Enterprise Integration Patterns in dem Praxis mit Apache Camel und Spring Integration

Anders Malmborg und Michael Haslgrübler

Einleitung

Derzeit zeichnet sich ein Trend weg von monolithischen in Richtung modularen Systemen ab, wurde bisher der gesamte Anwendungslogik von einem System zur Verfügung gestellt, werden zunehmend fachliche Teilprozesse in sowohl Architektur, Entwicklung und Betrieb geteilt betrachtet. Für den Endanwender erscheint es als ein Produkt, die "Nahtstellen" sind nicht sichtbar. Mit [Microservices] [microsvc] haben einige Unternehmen wie Amazon oder Netflix für Aufmerksamkeit gesorgt. Diese modularen Systeme versprechen Vorteile vor allem in der Skalierbarkeit und Unabhängigkeit bei der Entwicklung der Teilprodukte.

Obwohl Microservices es nicht zwangsweise benötigten, setzt sich auch ein weiteres Architekturmuster für die Skalierbarkeit durch, nämlich die [ereignisbasierte Architektur] [ea] wo Verarbeitungsschritte asynchron und parallel erfolgen.

Mit oder ohne modularer und/oder ereignisbasierter Architektur kommt man heutzutage an der Integration mit externen Systemen nicht vorbei, es werden synchrone Schnittstellen über WebServices als auch der asynchrone Datenaustausch in diversen Produkten nicht mehr wegzudenken, z.B.: Login via Facebook, Rechnungsmeldung zum Finanzamt, ...

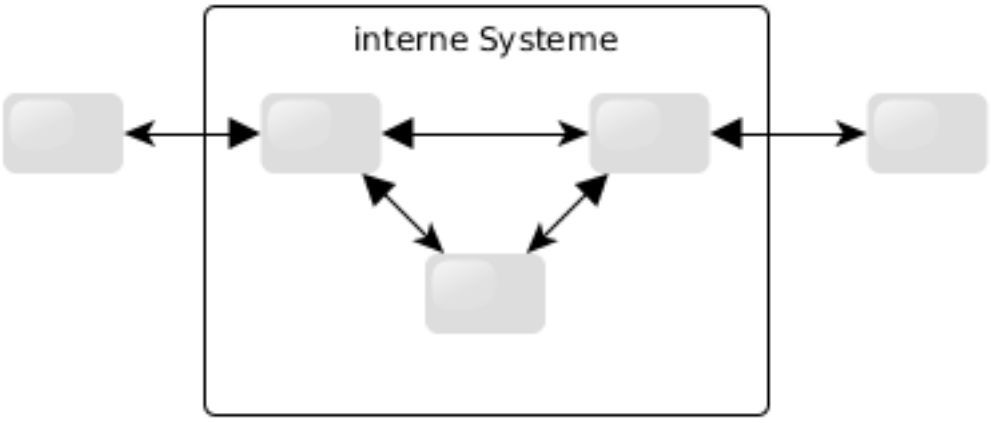

Abbildung 1: Interne und externe Systeme

Für die Interaktion zwischen internen als auch externen Systemem findet man im Buch [Enterprise Integration Patterns] [eip] Lösungsmustern, welche für eine Vielzahl von Problemen die durch diese Interaktion entstehen angewandt werden. Diese Enterprise Integration Patterns wurden in den Frameworks [Apache Camel] [camel] und [Spring Inte-

gration] [si] umgesetzt und werden in diesem Artikel anhand eines Praxisbeispiels erklärt und durchleuchtet.

Anhand des Fahrradshop, wird gezeigt wie man beide Frameworks einsetzen kann und typische Integrationsszenarien lösen kann.

Enterprise Integration Patterns

Die wichtigsten und häufigsten Integrationsmustern sind folgende 4 Arten (Styles): File Transfer, Shared Database, Remote Procedure Invocation/Call (RPC) und Messaging.

Im fiktiven Fahrradshop werden drei der vier Integrationsmuster (File Transfer, Remote Procedure Invocation und Messaging) anhand vom praxisorientierten Problemen eingesetzt.

File Transfer und Shared Database

File Transfers sind relativ einfach zu implementieren, bietet aber keine Möglichkeit synchrone Kommunikation damit zu implementieren.

Shared Database kann für einfache Fälle überlegt werden. Mit wachsenden Anzahl der beteiligten Systeme wird die enge Kopplung über Datenbank aber zu Problemen führen und es eignet sich außerdem oft nicht für die Kommunikation über Unternehmensgrenzen hinweg.

