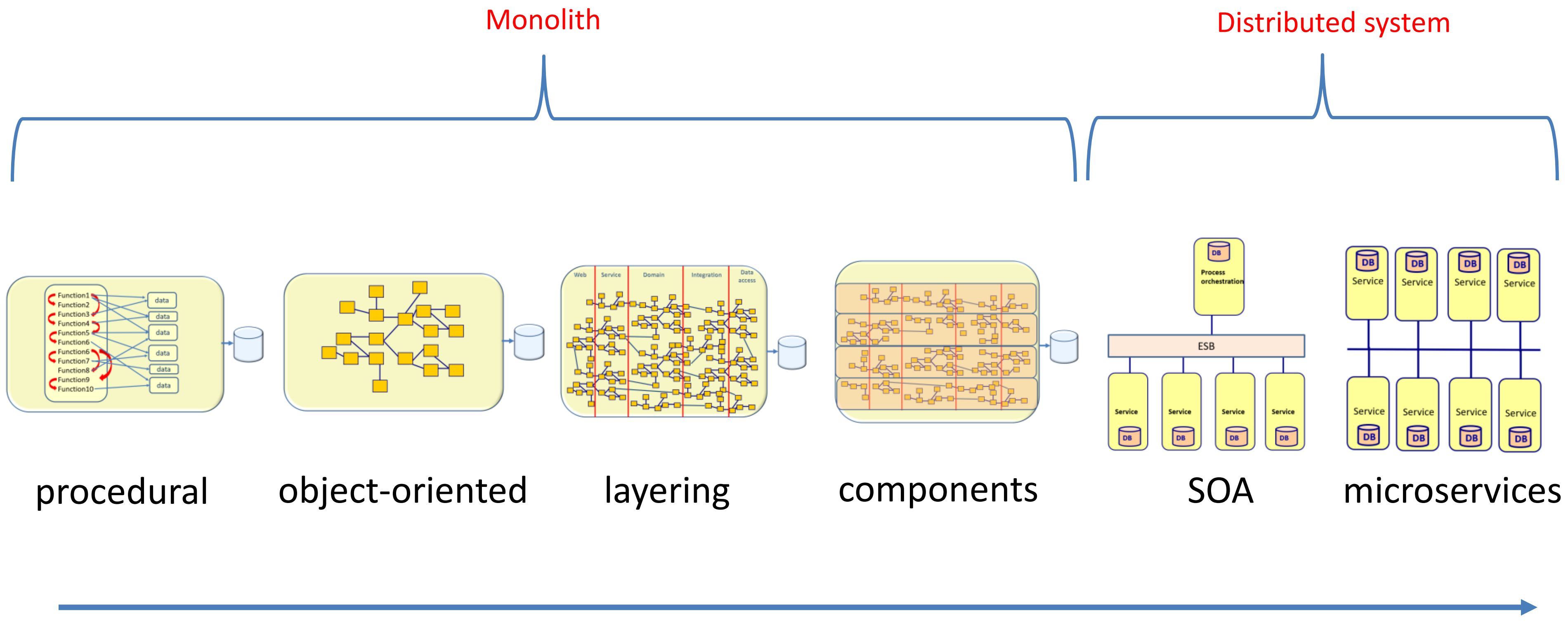
Lesson 5

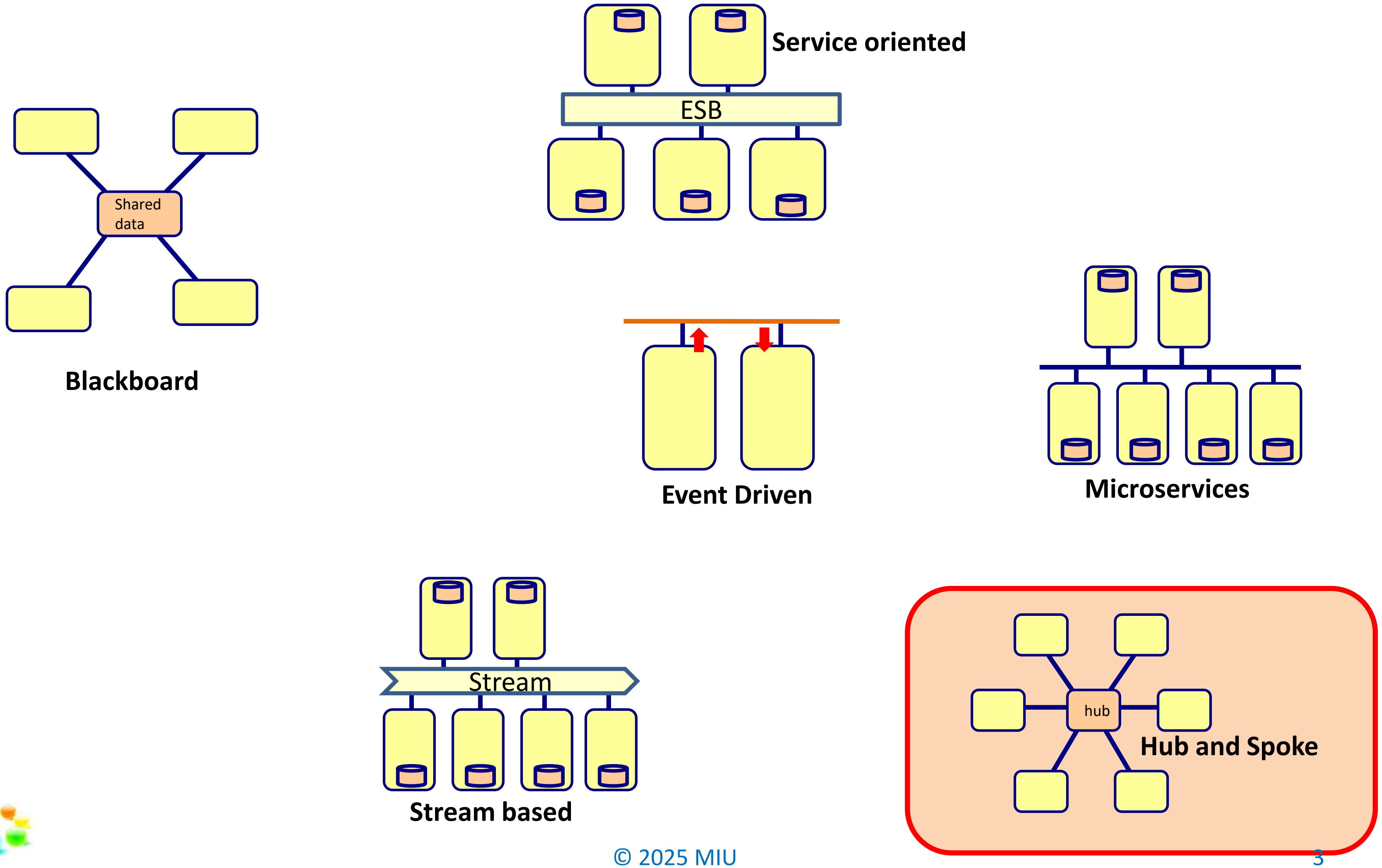
# SERVICE ORIENTED ARCHITECTURE INTEGRATION PATTERNS

# Architecture evolution



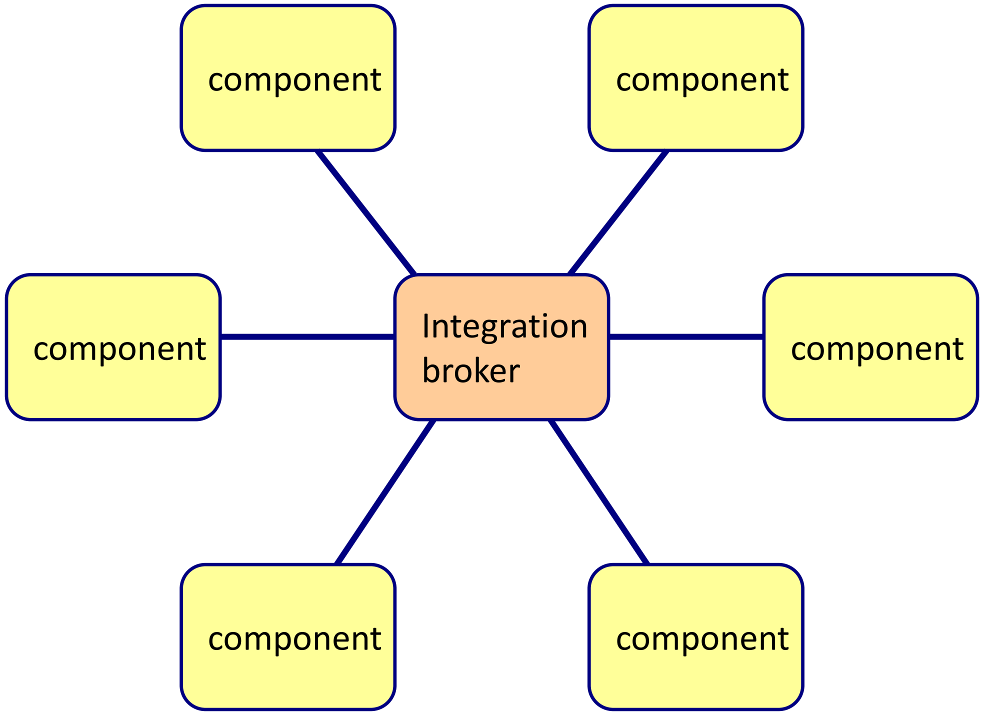
- Smaller and simpler parts
- More separation of concern
- More abstraction
- Less dependencies

# Architecture styles to connect applications



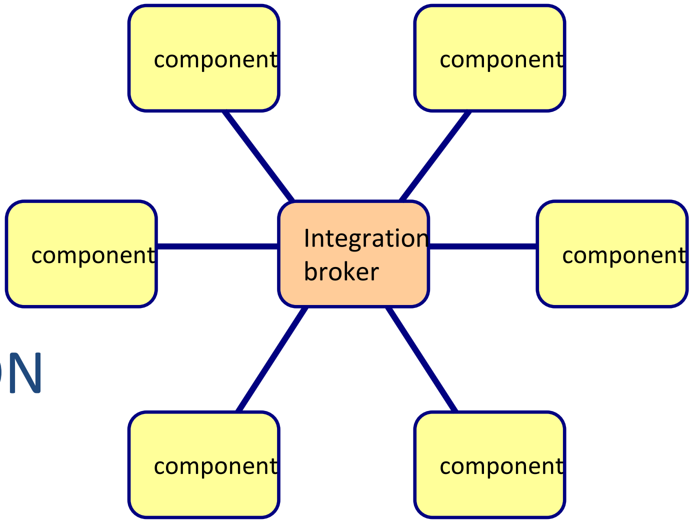
## Hub and Spoke

- Integration broker



# Hub and Spoke

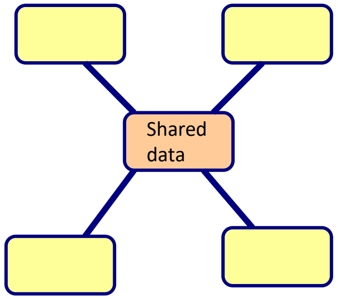
- Functionality:
  - Transport
  - Transformation
    - For example from XML to JSON
  - Routing
    - Send the message to a component based on certain criteria (content based routing, load balancing, etc.)
  - Orchestration
    - The business process runs within the integration broker



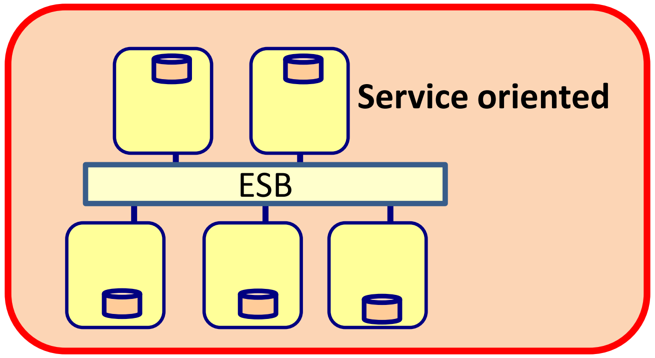
## Hub and spoke

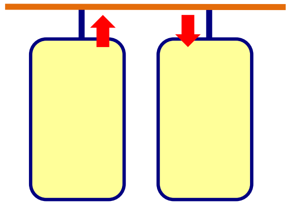
- Benefits
  - Separation of integration logic and application logic
  - Easy to add new components
  - Use adapters to plugin the integration broker
- Drawbacks
  - Single point of failure
  - Integration brokers are complex products
  - Integration broker becomes legacy itself

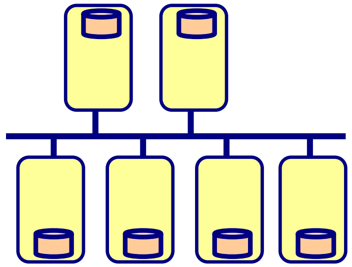
# Architecture styles to connect applications

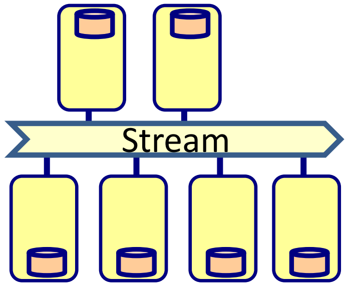


Blackboard

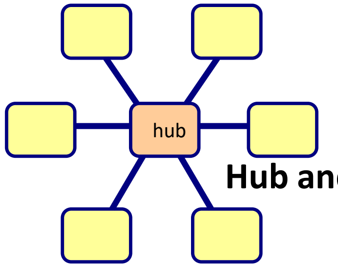



Event Driven


Microservices

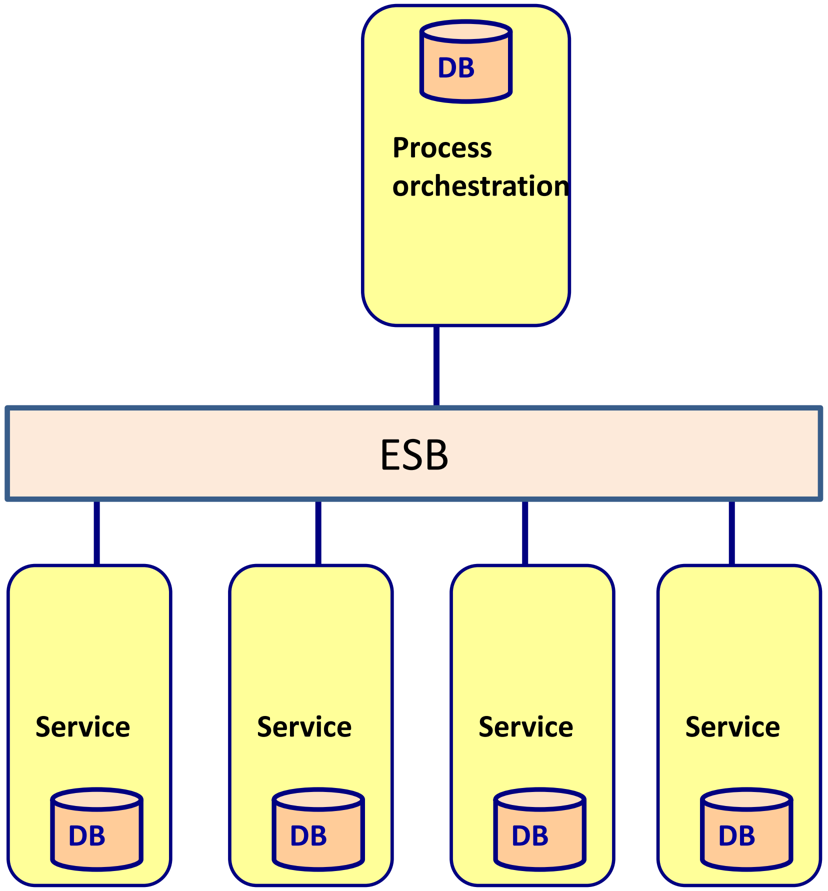


Stream based



Hub and Spoke

# Service Oriented Architecture

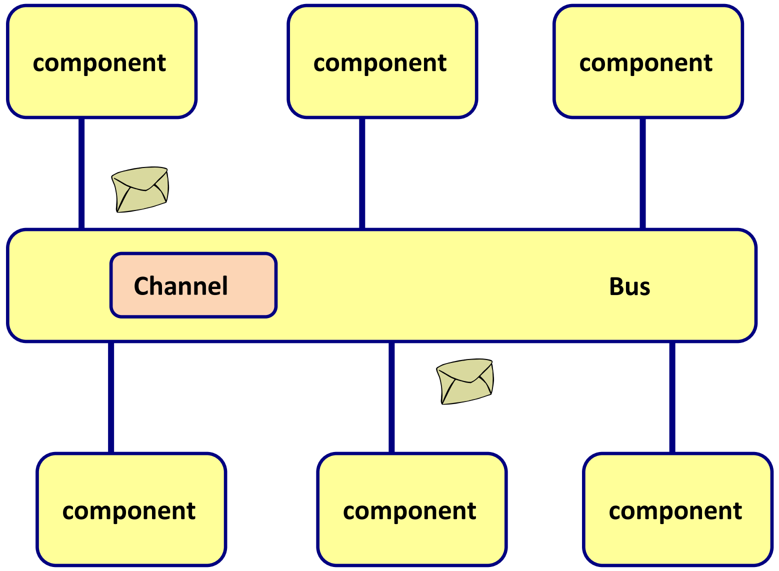


# 3 different aspects of a SOA

1. Communication through ESB
   - Standard protocols
2. Decompose the business domain in services
   - Often logical services
3. Make the business processes a 1st class citizen
   - Separate the business process from the application logic

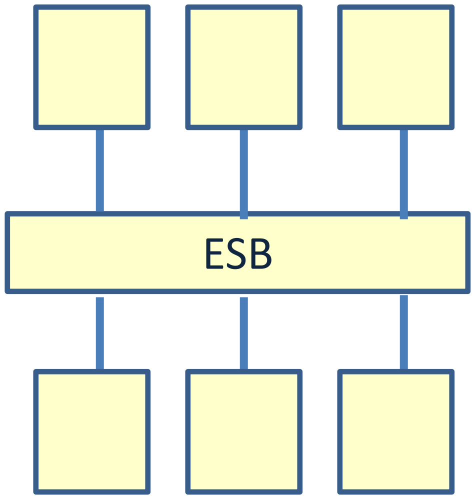
# Responsibility of the bus

- Routing
  - Static
  - Content based
  - Rule based
  - Policy based
- Message transformation
- Message enhancing/filtering
- Protocol transformation
  - Input transformation
  - Output transformation
- Service mapping
  - Service name, protocol, binding variables, timeout, etc.
- Message processing
  - Guaranteed delivery
- Process choreography
  - Business process
  - BPEL
- Transaction management
- Security



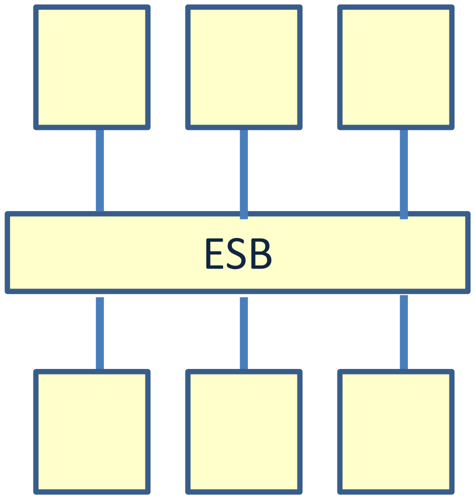
## Service Oriented Architecture

- Advantages
  - Independent services
  - Easy to add new services
  - Separation of business processes and service logic
  - Architecture is optimized for the business
  - Reuse of services
  - Architecture flexibility



