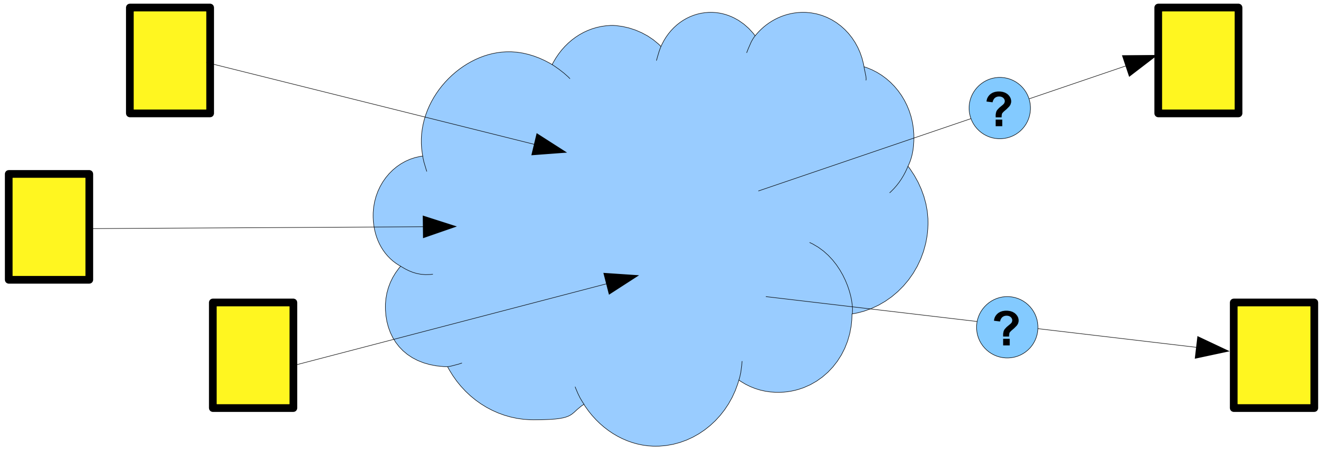
Message Driven Architecture with Spring

Mark Fisher, SpringSource/VMware
http://blog.springsource.com/author/markf
Twitter: @m_f_

Agenda

- Events
- Messaging
- Polling
- Scheduling
- Pipes and Filters
- Enterprise Integration Patterns
- Messaging in the Cloud

Events



Sending ApplicationEvents

1. Implement the callback to have the publisher injected by the container.

2. Define the bean, and use the publisher to send events at runtime.

```
public class MyEventPublisher implements ApplicationEventPublisherAware {

  private volatile ApplicationEventPublisher publisher;

  public void setApplicationEventPublisher(ApplicationEventPublisher aep) {
    this.publisher = aep;
  }

  public void send(String text) {
    Assert.notNull(this.publisher, “no publisher available”);
    this.publisher.publishEvent(new MyEvent(text));
  }

}
```

```
<bean  id="publisher" class="example.MyEventPublisher"/>
```

Receiving ApplicationEvents

1. Implement the ApplicationListener interface.

2. Define the bean, and it will be invoked at runtime when Events occur.

```
public class MyEventListener implements ApplicationListener {

  private final Log log = LogFactory.getLog(this.getClass());

  public void onApplicationEvent(ApplicationEvent event) {
    this.log.info("received event: " + event);
  }

}
```

```
<bean  id="listener" class="example.MyEventListener"/>
```

Filtering Event Types

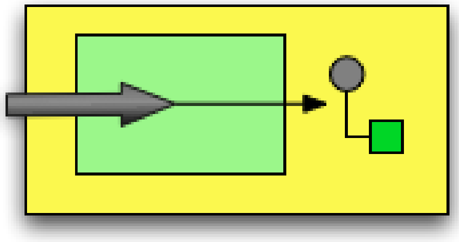
1. Implement the ApplicationListener interface ***with a parameterized type***.

2. Define the bean, and it will be invoked at runtime when that type of Event occurs.

```java
public class FooEventListener
             implements ApplicationListener<FooEvent> {

  private final Log log = LogFactory.getLog(this.getClass());

  public void onApplicationEvent(FooEvent event) {
    this.log.info("received FOO event: " + event);
  }

}
```

```xml
<bean  id="fooListener" class="example.FooEventListener"/>
```

Messaging



Sending JMS Messages

1. Inject an instance of Spring's JmsTemplate.

2. Provide the JMS ConnectionFactory in the JmsTemplate bean definition.

```
public class MessageSender {

  @Autowired
  private volatile JmsTemplate jmsTemplate;

  public void send(String message) {
    this.jmsTemplate.convertAndSend("example.queue", message);
  }

}
```

```
<bean  class="org.springframework.jms.core.JmsTemplate">
  <property name="connectionFactory" ref="connectionFactory"/>
</bean>

<bean id="connectionFactory"
 class="org.springframework.jms.connection.CachingConnectionFactory">
  ...
</bean>
```

Receiving JMS Messages

1. Define a “listener-container” within the context.

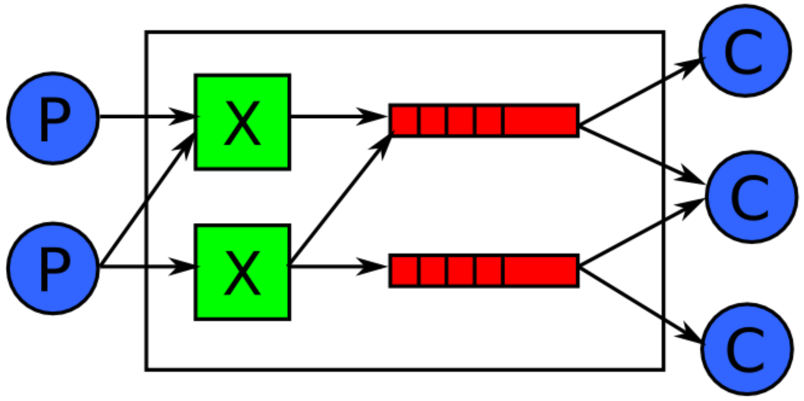
2. Point to any POJO to implicitly create a MessageListenerAdapter.

```
public class MyListener {

  private final Log log = LogFactory.getLog(this.getClass());

  public void log(String message) {
    this.log.info("received: " + message);
  }

}
```

```
<jms:listener-container>
  <jms:listener destination="example.queue" ref="listener" method="log"/>
</jms:listener-container>

<bean id="listener" class="example.MyListener"/>
```

