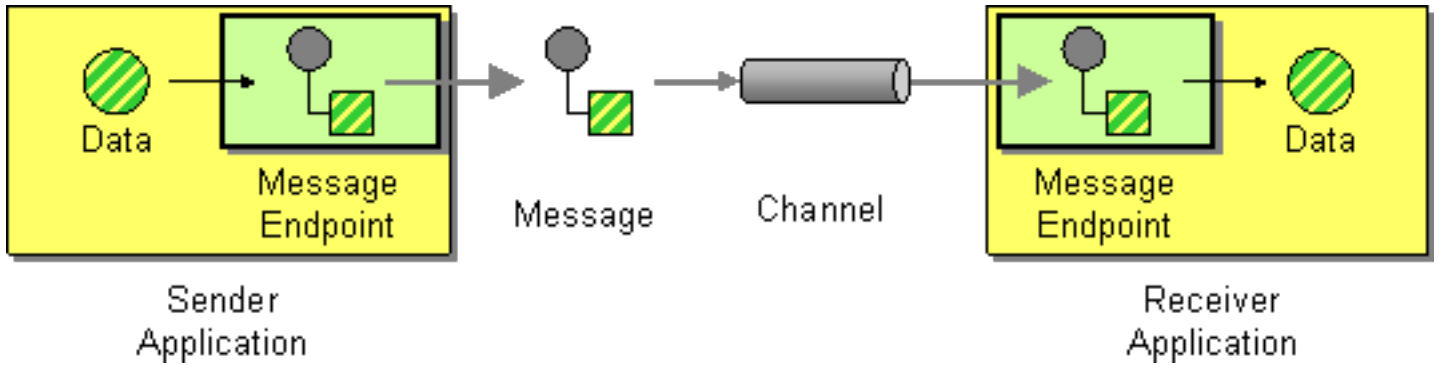
- **message endpoints** enable application services to send/receive messages to/from channels



The simplest integration

Message endpoints and channels enable realizing the simplest integration possible

- message endpoints for services to send/receive messages
- channels to transport messages from a service to another

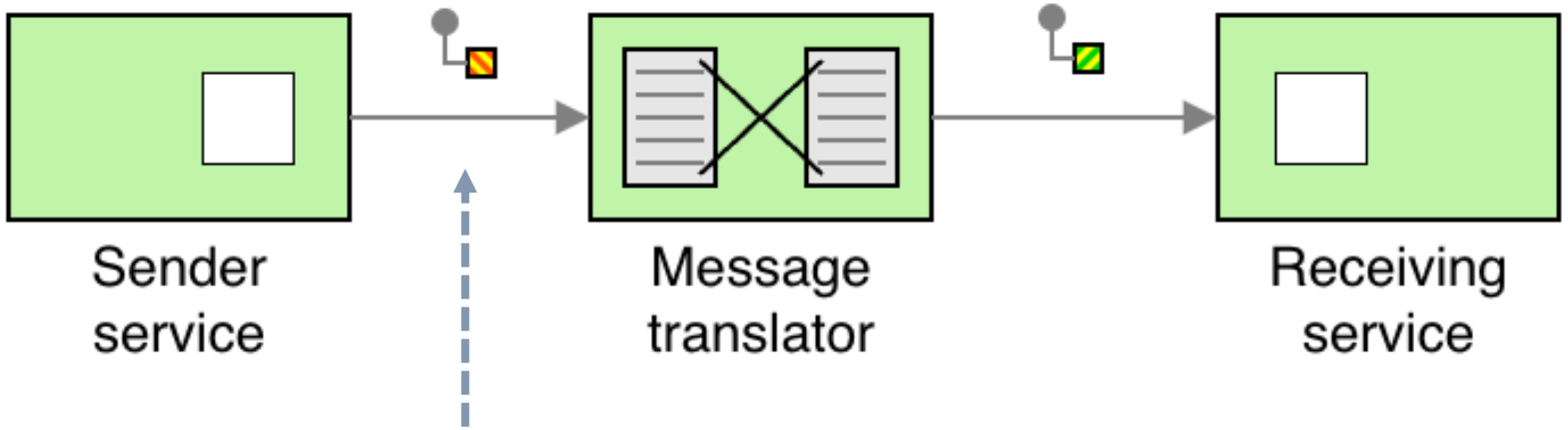


Is this enough for enterprise application integration?
What if, for instance, the receiver service expects a **different data format**?

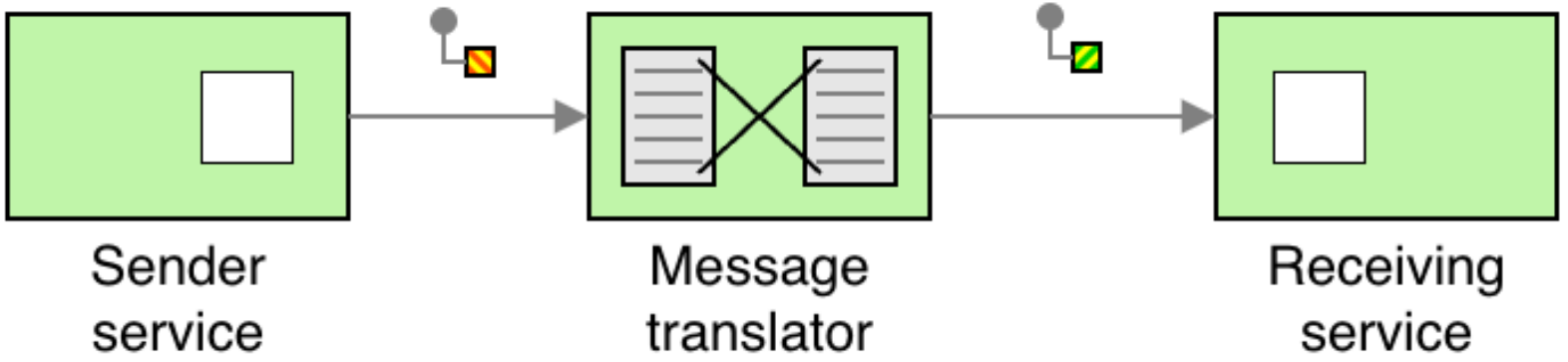
Message transformation

Sending and receiving services may expect different data types or formats

- We need a "**message translator**" the message somehow to suit the receiving services



Are we ready to integrate?



Message translators enable transforming messages, but other **other steps** may need to take place

For instance:

- How to **route** messages to different/multiple endpoints?
- How to **split** messages?
- How to **aggregate** messages?
- …