## Service Oriented Architecture

- Disadvantages
  - Complex ESB
  - Changing the business process while business processes are still running is very difficult
  - Most SOA's are build on top of monoliths



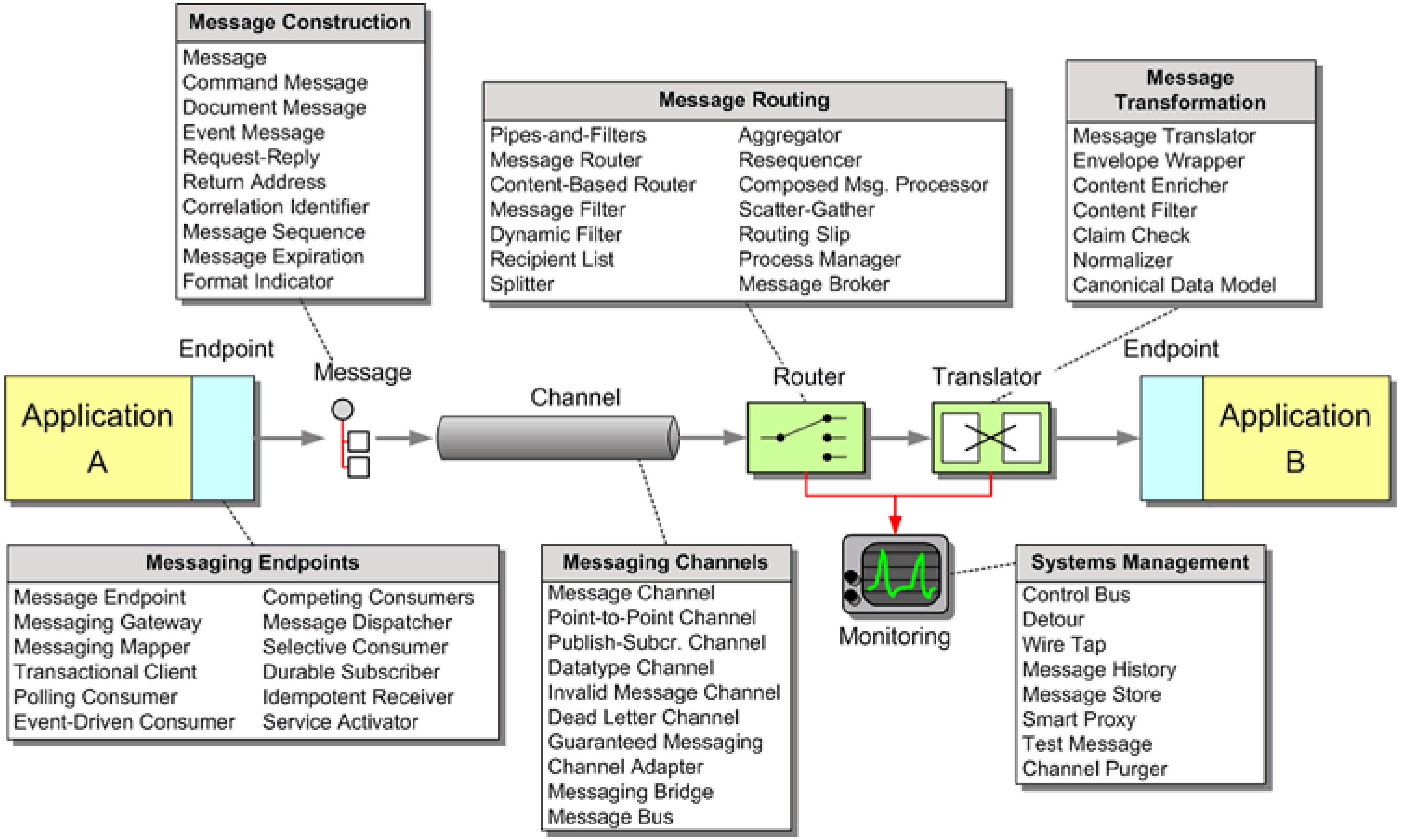
## Main point

- In a service oriented architecture the business processes run on the ESB and they orchestrate all activity within the SOA architecture..

- The more you are in tune with the laws of nature, the more support of nature you are able to enjoy.

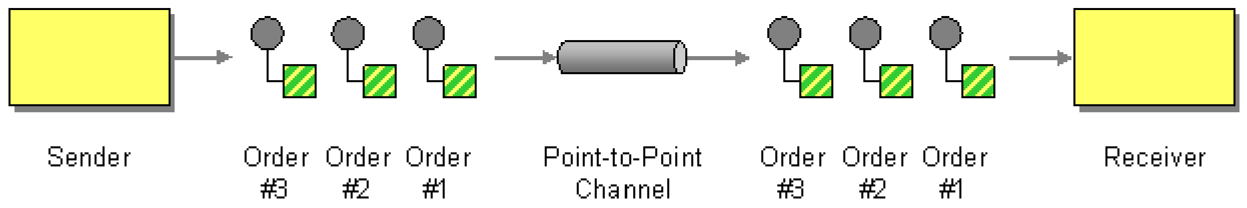
## ENTERPRISE INTEGRATION PATTERNS

# Enterprise Integration Patterns

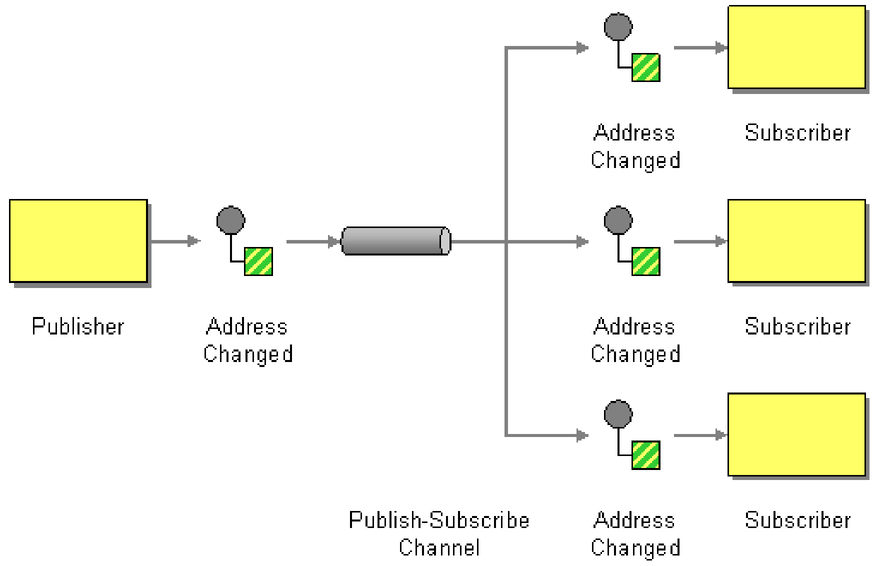


https://www.enterpriseintegrationpatterns.com/

# Messaging channel patterns

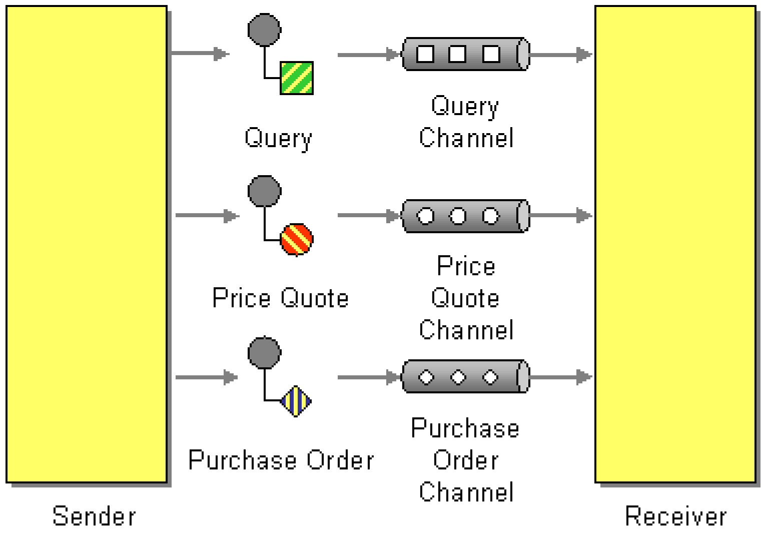


Point-to-point: only one receiver will receive the message



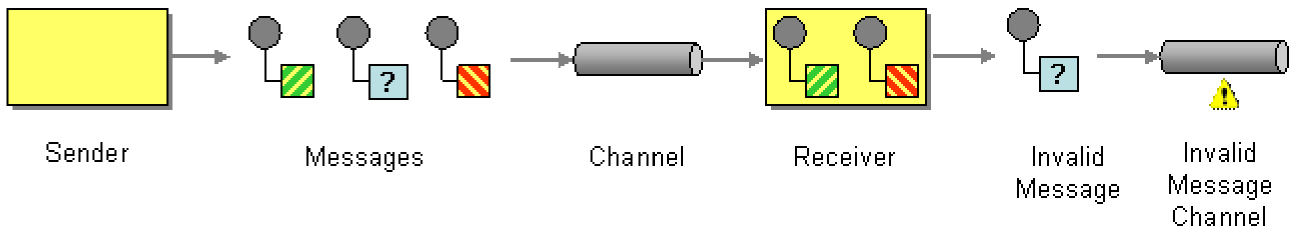
Publish-Subscribe : every subscriber will receive the message

# Messaging channel patterns

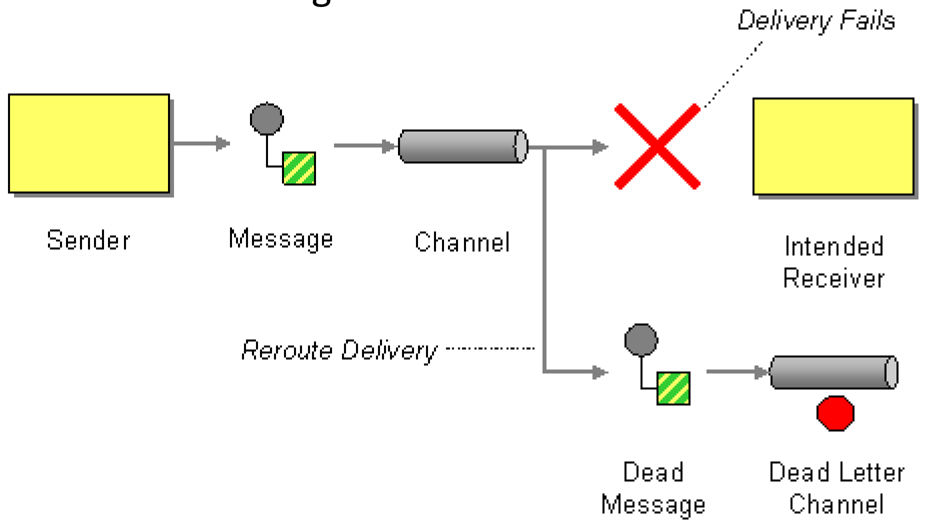


Datatype Channel : use a channel for each data type, so that the receiver know how to process it

# Messaging channel patterns

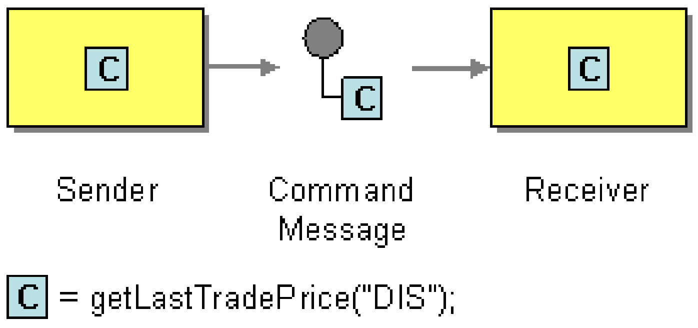


Invalid Message Channel : for messages that don't make sense for the receiver

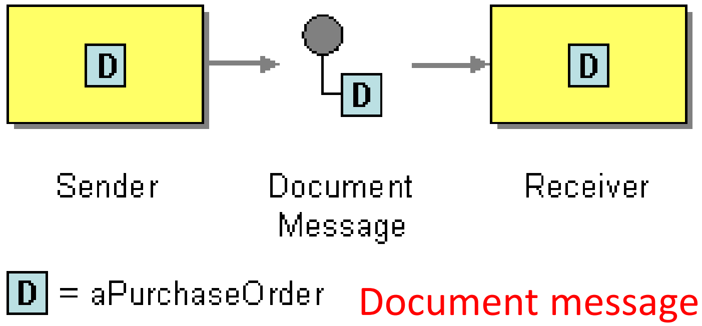


Dead Letter Channel : for messages that can't be delivered

# Message construction patterns

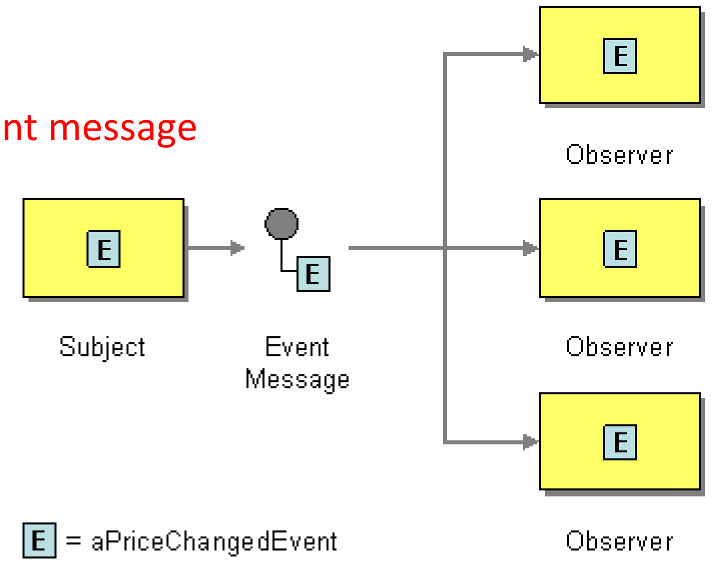


Command message

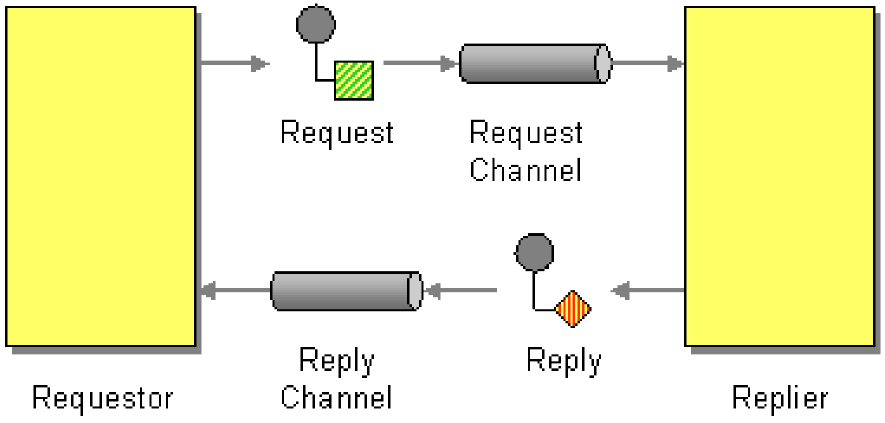


Document message

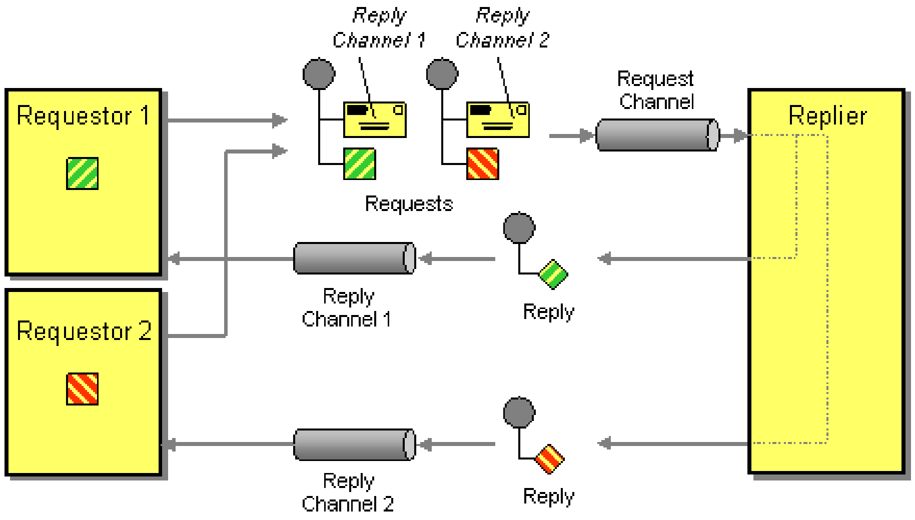
Event message



# Message construction patterns

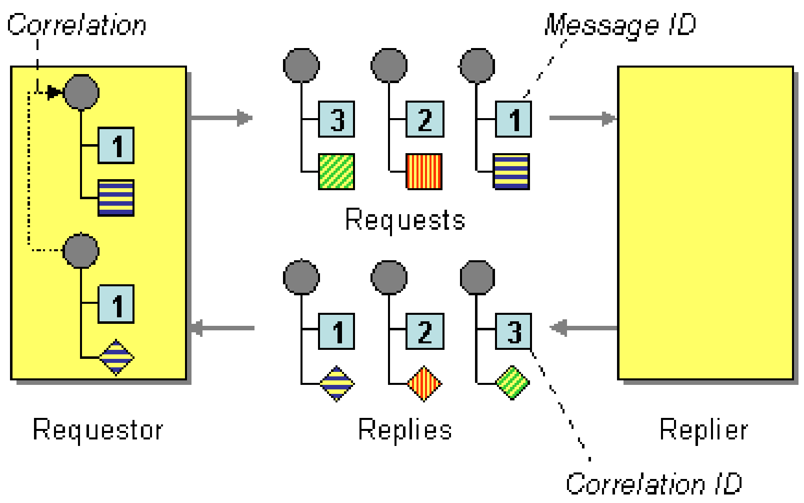


Request-Reply

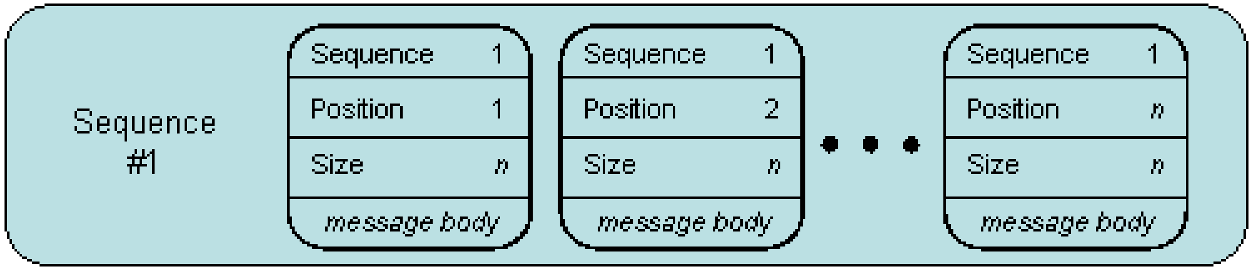


Return address

# Message construction patterns

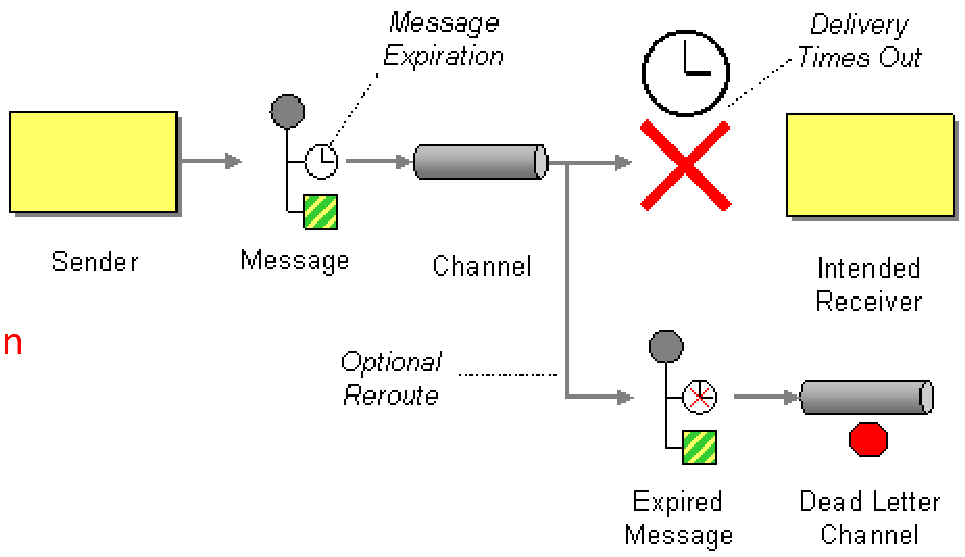


Each reply message should contain a Correlation Identifier, a unique identifier that indicates which request message this reply is for