Spring AMQP

- Similar to Spring's JMS support
- Does not hide AMQP behind JMS



- Exchanges
  - Where producers send Messages
  - May also send a routing key
- Queues
  - Where consumers receive Messages
  - A named FIFO buffer
- Bindings
  - Exchanges route to queues
  - Queues bind with routing keys/patterns

AMQP Messaging

1. Use AmqpTemplate instead of JmsTemplate (accepts exchange and routingKey).
2. Nothing changes on the listener side (just a POJO).

```java
public class MessageSender {

  @Autowired
  private volatile AmqpTemplate amqpTemplate;

  public void send(String message) {
    this.amqpTemplate.convertAndSend(
            "myExchange", "some.routing.key", message);
  }

}
```

```java
public class MyListener {

  private final Log log = LogFactory.getLog(this.getClass());

  public void log(String message) {
    this.log.info("received: " + message);
  }
}
```

```xml
<amqp:listener-container>
  <amqp:listener queue-names="foo" ref="listener" method="log"/>
</amqp:listener-container>

<bean id="listener" class="example.MyListener"/>
```

HTTP Messaging (Request/Reply)

1. Use RestTemplate, passing URI to methods based on HTTP methods

2. Configure HttpMessageConverters if out-of-the-box support is insufficient

```
public class HttpClient {

  private final String uri = “http://localhost/demo/{name}”;

  private final RestTemplate template = new RestTemplate();

  public String getResource(String name) {
    this.template.getForObject(uri, String.class, name);
  }

  public URI postResource(String name, Object resource) {
    this.template.postForLocation(uri, resource, name);
  }
}
```

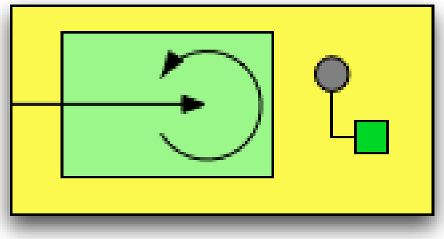
Sending Mail Messages

1. Create a SimpleMailMessage instance (or JavaMail MimeMessage).

2. Use MailSender (or JavaMailSender) configured with host/user/password, etc.

```
public class MailClient {

  @Autowired
  private volatile MailSender mailSender;

  public void send(String subject, String to, String text) {
    SimpleMailMessage message = new SimpleMailMessage();
    message.setSubject(subject);
    message.setTo(to);
    message.setText(text);
    this.mailSender.send(message);
  }

}
```

Polling



Lifecycle Management

Spring's Lifecycle interface provides basic support for any background task.

```
public interface Lifecycle {

  void start();

  void stop();

  boolean isRunning();
}
```

SmartLifecycle adds auto-startup and graceful shutdown capabilities.

```
public interface SmartLifecycle extends Lifecycle, Phased {

  boolean isAutoStartup();

  void stop(Runnable callback);
}
```

```
public interface Phased {
  int getPhase();
}
```

Task Execution

Spring provides a TaskExecutor whose signature matches Executor.

Several implementations are available: Threads can be managed, pooled, etc.

```
public class FilePoller implements Lifecycle {

  @Autowired @Qualifier("threadPool")
  private volatile Executor executor;

  public void start() {
    this.executor.execute(new Runnable() {
      public void run() {
        while (!this.stopRequested) {
          // poll a directory by calling listFiles()
          // send the list of Files to a handler
        }
      }
    });
  }
...
```

```
<task:executor id="threadPool" pool-size="5-25"/>

<bean class="example.FilePoller"/>
```

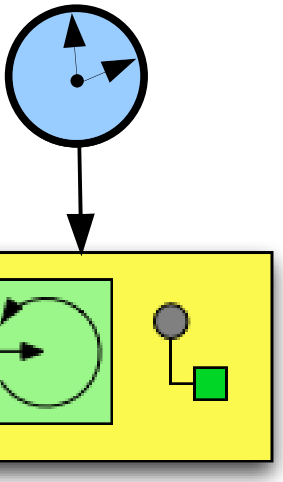
@Async

The @Async annotation implicitly adds the Executor support.

```
public class FilePoller implements Lifecycle {

  @Async
  public void start() {
    while (!this.stopRequested) {
      // poll a directory by calling listFiles()
      // send the list of Files to a handler
    }
  }
  ...
}
```

```
<task:annotation-driven executor="threadPool"/>

<task:executor id="threadPool" pool-size="5-25"/>

<bean class="example.FilePoller"/>
```

Scheduling



Task Scheduler and Trigger

Spring's TaskScheduler and Trigger provide an abstraction for scheduling tasks.

```
public interface TaskScheduler {

  ScheduledFuture schedule(Runnable task, Trigger trigger);
  ScheduledFuture scheduleAtFixedRate(Runnable task, long period);
  ScheduledFuture scheduleWithFixedDelay(Runnable task, long delay);
  ...
}
```

```
public interface Trigger {

  Date nextExecutionTime(TriggerContext triggerContext);

}
```

```
public interface TriggerContext {

  Date lastScheduledExecutionTime();
  Date lastActualExecutionTime();
  Date lastCompletionTime()
}
```

Trigger implementations include:
- PeriodicTrigger
- CronTrigger

Task Scheduling

TaskScheduler supports recurring, cancelable tasks.
As with TaskExecutor, Threads can be managed, pooled, etc.

```
public class FilePoller implements Lifecycle {

  @Autowired @Qualifier("scheduler")
  private volatile TaskScheduler scheduler;

  // other properties, e.g. 'task' and 'trigger'

  public void start() {
     this.task = this.scheduler.schedule(task, trigger);
  }

  public void stop() {
    if (this.task != null ) { this.task.cancel(true); }
  }
  ...
}
```

```
<task:scheduler id="scheduler" pool-size="10"/>
```

@Scheduled

The @Scheduled annotation implicitly adds the TaskScheduler support.