Remote Procedure Invocation und Messaging

Mit Remote Procedure Invocation und Messaging stehen anderseits skalierbare Lösungen für synchrone bzw. asynchrone Kommunikation zur Verfügung. Durch sogenannte Message Channels wird die Kommunikationsart abstrahiert und von der fachlichen Implementierung abgekoppelt. Das heißt das sofern sich die Anforderung ändern auch die Möglichkeit besteht auf Konfigurationsbasis den Kommunikationsmechansismus zu ändern ohne sein Produkt neu entwickeln zu müssen.

Messaging wird verwendet um zuverlässig und zeitnah Daten zwischen den Services auszutauschen. Bei asynchronen Schnittstellen werden die Daten über einen kürzeren oder längeren Zeitraum zwischengespeichert. Damit wird das Gesamtsystem fehlertoleranter - da nicht alle Services gleichzeitig zur Verfügung stehen müssen.

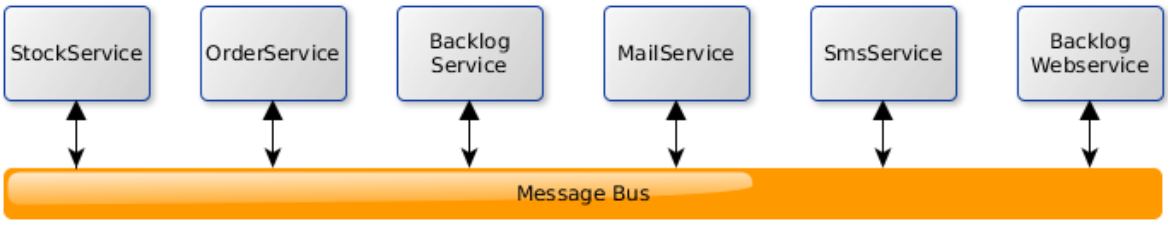

Abbildung 2: Messaging

Messages werden über Channels geschickt. Der Absender und Empfänger verbinden sich mit dem Channel. Falls es ein Absender und genau einen Empfänger gibt wird ein Point-to-Point Channel verwendet, ansonsten, bei mehreren Empfänger, für einen Message bietet sich der Publish-Subscribe Channel an.

Der Empfänger kann auch zur Laufzeit entschieden werden. In unserem Beispiel sollte nach der Verarbeitung des Auftrages eine Nachricht an den Kunden verschickt werden. Je nach Kunde sollte entweder eine SMS oder eine Mail verschickt werden. Die Entscheidung ob das Message zum SmsService oder MailService weitergereicht werden sollte, trifft hier ein Content-Based Router.

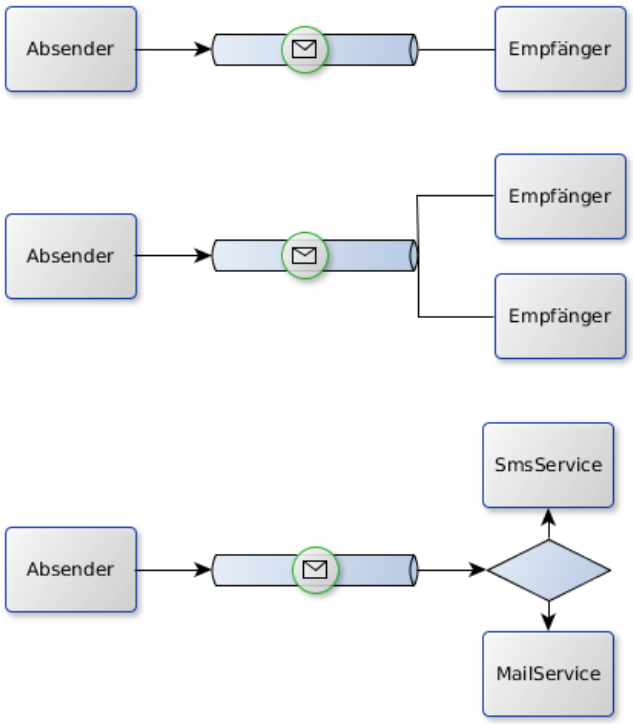

Abbildung 3: Channels

Nach diesem sehr gekürzten Übersicht von Enterprise Integration Patterns stellen wir jetzt die Anwendung vor.

