Spring Integration Introduction

Spring Integration

- Spring integration is technology widely used in different parts of Mercury and Quicksilver
- ?

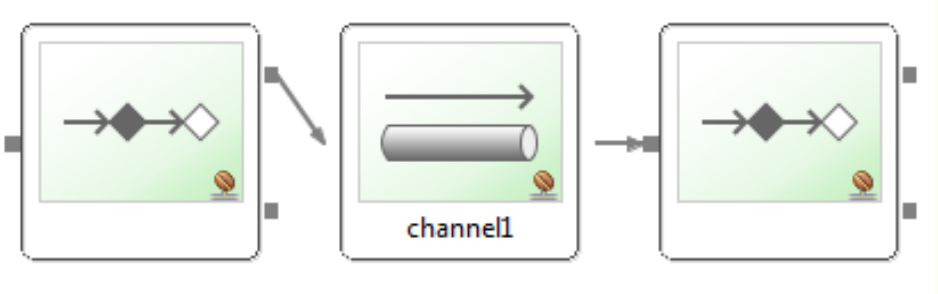
Spring integration allows you to :
1) Let application components exchange data through in-memory messaging
2) Integrate with external applications in a variety of ways through adapters

- Builds on Enterprise Integration Patterns for both

Ground rules

- Simple core API:
- A **Message** is sent by an endpoint
- **Endpoints** are connected to each other using **MessageChannles**
- An **endpoint** can receive **Messages** from a **MessageChannels**

  - by subscribing (passive) or pooling (active)



Message

- A **Message** consists of **MessageHeaders** and a **payload**

  - Some headers are pre defined

  - payload is just Java object

- A message is immutable

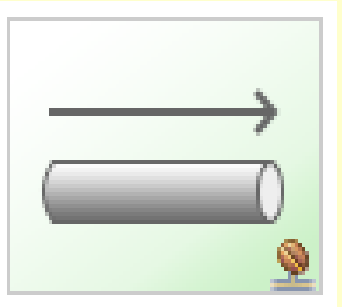
  - Let framework wrap payload for you or use a **MessageBuilder** to create it

- Each Message is created with unique ID

MessageChannels

- Connect message endpoints (loose coupling)
- Optional buffering, Interception
- Just spring beans
  - No broker needed
  - No persistance by default