Spring also provides <task:scheduled-tasks> as an alternative to annotations.

```
public class FilePoller {

  @Scheduled(cron="*/5 * 9-17 * * ?")
  public void poll() {
    // poll and send Files to a handler
  }

}
```

```
<task:annotation-driven scheduler="threadPool"/>

<task:scheduler id="scheduler" pool-size="10"/>

<bean class="example.FilePoller"/>
```

@Scheduled as Meta-annotation

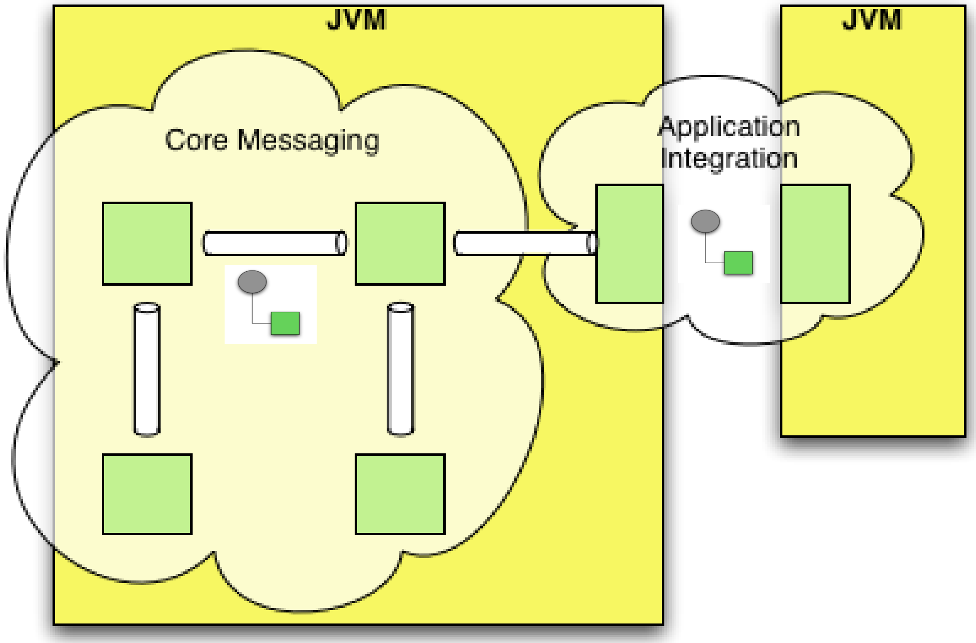
```
@Scheduled(cron="${schedules.daytime}")
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Daytime {
}
```

```
@Daytime
public void poll() {
  // poll and send Files to a handler
}
```

```
<context:property-placeholder
        location="/example/scheduling.properties"/>
```

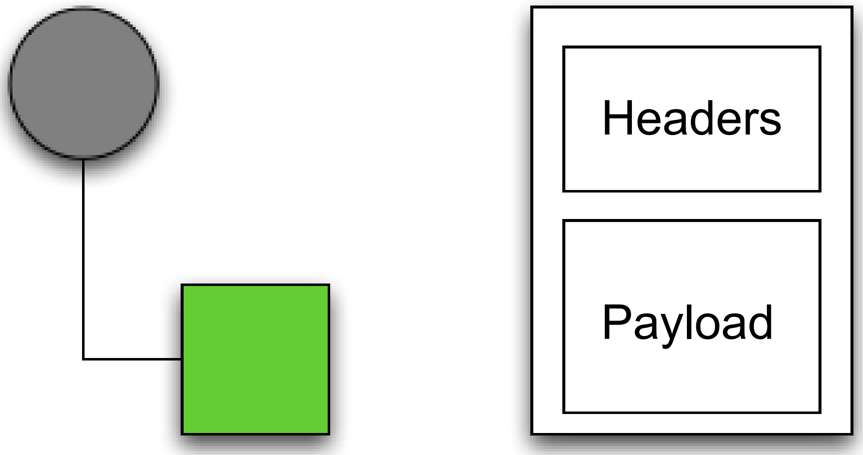
```
schedules.daytime=*/5 * 9-17 * * ?
```

Pipes and Filters



Message

- Payload can be any object
- Header values are stored in a Map



Message and Headers

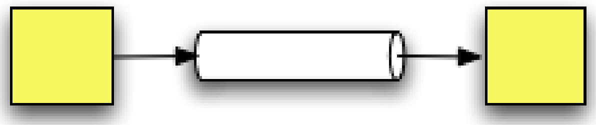
```
public interface Message<T> {

    MessageHeaders getHeaders();

    T getPayload();

}
```

```
Message<String> m1 = MessageBuilder.withPayload(“foo”)
        .setHeader(“itemId”, 123).build();

Message<String> m2 = MessageBuilder.fromMessage(m1)
        .setHeader(“itemId”, 456).build();
```

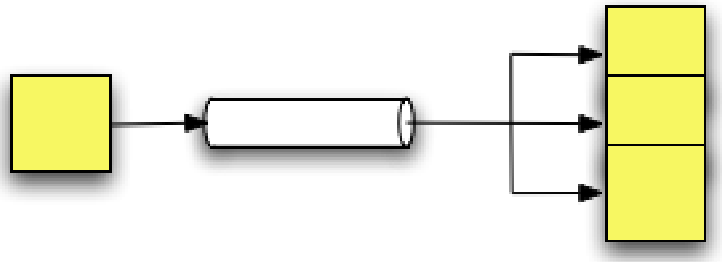
```
MessageHeaders headers = message.getHeaders();

long timestamp = headers.getTimestamp();

String value = headers.get(“someKey”, String.class);
```