Architekturmodell des Shops

Bei einer Bestellung wird zuerst geprüft ob eine ausreichende Menge des gewünschten Artikels im Lager des Fahrradshops vorhanden ist. Im Lager vorhandene Artikel werden ausgebucht und die fehlende Menge wird nachbestellt. Die Lieferanten senden Lieferscheine, die als Lagereingänge behandelt werden.

Das Architekturmodell in Kürze:

- StockService: verwaltet Lagerbestände. Lieferscheine von externen Lieferanten werden als CVS-Dateien eingespielt.

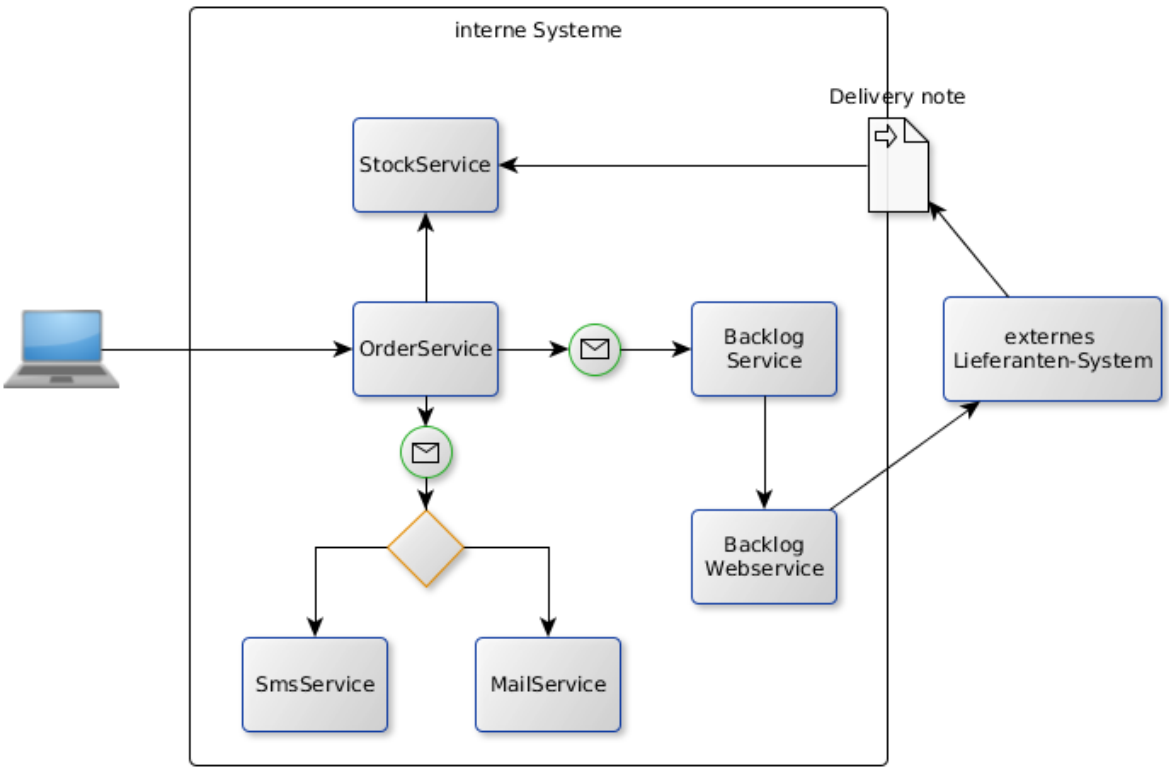

Abbildung 4: EIP im Fahrradshop

- OrderService: verwaltet Bestellungen. Stellt REST basierte Schnittstellen für eine Browserapplikation zur Verfügung.
- BacklogService: bestellt auf Lager fehlende Artikeln bei externen Lieferanten die SOAP Webservices dafür zur Verfügung stellen.
- Sms- bzw. MailService: sendet Bestellbestätigungen an den Kunden über SMS oder Mail.

Die Services werden so entwickelt, dass keine enge Kopplung besteht. Sie können damit in separaten Laufzeitumgebungen auf unterschiedlichen Servern betrieben werden.

