

Spring Integration Introduction

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
- Build on spring portfolio & programming model

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 between sync & async processing

Adapters

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

Adapters

Adapters shield application components from integration details

- External events produce incoming message
 - Incoming emails, new file (in folder), SOAP request, ...
- Components just deal with (payload of) message
 - Don't care about message origin nor destination
- Internal message can trigger external event
 - Calling a service, sending JMS message or email, ...

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

- Central to loose coupling
 - Runtime IoC
- Connect message endpoints
- Optional buffering, Interception
- Just spring beans
 - No broker needed
 - No persistence by default
 - May be backed by JMS or JDBC Message Store

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

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

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-executor id="pubSubExecutor" pool-size="10" />`

Channel Interceptors

- Can operate on messages
 - pre/post send , pre/post receive

```
<channel id="intercepted">  
  <interceptors>  
    <ref bean-"someInterceptorImplementation"/>  
  </interceptors>  
</channel>
```
- Selective global interceptor

```
<channel-interceptor ref="interceptorForXChannels" />
```
- Message immutable, cannot be changed in interceptor, we can build new one using `messageBuilder`

Wire Tap

- Standard pattern implemented as interceptor
- Copies incoming messages to given channel
 - 'Spy' on channel, good for debugging and monitoring
 - often used with logging channel adapter

```
<channel id="in">
```

```
  <interceptor>
```

```
    <wire-tap channel="logger" />
```

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

```
  <logging-channel-adapter channel="logger"  
    level="DEBUG" log-full-message="true" />
```

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
 - Basically outbound gateway for invoking bean method

Messaging Gateway

```
<gateway id="orederService"
  service-interface="com.example.OrderService"
  default-request-channel="orders" />

public interface OrderService {
    public OrderConfirmation submitOrder(Order order);
}
```

- Proxy for sending new messages
 - code doesn't depend on SI API
 - we can inject ordersService and call method and order will go to channel
- Temporary reply channel created automatically
 - Can also specify default-response channel

Gateway method signatures

- Gateway interface methods may return future
 - become async, non-blocking gateway then
 - Void method also supported
 - still called messaging gateway, even though that's technically a passive inbound adapter (as it is one-way)
 - Use annotation for per method configuration
- ```
public interface OrderService {
 @Gateway(requestChannel="orderChannel")
 public Future<Confirmation> submitOrder(Order order);
```

# 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
  - Specify method attribute if there are multiple methods

## Service Activator Methods

- Void and null-returning methods also supported
  - No response message then (one-way)
  - <outbound-channel-adapter> can be used as alternative for void method
- Can cause problems when inbound gateway expects reply message
  - Set **requires-reply** to true to throw an exception on null

# Gateways and Adapters

- Remember the difference:
  - 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

# Temporary Reply Channel (1)

- Temporary reply channels created automatically for inbound gateways if not explicitly defined
  - anonymous point to point channel
  - Set as 'replyChannel' message header (predefined, not needed to specify)
- Used by components that produce output when no explicit output channel is provided
  - like outbound gateways, service activators
  - output becomes reply message
- Reply channel removed automatically after receiving reply message

## Temporary Reply Channel (2)

- Advice: only define explicit reply channel when you need a channel definition
  - to refer to the channel by name and/or to change its type from point-to-point to publish-subscribe
- Rely on default temporary reply channel otherwise

# Integration Namespaces

- Spring integration has dedicated namespaces for different integration types
- For example :  
file, http, xml, jms, ip, twitter

## 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 find xsd

## Sample: Combining Components

Message is processed by service adapter and additionally 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" />
```

# 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)
```

```
 ...
```

```
}
```

# STS Visual Editor

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

# Summary

- Spring Integration provides the base components to implement EPI
  - To integrate application components
  - To integrate with external systems
- Allows for loosely coupled, event-driven architecture and separation of integration and processing logic

LAB