MessageChannel Types



- **Point-to-point**
- Only one receiver by message
- **DirectChannel**

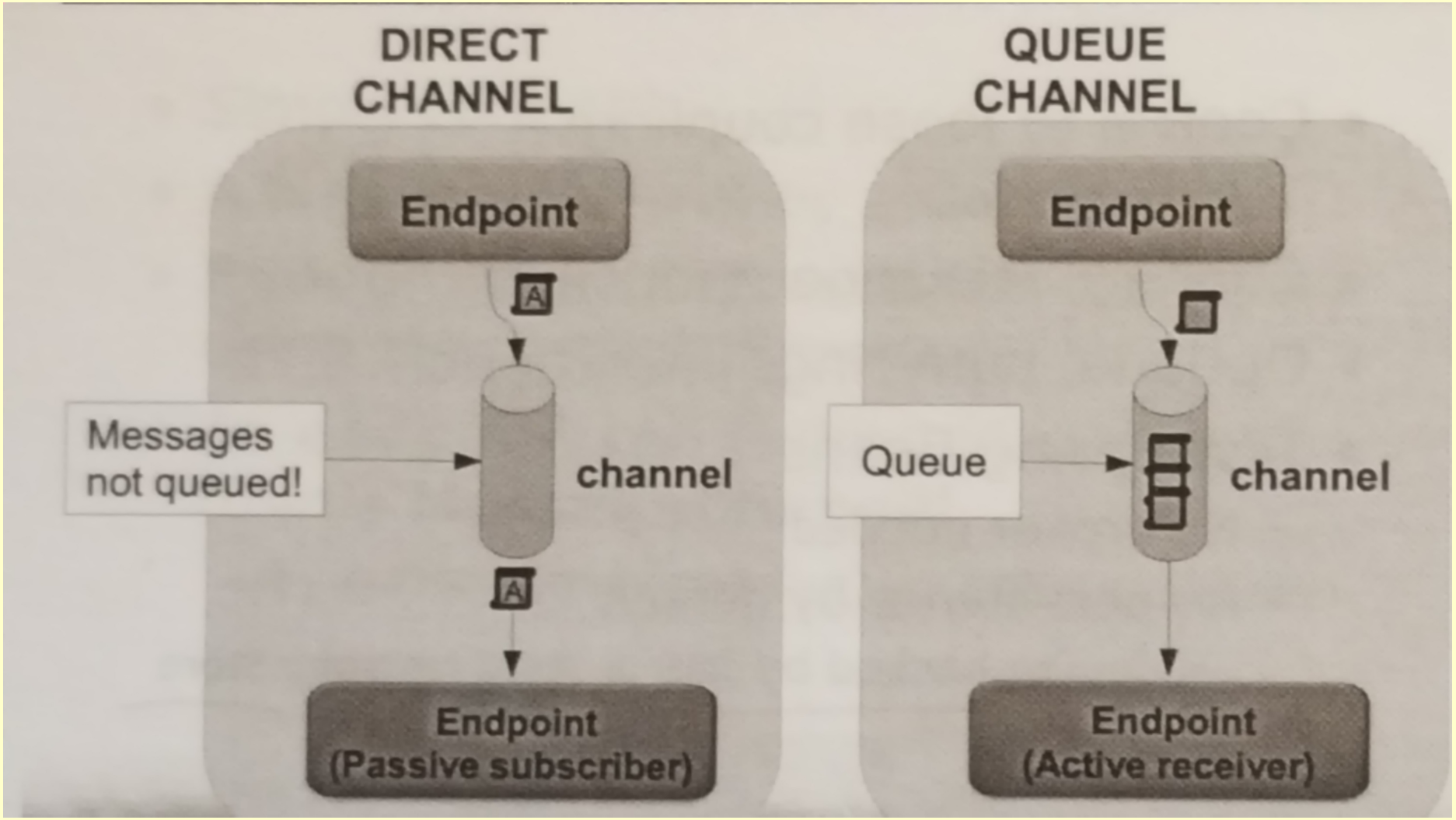
  - Message passed to receiver in senders's thread (return, exceptions), synchronous