Use Cases

CSV Import von Lieferscheine Die Lieferscheine werden als CSV Dateien aus einem Verzeichnis gelesen. Eine Zeile beinhaltet Typ, Bezeichnung, Teilenummer, Einkaufspreis sowie Anzahl gelieferte Teile:

```
1 FRAME; Road bike frame 60 cm;1935182366;103.95;2
2 DRIVE; Shimano HG LX;1935182439;31.85;6
```

Die Verarbeitungsschritte:

1. Zeilenweise einlesen
2. In Typ, Bezeichnung u.s.w aufteilen
3. Lagerbestände korrigieren

Bestellabwicklung Der Kunde stellt eine Warenkorb im Browser zusammen. Der Kunde sollte nach der Bestellannahme eine Bestätigung erhalten. In einer Systemkonfiguration ist hinterlegt ob der Kunde mittels SMS oder Mail die Bestätigung erhalten soll. Senden der Bestätigung sollte asynchron von der Verarbeitung stattfinden da die Bestätigung nicht so hohe Priorität wie (neue) Bestellungen hat.

1. Prüfen ob bestellte Artikeln auf Lager sind:
   - falls nicht einen Bestellwunsch erzeugen
   - Andernfalls den Lagerbestand reduzieren
2. Bestätigung senden

Nachschub fehlende Artikeln Sofern bestellte Artikeln nicht lagernd sind, werden sie mittels Webservices von externen Lieferanten nachbestellt.

Umsetzung mit Spring Integration und Channel

Beide Frameworks unterstützen die Mustern von Enterprise Integration Patterns und beziehen sich auch auf das Buch in der Dokumentation. Fallweise werden andere Namen verwendet, Spring Integration verwendet Beispielsweise Direct Channel statt Point-to-Point Channel.

Für die asynchrone Austausch von Nachrichten verwenden wir [Apache ActiveMQ] [amq]. Beide Frameworks integrieren problemlos mit ActiveMQ.

Anmerkung zu der Konfiguration von Spring Integration und Apache Camel

Spring Integration unterstützt "klassische" Spring XML Konfiguration als auch Annotationen. Apache Camel unterstützt Spring Konfiguration mit oder ohne Annotationen. Weiter kann Camel mit Java DSL statt oder zusätzlich zur Spring XML Konfiguration verwendet werden. Zwecks Vergleichbarkeit der beiden Frameworks wird hier immer nur Spring XML Konfiguration verwendet.

Java Service Implementierungen

Die Implementierungen der Services sind völlig unwissend von Spring Integration bzw. Apache Camel. Hier ein Teil der Implementierung von OrderService:

public Backlog handleOrder(Order order) {

```
1     List<BacklogItem> backlogItems = new ArrayList<BacklogItem
          >();
2     Customer customer = customerRepository.findByName(order.
          getCustomerName());
3     if (customer == null)
```

```
4       {
5           customer = new Customer(order.getCustomerName());
6       }
7       customer.getOrders().add(order);
8       for (OrderItem orderItem : order.getOrderItems()) {
9           orderItem.setStatus(OrderItemStatus.BACKLOG);
10          StockItem stockItem = stockService.getStockItem(
                orderItem.getItem().getNumber());
11          if (stockItem != null)
12          {
13              if (stockItem.getQuantity() > 0) {
14                  orderItem.setStatus(OrderItemStatus.CHECKED_OUT
                        );
15                  stockService.checkoutStockItem(stockItem);
16                  continue;
17              }
18          }
19          backlogItems.add(new BacklogItem(new Item(orderItem.
                getItem())));
20      }
21      customerRepository.save(customer);
22      return new Backlog(backlogItems);
23  }
```

In der Implementierung ist zu sehen, dass keine Verbindung mit dem BacklogService besteht.

CSV Import von Lieferscheine mit Spring Integration Da in der Spring Familie ausgereifte Funktionalität für CSV Verarbeitung in Form vom [Spring Batch] [sb] vorhanden ist, gibt es keine eigene Implementierung für CSV in Spring Integration. Die Konfiguration für Spring Batch

```
1 <bean id="deliveryNoteCsvReader" class="org.springframework.
      batch.item.file.FlatFileItemReader"
2       scope="prototype">
3       <property name="encoding" value="UTF-8" />
4       <property name="lineMapper">
5           <bean class="org.springframework.batch.item.file.
                mapping.DefaultLineMapper">
6               <property name="lineTokenizer">
7                   <bean
8                       class="org.springframework.batch.item.file.
                            transform.DelimitedLineTokenizer">
9                       <property name="delimiter" value=";" />
10                      <property name="names" value="itemType,name
                            ,number,price,quantity" />
11                  </bean>
12              </property>
13              <property name="fieldSetMapper">
```

```
14                    <bean class="eip.spring.integration.
                          StockItemFieldSetMapper" />
15                </property>
16            </bean>
17        </property>
18    </bean>
```

- DelimitedLineTokenizer: teilt jede Zeile in einzelne Felder.
- DefaultLineMapper: lässt den DelimitedLineTokenizer die Zeile zerlegen und StockItemFieldSetMapper daraus einen Objekt erzeugen
- FlatFileItemReader: liest die CSV Datei zeilenweise.