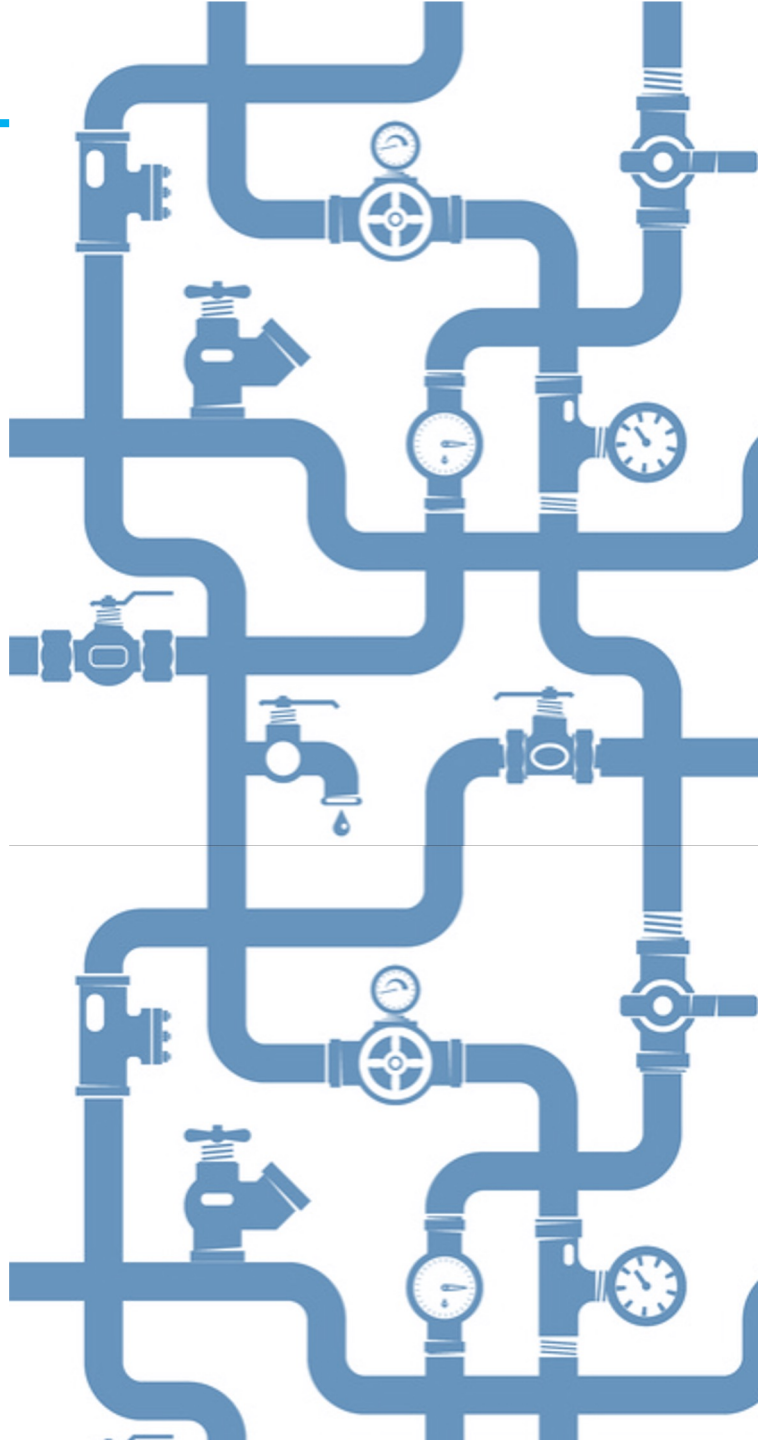
Pipes and filters

The **pipes and filters** architectural style (plus other EIPs) enable structuring the more complex integration needed by enterprise applications

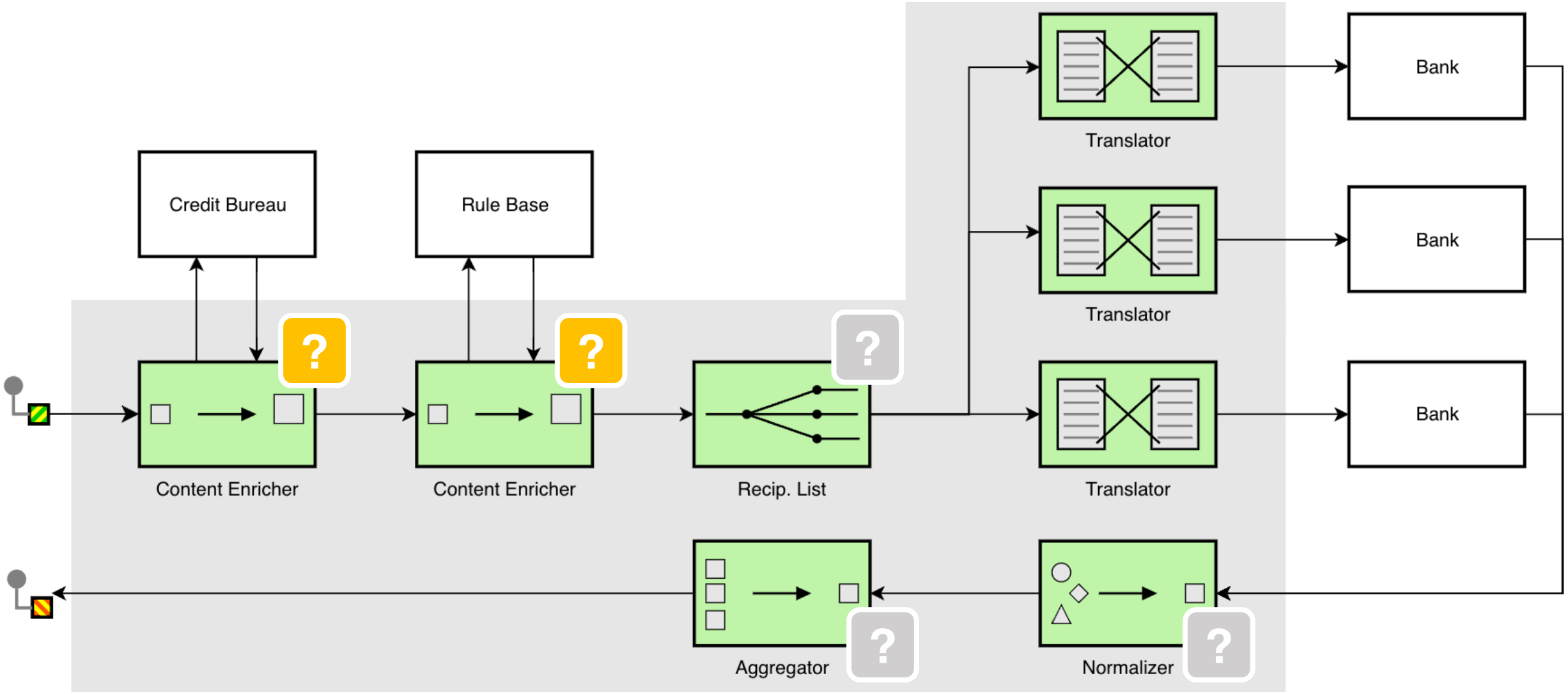
- Messages pass through multiple steps/components processing it (the **filters**)
- Components send message down the channels (**pipes**) they are connected to

Note that pipes and filters

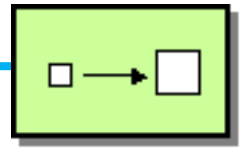
- all deal with the same message/channel abstraction, and
- can be composed flexibly depending on the circumstances



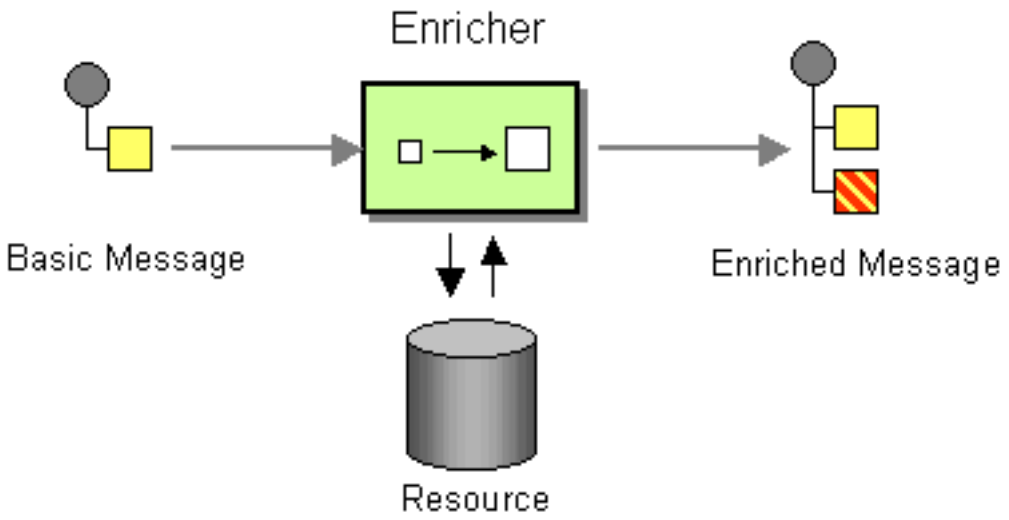
Pipes and filters: *Loan Broker* example



