Spring Integration

Enterprise Integration Patterns
Spring Integration – Goals
Main Components
SEDA
Architectures and other things
Conclusion
Trivadis Integration Blueprint V0.1

**Agenda**
- Enterprise Integration Patterns
- Spring Integration – Goals
- Main Components
- SEDA
- Architectures and other things
- Conclusion

---

**Spring Integration – Goals**
- Spring Integration is motivated by the following goals
  - Provide a simple model for implementing complex enterprise integration solutions
  - Facilitate asynchronous, parallel, message-driven behavior within a Spring-based application

---

**Patterns**
- Layer enforcement separation of concerns
- Interface contracts promote Loose coupling

---

Simple programming model to allow easy utilization of multi-core processors and multi-processor hardware

---

Pipe & filter components
Filter = Service, Router, or transformer
Channel adapaters connect in/out-bound transports
Trivadis Integration Blueprint V0.1

Super layering, Well Done !!

Agenda
- Enterprise Integration Patterns
- Spring Integration – Goals
- Main Components
- SEDA
- Architectures and other things
- Conclusion

---

Message
- A generic container for any
  - Java object + metadata
  - Payload of any type
  - Header = Metadata (e.g. timestamp, expiration, return address...)
- Developers can also store any arbitrary key value properties or attributes in the header.

```java
public interface Message<T> {
    Object getId();
    MessageHeader getHeader();
    T getPayload();
    boolean isExpired();
}
```

Message Channel
- Decouples producers from consumers
- Enforces data type consistency
- Provides a subscription strategy
  - Point-to-Point Channel, Publish/Subscribe Channel
- Enables message-based error handling
  - Invalid Message Channel, Dead Message Channel

---
**Channel implementation**

- The `SimpleChannel` implementation wraps a queue
  - capacity = 0: “direct-handoff” channel where a sender will block until the channel’s receive() method is called
  - capacity > 0: channel will store messages in its internal queue until capacity limit is reached and the send() method will return immediately even if no receiver is ready to handle the message

**PriorityChannel implementation**

- No FIFO, but
- allows for messages to be ordered within the channel based upon a priority
- By default the priority is determined by the ‘priority’ property within each message’s header
- However, for custom priority determination logic, a comparator of type `Comparator<Message<?>>` can be provided to the PriorityChannel’s constructor

**Configuring Message Channels**

```xml
<channel id="example Channel" />
<channel id="example Channel" capacity="100" />
<channel id="example Channel" publish-subscribe="true" />
<channel id="example Channel" publish-subscribe="true">
  <dispatcher-policy max-messages-per-task="25"
                     receive-timeout="10"
                     rejection-limit="3"
                     retry-interval="500"
                     should-fail-on-rejection-limit="false" />
</channel>
<channel id="number Channel" data-type="java.lang.Number" />
<priority-channel id="example Channel" data-type="example.Widget"
                  comparator-ref="widget Comparator" />
</channel>
```

**ChannelInterceptor**

```java
public interface ChannelInterceptor {
  boolean preSend(Message<?> message, MessageChannel channel);
  void postSend(Message<?> message, MessageChannel channel, boolean sent);
  boolean preReceive(MessageChannel channel);
  void postReceive(Message<?> message, MessageChannel channel);
}
```
ChannelInterceptorAdapter

Because it is rarely necessary to implement all of the interceptor methods
It provides no-op methods (the void methods are empty, and the boolean methods return true)

```java
public class CountingChannelInterceptor extends ChannelInterceptor {
    private final AtomicInteger sendCount = new AtomicInteger();

    @Override
    public boolean preSend(Message<?> message, MessageChannel channel) {
        sendCount.incrementAndGet();
        return true;
    }
}
```

Message Endpoint

- Provides an abstraction for message producers and consumers
  - Adapts input sources and output targets
  - Handles invocation of local services
- Cleanly separates messaging concerns from business components
  - Acts as a Messaging Gateway for the application
- Supports multiple consumer strategies
  - Polling or Event-driven
  - Selective Consumers, Competing Consumers

Configuring Message Endpoints

```xml
<endPoint input-channel="exampleChannel" handler-ref="exampleHandler"/>
<endPoint input-channel="exampleChannel" handler-ref="somePojO" handler-method="someMethod" default-output-channel="replyChannel"/>
<endPoint id="endpoint" input-channel="channel" handler-ref="handler" selector-ref="exampleSelector"/>
<endPoint id="endpoint" input-channel="exampleChannel" handler-ref="exampleHandler" schedule-period="3000"/>
<endPoint id="endpoint" input-channel="exampleChannel" handler-ref="exampleHandler" concurrency-core="5" max="25" queue-capacity="20" keep-alive="120"/>
```
Channel Adapter

- interacting with external systems or
- other components that are external to the messaging system
- As the name implies, the interaction consists of adapting the external system or component to send-to and/or receive-from a MessageChannel
- Source Adapters and Target Adapters
- JMS, File, FTP, RMI, Webservice, Stream, Mail, HTTP, ApplicationEvent

File Channel Adapters

```
<si:message-bus/>
<si:channel id="inputChannel"/>
<si:channel id="outputChannel"/>
<si:file-source id="fileSourceAdapter" directory="${java.io.tmpdir}/test-input" channel="inputChannel" poll-period="10000"/>
<si:file-target id="fileTargetAdapter" directory="${java.io.tmpdir}/test-output" channel="outputChannel"/>
```