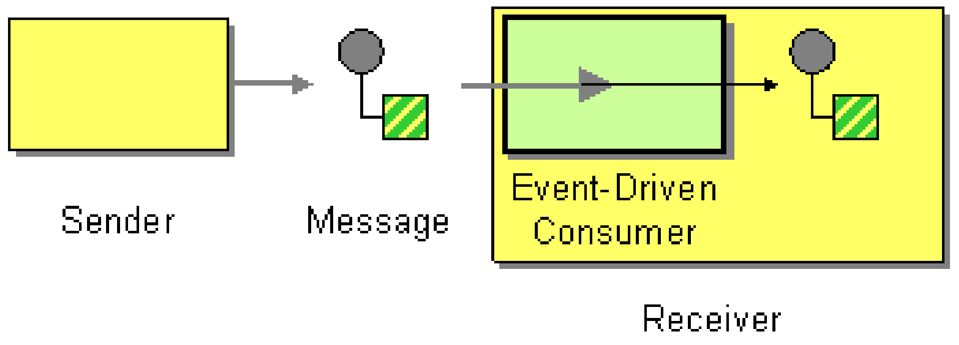
Whenever a large set of data may need to be broken into message-size chunks, send the data as a Message Sequence and mark each message with sequence identification fields.

# Message construction patterns

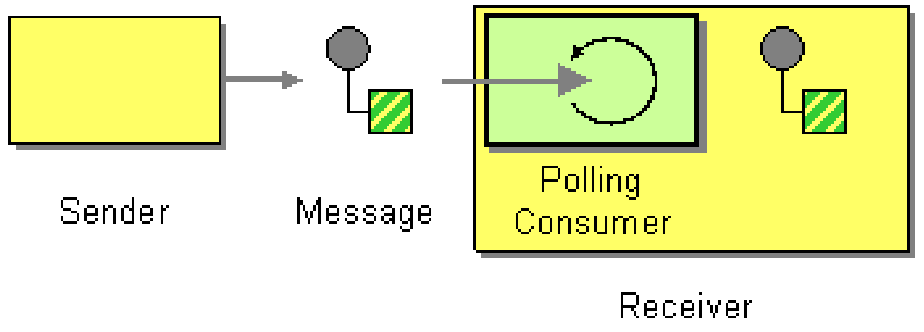


Message expiration

# Message Endpoint

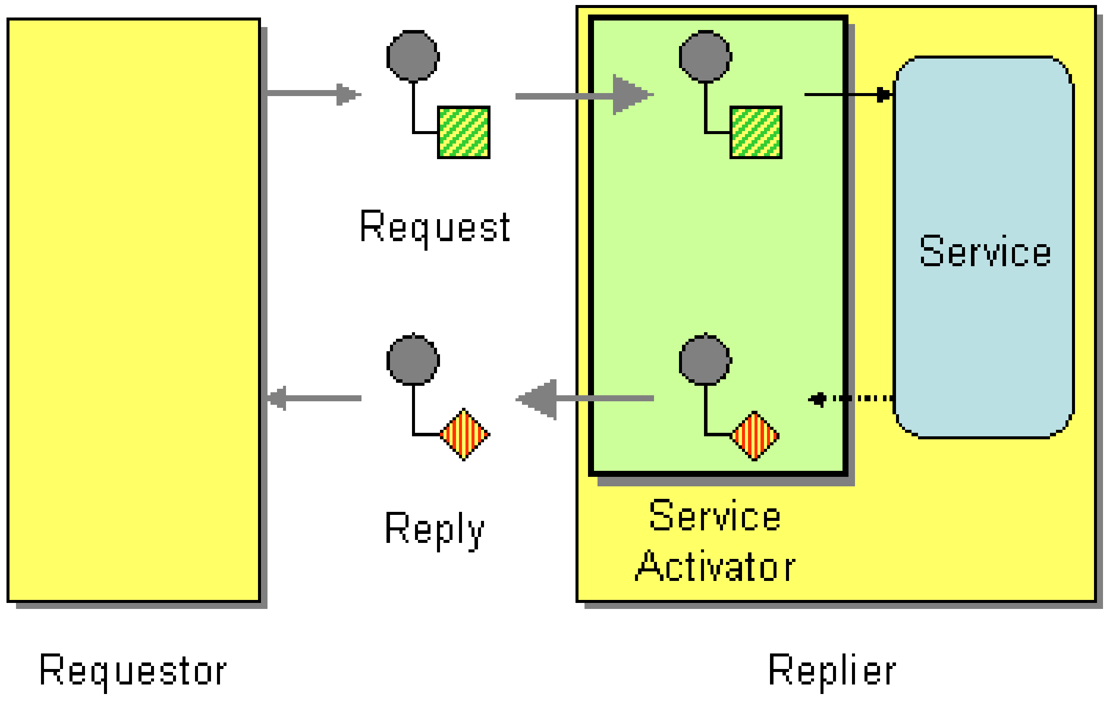


Event driven consumer



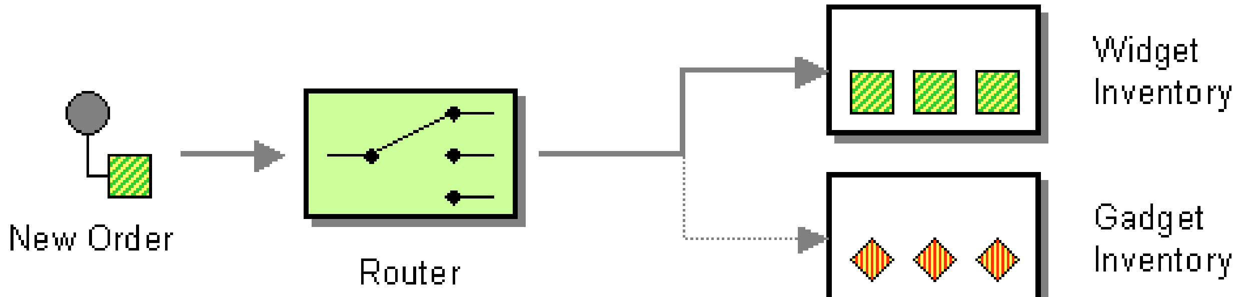
Polling consumer

# Message Endpoint



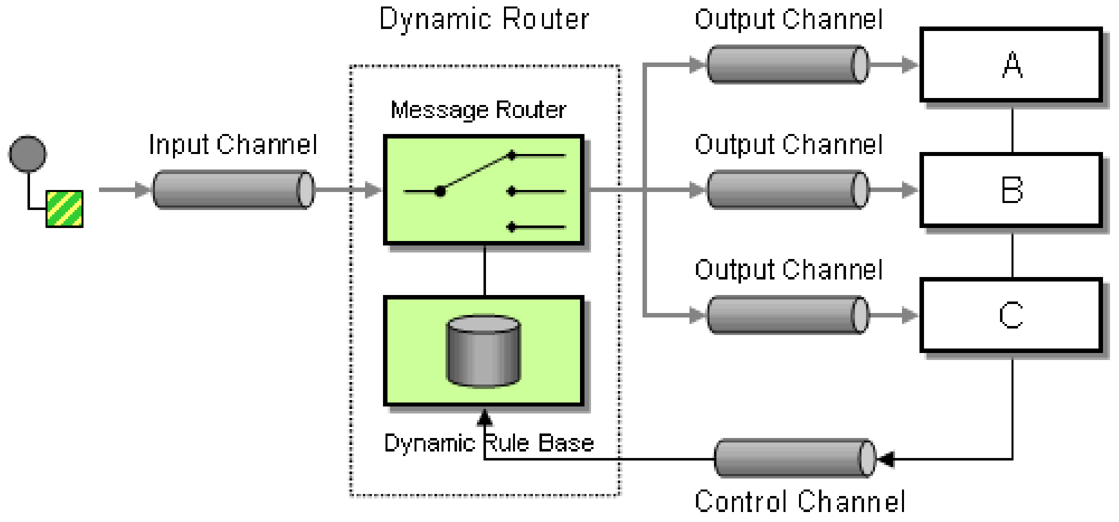
Service activator

# Message Routing

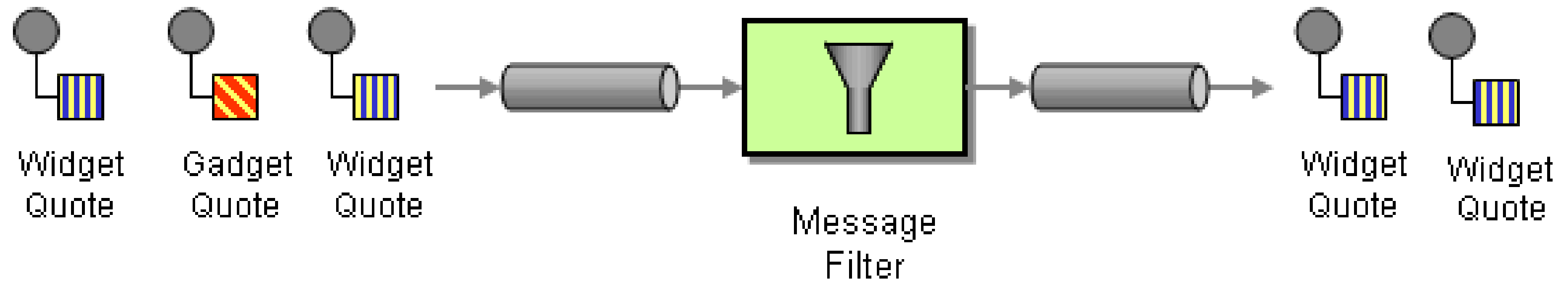


Content based router

Dynamic router

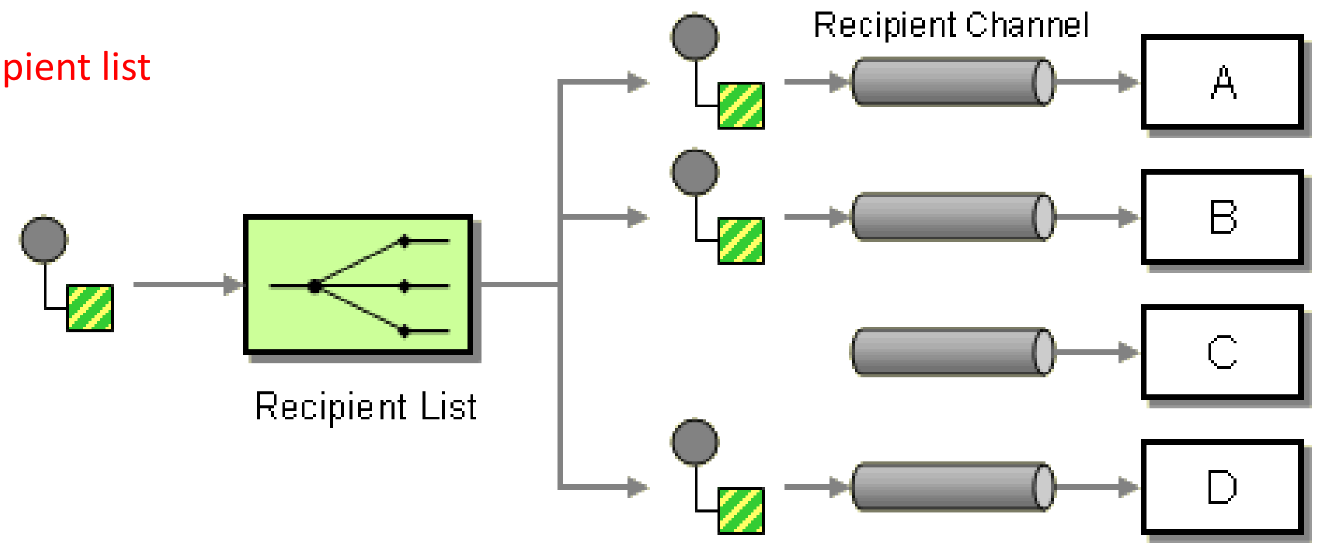


# Message Routing

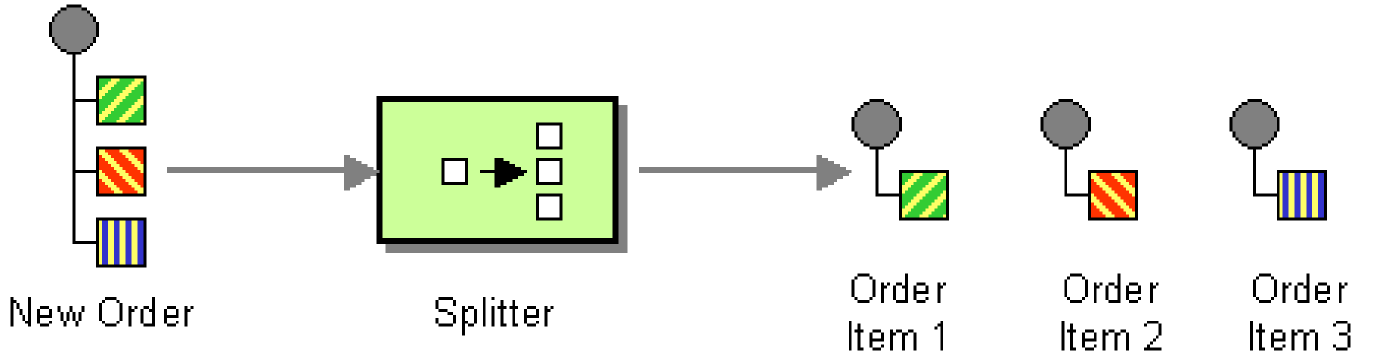


Message filter

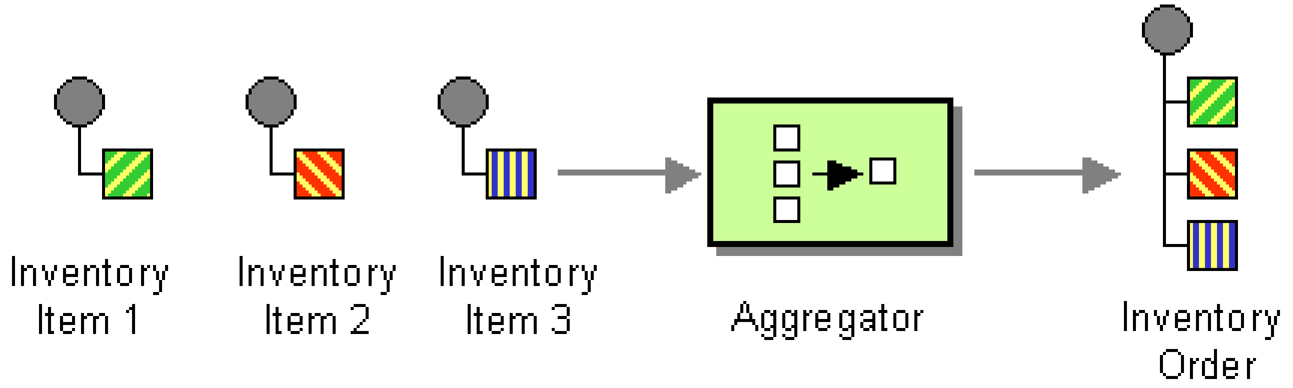
Recipient list



# Message Routing

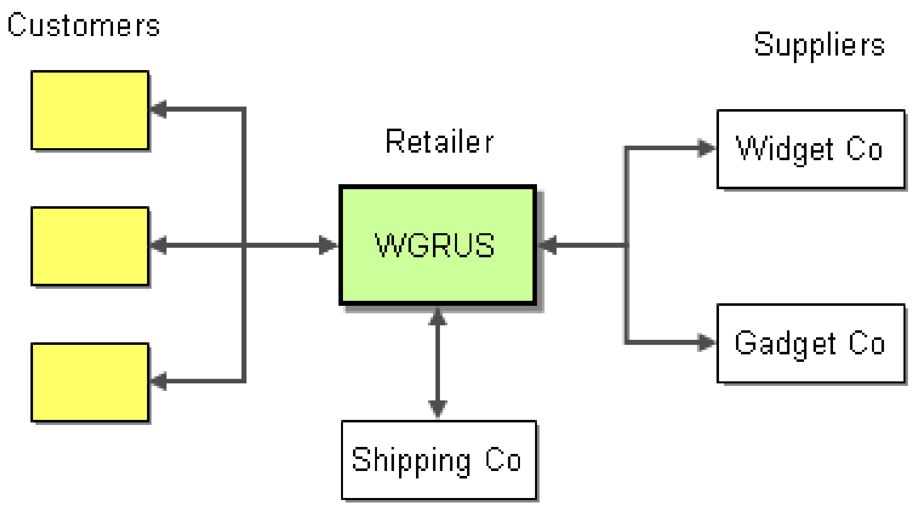


Splitter



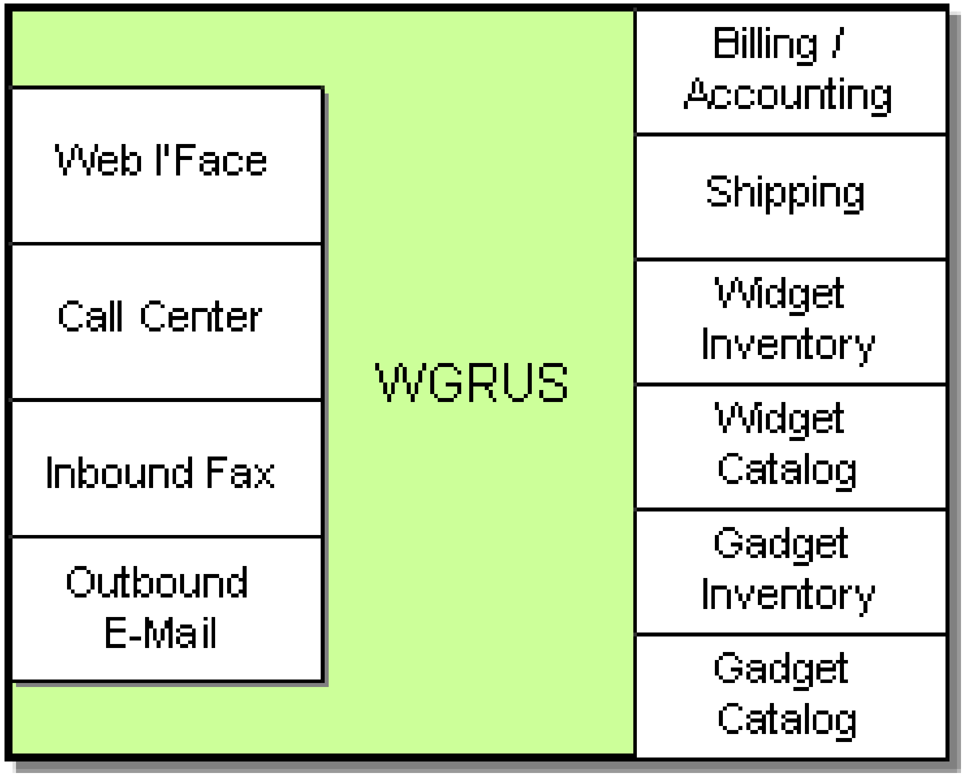
Aggregator

# Example: Widgets&Gatchets 'R Us (WGRUS)

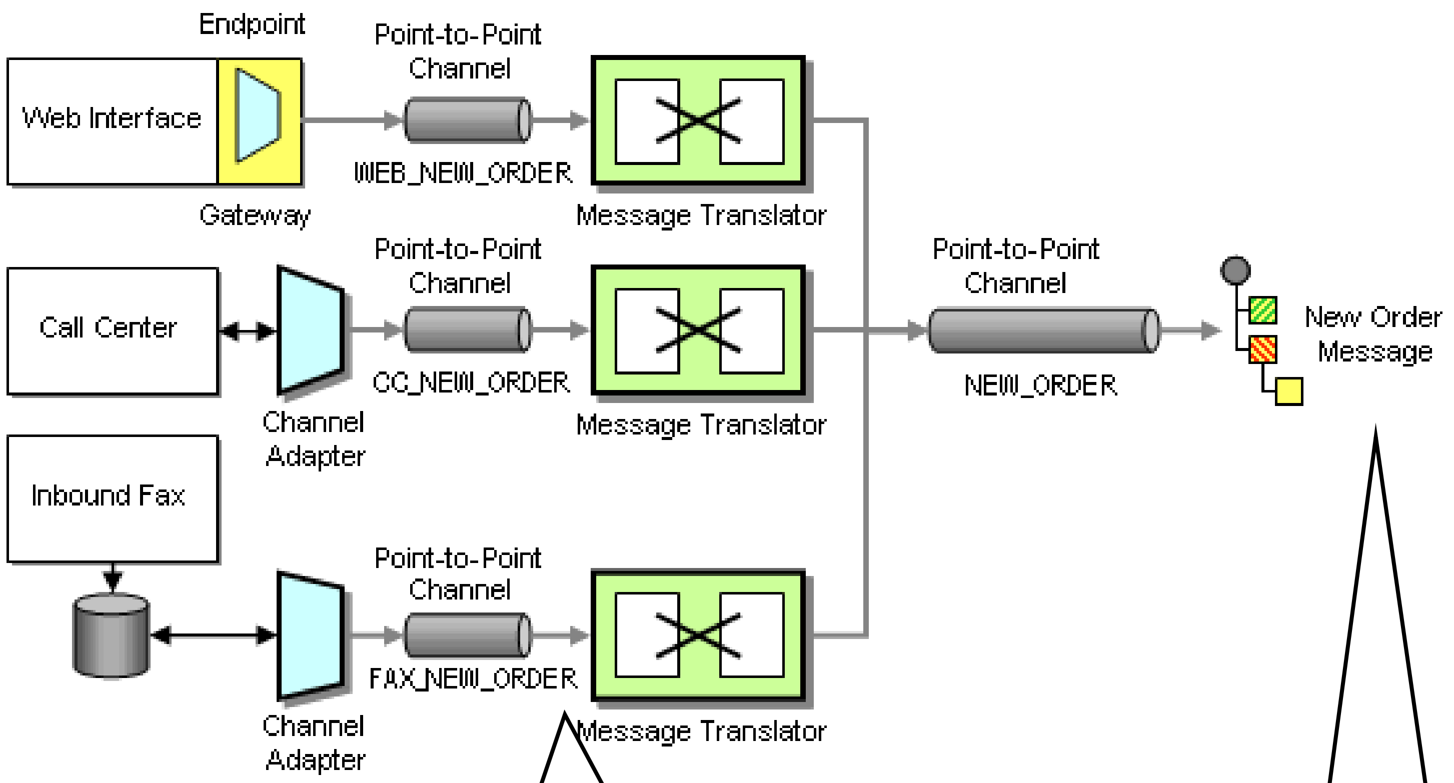


- **Take Orders**: Customers can place orders via Web, phone or fax
- **Process Orders**: Processing an order involves multiple steps, including verifying inventory, shipping the goods and invoicing the customer
- **Check Status**: Customers can check the order status
- **Change Address**: Customers can use a Web front-end to change their billing and shipping address
- **New Catalog**: The suppliers update their catalog periodically. WGRUS needs to update its pricing and availability based in the new catalogs.
- **Announcements**: Customers can subscribe to selective announcements from WGRUS.
- **Testing and Monitoring**: The operations staff needs to be able to monitor all individual components and the message flow between them.

# WGRUS internal IT infrastructure



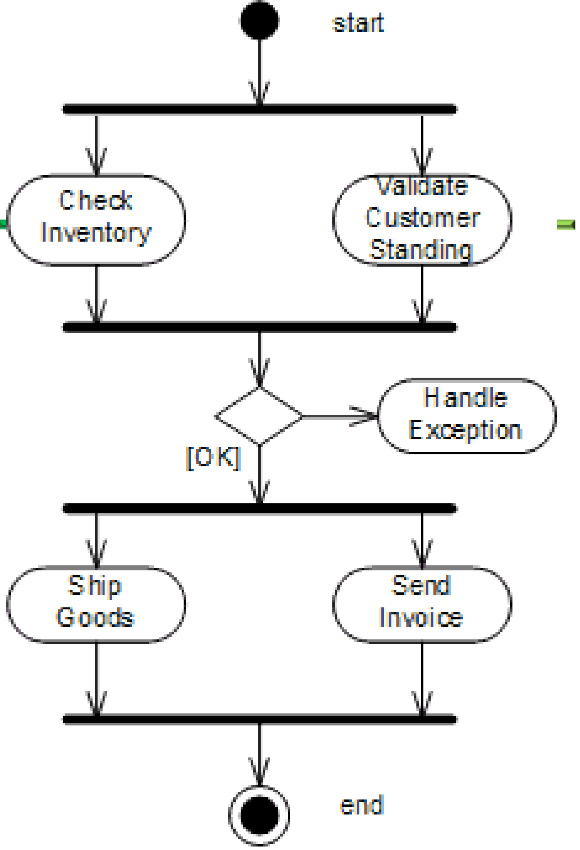
# Taking orders from 3 different channels