Da jetzt Spring Batch so weit konfiguriert ist, folgt nun Spring Integration. Der *inbound-channel-adapter* überwacht einen Verzeichnis, falls Dateien gefunden werden, werden sie auf die Reise auf den Channel *csvDeliveryNotesChannel* geschickt.

```
1 <int-file:inbound-channel-adapter id="
      deliveryNotesChannelAdapter"
2       directory="file:../eip-common/src/main/resources/
            deliverynotes" channel="csvDeliveryNotesChannel">
3       <int:poller fixed-rate="1000" />
4 </int-file:inbound-channel-adapter>
```

Ein ßervice-activator* horcht auf den *csvDeliveryNotesChannel* und leitet die Messages(hier Dateien) an den Spring Bean *deliveryNoteCsvImport* weiter.

```
1 <int:service-activator input-channel="csvDeliveryNotesChannel"
2       ref="deliveryNoteCsvImport" method="importCsv" />
```

Die Spring Konfiguration der verwendete Beans:

```
1 <bean id="deliveryNoteCsvImport" class="eip.spring.integration.
      DeliveryNoteCsvImport"
2       scope="prototype">
3       <constructor-arg name="reader" ref="deliveryNoteCsvReader"
            />
4       <constructor-arg name="writer" ref="stockWriterAdapter" />
5 </bean>
6
7 <bean id="stockWriterAdapter"
8       class="org.springframework.batch.item.adapter.
            ItemWriterAdapter">
9       <property name="targetObject" ref="stockService" />
10      <property name="targetMethod" value="addStockItem" />
11 </bean>
```

Die Java-Implementierung ist minimal, Spring Integration hier unterstützt von Spring Batch, muss lediglich konfiguriert werden.

Umsetzung mit Apache Camel Camel hat eine hohe Anzahl von Komponenten(Components). Diese werden in Form von URIs konfiguriert. Wir verwenden hier das File Component um den Verzeichnis für neue Dateien zu überwachen. Der Splitter teilt die Datei in einzelne Zeilen auf und zerlegt diese in Felder. Der Processor *csvToStockItemProcessor* erzeugt aus die Felder einen StockItem-Objekt. Zum Schluss werden die Objekte vom Spring Bean *stockService* auf der Datenbank gespeichert.

```
<camel:camelContext id="deliveryNoteImport">
    <camel:route>
        <camel:from uri="file://../eip-common/src/main/
            resources/deliverynotes?consumer.delay=1000&noop=
            true" />
        <camel:split streaming="true">
            <camel:tokenize token="\n" xml="false" />
            <camel:unmarshal>
                <camel:csv delimiter=";" />
            </camel:unmarshal>
            <camel:process ref="csvToStockItemProcessor" />
            <camel:bean ref="stockService" method="addStockItem
                "/>
        </camel:split>
    </camel:route>
</camel:camelContext>
```

Die Konfiguration für CSV Import ist bei Apache Camel um einiges kompakter.

Entkopplung der Bestellbestätigung mittels Apache ActiveMQ

Der Kunde sollte nach der Bestellannahme eine Bestätigung erhalten. In einer Systemkonfiguration ist hinterlegt ob der Kunde mittels SMS oder Mail die Bestätigung erhalten soll. Senden der Bestätigung sollte asynchron von der Verarbeitung stattfinden da die Bestätigung nicht so hohe Priorität wie (neue) Bestellungen hat. Daher wird ActiveMQ als JMS Implementierung eingesetzt und dient als Entkopplung zwischen OrderService und Sms- bzw. MailService.

Weiter sollte der OrderService nicht mit der Entscheidung ob SMS oder Mail angebracht ist bzw. die dafür notwendige Parametern für SMS oder Mail Versand beschäftigt werden.