Message Channel

- Decouples Producers from Consumers
- Provides extension point for interceptors
- May be ***Point-to-Point***



- Or ***Publish/Subscribe***



Message Channel Types

```
<channel id="sync-p2p"/>

<channel id="async-p2p">
   <dispatcher task-executor="someThreadPool" />
</channel>


<channel id="async-buffering-p2p">
   <queue capacity="50" />
</channel>


<publish-subscribe-channel id="sync-pubsub" />


<publish-subscribe-channel id="async-pubsub"
                           task-executor="someThreadPool" />
```

Sending Messages

```
public interface MessageChannel {

     boolean send(Message<?> message);

     boolean send(Message<?> message, long timeout);

}
```

```
MessagingTemplate template = new MessagingTemplate();

template.send(someChannel, message);

template.send("fooChannel", message);

template.convertAndSend(someChannel, "hello");

template.convertAndSend("fooChannel", someObject);


template.setSendTimeout(5000);
template.setDefaultChannel(someChannel);
template.convertAndSend(someObject);
```

Gateway Proxy

```java
public interface MyGateway {

  void send(String text);


  String send(@Payload Foo foo, @Header(“bar”) String s);

}
```

```xml
<gateway id="gateway"
            service-interface="example.MyGateway"
            default-request-channel="someChannel" />
```

Message Publishing Interceptor

- Non-invasive, AOP-based implementation
- Uses SpEL for generating payload and headers

```
@Publisher(channel="confirmations") // payload = #return
public String createBooking(BookingRequest request) {
    ...
}

@Publisher(payload="#args.bookingId", channel="cancellations")
public void cancelBooking(String bookingId) {
    ...
}
```

@Publisher as a Meta-Annotation

- Define a custom annotation

```
@Publisher(channel="auditChannel")
public @interface Audit {}
```

- Apply to methods, no channel required

```
@Audit
public User createUser(String name) {...}

@Audit
public User deleteUser(long userId) {...}
```

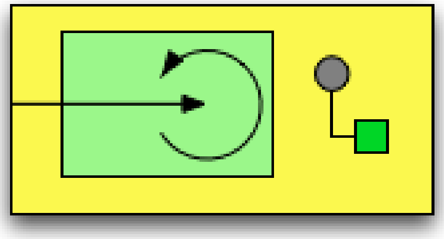
Receiving Messages

- Inversion of Control
  - Endpoints delegate to Spring-managed objects
  - Framework handles message reception and method invocation (including conversion)
  - Similar but more abstract than Spring JMS
- Clean separation of Code and Configuration

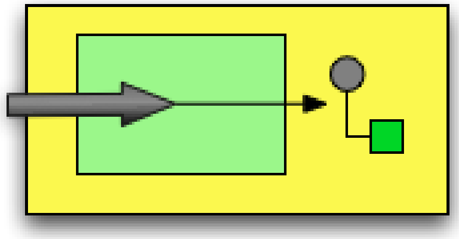
```
<service-activator input-channel="requests"
                   ref="loanBroker"
                   method="processRequest"
                   output-channel="quotes"/>
```

Message Endpoint

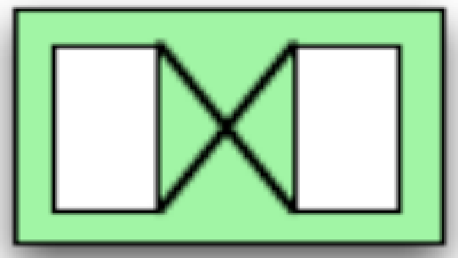
- Producers send Messages to a Message Channel
- Depending on their type, Message Channels may have ***Polling Consumers***

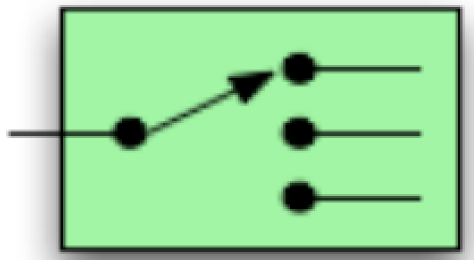


- Or ***Event-Driven Consumers***



Enterprise Integration Patterns





Message Endpoint Types

- Transformer
  - Convert payload or modify headers
- Filter
  - Discard messages based on boolean evaluation
- Router
  - Determine next channel based on content
- Splitter
  - Generate multiple messages from one
- Aggregator
  - Assemble a single message from multiple

Filtering and Routing

- Filter returns a boolean

```
<filter input-channel="customers"
        ref="customerRegistry"
        method="isVip"
        output-channel="vips"
        discard-channel="nonVips"/>
```

- Router returns a channel name (or map key)

```
<router input-channel="customers"
        ref="customerRegistry"
        method="getStatus">
   <mapping value="1" channel="platinum"/>
</router>
```

- Other routers included out of the box:

   recipient-list, payload-type, header-value, xpath, ...

Splitting and Aggregating

- Splitter returns a Collection or Array

```
<splitter input-channel="orders"
          ref="orderRepository"
          method="getLineItems"
          output-channel="lineItems"/>
```

- Aggregator accepts a Collection or Array

```
<aggregator input-channel="processedItems"
            ref="orderRepository"
            method="generateConfirmation"
            output-channel="confirmations"/>
```

- Default Splitter and Aggregator require no ref/method
- Aggregator also has ReleaseStrategy and CorrelationStrategy

Annotation Support

- Alternative to XML

```
@ServiceActivator(inputChannel="accounts")
public void createAccount(Account account) {…}

@Filter(inputChannel="customers", outputChannel="vips")
public boolean isVip(Customer customer) {…}

@Splitter(inputChannel="orders", outputChannel="lineItems")
public List<LineItem> getLineItems(Order order) {…}
```

Expression Language Support

- Alternative option for ref/method in endpoints

```
<filter input-channel="customers"
        expression="payload.vip"
        output-channel="vips"
        discard-channel="nonVips"/>
```