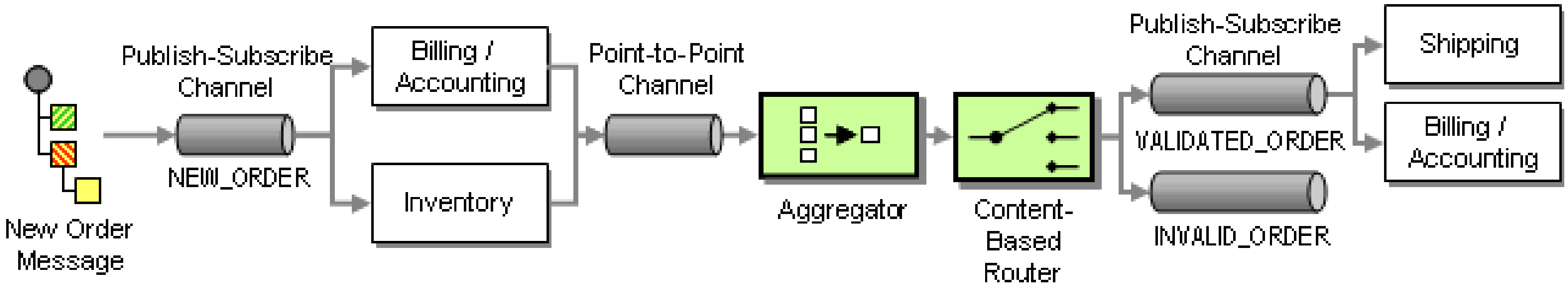


Application specific messages

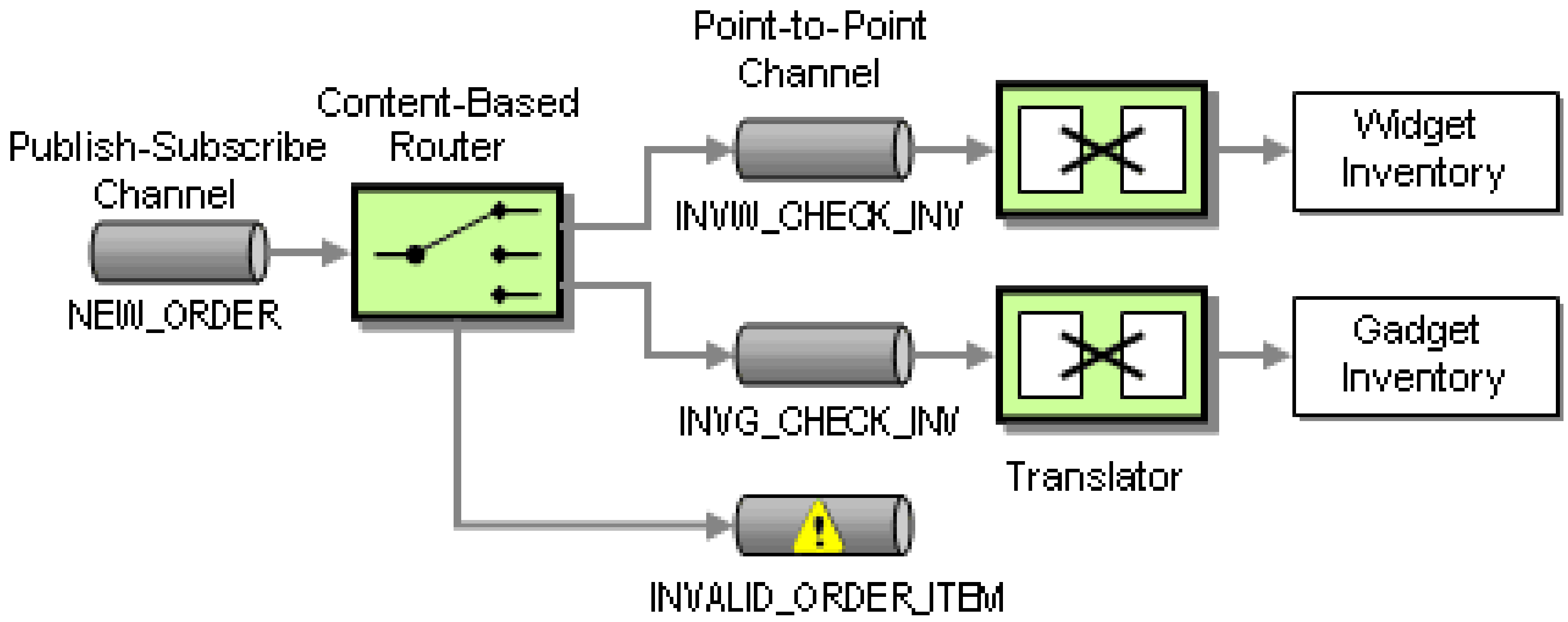
Canonical Data Model

# Order Processi



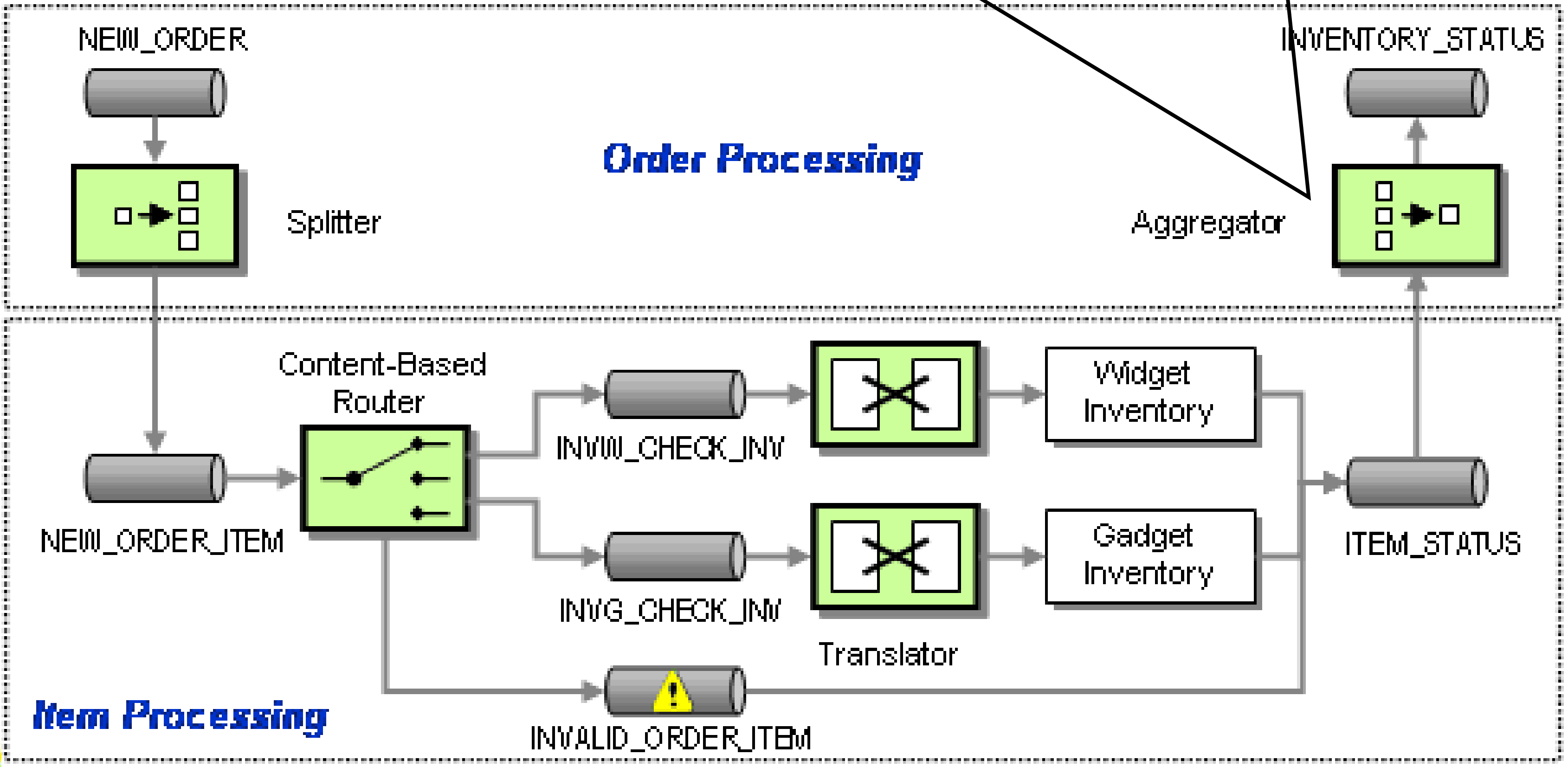


# Routing the inventory request

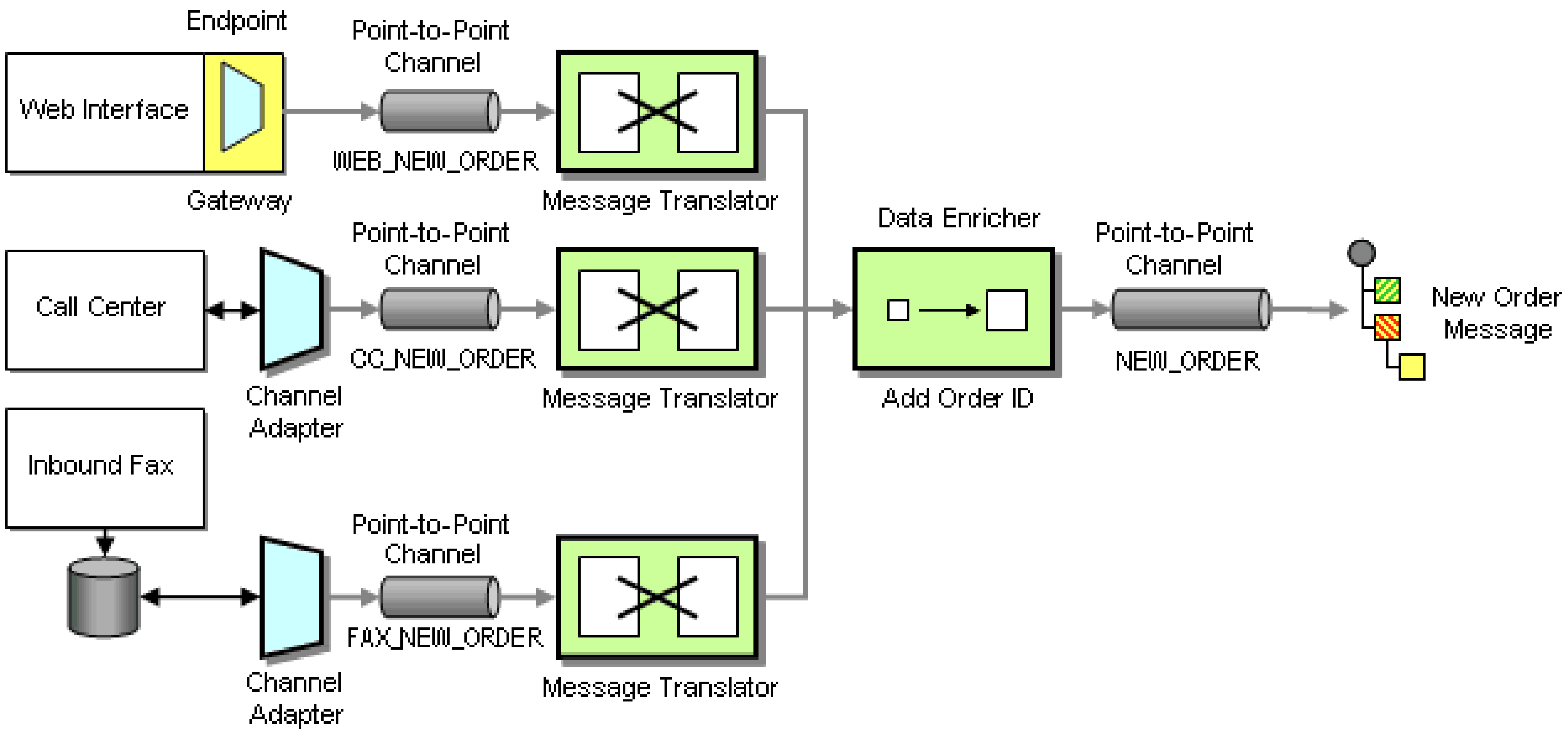


# Orders can contain multiple items

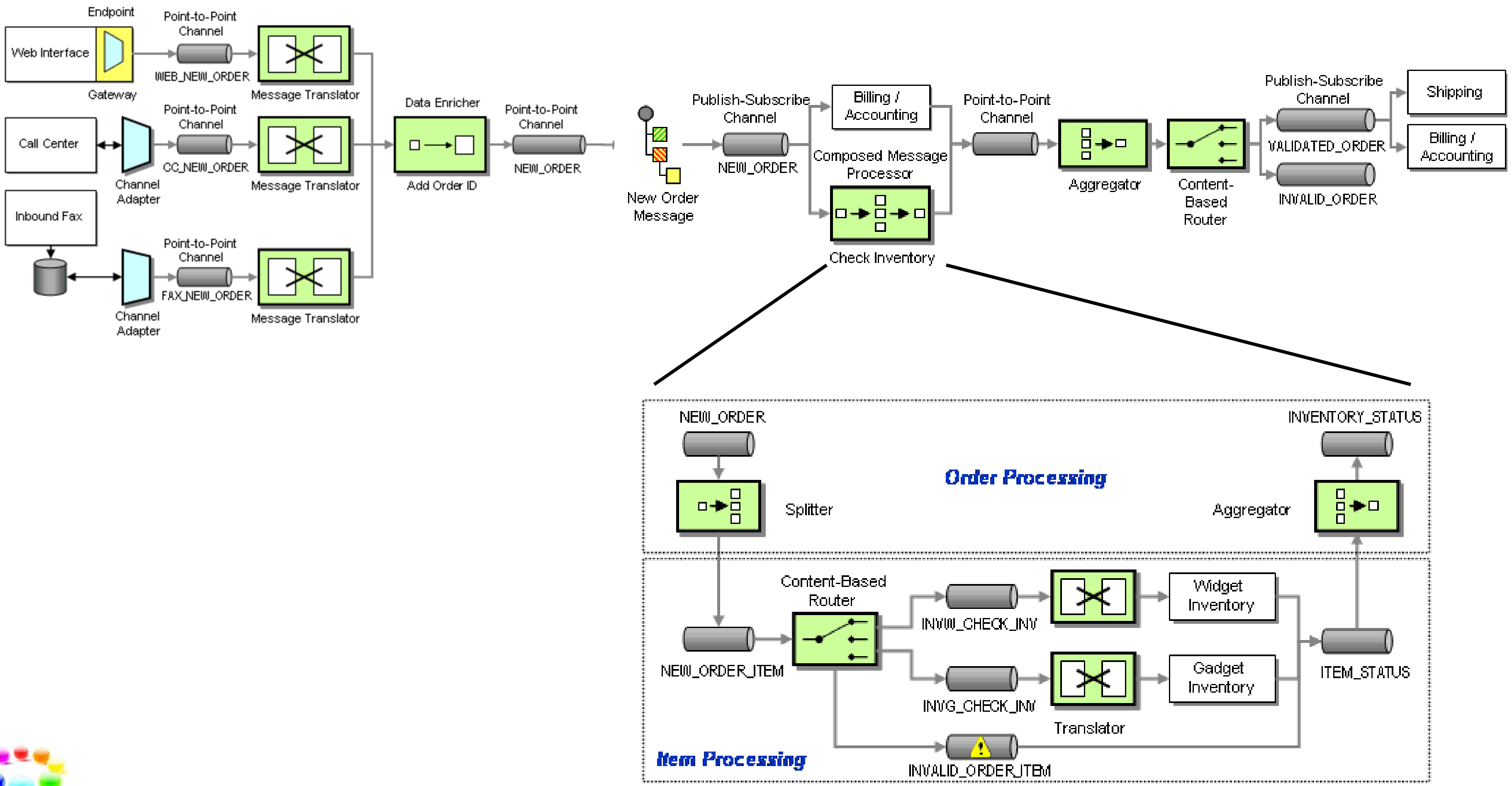
1. Correlation: which messages belong together? We need an unique order ID
2. Completeness: how do we know that all messages are received? Count
3. Aggregation algorithm: how do we combine the individual messages into one result message? Append based on order ID



# Add an unique order ID



# Result so far



## Main point

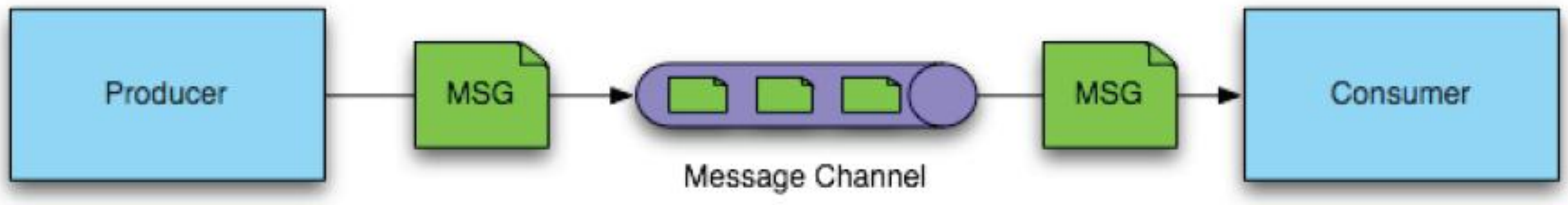
- There are many different integration patterns that you can use to implement the integration logic between components and systems.

- The unified field is the field of all possibilities.

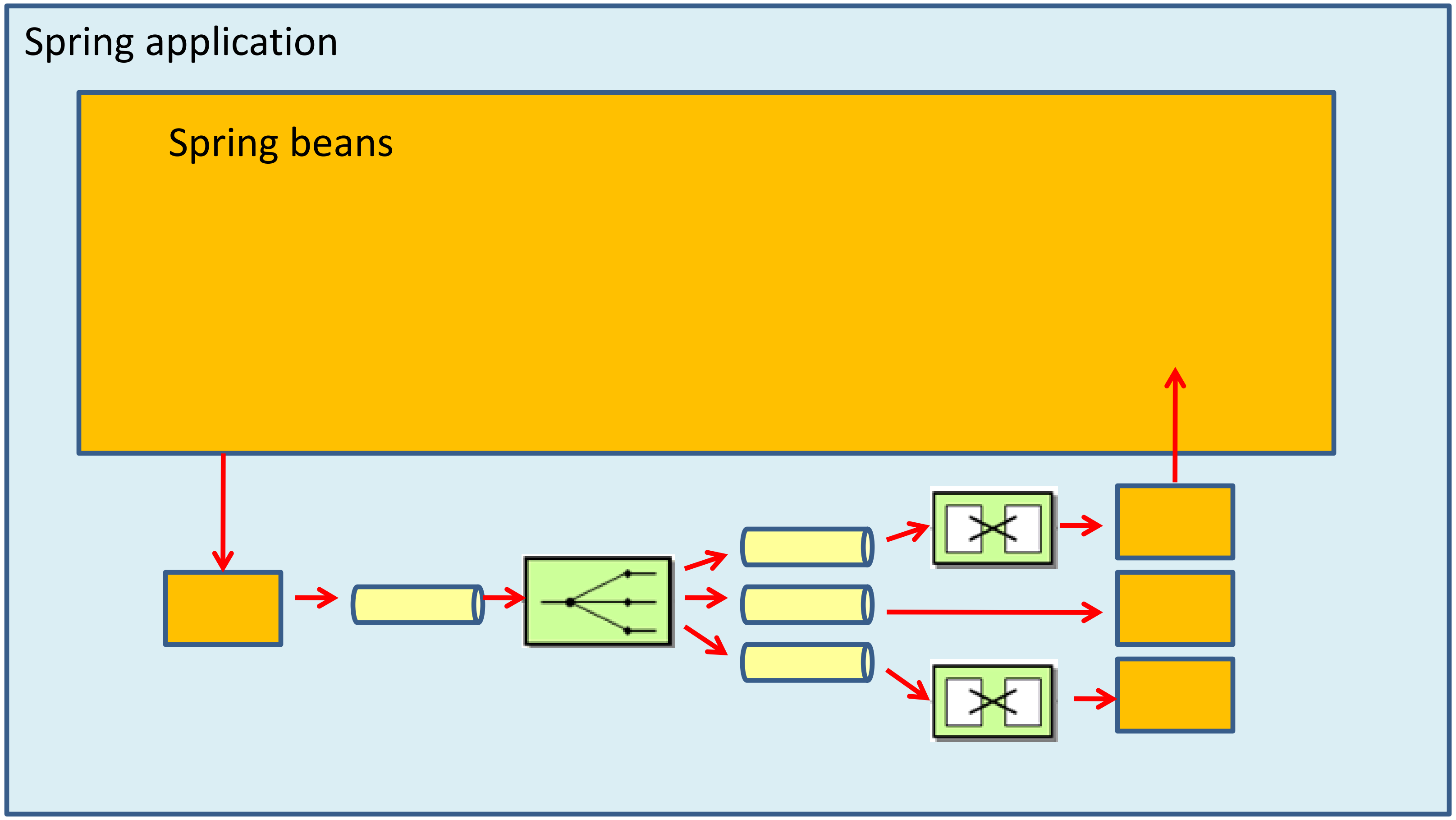
## SPRING INTEGRATION

# What is Spring Integration?

- Provides a simple model to implement complex enterprise integration solutions
- Facilitate asynchronous, parallel, message-driven behavior within a Spring-based application