ActiveMQ wird für beide Implementierung gleich konfiguriert:

```
<amq:broker useJmx="false" persistent="false">
    <amq:transportConnectors>
        <amq:transportConnector uri="vm://localhost" />
    </amq:transportConnectors>
</amq:broker>
```

Umsetzung mit Spring Integration Zuerst wird die Bestellbestätigung *Notification* in entweder einer *SmsNotification* oder einer *MailNotification* umgewandelt. Dafür wird einen *Transformer* implementiert:

```
1  public Message<?> transform(Message<?> message) {
2      Notification notification = (Notification) message.
           getPayload();
3      Notification outNotification;
4      if (customerConfig.getContactChannel(notification.
           getCustomer()).equals(SMS))
5          outNotification = new SmsNotification(notification.
               getCustomer(),
6                                                 notification.
                                                      getMessage(),
7                                                 customerConfig.
                                                      getSmsNumber(
                                                      notification.
                                                      getCustomer())
                                                      );
8      else
9          outNotification = new MailNotification(notification.
               getCustomer(),
10                                                 notification.
                                                      getMessage(),
11                                                 customerConfig.
                                                      getMailAddressNumber
                                                      (notification
                                                      .getCustomer
                                                      ()),
12                                                 customerConfig.
                                                      getMailSubject
                                                      (notification
                                                      .getCustomer
                                                      ()));
13     return MessageBuilder.withPayload(outNotification).build();
14 }
```

Die Spring Konfiguration schaut wie folgt aus:

```
1 <int:transformer id="notificationTransformer" input-channel="
    transformerChannel"
2     method="transform" output-channel="routingChannel">
3     <bean class="eip.spring.integration.NotificationTransformer
          " />
4 </int:transformer>
```

Das Versenden erfolgt durch den Sms- bzw. MailService. Dazu wird ein *Router* verwendet:

```
1 <int:payload-type-router input-channel="routingChannel">
2     <int:mapping type="eip.common.services.SmsNotification"
3         channel="smsOutQueue" />
4     <int:mapping type="eip.common.services.MailNotification"
5         channel="mailOutQueue" />
6 </int:payload-type-router>
```

Die Art der Messages, *SmsNotification* oder *MailNotification* entscheidet über den zu verwendenden Channel.

Zum Schluss die Konfiguration für die JMS Anbindung . hier für SMS:

```
<int-jms:outbound-channel-adapter id="smsJms"
    channel="smsOutQueue" destination="smsJmsQueue" />

<bean id="smsJmsQueue" class="org.apache.activemq.command.
    ActiveMQQueue">
    <constructor-arg value="queue.sms" />
</bean>

<int-jms:message-driven-channel-adapter
    id="smsIn" destination="smsJmsQueue" channel="smsInQueue"
        />
<int:channel id="smsInQueue" />

<int:service-activator input-channel="smsInQueue"
    ref="smsServiceMock" method="send" />
```

- jms:outbound-channel-adapter schiebt die Messages vom *smsOutQueue* zum *smsJmsQueue*.
- smsJmsQueue: definiert ein ActiveMQ Queue *queue.sms*.
- jms:message-driven-channel-adapter nimmt den Message vom ActiveMQ und gibt es an den *service-activator*.
- service-activator: ruft der Spring Bean auf mit dem Parameter SmsNotification.

Umsetzung mit Camel Wie beim Spring Integration wird zuerst die *Notification* in einen *SmsNotification* bzw. *MailNotification* umgewandelt. Mit Camel wird es als ein *Processor* implementiert:

```
public void process(final Exchange exchange) throws Exception {
    Notification notification = exchange.getIn()
            .getBody(Notification.class);
    if (notification.getContactChannel(notification.getCustomer
        ()).equals(SMS))
        exchange.getIn().setBody(
                new SmsNotification(notification.getCustomer(),
                                    notification.getMessage(),
                                    customerConfig.getSmsNumber
                                        (notification.
                                        getCustomer())));
    else
        exchange.getIn().setBody(
                new MailNotification(notification.getCustomer()
                    ,
                                     notification.getMessage(),
```

```
13                                                  customerConfig.
                                                        getMailAddressNumber(
                                                        notification.
                                                        getCustomer()),
14                                                  customerConfig.
                                                        getMailSubject(
                                                        notification.
                                                        getCustomer()));
15  }
```

Die Entscheidung wohin damit, fordert in Camel folgende Java Implementierung:

```
1  public String slip(final Notification notification,
2          @Properties Map<String, Object> properties) {
3      // End routing by returning null, otherwise endless loop
4      // The current endpoint is in the properties.
5      // First run - where routing should be done - it will be
           null
6      if (properties.get(Exchange.SLIP_ENDPOINT) != null) {
7          return null;
8      }
9      if (notification instanceof SmsNotification) {
10         return "activemq:sms";
11     } else if (notification instanceof MailNotification) {
12         return "activemq:mail";
13     }
14     return null;
15 }
```