  - passive subscriber
- **QueueChannel**

  - Message is queued, sending doesn't block

  - Active receiver polls from different thread

Direct vs Queue Channel

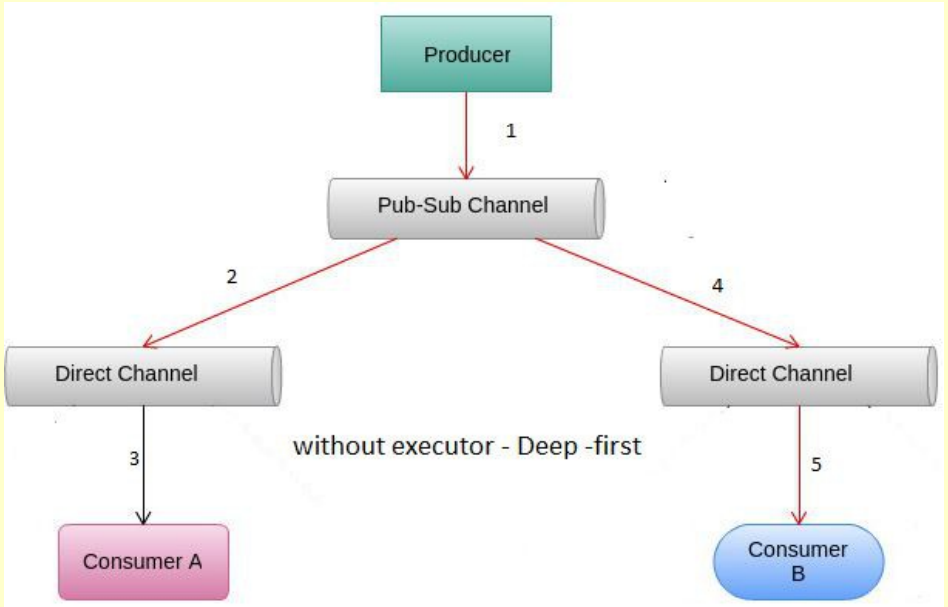


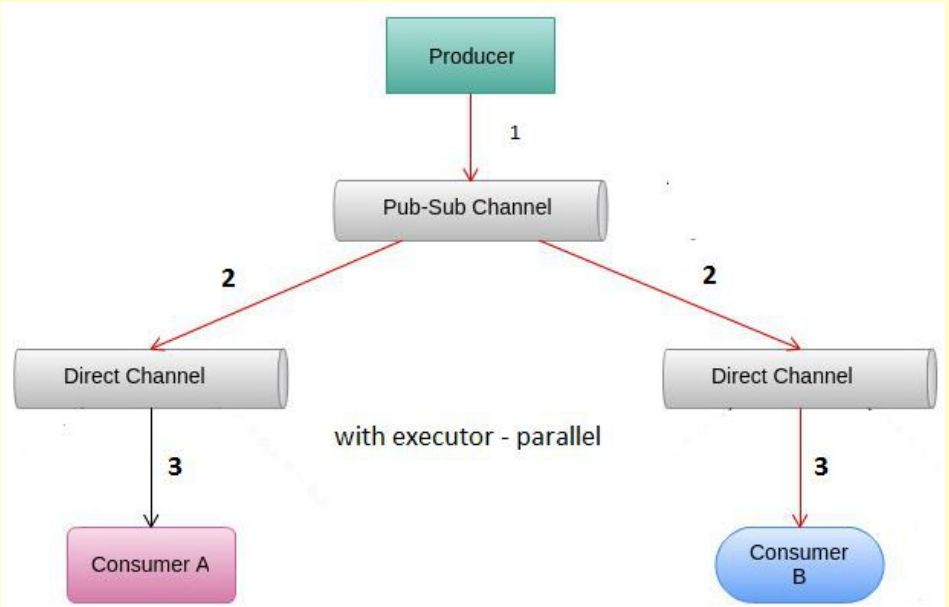
MessageChannel Types

- **Publish-subscribe**
- Multiple receivers per message
- **PublishSubscribeChannel**

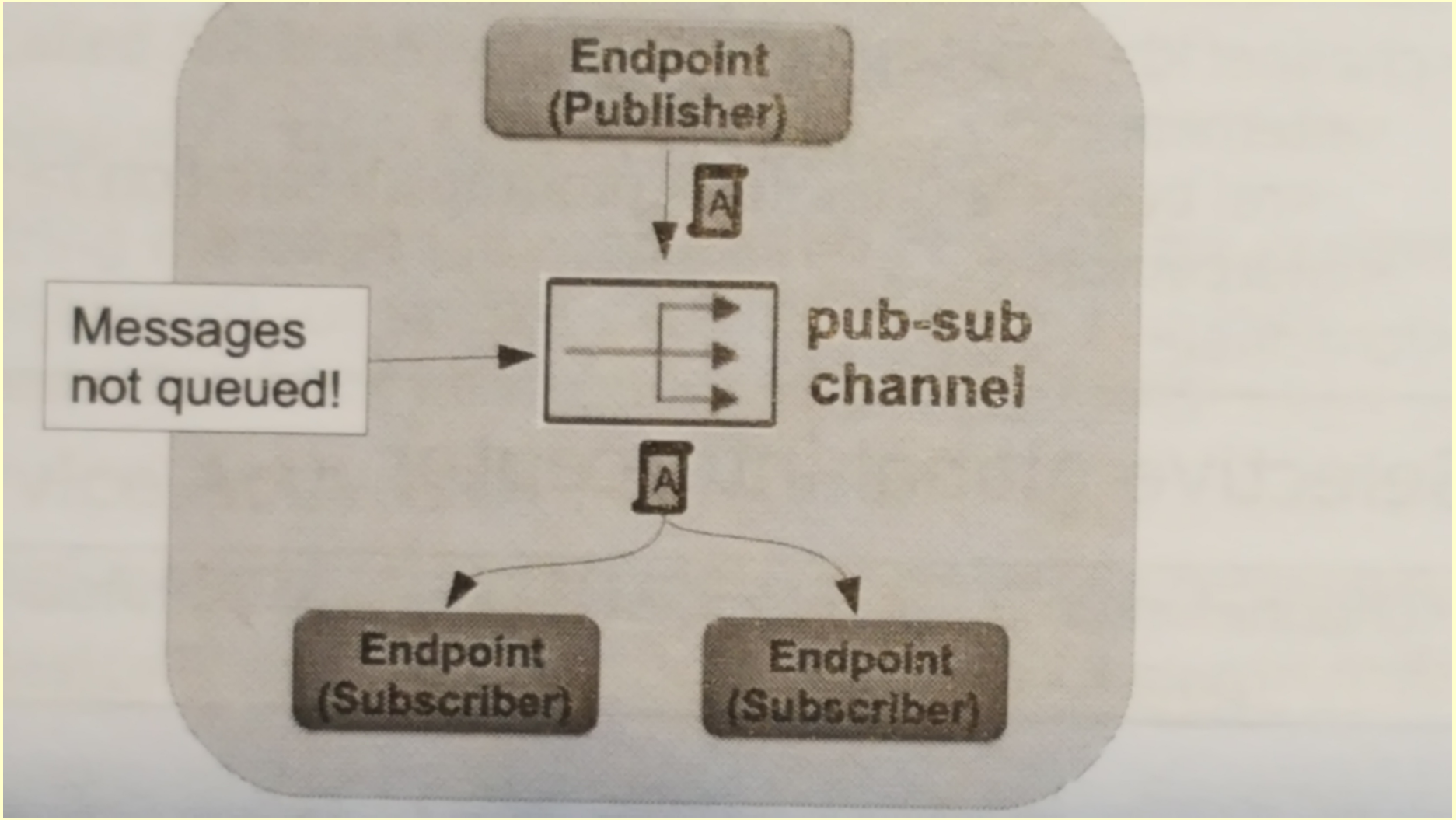
  - Receivers invoked one by one in senders thread