Content Enricher



When sending messages from a service to another, the target service may require more information than the source service can provide

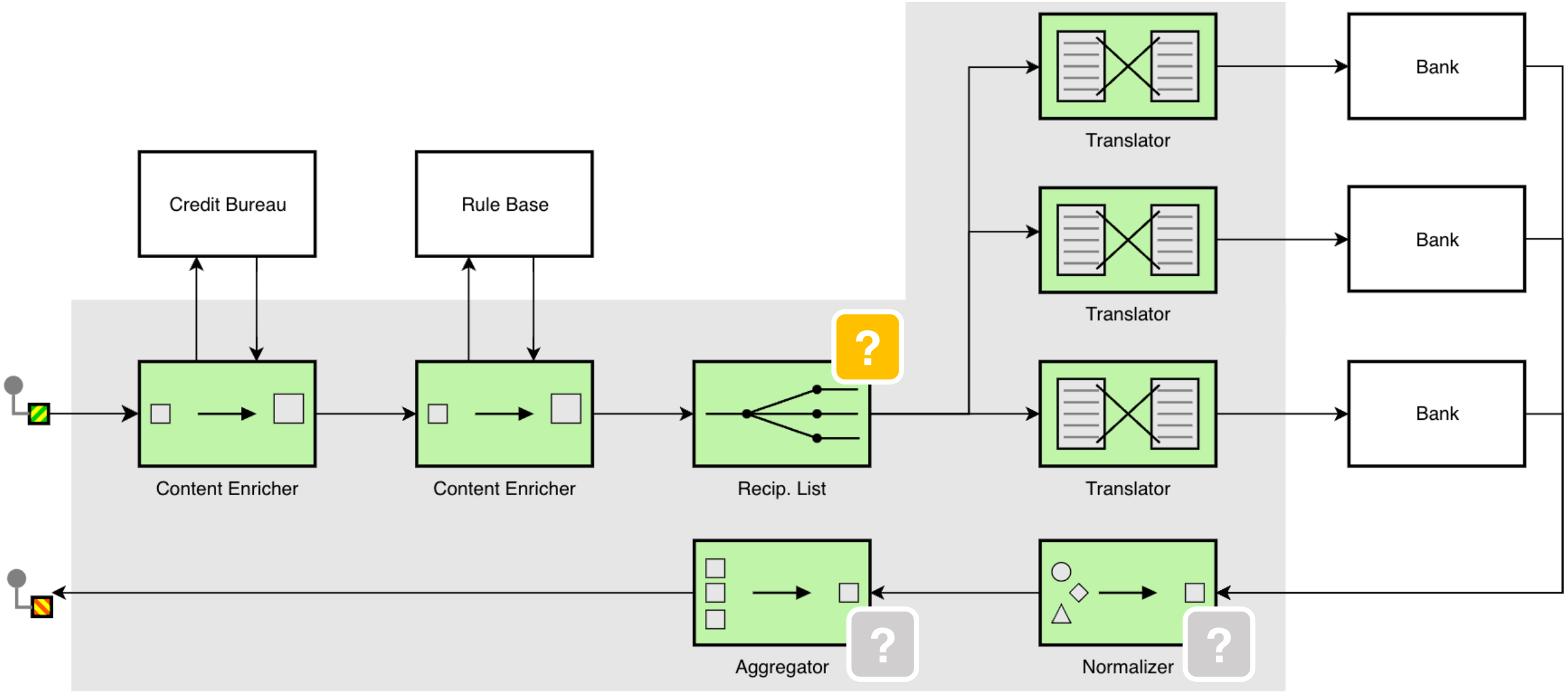


The **Content Enricher** uses information inside the incoming message (e.g. key fields) to retrieve data from an external source.

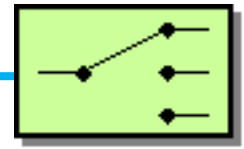
- Retrieved data is appended to the messagge
- Original information may be carried over or no longer needed

  → depends on the specific needs of the receiving service

Pipes and filters: *Loan Broker* example



Message routing



We know how to send messages over a channel and apply processing/filtering steps

→ each filter is connected to an incoming and outgoing channel

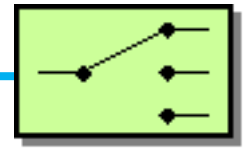
Endpoints are however kind of hardcoded

- Lack of flexibility
- What if an endpoint changes? // quite common in cloud-native scenarios

We need a better solution for **message routing**, as **routers** would know where and how to send messages, e.g.,

- **content-based router** could route messages messages based on
  - message type (in headers)
  - message content (in body)
- **context-based router** could route based on conxtextual information
  - retrieved from a central configuration location
  - e.g., in “testing” context, messages would be routed differently than in “production”

Message routing (2)



In general, a **message router** is connected to multiple channels and contains the logic to decide which channel it should send to

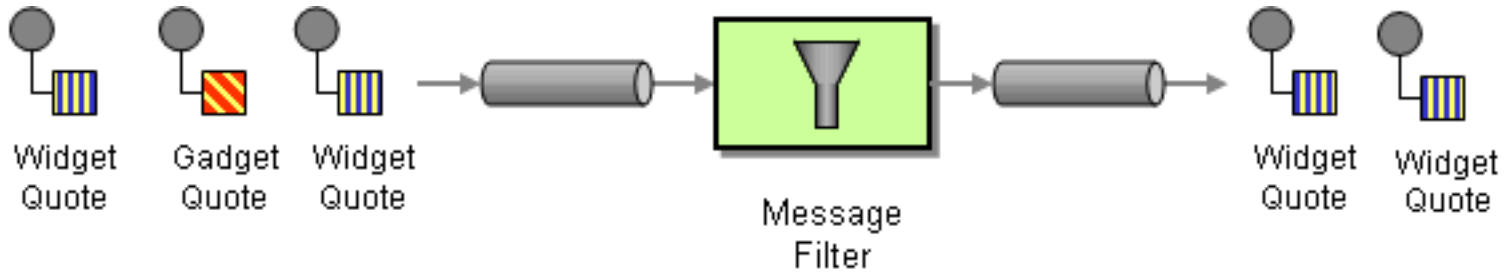
Message routing pattern	Consumed msgs	Published msgs	Stateful?
Message Filter	1	0 or 1	No*
Content-based Router	1	1	No*
Recipient List	1	multiple (incl. 0)	No
Splitter	1	multiple	No
Aggregator	multiple	1	Yes

* typically stateless, but sometimes could be stateful
(we also discuss a concrete example later)

Message filtering

Example: suppose that an enterprise applications sells gadgets and widgets, also sending price changes/promotions to large customers

→ what if some customers are interested only in widget quotes?

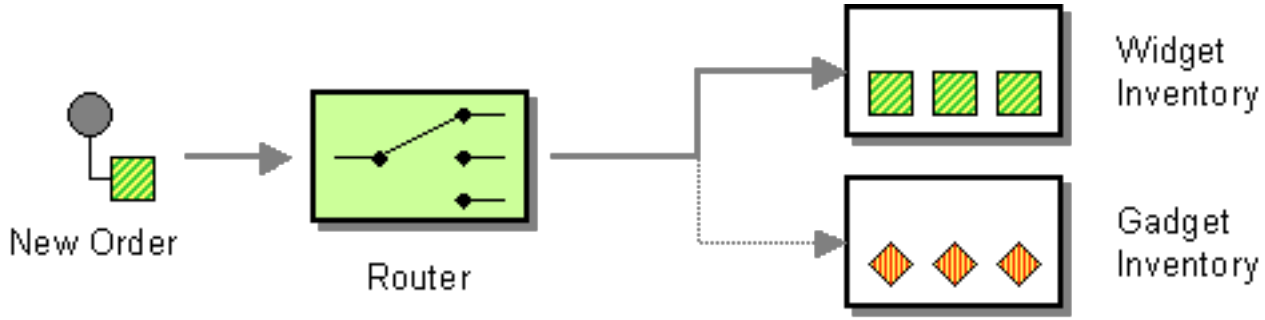


A **message filter** can eliminate undesired messages from a channel based on given criteria