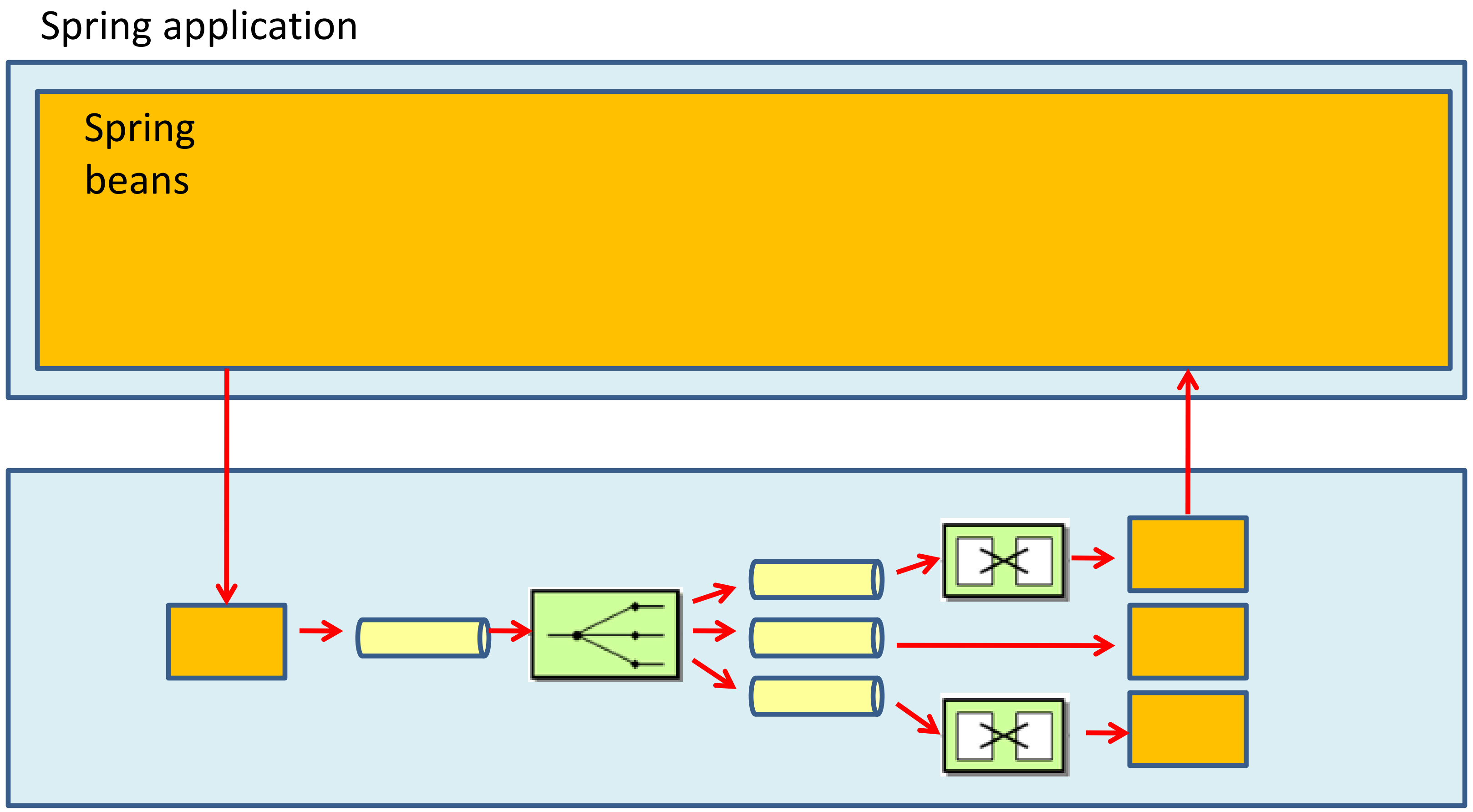


# Using Spring Integration



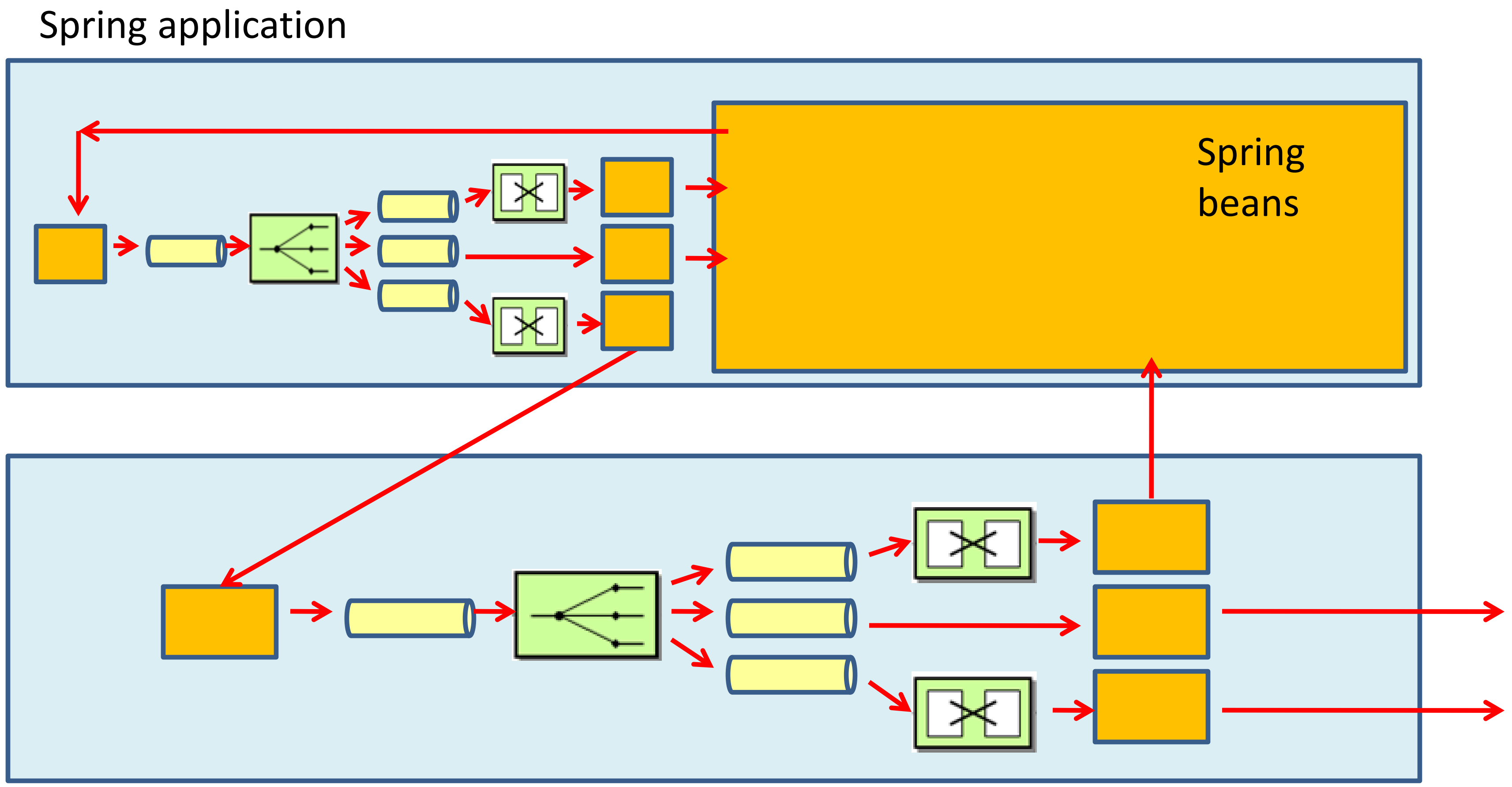
- Use SI inside your application

# Using Spring Integration



- Use SI outside your application

# Using Spring Integration

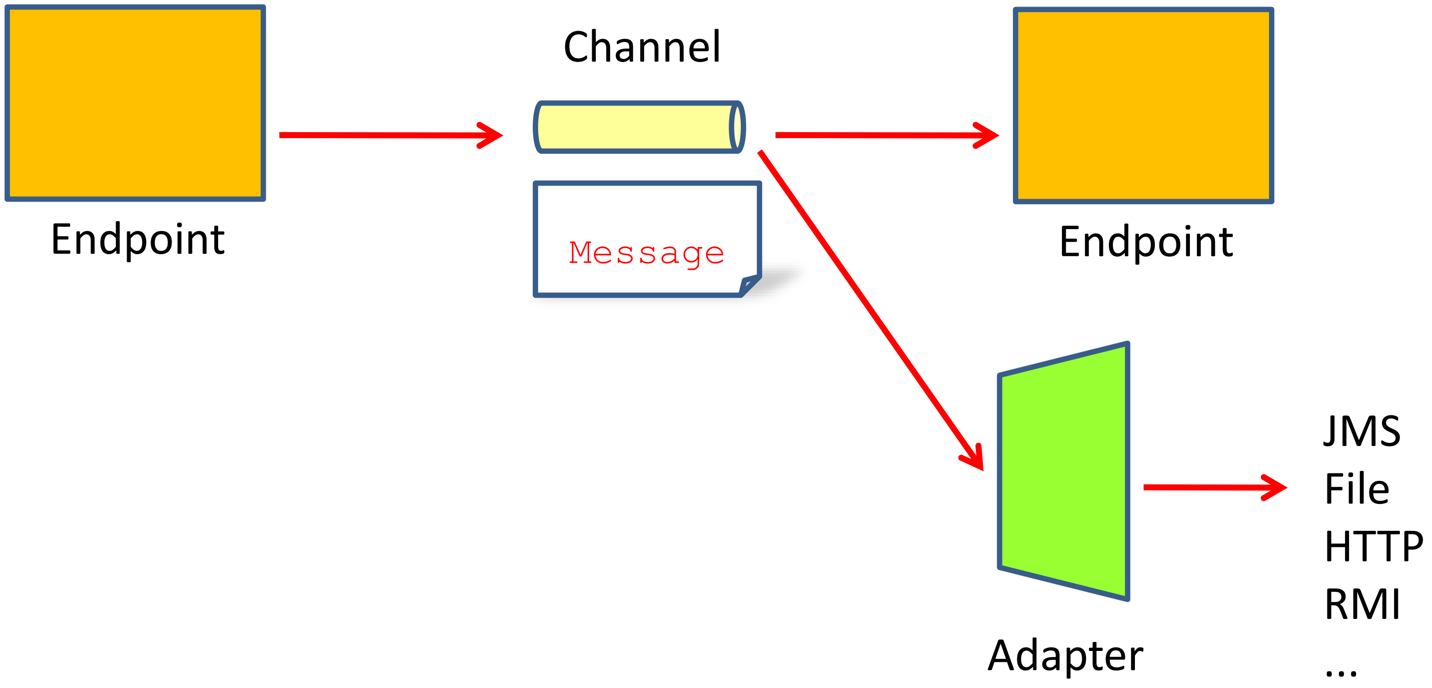


- Use SI inside and outside your application

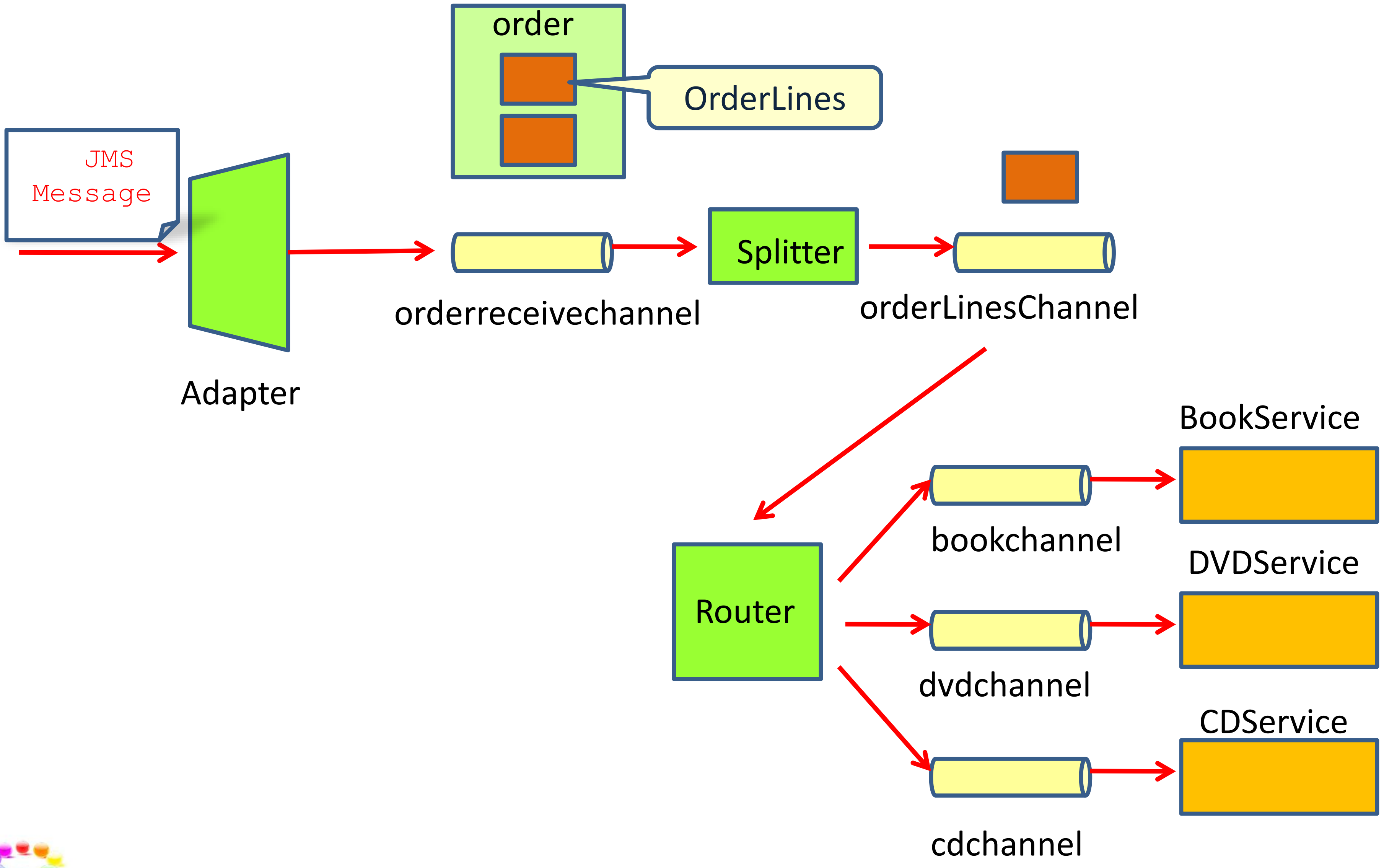
# Difference with an ESB

- ESB’s run within its own VM
  - Spring Integration can run within an application
- You have to install ESB’s
  - Spring integration is a library
- You have to start (and stop) ESB

# Basic components



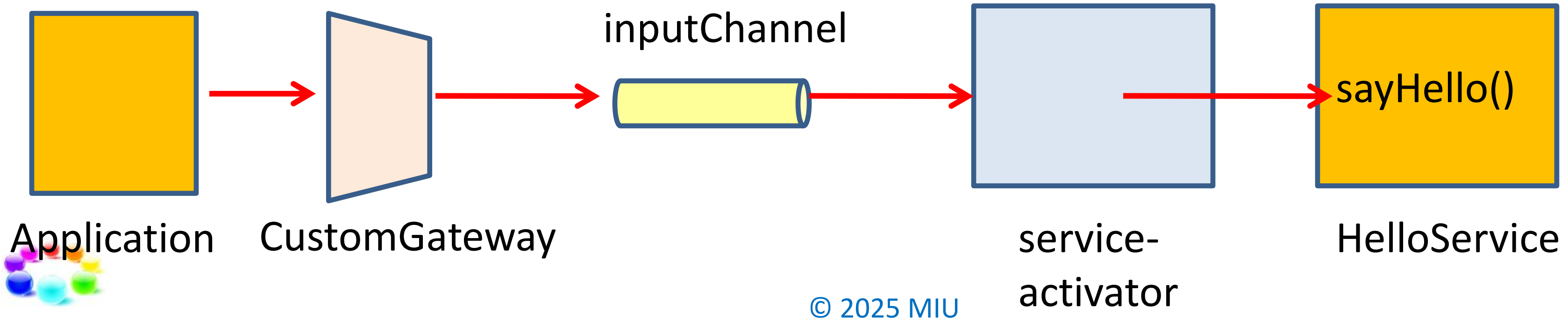
# Spring Integration example



# Spring integration Hello World

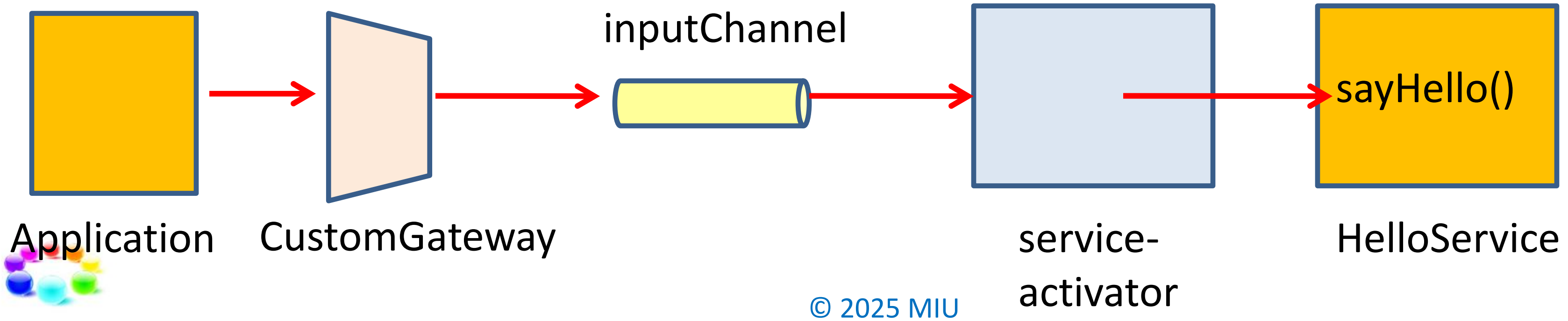
```
public class HelloService {
  public void sayHello(String name){
    System.out.println("Hello "+name);
  }
}
```

```
public interface CustomGateway {

  public void process(String message);
}
```



# Integration-context.xml

```xml
<int:gateway service-interface="integration.CustomGateway"
             default-request-channel="inputChannel">
  <int:method name="process" />
</int:gateway>

<int:channel id="inputChannel" />

<int:service-activator
   input-channel="inputChannel" ref="helloService" method="sayHello" />

<bean id="helloService" class="integration.HelloService" />
```

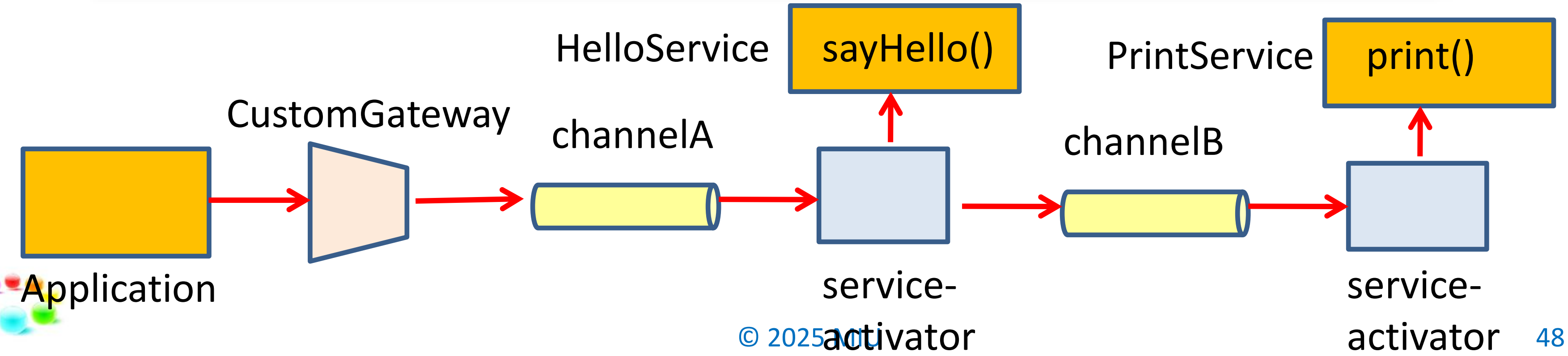


# The application

```java
@SpringBootApplication
@ImportResource("integration-context.xml")
public class SpringIntegrationProjectApplication implements CommandLineRunner {

  @Autowired
  private CustomGateway gateway;


  public static void main(String[] args) {
   SpringApplication.run(SpringIntegrationProjectApplication.class, args);
  }

  @Override
  public void run(String... args) throws Exception {
    gateway.process("World");
  }
}
```

# Extending the application

```
<int:gateway service-interface="integration.CustomGateway"
  default-request-channel=" channelA">
  <int:method name="process" />
</int:gateway>

<channel id="channelA"/>
<channel id="channelB"/>

<service-activator input-channel="channelA"
                   output-channel="channelB"
                   ref="helloService"
                   method="sayHello"/>
<service-activator input-channel="channelB"
                   ref="printService"
                   method="print"/>

<beans:bean id="helloService" class="integration.HelloService"/>
<beans:bean id="printService" class="integration.PrintService"/>
```



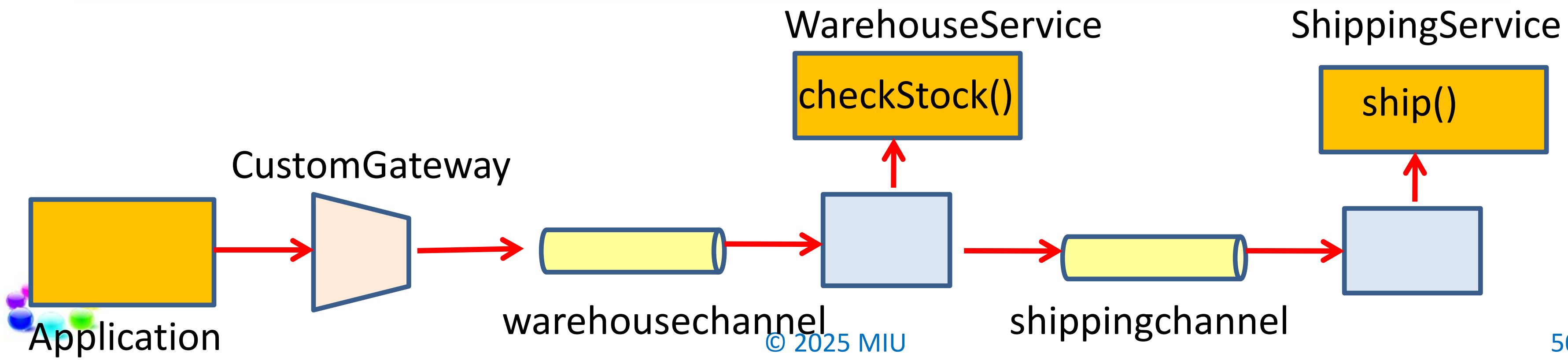
# Extending the application

```
public class HelloService {

  public String sayHello(String name) {
    System.out.println("HelloService: receiving name "+name);
    return "Hello "+ name;
  }
}
```

```
public class PrintService {

  public void print(String message) {
   System.out.println("Printing message: "+ message);
  }
}
```

# Sending an Order

```
<int:gateway service-interface="integration.CustomGateway"