- Mapping between Messages and Methods

```
public void createAccount(
 @Payload("firstName") String firstName,
 @Payload("lastName") String lastName,
 @Header("userInfo.account.id") String accountId) {...}
```

Groovy Support

- Another option for endpoints

```
<router input-channel="customers"
  <groovy:script location="file:/scripts/router.groovy"
                 refresh-check-delay="30000"/>
</router>
```

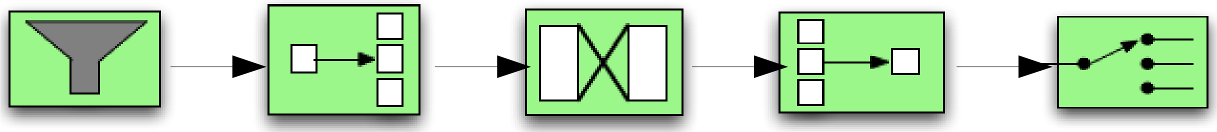
- Script has access to Message:

```
payload.isVip || headers.vip ? 'vips' : 'nonVips'
```

Chain

- When explicit channels are not necessary

```
<chain input-channel="orders">
  <filter expression="@orderValidator.isValid(payload)"/>
  <splitter/>
  <transformer ref="orderItemEnricher"/>
  <aggregator/>
  <header-value-router header-name="vendorId"/>
</chain>
```



Channel Adapters and Messaging Gateways

- JMS
- AMQP
- TCP
- UDP
- File/FTP/SFTP
- RMI
- RSS
- Redis
- HTTP (REST)
- WS (SOAP/POX)
- Mail (POP3/IMAP/SMTP)
- JDBC
- Twitter
- XMPP
- SMPP
- Spring Events

Channel Adapters (one-way)

```
<file:inbound-channel-adapter channel="fromFile"
                              directory="${java.io.tmpdir}/input"
                             filename-pattern="[a-z]+.txt">
    <si:poller fixed-delay="5000"/>
</file:inbound-channel-adapter>


<jms:outbound-channel-adapter channel="toJms"
                              destination="exampleQueue"/>
```

Gateways (request-reply)

```
<http:outbound-gateway request-channel="httpRequests"
            url="http://trafficexample.org/{zipCode}">
   <http:uri-variable name="zipCode"
                      expression="payload.address.zip"
</http:outbound-gateway>


<ws:outbound-gateway request-channel="weatherRequests"
              uri="http://weatherexample.org"
             marshaller="jaxb2Marshaller"/>
```

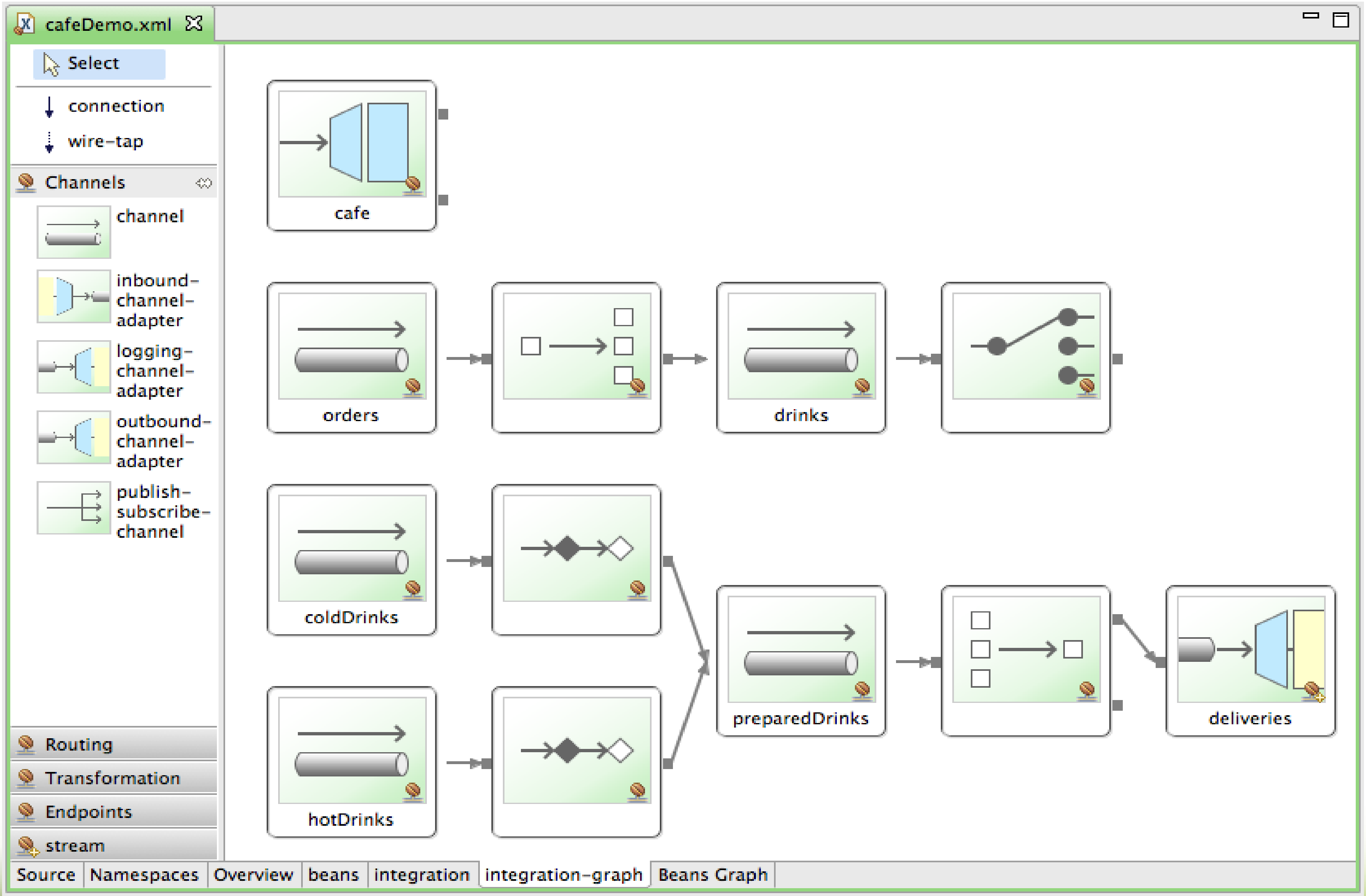
XML Support

- XPath Router
- XPath Splitter
- XPath Transformer
- XPath Header Enricher
- XSLT Transformer
- OXM Marshalling Transformer
- OXM Unmarshalling Transformer