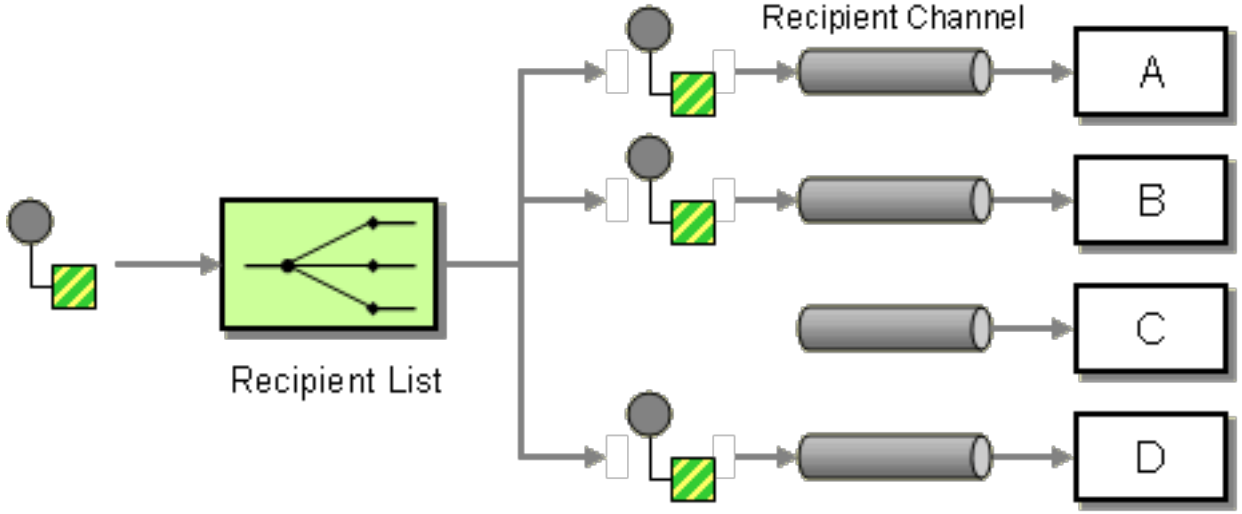
- Only one output channel
- If a message matches the given criteria, it is routed to the output channel
- Otherwise, it is discarded

Routing messages to multiple destinations

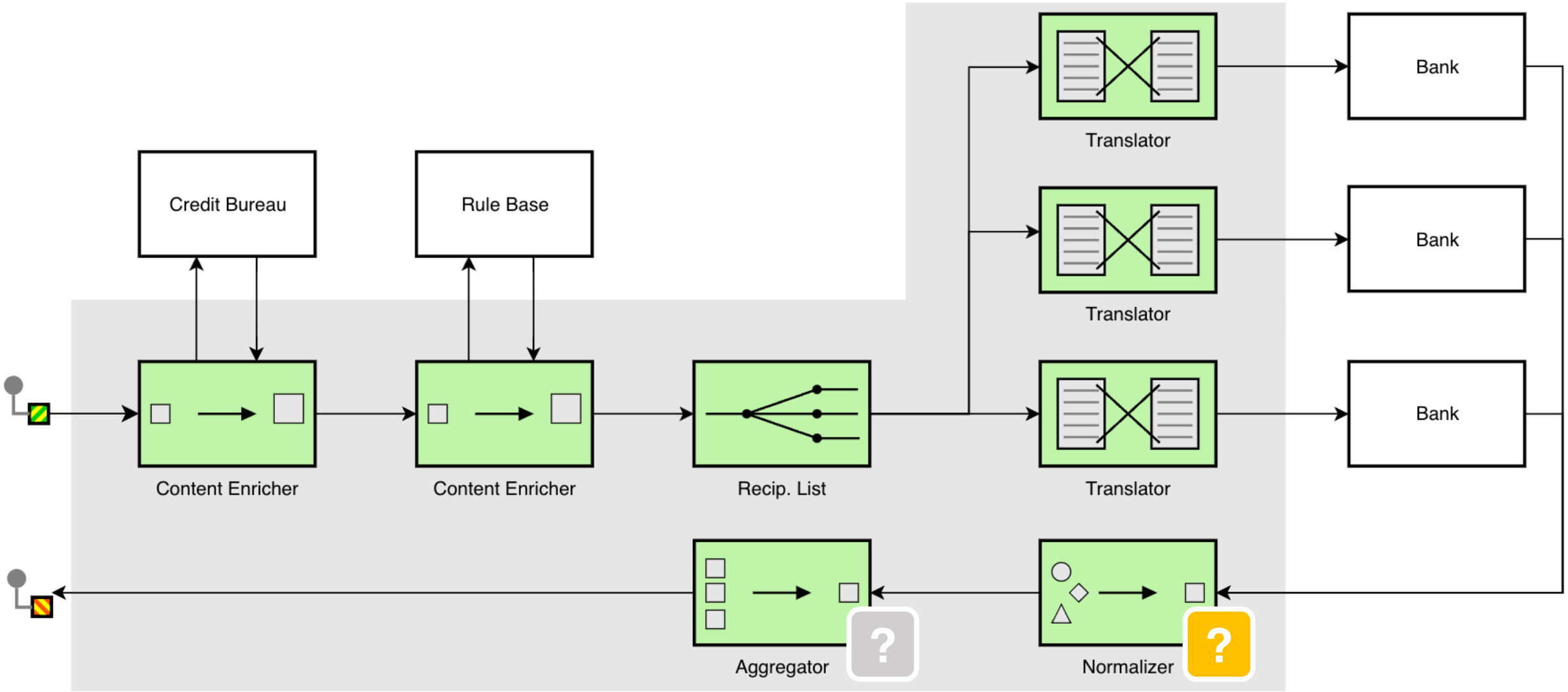
A **content-based router** enables routing each message to the correct recipient based on the message content



A **recipient list** inspects an incoming message, determines the list of desired recipients, and forwards the message to all channels associated with the recipients in the list



Pipes and filters: *Loan Broker* example

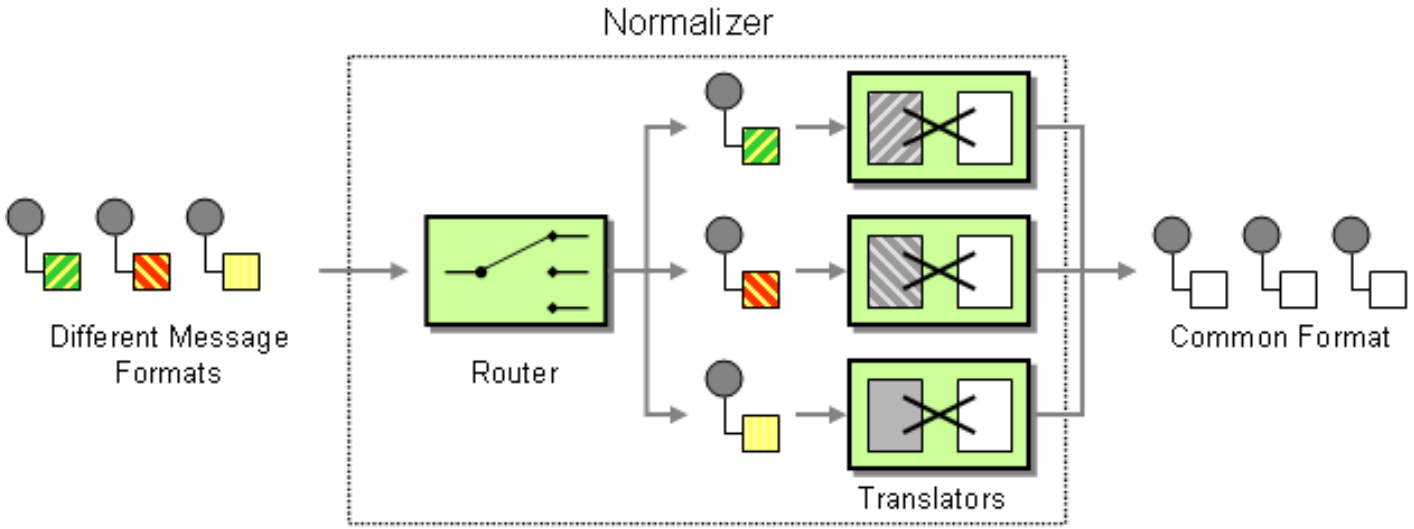


Normalizer: a composite EIP

In service integration, it can happen that the “same data” is received in “different formats”