  default-request-channel="warehousechannel">
  <int:method name="process" />
</int:gateway>

<int:channel id="warehousechannel" />
<int:channel id="shippingchannel" />

<int:service-activator
  input-channel="warehousechannel" output-channel="shippingchannel"
  ref="warehouseservice" method="checkStock" />

<int:service-activator
  input-channel="shippingchannel" ref="shippingservice" method="ship" />

<bean id="warehouseservice" class="integration.WarehouseService" />
<bean id="shippingservice" class="integration.ShippingService" />
```



# The services

```
public class WarehouseService {

  public Order checkStock(Order order) {
    System.out.println("WarehouseService: checking order "+order.toString());
    return order;
  }
}
```

```
public class ShippingService {
  public void ship(Order order) {
    System.out.println("shipping: "+ order.toString());
  }
}
```

```
public class Order {
  private String orderNumber;
  private double amount;

  public String toString(){
    return "order: nr="+orderNumber+" amount="+amount;
  }
...
}
```

# The application

```
@SpringBootApplication
@ImportResource("integration-context.xml")
public class SpringIntegrationProjectApplication implements CommandLineRunner {

   @Autowired
   private CustomGateway gateway;


   public static void main(String[] args) {
    SpringApplication.run(SpringIntegrationProjectApplication.class, args);
   }

   @Override
   public void run(String... args) throws Exception {
     Order order = new Order("H-234-X56",1245.75);
     Message<Order> message = MessageBuilder.withPayload(order).build();
     gateway.process(message);
   }
}
```

```
WarehouseService: checking order order: nr=H-234-X56 amount=1245.75
shipping: order: nr=H-234-X56 amount=1245.75
```

## MESSAGE CHANNELS

# Direct Channel

```
<channel id="inputChannel"/>
```

Default point-to-point channel

- Point-to-point
- When there are multiple handlers subscribed to the same channel
  - A “round-robin” loadbalancer balances the messages
  - The loadbalancer will automatically send the message to a subsequent handler if the preceding handler throws an exception (failover)

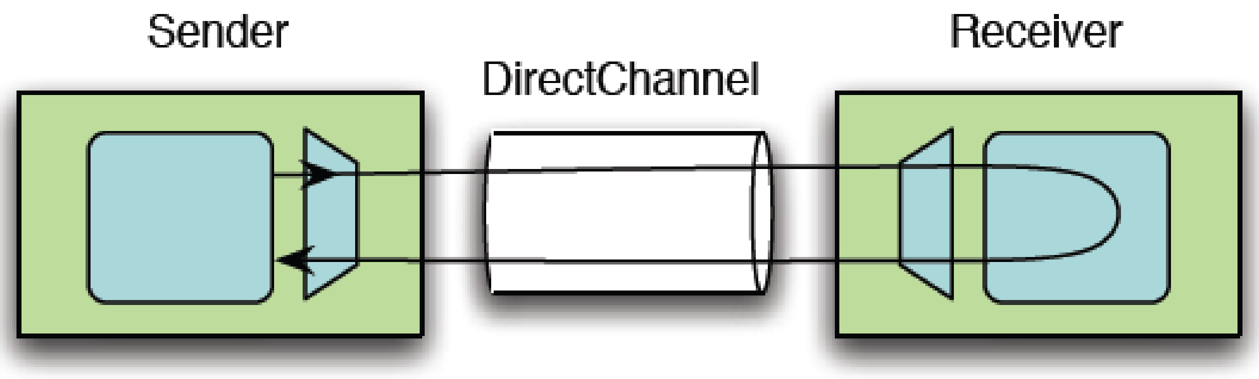
```
<channel id="failFastChannel">
    <dispatcher failover="false"/>
</channel>

<channel id="channelWithFixedOrderSequenceFailover">
    <dispatcher load-balancer="none"/>
</channel>
```

No failover

No round-robin balancer

# Synchronous

- A direct default channel is synchronous

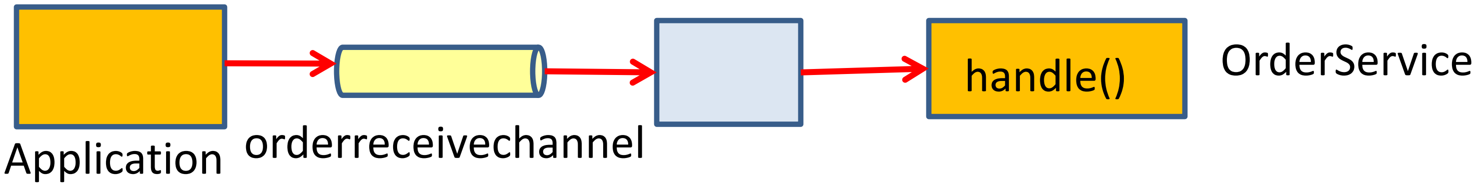


# Synchronous

```xml
<channel id="orderreceivechannel" />

<service-activator input-channel="orderreceivechannel"
                   ref="orderservice" method="handle" />
<beans:bean id="orderservice" class="integration.OrderService" />
```

```java
public class OrderService {
  public void handle(Order order) throws Exception {
    System.out.println("OrderService receiving order: "+ order.toString());
    Thread.sleep(5000);
  }
}
```