Die Spring Konfiguration dazu:

```
1  <camel:camelContext id="activeMqTest">
2      <camel:proxy id="notificationService" serviceInterface="eip
           .common.services.NotificationService"
3          serviceUrl="direct:notification" />
4
5      <camel:route>
6          <camel:from uri="direct:notification"/>
7          <camel:bean ref="notificationEnricher"/>
8          <camel:dynamicRouter>
9              <camel:method ref="notificationRouter" method="slip
                   "/>
10         </camel:dynamicRouter>
11     </camel:route>
12     <camel:route>
13         <camel:from uri="activemq:sms" />
14         <camel:bean ref="smsServiceMock" />
15     </camel:route>
16     <camel:route>
17         <camel:from uri="activemq:mail" />
```

```
18          <camel:bean ref="mailServiceMock" />
19      </camel:route>
20  </camel:camelContext>
21
22  <bean id="notificationEnricher" class="eip.camel.
        NotificationEnricher"/>
23  <bean id="notificationRouter" class="eip.camel.
        NotificationRouter"/>
24
25  <bean id="smsServiceMock" factory-method="mock" class="org.
        mockito.Mockito">
26      <constructor-arg value="eip.common.services.SmsService" />
27  </bean>
28  <bean id="mailServiceMock" factory-method="mock" class="org.
        mockito.Mockito">
29      <constructor-arg value="eip.common.services.MailService" />
30  </bean>
```

Nachschub fehlende Artikeln mit SOAP Web Service

Sofern eine Bestellung nicht mit dem Lagerbestand abgedeckt werden kann, werden die Teile im Backlog abgelegt und es wird eine Bestellung beim Lieferanten durchgeführt. Die Bestellung sofern erfolgreich wird mit einer Bestellnummer quittiert. Beide SOAP Clients sowohl von Camel als auch Spring setzen auf eine Generierung mit Java Code auf. Die Basis für diese Generierung ist die Beschreibung des Web Services in der Web Service Description Language (WSDL).

Umsetzung mit Spring Integration Spring empfiehlt eine Code-Generierung mit dem Maven jaxb2-plugin, die WSDL-Datei des WebServices wird definiert und in welchen Package die generierten Paketen liegen sollen. Durch die Generierung stehen uns die Basisdatentypen für die Service Interaktion zur Verfügung. Die einzelnen Operation des WebService welche verwendet werden wollen, müssen explizit implementiert werden. Unser WebService Client leitet von der Klasse *WebServiceGatewaySupport* ab welche uns Basismethoden zur Interaktion zur Verfügung stellt. Die gewünschte Operation des WebService muss normalerweise definiert werden, da unser Service jedoch nur eine Operation zur Verfügung stellt, ist das hier nicht notwendig. Die Operations-Payload ist unserem Fall die Bestellung.

```
1  public class PartsOrderService extends
       WebServiceGatewaySupport {
2      public OrderResponse order(OrderRequest orderRequest) {
3          OrderResponse response = (OrderResponse)
               getWebServiceTemplate().marshalSendAndReceive(
               orderRequest);
4          return response;
5      };
6  }
```

Um den WebService Client letztendlich auch zu verwenden ist es notwendig zwei Spring-Beans zu definieren. Erstens einen Marshaller für die Verarbeitung von Java Objekten zu XML und vice versa und zweitens den Service selbst. Der Service benötigt für die Funktionsfähigkeit den Marshaller und die URI des WebService.

```
<bean id="marshaller" class="org.springframework.oxm.jaxb.
    Jaxb2Marshaller">
    <property name="contextPath" value="parts.eip" />
</bean>

<bean id="webserviceTemplate" class="parts.eip.
    PartsOrderService" >
    <property name="defaultUri" value="http://localhost:8080/
        eip-webservice-camel/partsOrder"></property>
    <property name="marshaller" ref="marshaller" />
    <property name="unmarshaller" ref="marshaller" />
</bean>
```

Umsetzung mit Camel Bei einer Umsetzung mit Camel kommt das Apache Framework für Open-Source Services kurz Apache CXF [cxf] zur Verwendung. Analog zu Spring Integration wird hier ein Maven Plugin für die Codegenerierung verwendet.