→ e.g., if received from different services

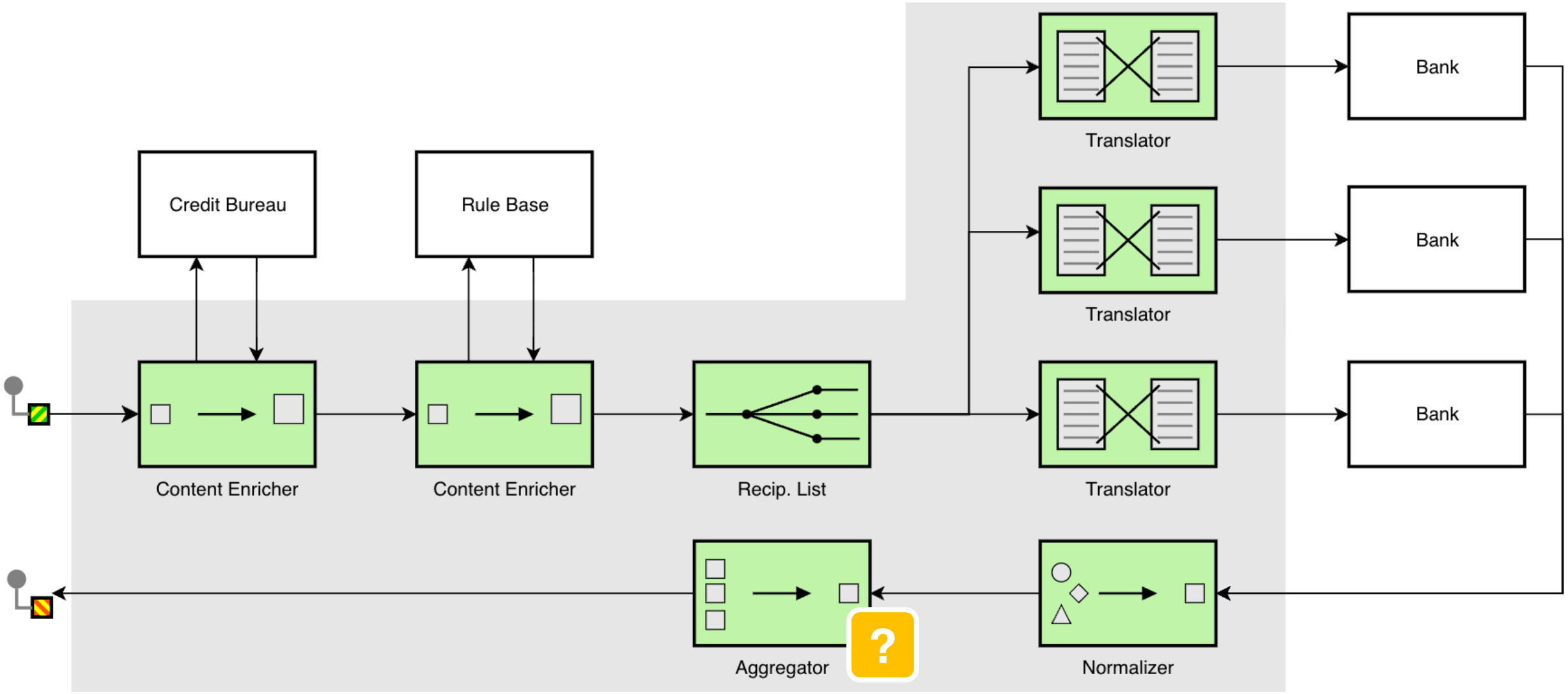
A **normalizer** enable translating messages to match a common data format



A normalizer can be realized as a **composition** of other EIPs

- A **message router** routing incoming messages to the most suited translator
- A set of **translators** transforming incoming (different) messages to a common format

Pipes and filters: *Loan Broker* example



Scattering / gathering messages // still message routing

Rather than defining long sequential integrations, some integration steps may go **in parallel**

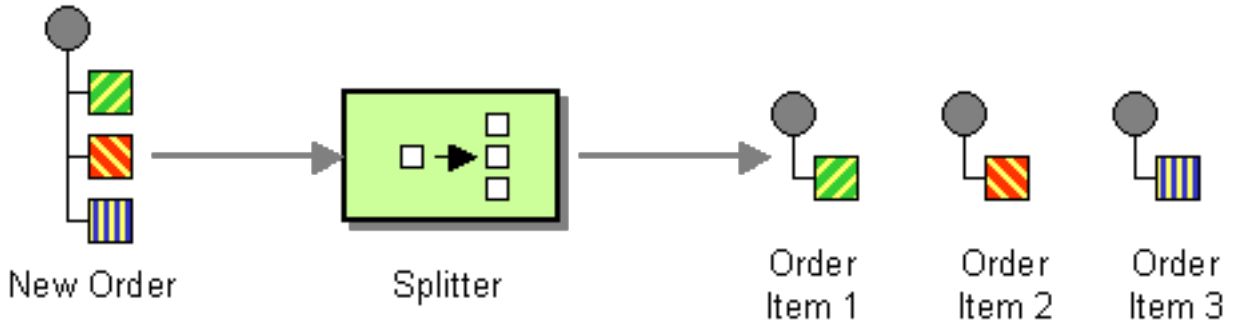
- e.g., when sending the loan request to multiple different banks in our example
- Independent processes , which can be **splitted** executed in parallel
- The results of such processes can be **aggregated** and decisions made on how to proceed

Message **splitters** and **aggregators** come exactly to this purpose

Message routing pattern	Consumed msgs	Published msgs	Stateful?
Message Filter	1	0 or 1	No*
Content-based Router	1	1	No*
Recipient List	1	multiple (incl. 0)	No
Splitter	1	multiple	No
Aggregator	multiple	1	Yes

Splitter

How can we process a message if it contains multiple elements, each of which may have to be processed in a different way?

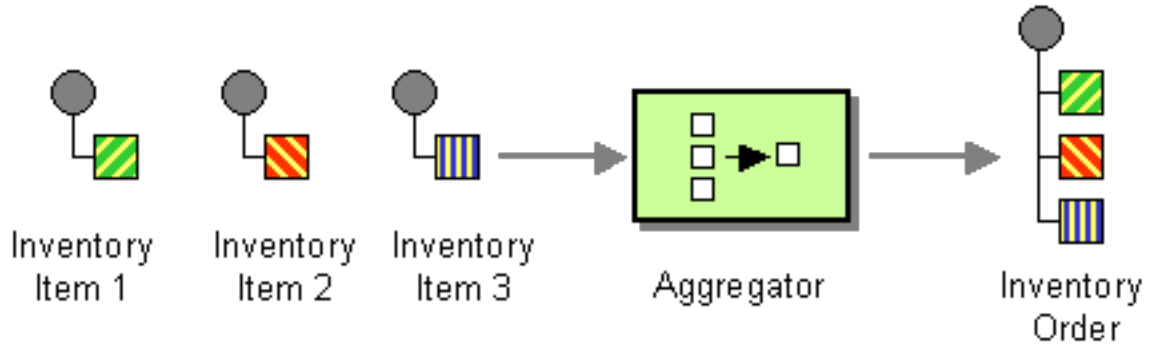


A **splitter** enables breaking out the composite message into a series of individual messages