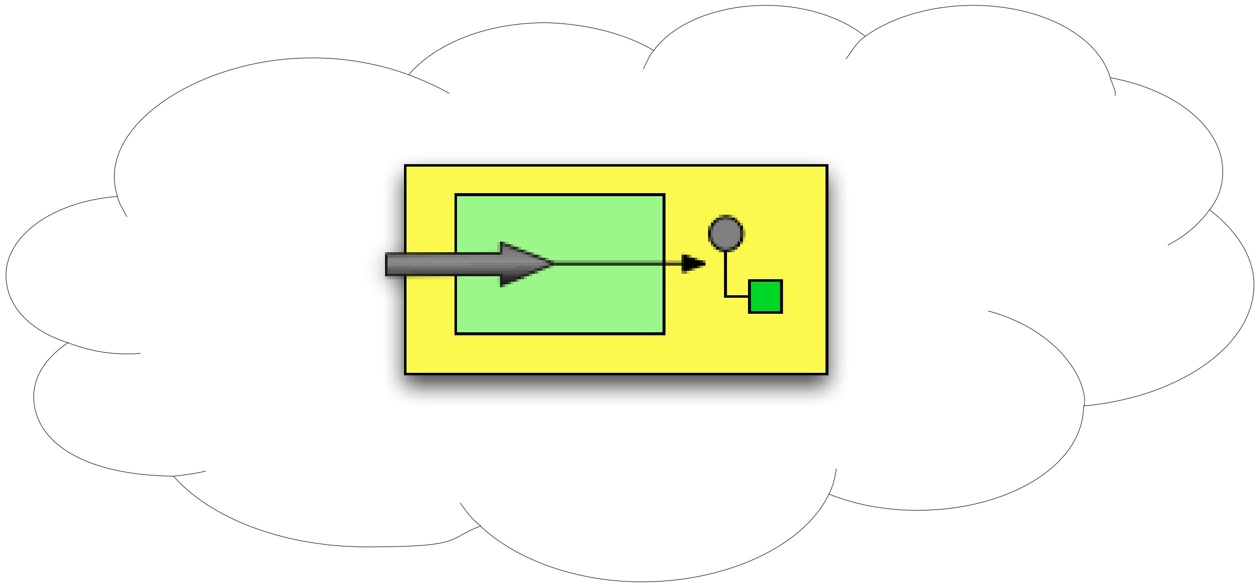
Spring Integration and Other Spring Projects

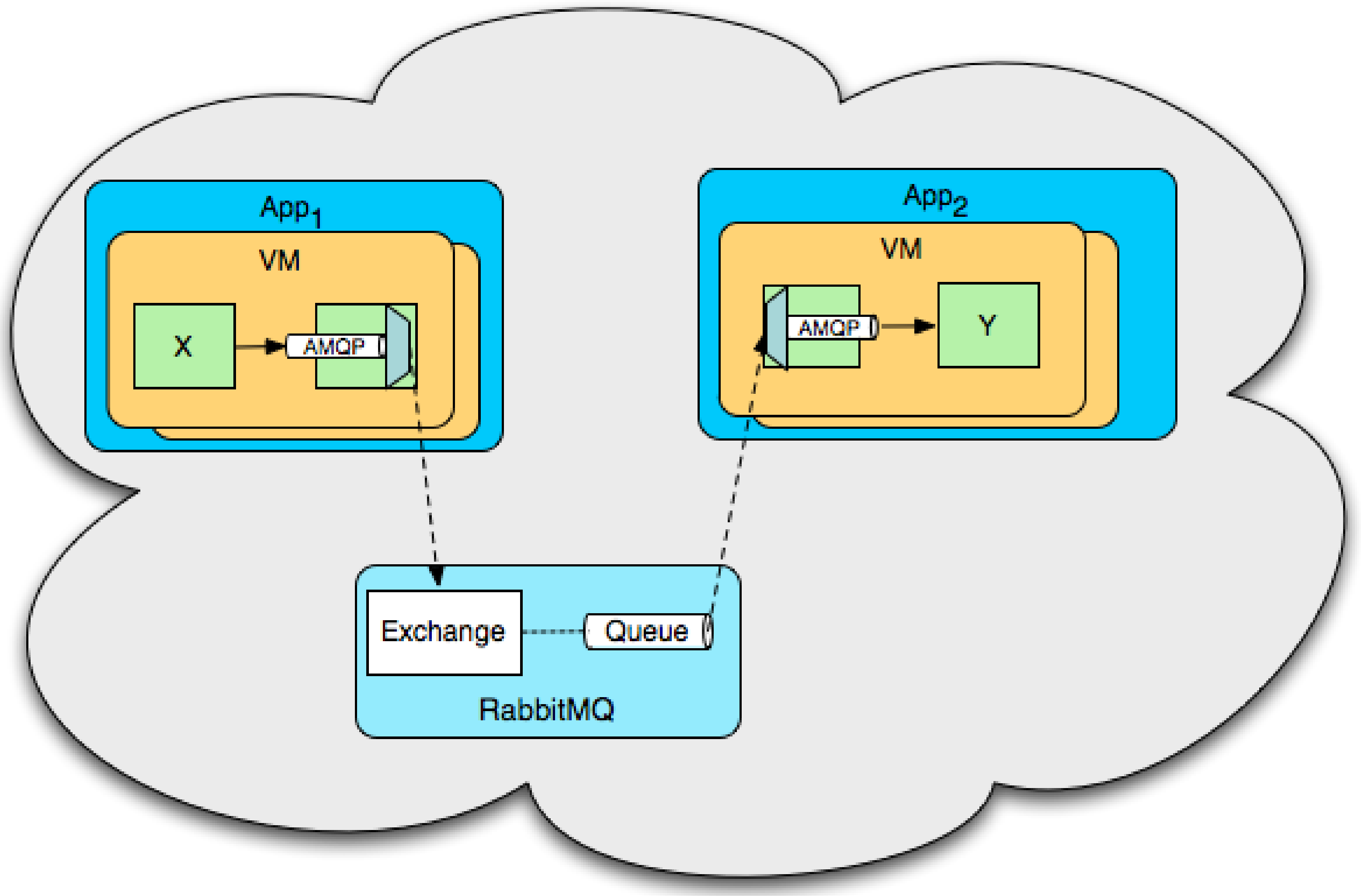
- Spring Integration for .NET
- Spring AMQP: Channel Adapters
- Spring BlazeDS (Flex): Messaging Adapter
- Spring Batch: Jobs ↔ Events
- Spring Web Services: Gateways
- Spring Security: Channel Interceptor
- Spring Roo: Addon
- SpringSource Tool Suite
- Cloud Foundry