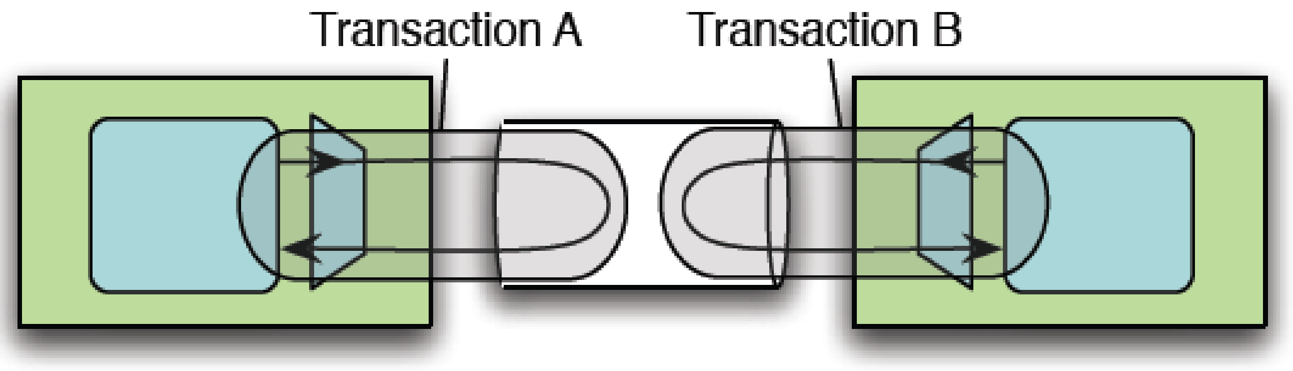
Sleep 5 seconds



```
time before sending message =8:54:15
OrderService receiving order: order: nr=H-234-X56 amount=1245.75
time after sending message =8:54:20
```

# QueueChannel: Asynchronous

- A queue channel is asynchronous



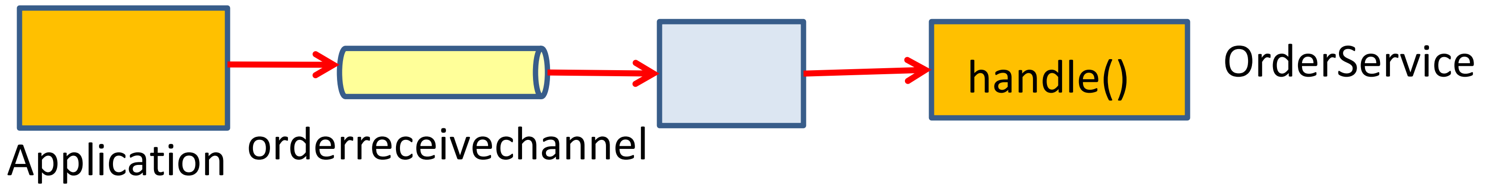
# QueueChannel

```
<channel id="orderreceivechannel" >
  <queue capacity="25"/>
</channel>

<service-activator input-channel="orderreceivechannel" ref="orderservice"
                   method="handle" >
  <poller>
    <interval-trigger interval="200"/>
  </poller>
</service-activator>

<beans:bean id="orderservice" class="integration.OrderService" />
```

Add a queue

Now we need a poller

```
time before sending message =9:22:30
time after sending message =9:22:30
OrderService receiving order: order: nr=H-234-X56 amount=1245.75
```



# Datatype channel

```
<channel id="numberChannel" datatype="java.lang.Number"/>
```

Datatype Channel that only accepts messages containing a certain payload type

```
<channel id="stringOrNumberChannel"
         datatype="java.lang.String,java.lang.Number"/>
```

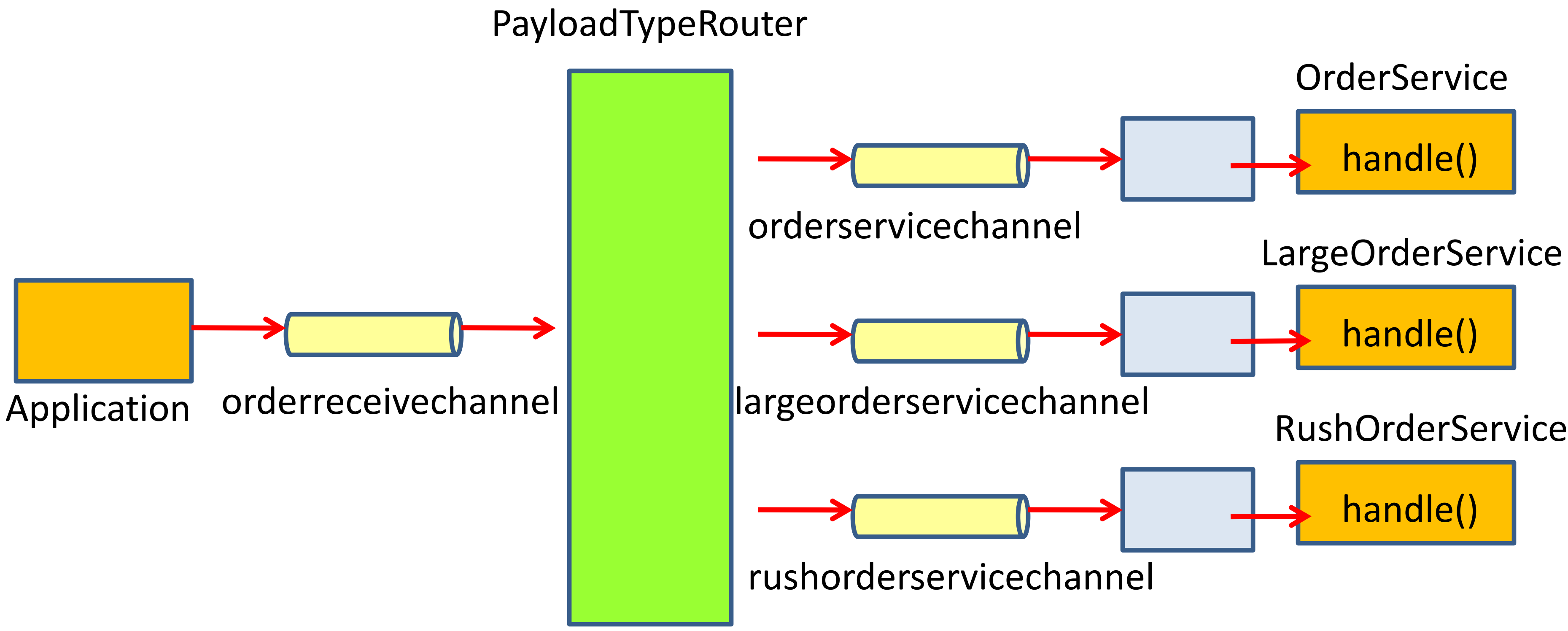
Accept multiple types

## ROUTER

# Routers

- Build-in routers
  - PayloadTypeRouter
  - HeaderValueRouter
  - RecipientListRouter
- Custom router

# PayloadTypeRouter



# PayloadTypeRouter

```xml
<channel id="orderreceivechannel" />
<channel id="orderservicechannel" />
<channel id="rushorderservicechannel" />
<channel id="largeorderservicechannel" />

<payload-type-router input-channel="orderreceivechannel">
  <mapping type="integration.Order" channel="orderservicechannel" />
  <mapping type="integration.RushOrder" channel="rushorderservicechannel" />
  <mapping type="integration.LargeOrder" channel="largeorderservicechannel" />
</payload-type-router>

<service-activator input-channel="orderservicechannel"
                   ref="orderservice" method="handle" />

<service-activator input-channel="rushorderservicechannel"
                   ref="rushorderservice" method="handle" />

<service-activator input-channel="largeorderservicechannel"
                   ref="largeorderservice" method="handle" />

<beans:bean id="orderservice" class="integration.OrderService" />
<beans:bean id="rushorderservice" class="integration.RushOrderService" />
<beans:bean id="largeorderservice" class="integration.LargeOrderService" />
```

# The Payload types

```java
public class Order {
   private String orderNumber;
   private double amount;

   public String toString(){
      return "order: nr="+orderNumber+" amount="+amount;
   }
   ...
}
```

```java
public class RushOrder extends Order{
   public RushOrder(String orderNumber, double amount) {
      super(orderNumber, amount);
   }
}
```

```java
public class LargeOrder extends Order{
   public LargeOrder(String orderNumber, double amount) {
      super(orderNumber, amount);
   }
}
```

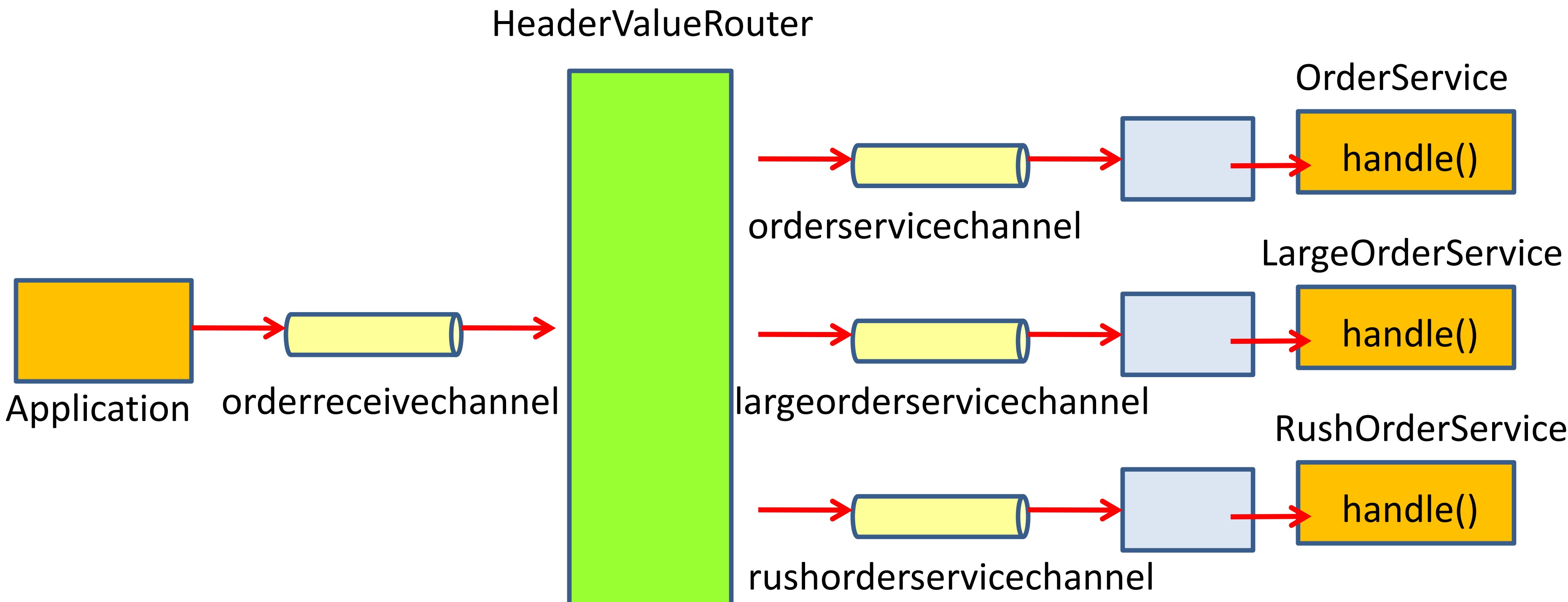
## The services

```
public class OrderService {
  public void handle(Order order) {
    System.out.println("OrderService receiving order: "+ order.toString());
  }
}
```

```
public class LargeOrderService {
  public void handle(Order order) {
    System.out.println("LargeOrderService receiving order: "+ order.toString());
  }
}
```

```
public class RushOrderService {
  public void handle(Order order) {
    System.out.println("RushOrderService receiving order: "+ order.toString());
  }
}
```

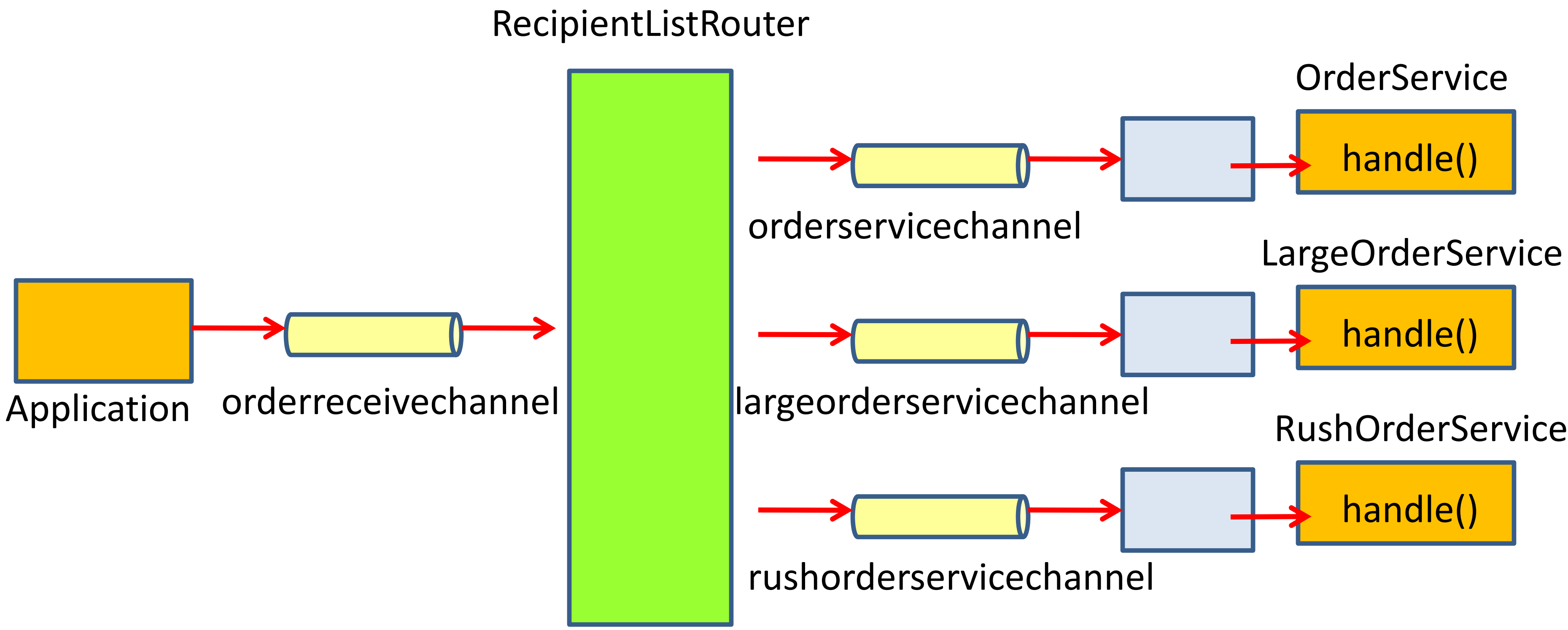
# HeaderValueRouter



# HeaderValueRouter

```xml
<channel id="orderreceivechannel" />
<channel id="orderservicechannel" />
<channel id="rushorderservicechannel" />
<channel id="largeorderservicechannel" />

<header-value-router input-channel="orderreceivechannel"
                     header-name="orderType">
  <mapping value="normal" channel="orderservicechannel" />
  <mapping value="rush" channel="rushorderservicechannel" />
  <mapping value="large" channel="largeorderservicechannel" />
</header-value-router>

<service-activator input-channel="orderservicechannel"
                   ref="orderservice" method="handle" />

<service-activator input-channel="rushorderservicechannel"
                   ref="rushorderservice" method="handle" />

<service-activator input-channel="largeorderservicechannel"
                   ref="largeorderservice" method="handle" />

<beans:bean id="orderservice" class="integration.OrderService" />
<beans:bean id="rushorderservice" class="integration.RushOrderService" />
<beans:bean id="largeorderservice" class="integration.LargeOrderService" />
```

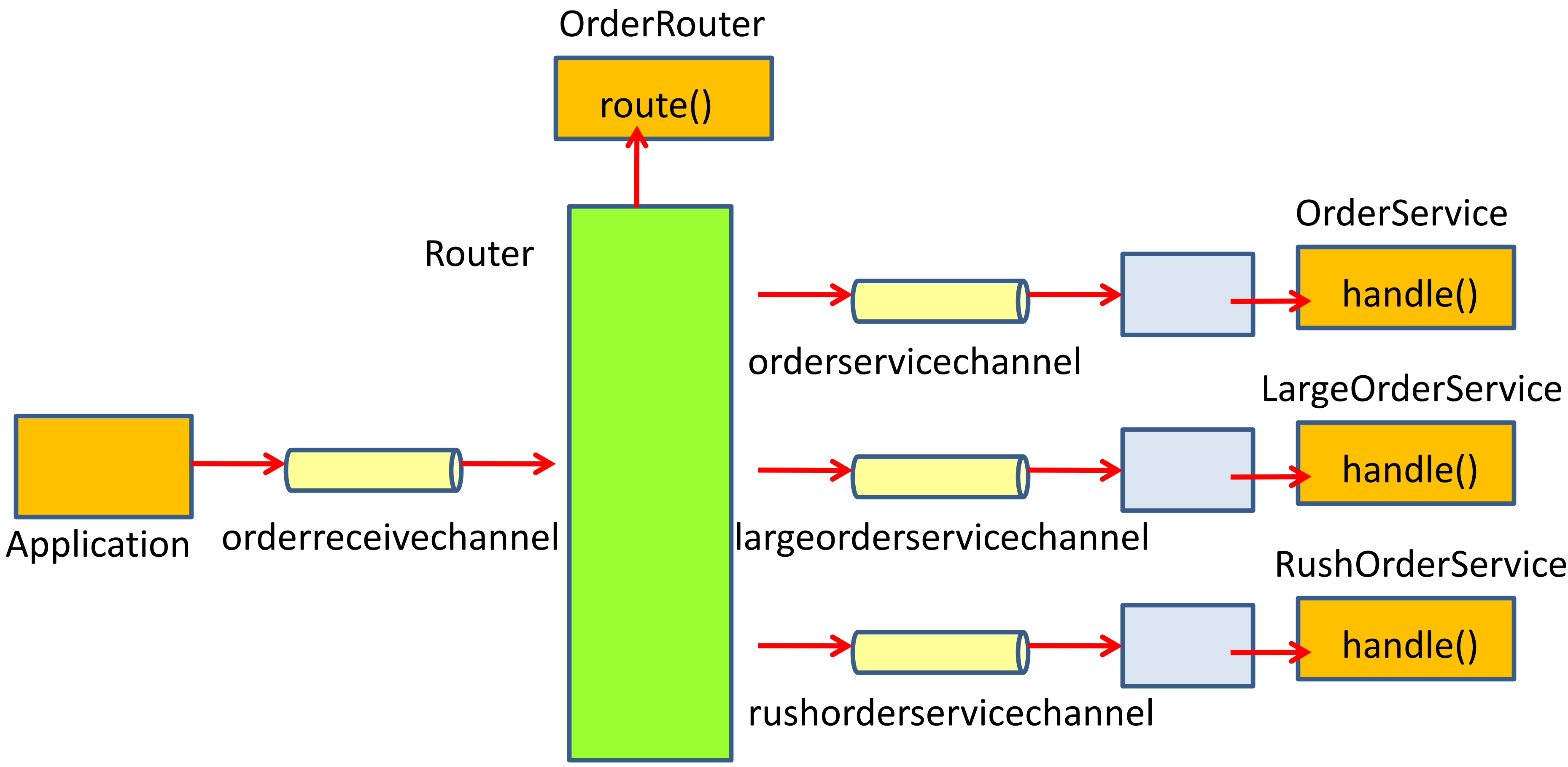
# RecipientListRouter



# RecipientListRouter

```
<channel id="orderreceivechannel" />
<channel id="orderservicechannel" />
<channel id="rushorderservicechannel" />
<channel id="largeorderservicechannel" />

<recipient-list-router id="customRouter" input-channel="orderreceivechannel"
                   apply-sequence="true">
  <recipient channel="orderservicechannel" />
  <recipient channel="rushorderservicechannel" />
  <recipient channel="largeorderservicechannel" />
</recipient-list-router>

<service-activator input-channel="orderservicechannel"
ref="orderservice" method="handle" />

<service-activator input-channel="rushorderservicechannel"
ref="rushorderservice" method="handle" />

<service-activator input-channel="largeorderservicechannel"
ref="largeorderservice" method="handle" />

<beans:bean id="orderservice" class="integration.OrderService" />
<beans:bean id="rushorderservice" class="integration.RushOrderService" />
<beans:bean id="largeorderservice" class="integration.LargeOrderService" />
```