- Each output message contains a different portion of the original message
- Each output message can be processed differently
  - e.g., exploiting content-based routers to ship them to different processing chains

Aggregator

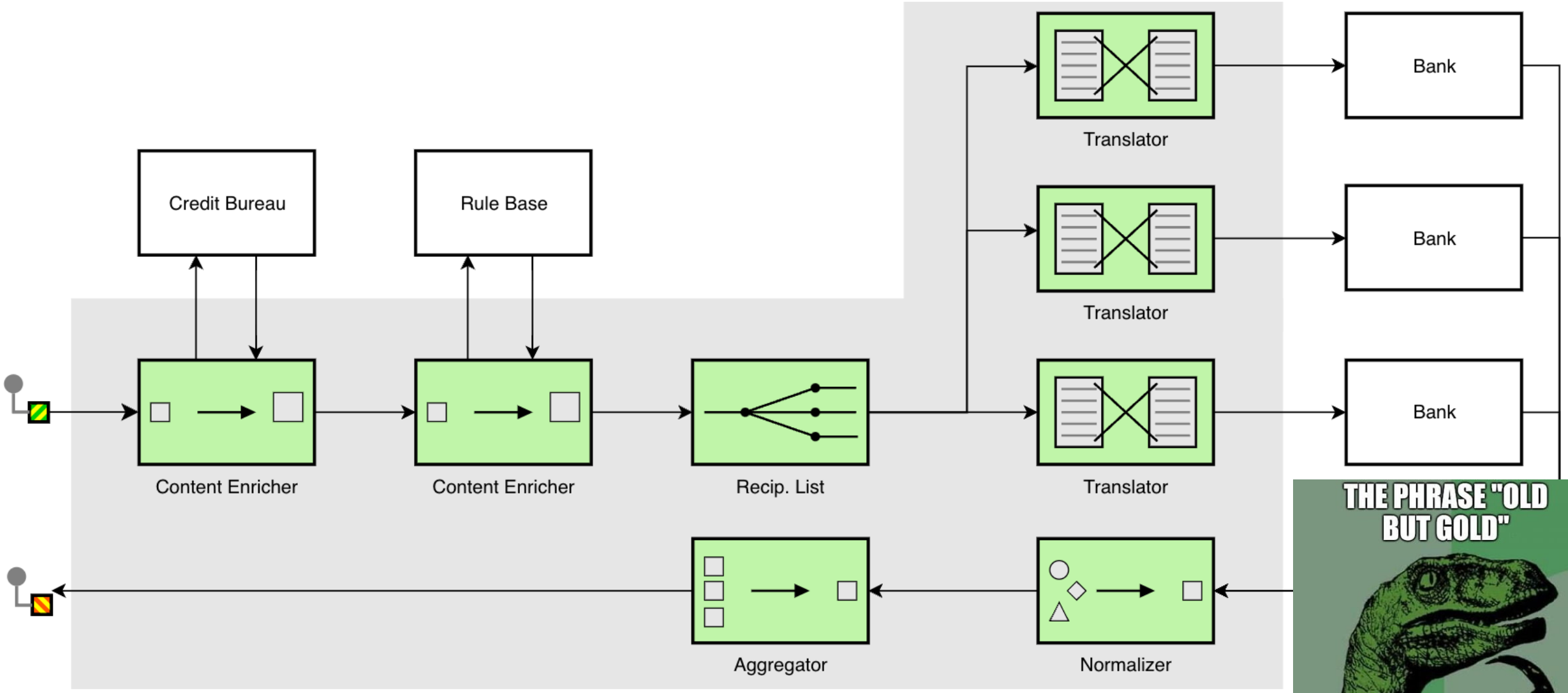
How do we combine the results of individual, but related messages so that they can be processed as a whole?



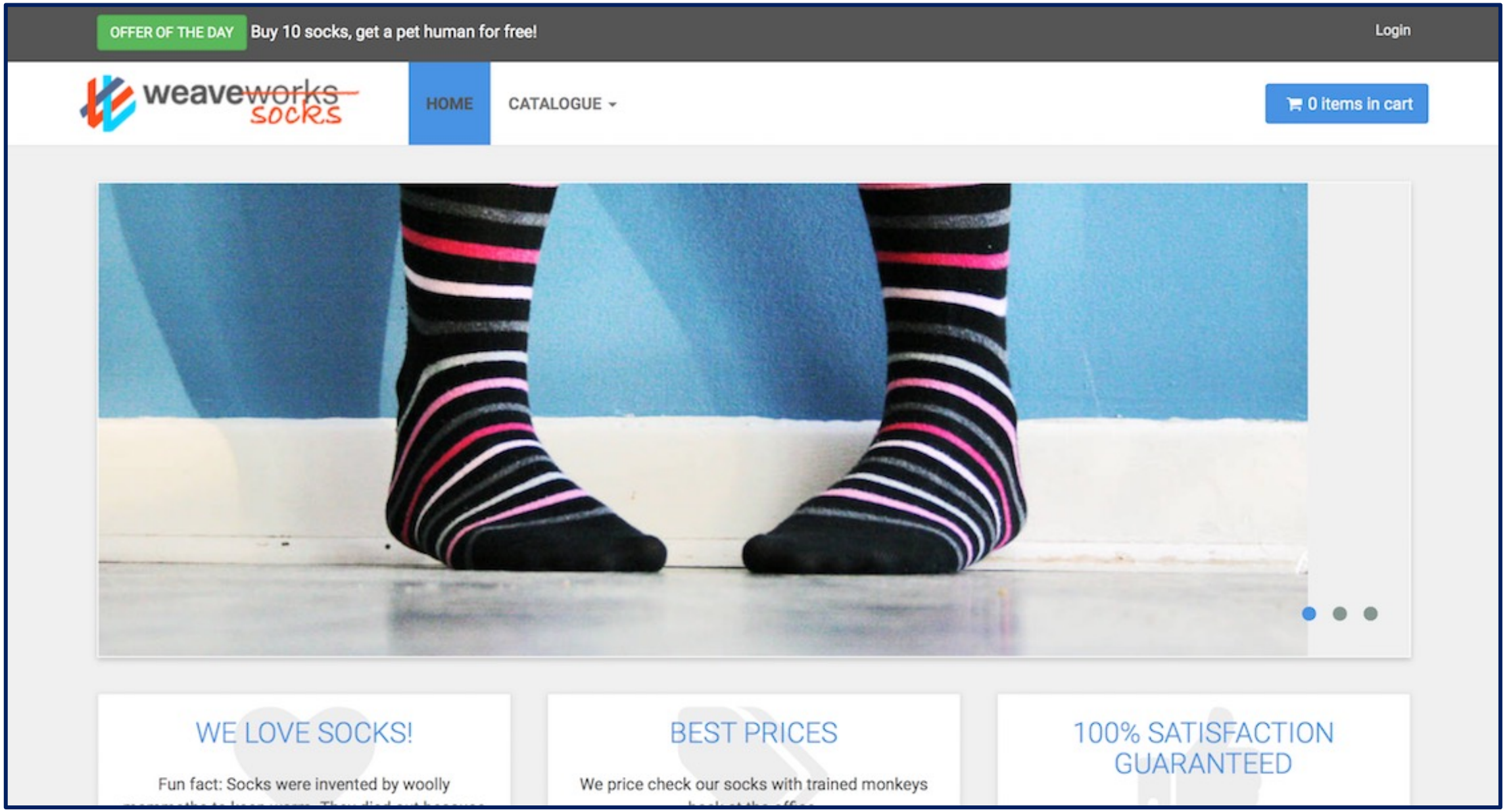
A (stateful) **aggregator** enables collecting and storing individual messages until a complete set of related messages has been received.