  - Or invoked in parallel on different threads using TaskExecutor





Publish Subscribe



Defining Channels

- **DirectChannel** (sync)

  <channel id="incoming">
- **QueueChannel** (async)

  <channel id="orderedNotifications">

    <queue capacity="10"/>

  </channel>

Defining Channels

- **PublishSubscribeChannel** (sync)

  <publish-subscribe-channel id="statistics" />
- **PublishSubscribeChannel** (async)

  <publish-subscribe-channel id="appEvents"

  task-executor="pubSubExecutor" />

  <task-exacutor id="pubSubExecutor" pool-size="10" />

Message Endpoints

- **Channel Adapter**: One way integration

  - message enters or leaves application

  - Called 'inbound' or 'outbund'

- **Gateway**: Two way integration

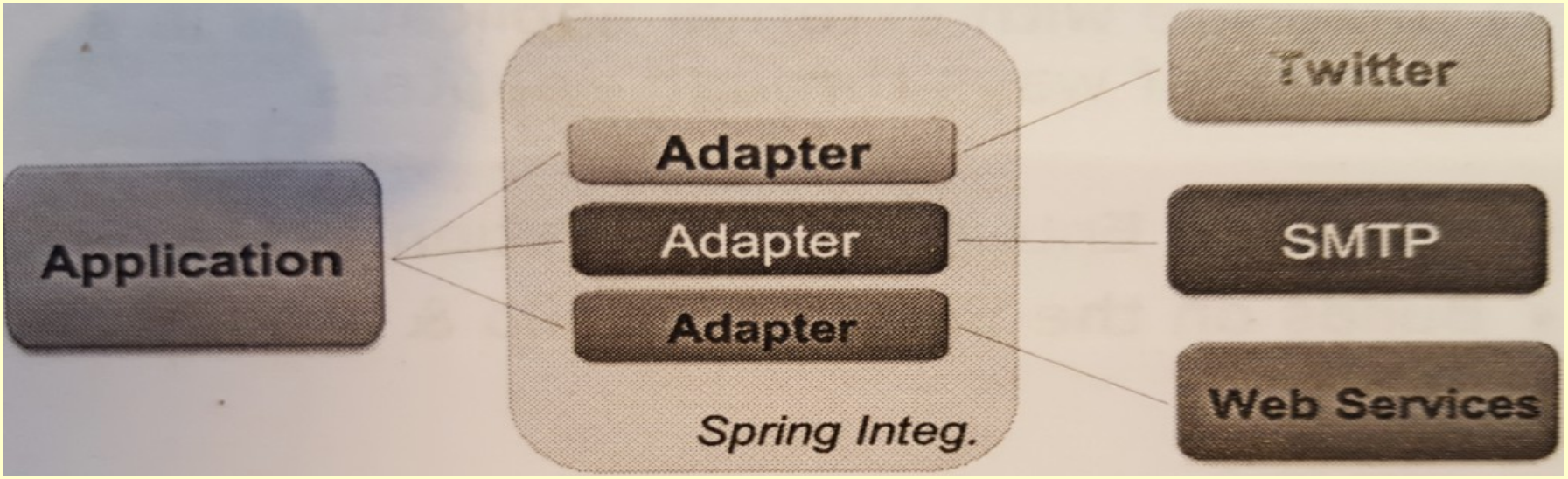
  - Bring message into application and wait for response (inbound) or invoke external system and feed response back into application (outbound)