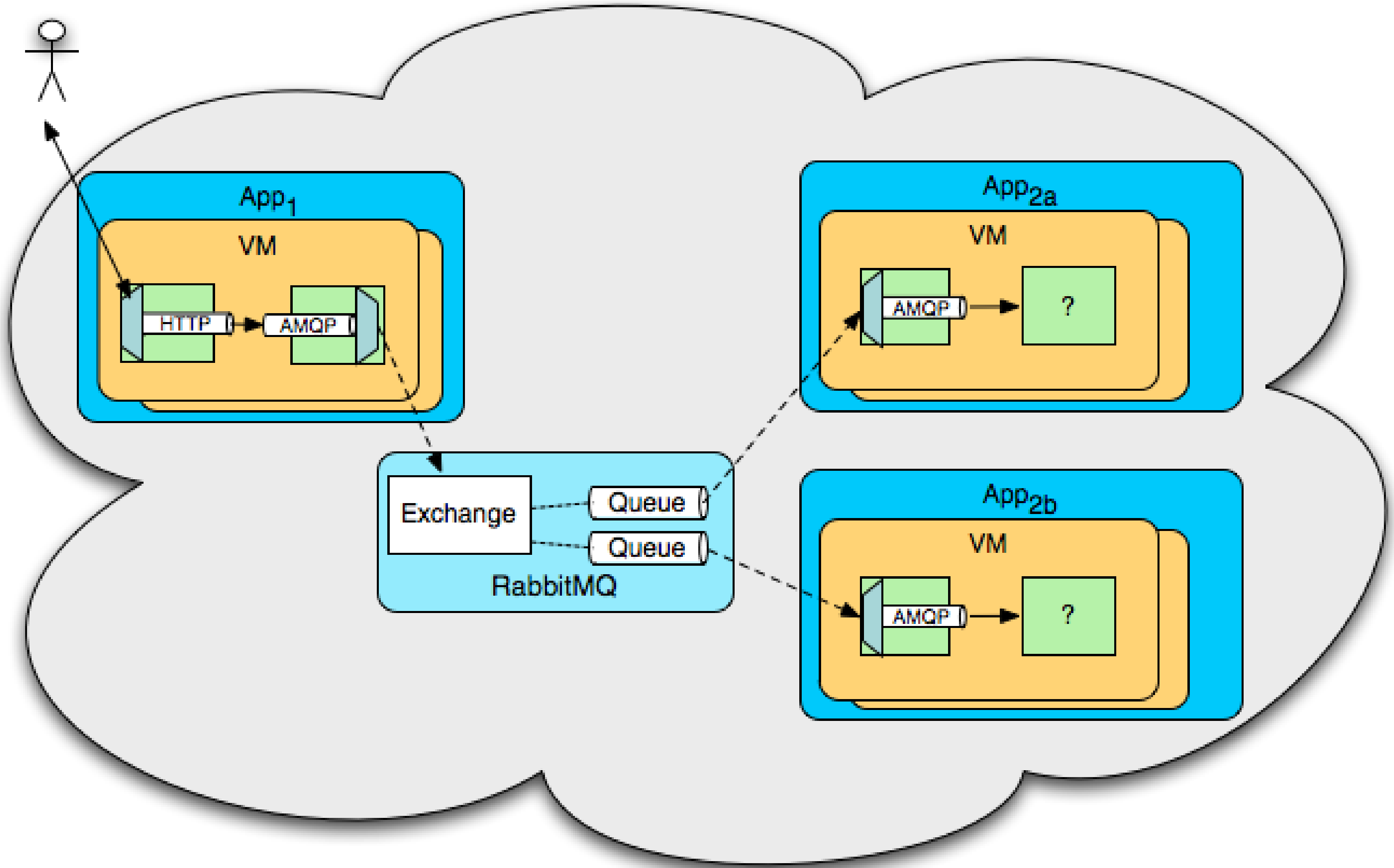
STS Visual Editor



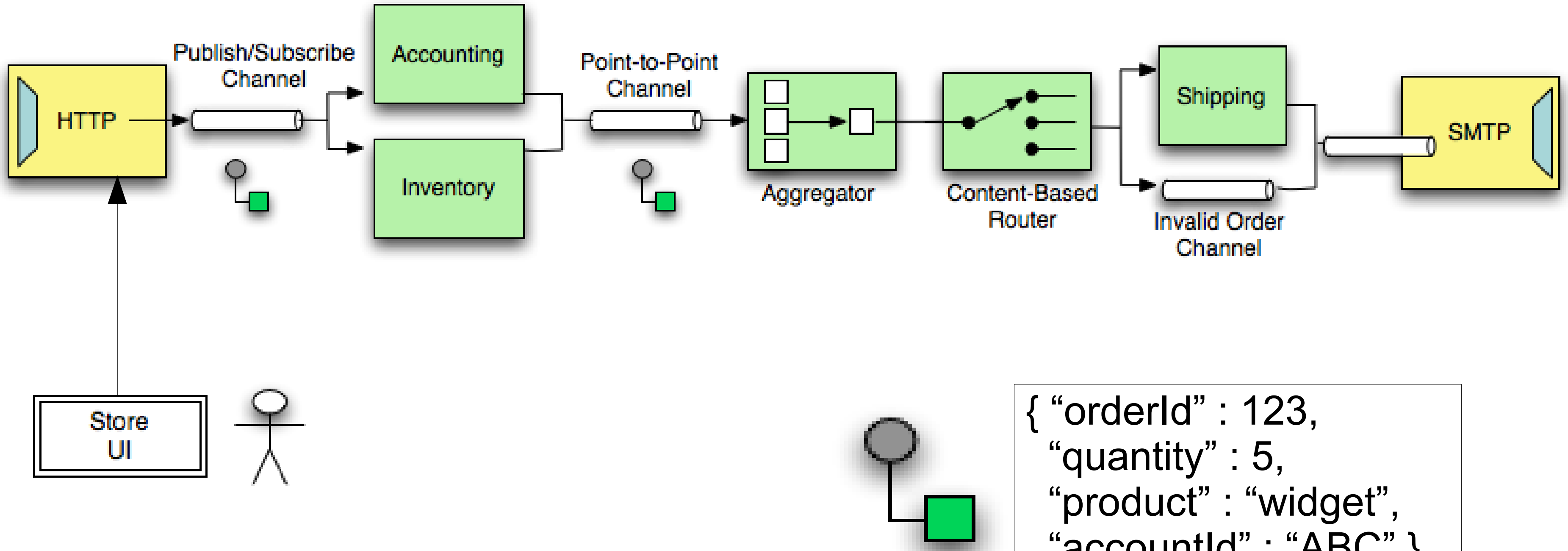
Messaging in the Cloud




App1
VM
X
AMQP
App2
VM
AMQP
Y
Exchange
Queue
RabbitMQ

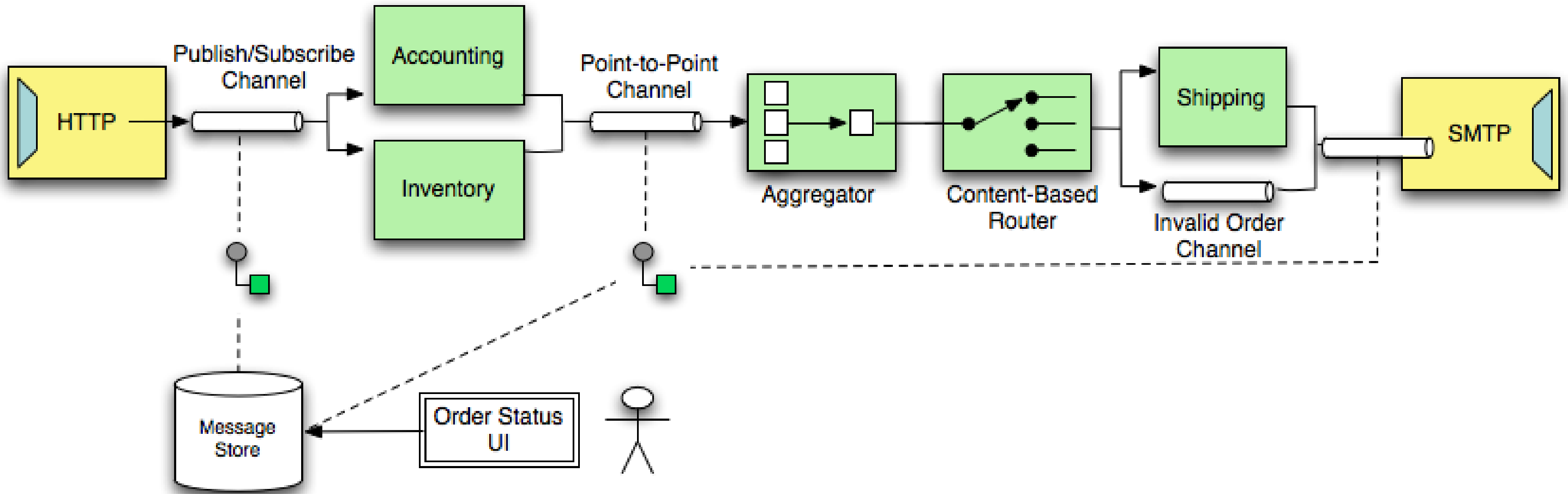

App1
VM
HTTP
AMQP
Exchange
Queue
Queue
RabbitMQ
App2a
VM
AMQP
?
App2b
VM
AMQP
?

Sample Application: WGRUS

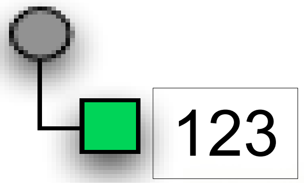


{ “orderId” : 123,
“quantity” : 5,
“product” : “widget”,
“accountId” : “ABC” }

Status Checking with Message Store



```
{ “orderId” : 123,
  “quantity” : 5,
  “product” : “widget”,
  “accountId” : “ABC” }
```



Links

- Spring Framework Documentation
  - http://static.springsource.org/spring/docs/3.0.x
- Spring AMQP
  - http://www.springsource.org/spring-amqp
- Spring Integration
  - http://www.springsource.org/spring-integration
- Enterprise Integration Patterns
  - http://enterpriseintegrationpatterns.com
- Sample Code
  - http://github.com/SpringSource/cloudfoundry-samples
  - http://git.springsource.org/spring-integration/samples