Trivadis Integration Blueprint V0.1

Spring Integration

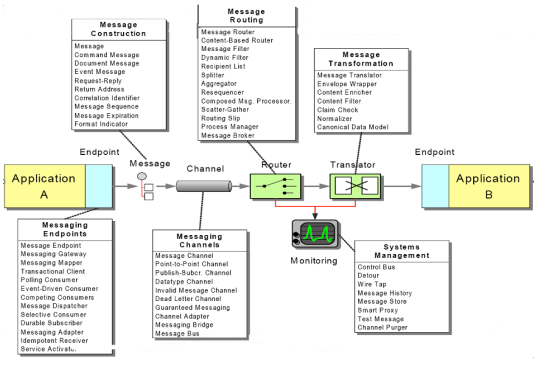
Peter Welkenbach
Principal Consultant
Trivadis
peter.welkenbach@trivadis.com

trivadis
makes IT easier.

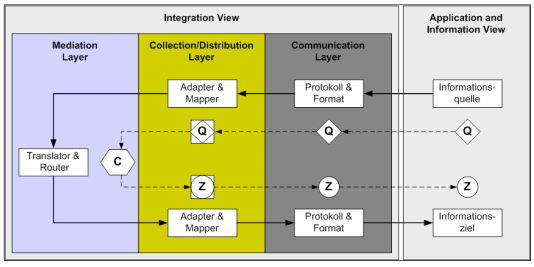
Agenda

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

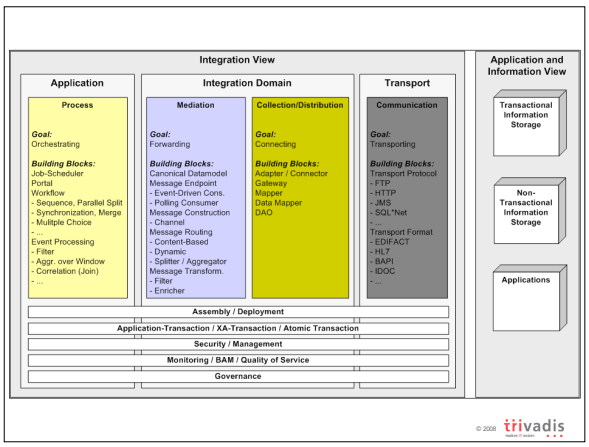
EAI Patterns (Hohpe, Woolf 2004)



Integration - Flow



Trivadis Integration Blueprint V0.1



Agenda

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

Spring Integration – Goals

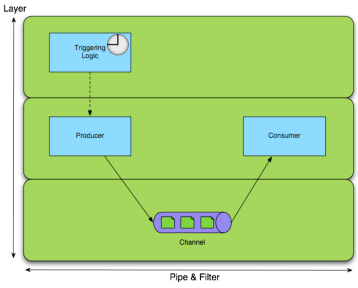
- Spring Integration is motivated by the following goals
  - Provide a simple model for implementing complex enterprise integration solutions
  - Facilitate asynchronous, parallel, message-driven behavior within a Spring-based application

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

Patterns

Layers enforce separation of cooncerns

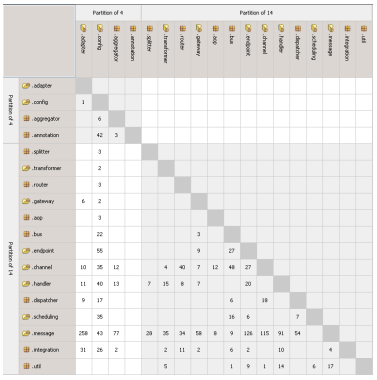
Interface contracts promote Loose coupling



Pipes decouple components

Filter = Service, Router, or transformer

Channel adapeters connect in/out-bound transports



Super layering, Well Done !!