- **Service Activator**

  - Call method and wrap message into response channel

Adapters

- Connect your application to the outside world
  - Remoting, REST, WS, File & FTP, SMTP, Twitter, ...
  - Also Mirrorlake, PCS, Jupiter, ION, ...
  - For accepting input or producing output



Adapters

- External events produce incomming message

  - Incoming emails, new file (in folder), SOAP request

- Internal message can trigger external event

  - Calling a service, sending JMS message or email

Gateways and Adapters

- Remember the dfference:

  - Channel adapter is one way (in or out)

  - Inbound Gateway awaits internal reply and returns it in-band

  - Outbound Gateway awaits external response and puts it on a channel in invoking thread

- Often use or add message headers

  - inbound HTTP: copies request header to SI headers

  - outbound JMS: copies SI headers to JMS headers

Service Activator

```
<service-activator ref="orderProcessor"
    input-channel="orders" output-channel="confirmations" />
<beans:bean id="orderProcessor" class="broker.OrderProcessor" />
```

- Invoke bean method for incoming message
- Take massage from channel execute business logic and put returned object to new channel

Integration Namespaces

- Spring integration has dedicated namespaces for different integration types
- For example :

  - file, http, xml, jms, ip, twitter

- In Mercury we have own namespaces

  - pcs, jupiter, ml

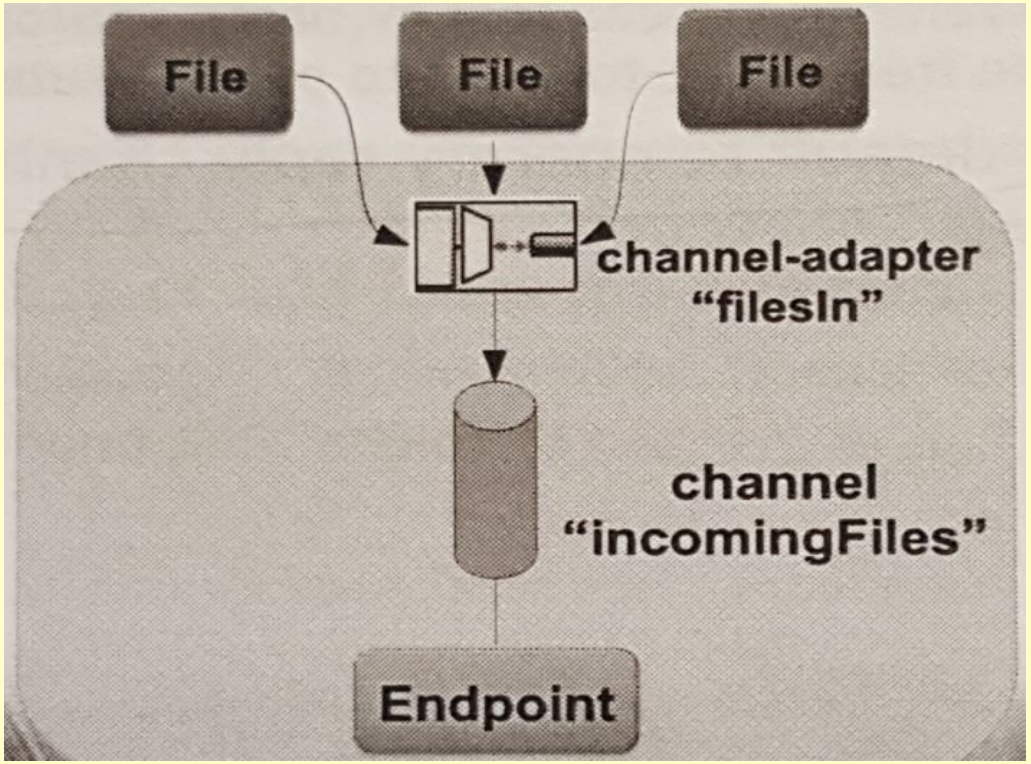
```
<jupiter:publisher id="publisher" inputChannel="channelName"
serverUrl="${host}" userName="${username}"

userPassword="${password}" />
```