Der wesentliche Unterschied zu Spring Integration ist dass hier ein Java-Interface definiert wird was eine Java Beschreibung des WebServices ist und bereits die Operationen des WebService als Methoden definiert sind.

```
@WebService(targetNamespace = "http://eip.parts", name = "
    PartsOrder")
@XmlSeeAlso({ObjectFactory.class})
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
public interface PartsOrder {

    @WebResult(name = "OrderResponse", targetNamespace = "http
        ://eip.parts", partName = "OrderResponse")
    @WebMethod(operationName = "Order")
    public OrderResponse order(
        @WebParam(partName = "OrderRequest", name = "
            OrderRequest", targetNamespace = "http://eip.parts
            ")
        OrderRequest orderRequest
    );
}
```

Um den WebService schlussendlich zu verwenden ist es noch notwendig im Spring Context den WebService Client zu definieren. Hierbei wird das Java-Interface mit der URI des WebService verknüpft und kann nunmehr verwendet werden.

```
<jaxws:client id="partsOrderServiceClient" serviceName="
    partsOrderService" endpointName="partsOrderEndpoint"
    address="http://localhost:8080/eip-webservice/partsOrder"
    serviceClass="parts.eip.PartsOrder">
```

```
</jaxws:client>
```

Verwendung Bei der Einbindung von Fremdsystem sollte man beachten dass diese womöglich nicht verfügbar sind auch wenn die eigene Applikation zur Verfügung steht, dadurch ist es sinnvoll diese von einander zu entkoppeln. Da ansonsten die Verfügbarkeit der eigenen Applikation von dem Fremdsystem abhängt und wenn das Fremdsystem nicht zur Verfügung steht auch die eigene Applikation gar nicht oder nur eingeschränkt zur Verfügung steht. Der Scheduler vom Spring Framework bietet eine einfache Möglichkeit diese Entkopplung zu erreichen.

Mit der folgenden Spring Konfiguration Datei, wird der Scheduler definiert. Was vom Scheduler zu steuern ist wird über Annotationen direkt im Java Code gesteuert.

```
<task:annotation-driven scheduler="myScheduler"/>
<task:scheduler id="myScheduler" pool-size="1" />
```

Wir definieren in unserem Backlog Service eine Methode welche periodisch abgearbeitet wird, wobei erst nach einer Sekunde nachdem die Verarbeitung abgeschlossen ist eine neue Verarbeitung startet. Unsere Methode überprüft ob sich in unseren Backlog Element befinden die bestellt werden können. Sofern dies der Fall ist wird eine Bestellung getätigt, falls nicht bleibt das Element im Backlog enthalten.

```
@Scheduled(fixedDelay = 1000)
public void processBacklog() {
        if (getBacklogItems().size() > 0) {
            BacklogItem backlogItem = getBacklogItems().get(0);
            OrderResponse orderResponse = getSOAPWebServiceClient().order(toOrderRequest(backlogItem));
            if (orderResponse != null && isValidOrderNumber(orderResponse.getOrderNumberUuid())) {
              list.remove(backlogItem);
            }
        }
}
```

Fazit

Die Enterprise Integration Patterns definieren einen Katalog von Mustern für modulare Systeme und ereignisorientierte Architekturen. Damit können Systeme flexibler und skalierbar umgesetzt werden, was jedoch Komplexität und Mehraufwand zur Folge hat. Mit den beiden hier vorgestellten Frameworks Spring Integration und Camel kann man den Mehraufwand deutlich reduzieren und die Komplexität den Frameworks zum Teil überlassen.

Beide Frameworks sind ausgereift, gut dokumentiert und vielfältig erprobt.

Die Investition in ein solches Framework ist ab mittlerer Systemgröße zu empfehlen, einmal vorhanden und verstanden, werden Sie eine Vielzahl von Anwendungsfälle und Möglichkeiten entdecken und schätzen.

Referenzen

[microsvc]: http://martinfowler.com/articles/microservices.html “Microservices, Martin Fowler” [eip]: http://www.enterpriseintegrationpatterns.com/ “Enterprise Integration Patterns, Gregor Hohpe & Bobby Woolf” [ea]: http://en.wikipedia.org/wiki/Event-driven_architecture “Event-driven Architecture” [camel]: http://camel.apache.org/ “Apache Camel” [si]: http://projects.spring.io/spring-integration/ “Spring Integration” [amq]: http://activemq.apache.org/ “Apache ActiveMQ” [sb]: http://projects.spring.io/spring-batch/ “Spring Batch” [cxf]: http://cxf.apache.org/ “Apache CXF: An Open-Source Services Framework”