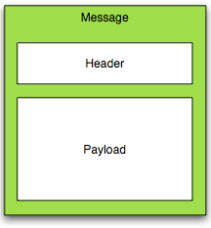
Agenda

- 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.

```
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
  - FIFO
- 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

Configuring Message Channels

```
<channel id="exampleChannel"/>
<channel id="exampleChannel" capacity="100"/>
<channel id="exampleChannel" publish-subscribe="true"/>
<channel id="numberChannel" datatype="java.lang.Number"/>
```

```
<channel id="exampleChannel" 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>
```


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

```
<priority-channel id="exampleChannel"
                  datatype="example.Widget"
                  comparator-ref="WidgetComparator"/>
```


ChannelInterceptor

```
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);
}
```

```
<channel id="exampleChannel">
      <interceptor ref="trafficMonitoringInterceptor"/>
</channel>
```

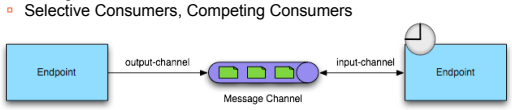
ChannelInterceptorAdapter

- 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)

```
public class CountingChannelInterceptor
    extends ChannelInterceptorAdapter {
    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

```
<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>
```


Configuring Message Endpoints

```
<endpoint input-channel="exampleChannel"
          handler-ref="exampleHandler"/>
    <schedule period="5000"/>
</endpoint>
```

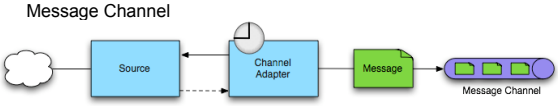
```
<endpoint input-channel="exampleChannel"
          handler-ref="exampleHandler"/>
    <concurrency core="5"
                 max="25"
                 queue-capacity="20"
                 keep-alive="120"/>
</endpoint>
```

Channel Adapter

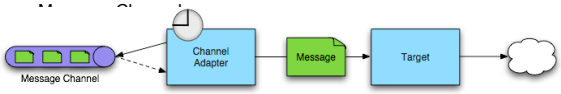
- interacting with external systems or
- other components that are external to the messaging system
- As the name implies, the interaction consists of adapting the external system or component to send-to and/or receive-from a MessageChannel
- Source Adapters and Target Adapters
- JMS, File, FTP, RMI,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

Channel Adapter

- Connect a source to the messaging system so it can send to a Message Channel



- Connect a target to the messaging system so it can receive from



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"/>
```

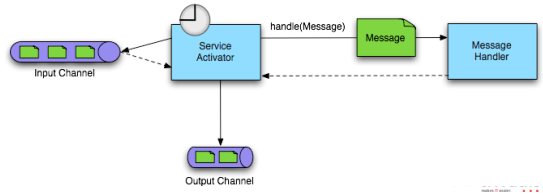

Delegating Target Adapters

```
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



MessageSelector

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

```
public interface MessageSelector {
   boolean accept(Message<?> message);
}
```

- `MessageEndpoint` 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 - PayloadTypeSelector

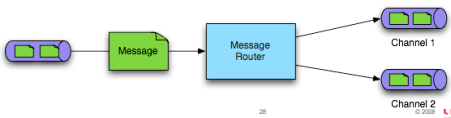
- `PayloadTypeSelector` provides similar functionality to Datatype Channels except that in this case the type-matching can be done by the endpoint rather than the channel

```
PayloadTypeSelector selector =
      new PayloadTypeSelector(String.class, Integer.class);

assertTrue( selector.accept(new StringMessage("example")));

assertTrue( selector.accept(new GenericMessage<Integer>(123)));

assertFalse( selector
         .accept(new GenericMessage<MyObject>(myObject)));
```

- `UnexpiredMessageSelector` only accepts messages that have not yet expired

Message Router

- Particular type of `MessageHandler`
- 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



Router

```
<context:component-scan
   base-package="org.springframework.integration.samples.oddeven"/>

<message-bus auto-create-channels="true" />

<annotation-driven/>
```

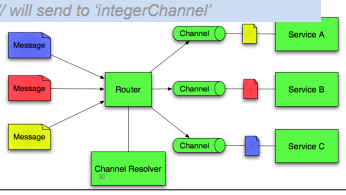
```
@MessageEndpoint
public class Counter {
   private AtomicInteger count = new AtomicInteger();

   @Polled(period=3000)
   public int getNumber() {
      return count.incrementAndGet();
   }

   @Router
   public String resolveChannel(int i) {
      if (i % 2 == 0) {
            return "even";
      }
      return "odd";
   }
}
```