Sample : Inbound File Adapter

- It is triggered on new files in directory

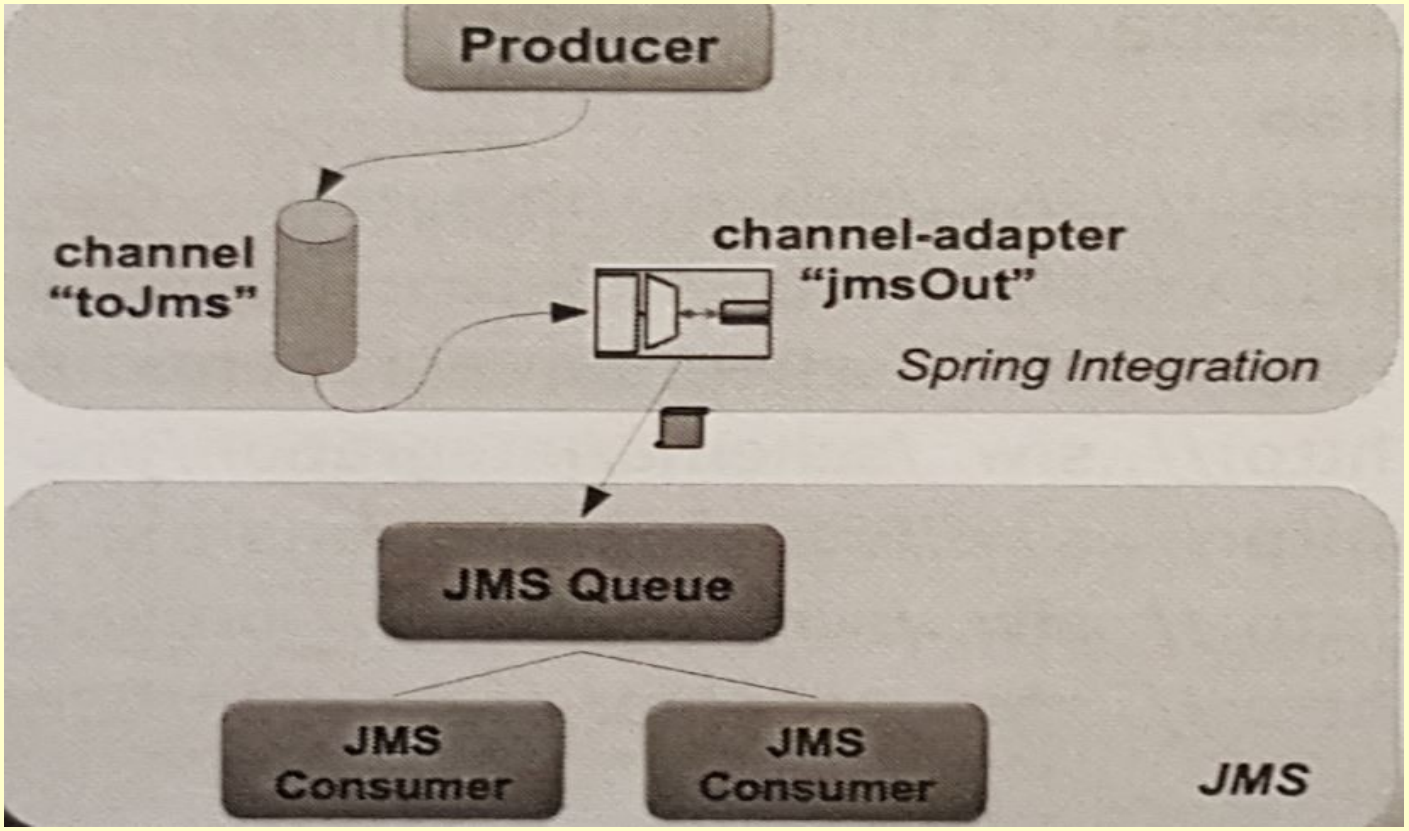
```
<int-file:inbound-channel-adapter id="filesin"
    channel="incommingFiles"
    directory="file:C:/inputResource" />
```