- It then publishes a single message distilled from the individual messages
- The published message can be processed as a whole

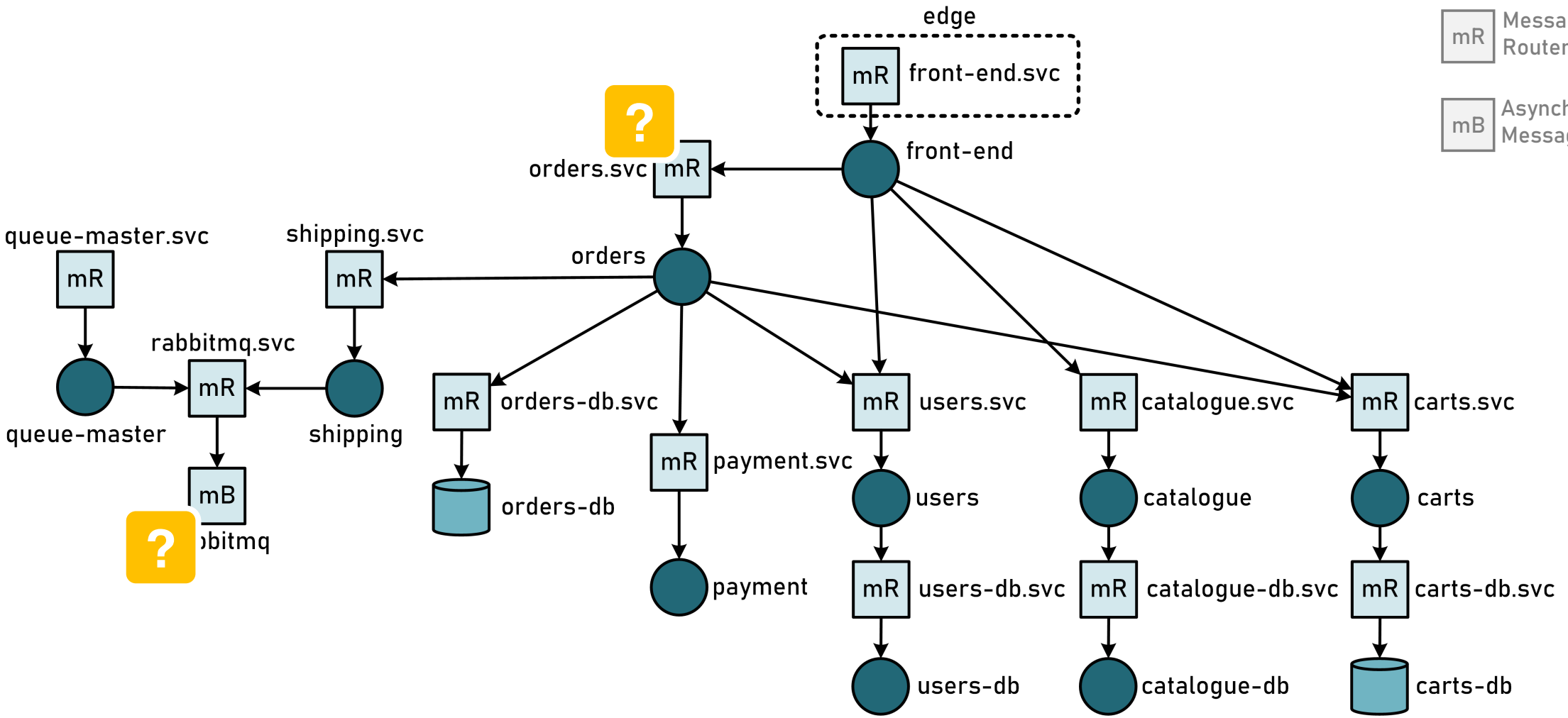
Pipes and filters: *Loan Broker* example