# Custom Router bean



# Custom Router bean

```xml
<channel id="orderreceivechannel" />
<channel id="orderservicechannel" />
<channel id="rushorderservicechannel" />
<channel id="largeorderservicechannel" />

<router method="route" input-channel="orderreceivechannel">
  <beans:bean class="integration.OrderRouter" />
</router>

<service-activator input-channel="orderservicechannel"
ref="orderservice" method="handle" />

<service-activator input-channel="rushorderservicechannel"
ref="rushorderservice" method="handle" />

<service-activator input-channel="largeorderservicechannel"
ref="largeorderservice" method="handle" />

<beans:bean id="orderservice" class="integration.OrderService" />
<beans:bean id="rushorderservice" class="integration.RushOrderService" />
<beans:bean id="largeorderservice" class="integration.LargeOrderService" />
```

# The router bean

```
public class OrderRouter {
  public String route(Order order) {
    String destinationChannel = null;
    if (order.isRush())
      destinationChannel = "rushorderservicechannel";
    else if (order.getAmount() > 20000)
      destinationChannel = "largeorderservicechannel";
    else
      destinationChannel = "orderservicechannel";
    return destinationChannel;
  }
}
```

```
RushOrderService receiving order: order: nr=H-234-X56 amount=1245.75
OrderService receiving order: order: nr=H-234-X57 amount=600.65
LargeOrderService receiving order: order: nr=H-234-X58 amount=50600.65
```

## The router bean: multiple return values

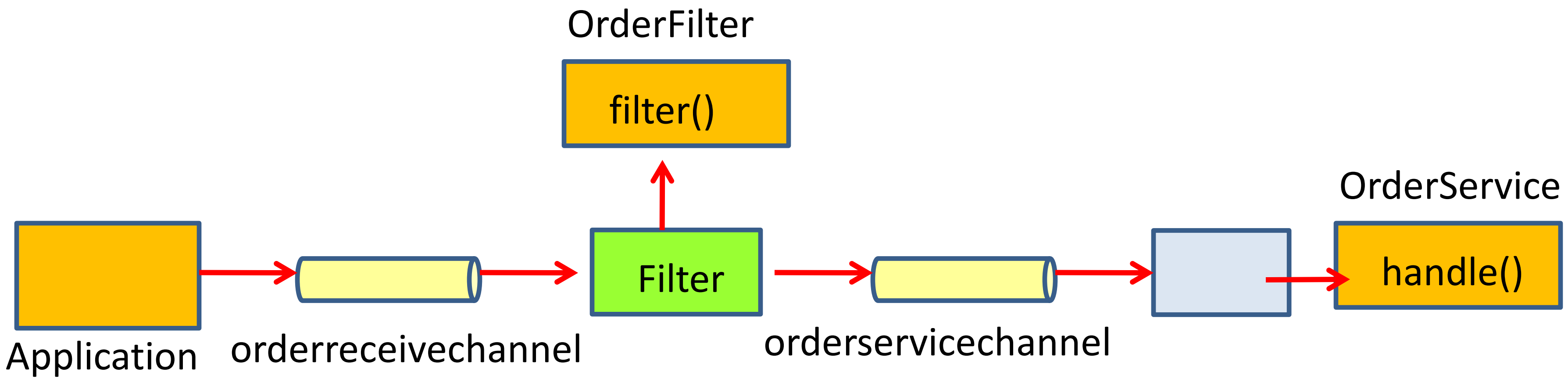
```java
public class OrderRouter {
  public List<String> route(Order order) {
    List<String> destinationChannels = new ArrayList<String>();
    if (order.isRush())
      destinationChannels.add("rushorderservicechannel");
    if (order.getAmount() > 20000)
      destinationChannels.add("largeorderservicechannel");
    destinationChannels.add("orderservicechannel");
    return destinationChannels;
  }
}
```

```
RushOrderService receiving order: order: nr=H-234-X56 amount=1245.75
OrderService receiving order: order: nr=H-234-X56 amount=1245.75
OrderService receiving order: order: nr=H-234-X57 amount=600.65
RushOrderService receiving order: order: nr=H-234-X58 amount=50600.65
LargeOrderService receiving order: order: nr=H-234-X58 amount=50600.65
OrderService receiving order: order: nr=H-234-X58 amount=50600.65
```

## FILTER

# Filter

```
<channel id="orderreceivechannel" />
<channel id="orderservicechannel" />

<filter input-channel="orderreceivechannel" output-channel="orderservicechannel"
        ref="orderfilter" method="filter"/>

<service-activator input-channel="orderservicechannel"
                   ref="orderservice" method="handle" />

<beans:bean id="orderservice" class="integration.OrderService" />
<beans:bean id="orderfilter" class="integration.OrderFilter" />
```



## The Filter class

```
public class OrderFilter {
  public boolean filter(Order order) {
    if (order.getAmount() > 800)
      return true;
    else
      return false;
  }
}
```

# What to do with rejected messages?

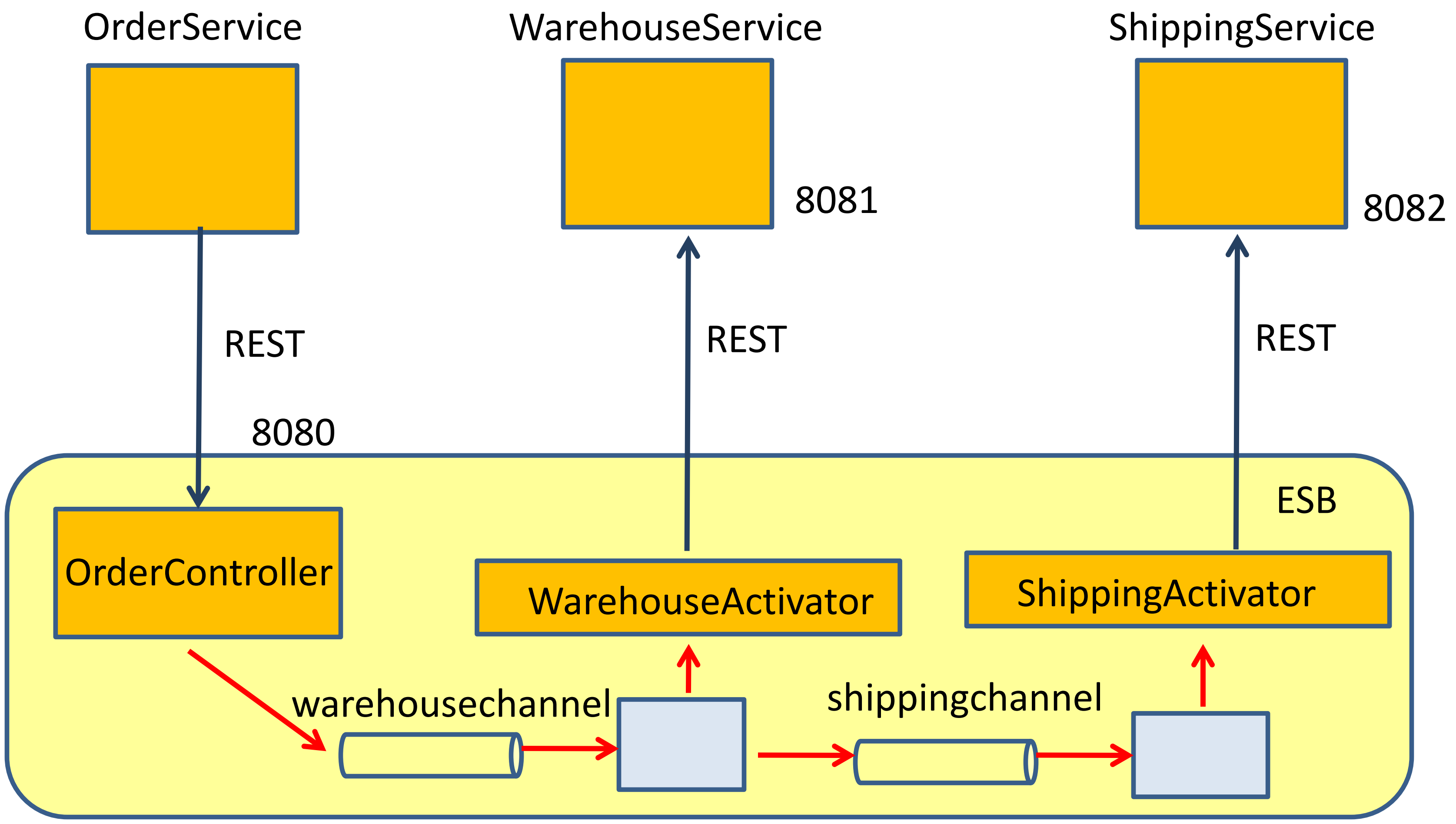
```
<filter input-channel="orderreceivechannel" output-channel="orderservicechannel"
        ref="orderfilter" method="filter" throw-exception-on-rejection="true"/>
```

Throw an exception if a message is rejected

```
<filter input-channel="orderreceivechannel" output-channel="orderservicechannel"
        ref="orderfilter" method="filter" discard-channel="rejectedMessages"/>
```

Send rejected messages to another channel

# ESB with spring integration



# ESB configuration

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/integration"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/integration
      http://www.springframework.org/schema/integration/spring-integration.xsd">

  <channel id="wharehousechannel"/>
  <channel id="shippingchannel"/>

  <service-activator input-channel="wharehousechannel"
                output-channel="shippingchannel"
                ref="warehouseservice"
                method="checkStock"/>

  <service-activator input-channel="shippingchannel"
                ref="shippingservice"
                method="ship"/>

  <beans:bean id="warehouseservice" class="esb.WarehouseActivator"/>
  <beans:bean id="shippingservice" class="esb.ShippingActivator"/>

</beans:beans>
```

# OrderController

```
@RestController
public class OrderController {
   @Autowired
   @Qualifier("wharehousechannel")
   MessageChannel warehouseChannel;

   @PostMapping("/orders")
   public ResponseEntity<?> receiveOrder(@RequestBody Order order) {
      Message<Order> orderMessage = MessageBuilder.withPayload(order).build();
      warehouseChannel.send(orderMessage);
      return new ResponseEntity<Order>(order, HttpStatus.OK);
   }
}
```

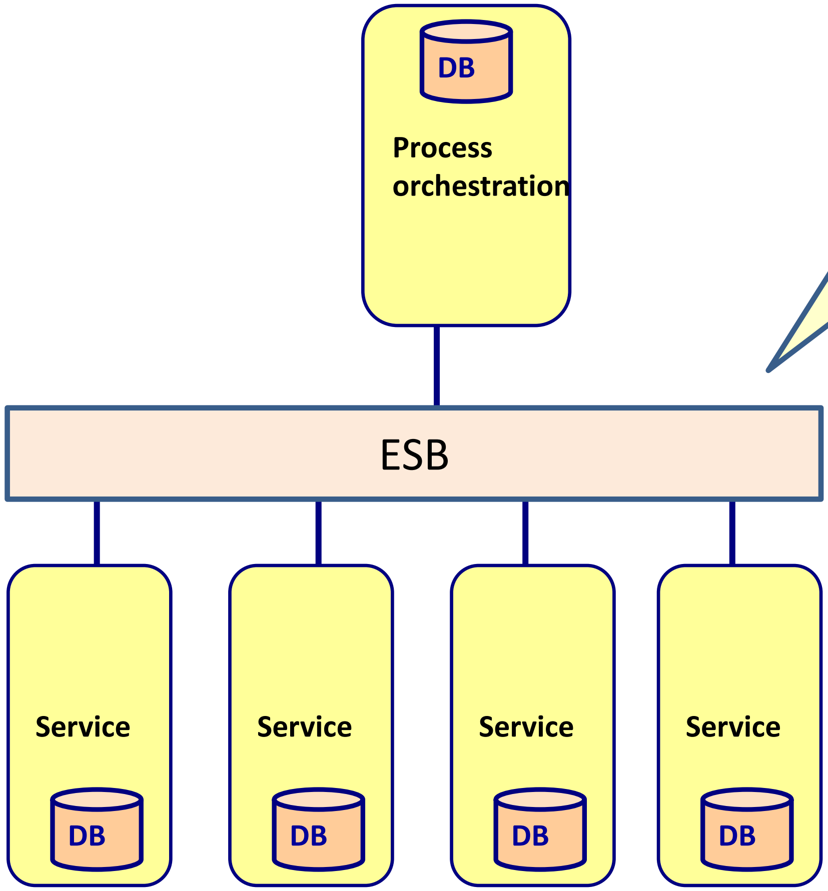
# The activator beans

```java
public class WarehouseActivator {

  @Autowired
  RestTemplate restTemplate;

  public Order checkStock(Order order) {
    System.out.println("WarehouseService: checking order "+order.toString());
    restTemplate.postForLocation("http://localhost:8082/orders", order);
    return order;
  }
}
```

```java
public class ShippingActivator {
  @Autowired
  RestTemplate restTemplate;

  public void ship(Order order) {
    System.out.println("shipping: "+ order.toString());
    restTemplate.postForLocation("http://localhost:8081/orders", order);
  }
}
```

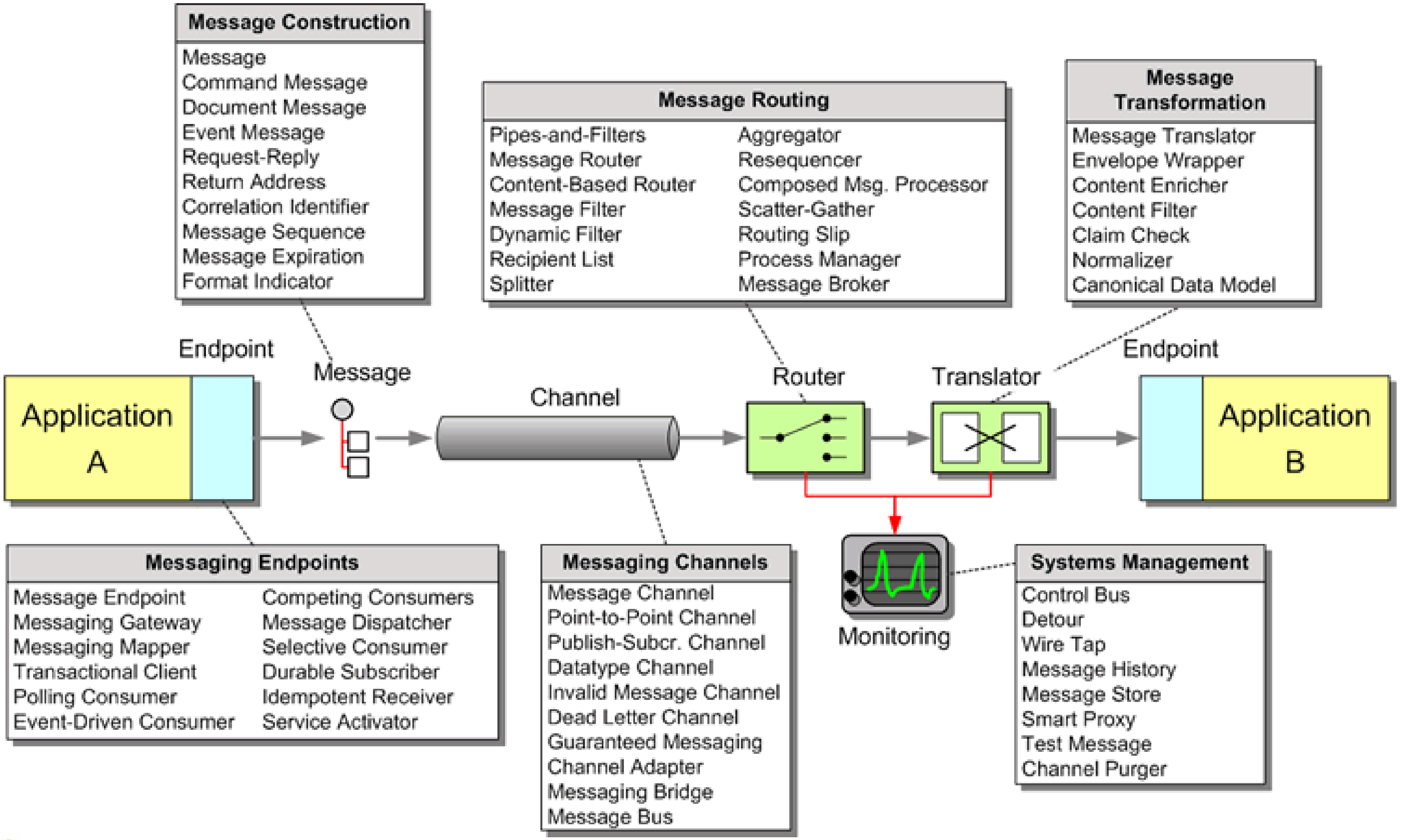
## SUMMARY

# Service Oriented Architecture



An ESB works only when the scope is small and the integration logic is simple

# Enterprise Integration Patterns



**Message Construction**
Message
Command Message
Document Message
Event Message
Request-Reply
Return Address
Correlation Identifier
Message Sequence
Message Expiration
Format Indicator

**Message Routing**
Pipes-and-Filters
Message Router
Content-Based Router
Message Filter
Dynamic Filter
Recipient List
Splitter
Aggregator
Resequencer
Composed Msg. Processor
Scatter-Gather
Routing Slip
Process Manager
Message Broker

**Message Transformation**
Message Translator
Envelope Wrapper
Content Enricher
Content Filter
Claim Check
Normalizer
Canonical Data Model

**Messaging Endpoints**
Message Endpoint
Messaging Gateway
Messaging Mapper
Transactional Client
Polling Consumer
Event-Driven Consumer
Competing Consumers
Message Dispatcher
Selective Consumer
Durable Subscriber
Idempotent Receiver
Service Activator

**Messaging Channels**
Message Channel
Point-to-Point Channel
Publish-Subcr. Channel
Datatype Channel
Invalid Message Channel
Dead Letter Channel
Guaranteed Messaging
Channel Adapter
Messaging Bridge
Message Bus

**Systems Management**
Control Bus
Detour
Wire Tap
Message History
Message Store
Smart Proxy
Test Message
Channel Purger

# Connecting the parts of knowledge with the wholeness of knowledge

1. By externalizing integration logic from the application into an ESB, the applications become more loosely coupled.
2. Integration logic can be designed with a basic set of integration patterns.

---

3. **Transcendental consciousness** is the field that connects everything together.
4. **Wholeness moving within itself:** In Unity Consciousness, one realizes that everything else in creation is connected at the field of pure consciousness