Delegating Target Adapters

```
MethodInvokingTarget target = new MethodInvokingTarget();
target.setObject(new ExampleTarget());
target.setMethod("publish");
DefaultTargetAdapter adapter = new DefaultTargetAdapter(target);
bus.registerHandler("adapter", adapter);
new Subscription(channel));
channel.send(new StringMessage("foo"));
```

channel decouples method invocation
Service Activator

- A Message Endpoint that invokes a service
- Supports multiple communication styles
  - one-way and request-reply
  - synchronous and asynchronous
- The service is unaware of the messaging system

![Service Activator diagram]

MessageSelector

- **reactive routing** to determine what messages the handler should receive
  - Datatype Channel and Message Router provide **proactive routing**

- can be configured with 0 or more selectors,
  - will only receive messages that are accepted by each selector
  - a couple of common selector implementations are provided

![MessageSelector diagram]

Message Router

- Particular type of **Message Handler**
- Route messages to message channels
- Isolate routing strategy from business logic
- Provide a dynamic alternative to publish/subscribe channels
- Accommodate complex messaging scenarios
  - Splitter, Aggregator, Resequencer

![Message Router diagram]
**PayloadTypeRouter**

```java
PayloadTypeRouter router = new PayloadTypeRouter();
router.setChannelMappings(channelMappings);
Message<String> message1 = new StringMessage("test");
router.handle(message1); // will send to 'stringChannel'
Message<Integer> message2 = new GenericMessage<Integer>(123);
router.handle(message2); // will send to 'integerChannel'
```

**RecipientListRouter**

```java
RecipientListRouter router = new RecipientListRouter();
Message<String> message = new StringMessage("test");
router.handle(message); // will send to channel1 and channel2
```

**MessageHandler**

- A generic interface defines the simple but common behavior of processing a received Message
- Many of the internal base messaging components implement this top-level interface
  - Routers, Transformers, Service Invokers
- Implementations do not necessarily return a reply Message (routers, void-returning service invokers)
Example: Message Translator

- Convert payload type
- Enrich message content
- Filter message content
- Normalize message format
  - Multiple clients may send multiple versions
  - The application may expect a canonical format

MessageHandlerChain

- MessageHandlers can be linked together

```java
MessageHandlerChain chain = new MessageHandlerChain();
chain.add(new Handler1());
chain.add(new Handler2());
chain.add(new Handler3());
Message result = chain.handle(new StringMessage("foo"));
```

Message Bus

- The Message Bus passes the messages from the channel to the handler (is a mediator)
- manage registration of the MessageChannels and MessageHandlers
- creation and lifecycle management of message dispatcher
- Each channel has a DispatcherPolicy
- activation of handler subscriptions
- configuration of thread pools
Agenda

- Enterprise Integration Patterns
- Spring Integration – Goals
- Main Components
- SEDA
- Architectures and other things
- Conclusion

SEDA

- Matt Welsh, Ph.D. thesis work at UC Berkeley
- Decompose service into stages separated by queues
  - Each stage performs a subset of request processing
  - Stages internally event-driven
- Each stage contains a thread pool to drive stage execution
  - However, threads are not exposed to application
  - Dynamic control grows/shrinks thread pool

SEDA - Staged Event Driven Systems

- Alternative to thread-per-request server model
- Controlled number of threads per handler
- Ideal for short-lived tasks and high # of requests
The cool spring integration universe

we call it
Spring-Otopia

Spring Integration

uses

triggers

Spring Batch

uses

uses

Spring

Spring Integration and Spring Batch #2

<table>
<thead>
<tr>
<th>Framework</th>
<th>Building Block</th>
<th>Runtime</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spring</td>
<td>Component</td>
<td>Component Container</td>
<td>CDA</td>
</tr>
<tr>
<td>Spring Integration</td>
<td>Service</td>
<td>ESB Container</td>
<td>SOA</td>
</tr>
<tr>
<td>Spring Batch</td>
<td>Batch Script/Job</td>
<td>Job Container</td>
<td>JOA</td>
</tr>
</tbody>
</table>

CDA - Component Oriented Architecture  
SOA - Service Oriented Architecture  
JOA - Job Oriented Architecture

Integration View

Application

Integration Domain

Transport

Spring Integration and Spring Batch #3

<table>
<thead>
<tr>
<th>Spring Integration</th>
<th>Spring Batch</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOA</td>
<td>JOA</td>
</tr>
<tr>
<td>Objects</td>
<td>Data</td>
</tr>
<tr>
<td>Bus</td>
<td>Hub</td>
</tr>
<tr>
<td>Distributed</td>
<td>Local</td>
</tr>
<tr>
<td>single Messages</td>
<td>Bulk Data</td>
</tr>
<tr>
<td>State Machine</td>
<td>Transactional</td>
</tr>
</tbody>
</table>
Conclusion - Spring Integration

- Spring framework extension for easy implementation of
  - ESB
  - SEDA
  - develop for multicores
- Simple and powerful
- Easy to extend
- Spring & Spring Integration a winning team

Thank you!

Upcoming event
High Performance Computing
16.9.2008, 14:30h
Oracle, Baden-Dättwil
http://www.trivadis.com/events/kommende-events.html