Sample: Outbound JMS Adapter

Adapter translate to jms message

```
<int-jms:outbound-channel-adapter id="jmsOut"
    channel="toJms" destination="jmsQueue" />
```

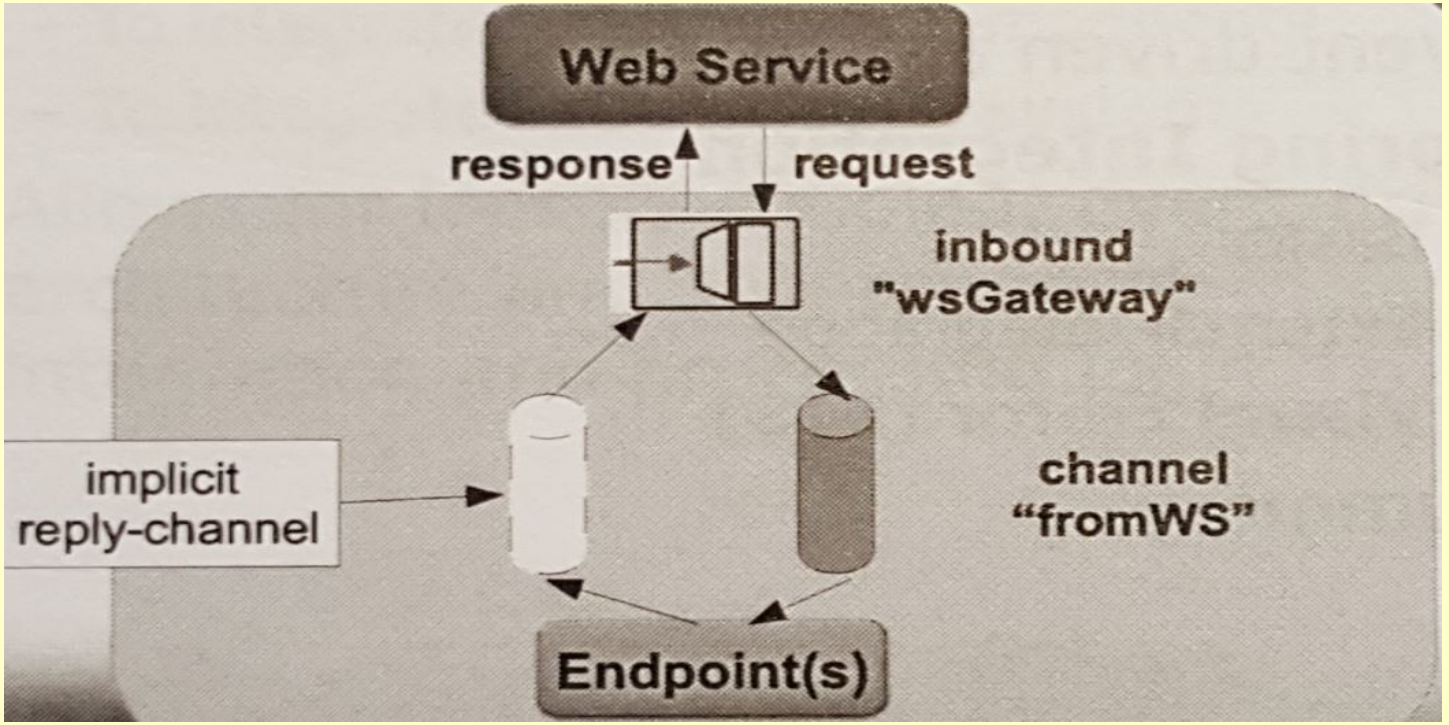


Sample : Inbound Web Service Gateway

<**int-ws:inbound-gateway** id="wsGateway" channel="fromWS" marshaller="jaxb2" unmarshaller="jaxb2" />