Send to channel id="even"

Send to channel id="odd"

PayloadTypeRouter

```
channelMappings.put(String.class, stringChannel);
channelMappings.put(Integer.class, integerChannel);

PayloadTypeRouter router = new PayloadTypeRouter();
router.setChannelMappings(channelMappings);

Message<String> message1 = new StringMessage("test");
Message<Integer> message2 = new GenericMessage<Integer>(123);

router.handle(message1); // will send to 'stringChannel'
router.handle(message2); // will send to 'integerChannel'
```



RecipientListRouter

```
List<MessageChannel> channels =
                              new ArrayList<MessageChannel>();
channels.add(channel1);
channels.add(channel2);

RecipientListRouter router = new RecipientListRouter();
router.setChannels(channels);

Message<String> message = new StringMessage("test");

// will send to channel1 and channel2
router.handle(message);
```

Comparable to a dynamic publish/subscribe

It's cool man

MessageHandler

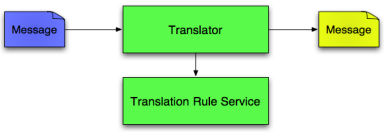
- A generic interface defines the simple but common behavior of processing a received Message

```
public interface MessageHandler {
   Message handle(Message 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

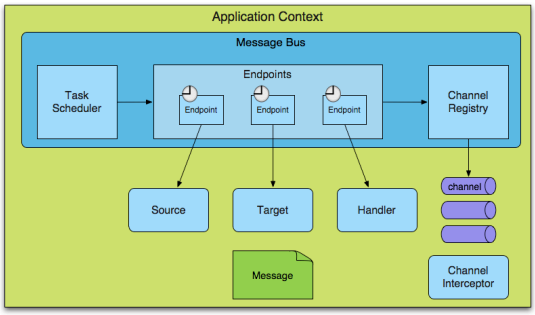


MessageHandlerChain

- MessageHandlers can be linked together

```
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



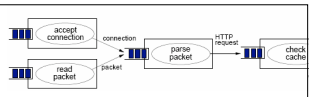
Message Bus

- *The MessageBus 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

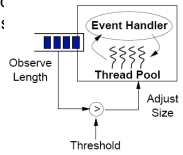
Agenda

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

SEDA

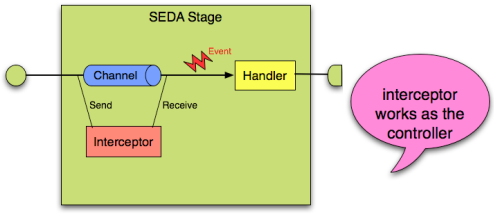


- Matt Welsh, Ph.D. thesis work at UC Berkeley
- Decompose service into *stages* separated by *queues*
  - Each stage performs a subset of request processing
  - Stages internally event-driven
- Each stage contains a *thread pool* to drive stage execution
  - However, threads are not exposed to applic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



SEDA - Staged Event Driven Systems

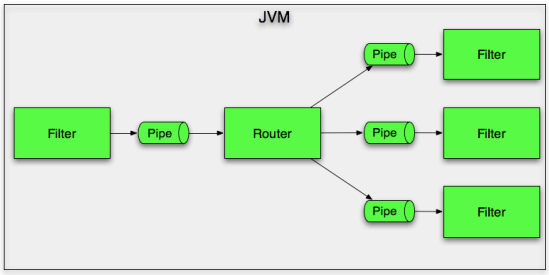
- Alternative to thread-per-request server model
- Controlled number of threads per handler
- Ideal for short-lived tasks and high # of requests