Newer examples: *Sock Shop*

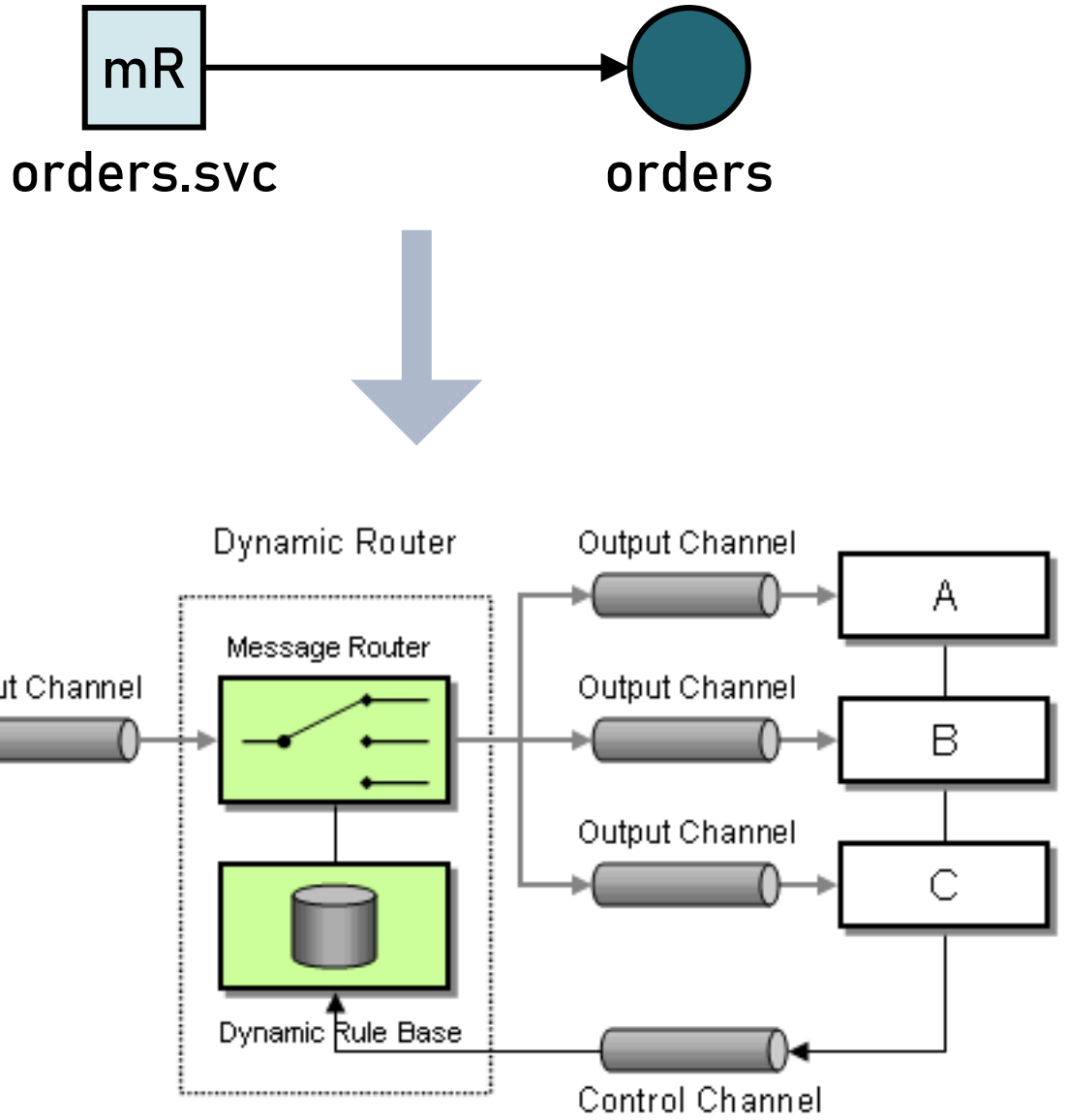


Newer examples: *Sock Shop*

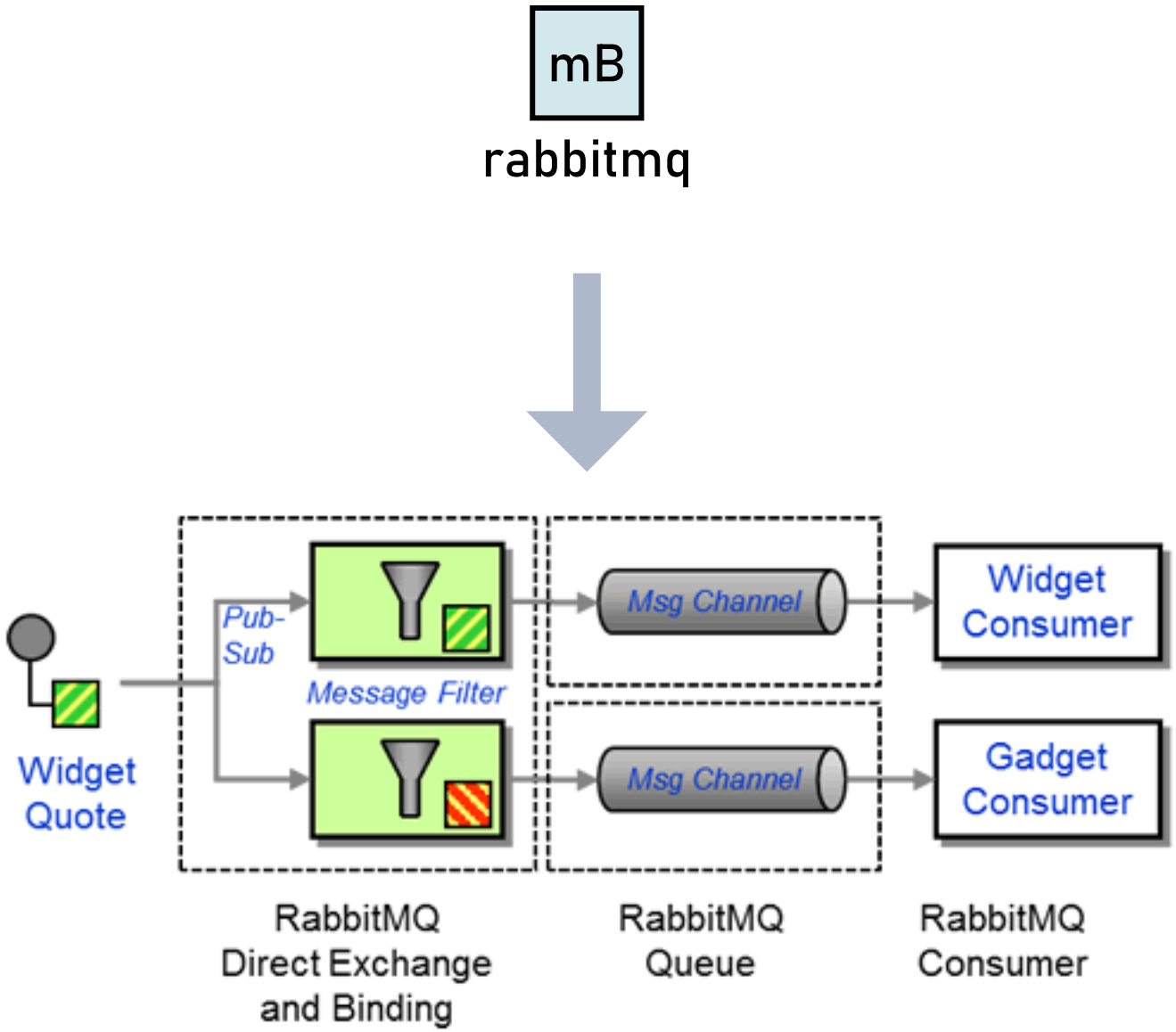


https://microservices-demo.github.io

Kubernetes services → load balancers → routers



RabbitMQ



Source: https://www.enterpriseintegrationpatterns.com/patterns/messaging/Filter.html

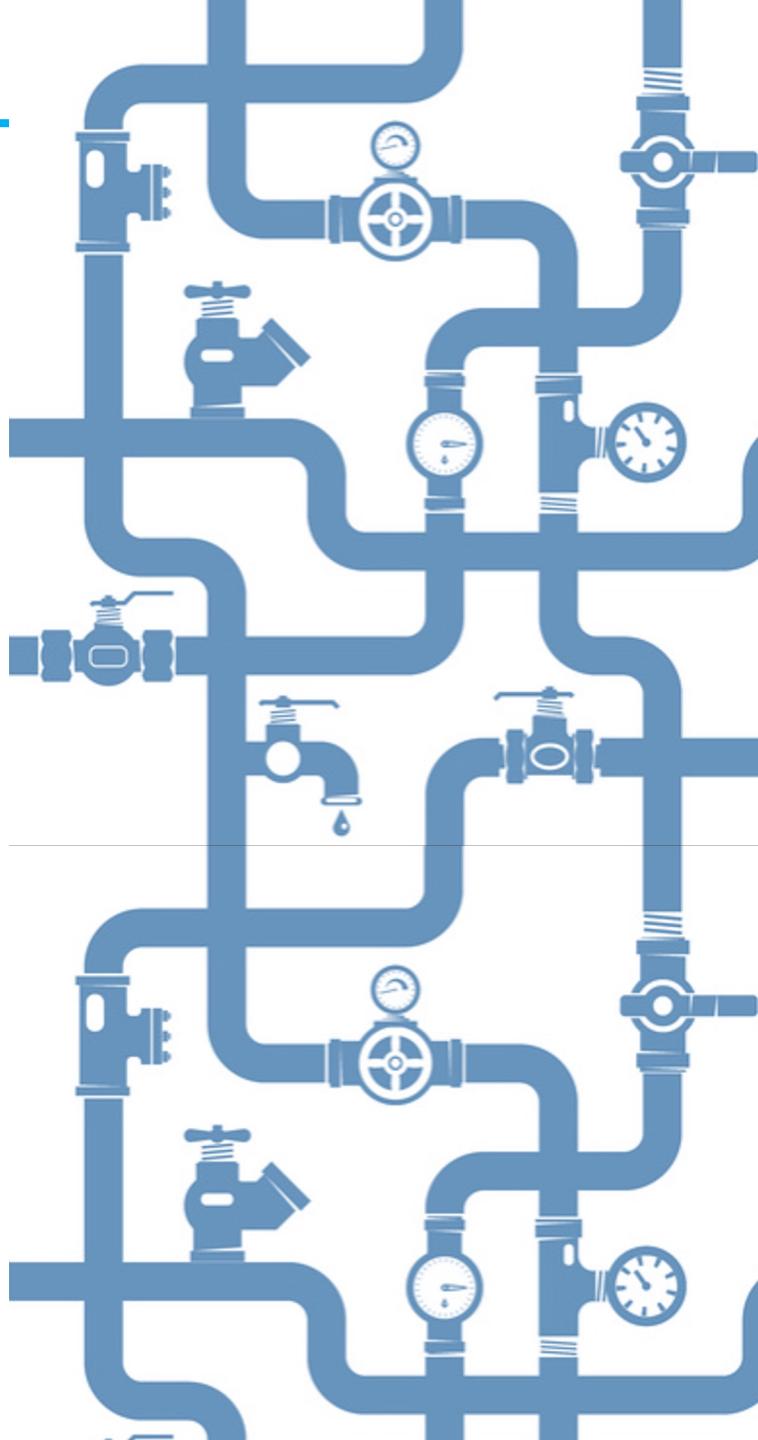
The road so far…

EIPs enable integrating multiple heterogeneous services

- Pipes and filters architectural style
- Dealing with complexity in a coherent way
- Composable message-oriented patterns

Currently used to structure **enterprise-level cloud-native apps**

- Microservices
- Kubernetes-based deployment
- Apache Camel
- Red Hat Jboss Fuse
- Serverless architectures
- …and many others!



…and the road ahead!

- How to **validate** designed integrations?
- What about known **architectural smells** and **refactorings**?
- What about **security**? Are integrated applications secure-by-design?
- How to **explain and isolate failures** in integrated applications?
- How to automatically deploy integrated applications, **serverlessly**?