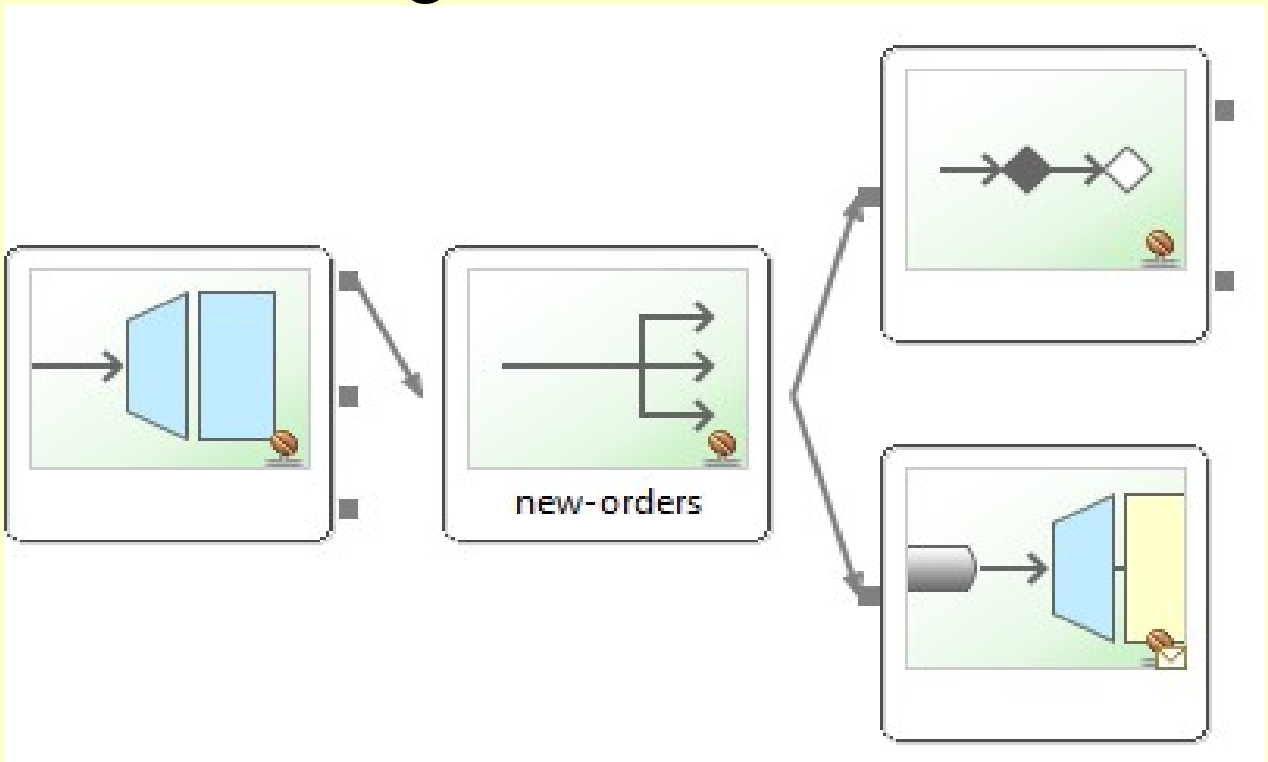
<**oxm:jaxb2-marshaller** id="jaxb2" contextPath="com.example.xml" />

- In com.example.xml we can fing xsd



STS Visual Editor

- SpringSource Tool Suite includes a visual editor for Spring Integration flows
- Select 'integration-graph' tab on open Spring Integration configuration file



Sample: Combining Components - call

```
@Controller
public class OrderController {
    @Autowired OrderService orderService;

    @PostMapping("/orders")
    @ResponseStatus(CREATED)
    public void placeOrder(@RequestBody Order order)
        Confirmation conf = orderService.submitOrder(order)
    ...
}
```

Sample: Combining Components

Message is processed by service adapter and additionaly send to jms queue :

```
<gateway default-request-channel="new-orders"
    service-interface="com.example.OrderService" />
<publish-subscribe-channel id="new-orders"/>
<service-activator input-channel="new-orders" requires-reply="true" ref="orderProcessor"
method="processOrder" />
<int-jms:outbound-channel-adapter channel="new-orders" destination-name="queue.orders" />
```

Benefits

- Loose coupling between components
  - Small focused components
  - Eases testing, reuse
- Event-driven architecture
  - No hard-coded process flow
  - Easy to change or expand
- Separates integration and processing logic
  - Framework handle routing, transformation ...
  - Easily switch beetwen sync & async processing

Benefits

- Loose coupling between components
  - Small focused components
  - Eases testing, reuse
- Event-driven architecture
  - No hard-coded process flow
  - Easy to change or expand
- Separates integration and processing logic
  - Framework handle routing, transformation ...
  - Easily switch beetwen sync & async processing

Next time

- More Endpoint types (bridge, transformer, router, filter)
- Error handling
- Chains
- Using SpEL

LAB