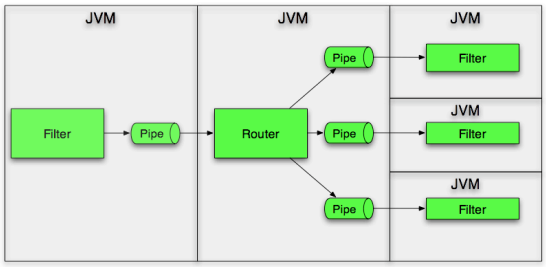
Agenda

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

Develop and Test in one JVM

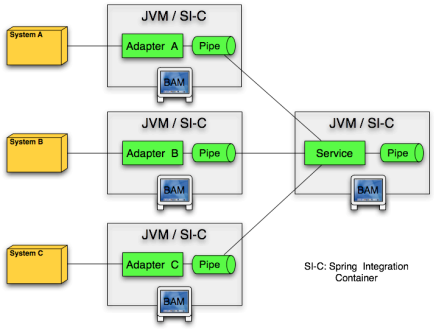


Deploy to different JVMs

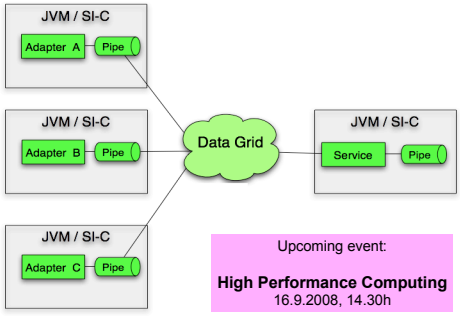


different GC strategies and Heap size per JVM possible

Integration Architecture – host adapters per system



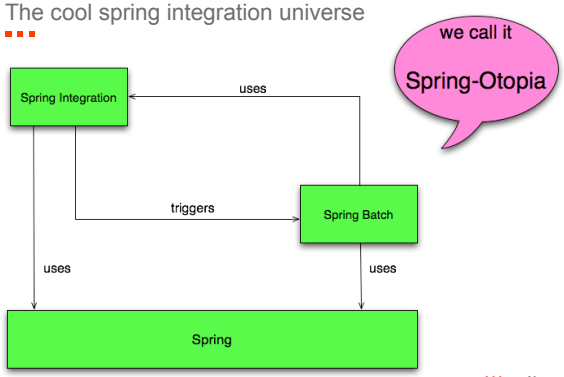
Integration Architecture – Data Grid

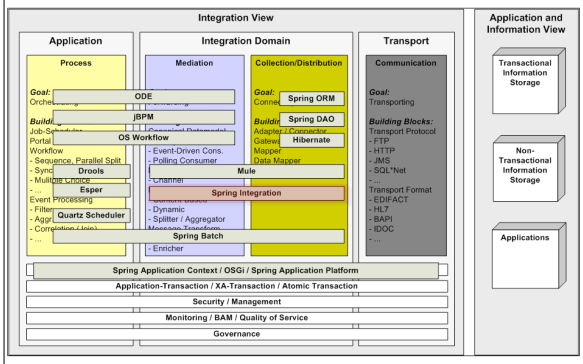


Upcoming event:

High Performance Computing
16.9.2008, 14.30h

The cool spring integration universe





Spring Integration and Spring Batch #2

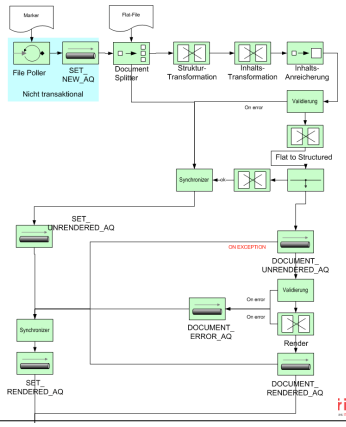
Framework	Building Block	Runtime	Type
Spring	Component	Component Container	COA
Spring Integration	Service	ESB Container	SOA
Spring Batch	Batch Script/Job	Job Container	JOA

COA - Compponent Oriented Architecture
SOA - Service Oriented Architecture
JOA - Job Oriented Architecture

Spring Integration and Spring Batch #3

Spring Integration	Spring Batch
SOA	JOA
Objects	Data
Bus	Hub
Distributed	Local
single Messages	Bulk Data
State Machine	Transactional

Protoype



Agenda

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

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