Zusammenfassung

Kein IT-System kommt ohne Interaktion mit anderen Systemen aus, sei es via Web-Services oder auch durch Import/Export von Dateien in CSV, XML oder sonstige Formaten.

Für die meisten Technologien gibt es entweder Unterstützung in Form von Standards wie JAX-WS, für Web-Services, oder Framework-Lösungen wie Spring Batch, für CSV Verarbeitung. Für die Interaktion zwischen und in IT-System beschreibt das Standardwerk [Enterprise Integration Patterns] [eip] Ansätze um Skalierung und Erweiterbarkeit zu gewährleisten.

Diese Enterprise Integration Patterns wurden in den Frameworks [Camel] [camel] und [Spring Integration] [si] umgesetzt und werden hier anhand eines Praxisbeispiels, eines Fahrradshops, erklärt und durchleuchtet.

Anhand des Fahrradshop, wird gezeigt wie man beide Frameworks einsetzen kann und typische Integrationsszenarieren lösen kann:

- Lieferscheinverarbeitung
- Lagerverwaltung
- Bestellsystem
- Benachrichtigungssystem

Domainmodell

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 Domainmodell in Kürze:

- OrderService: verwaltet Bestellungen.
- BacklogService: bestellt auf Lager fehlende Artikeln beim Lieferanten.
- StockService: verwaltet Lagerbestände.
- Sms- bzw. MailService: sendet Bestellbestätigungen an den Kunden über SMS oder Mail.

Use Cases

CSV Import von Bestellungen

Die Bestellaufträge werden als CSV Dateien aus einem Verzeichnis gelesen. Ein Auftrag besteht aus Auftrags-Kopf(ORDER) und Auftrags-Positionen(ITEM). Folgend ein Beispiel im CSV Format:

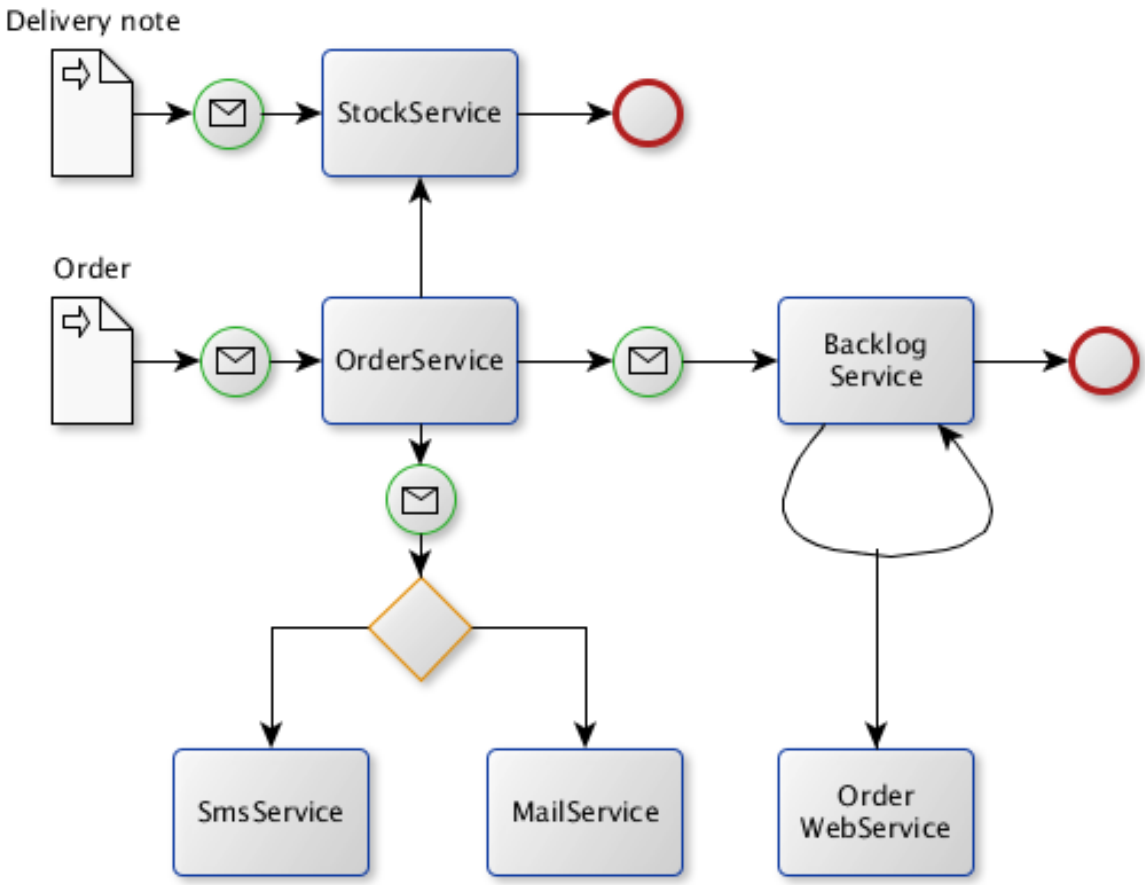

Figure 1: EIP im Fahrradshop

```
ORDER;Bike support;1
ITEM;FRAME;Road bike frame 60 cm;1935182366
ITEM;DRIVE;Shimano HG LX;1935182439
ORDER;Bike specialists;2
ITEM;WHEEL;Spoke 28 inches;098876
```

Die Verarbeitungsschritte:

1. CSV Datei lesen, in Einzelaufträge aufteilen (1 ORDER, n ITEM)
2. Prüfen ob bestellte Artikeln auf Lager sind:
   - falls nicht einen Bestellwunsch erzeugen
   - Andernfalls den Lagerbestand reduzieren

Eine weitere nicht funktionale Anforderung ist, dass der OrderService nicht den BacklogService "kennen" sollte. Diese lose Kopplung wird es ermöglichen die Systeme in die Zukunft getrennt zu betreiben.

Java Service Implementierungen

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

public Backlog handleOrder(Order order) {

```
    List<BacklogItem> backlogItems = new ArrayList<BacklogItem>();
    Customer customer = customerRepository.findByName(order.getCustomerName());
    if (customer == null)
    {
        customer = new Customer(order.getCustomerName());
    }
    customer.getOrders().add(order);
    for (OrderItem orderItem : order.getOrderItems()) {
        orderItem.setStatus(OrderItemStatus.BACKLOG);
        StockItem stockItem = stockService.getStockItem(orderItem.getItem().getNumber());
        if (stockItem != null)
        {
            if (stockItem.getQuantity() > 0) {
                orderItem.setStatus(OrderItemStatus.CHECKED_OUT);
                stockService.checkoutStockItem(stockItem);
                continue;
            }
        }
```

```
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.

Anmerkung zu der Konfiguration von Spring Integration und Camel

Spring Integration unterstützt "klassische" Spring XML Konfiguration als auch Annotationen. 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.

Umsetzung 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="orderCsvReader" class="eip.spring.integration.
       OrderFlatFileItemReaderDelegate"
2        scope="prototype">
3        <constructor-arg>
4            <bean class="org.springframework.batch.item.file.
                 FlatFileItemReader">
5                <property name="encoding" value="UTF-8" />
6                <property name="lineMapper">
7                    <bean
8                        class="org.springframework.batch.item.file.
                             mapping.DefaultLineMapper">
9                        <property name="lineTokenizer">
10                           <bean
11                               class="org.springframework.batch.
                                    item.file.transform.
                                    PatternMatchingCompositeLineTokenizer
                                    ">
12                               <property name="tokenizers">
13                                   <map>
14                                       <entry key="ORDER*">
15                                           <bean
16                                               class="org.
                                                    springframework.
                                                    batch.item.file.
                                                    transform.
```

```
                                                    DelimitedLineTokenizer
                                                    ">
17                                                  <property name="
                                                    delimiter" value
                                                    =";" />
18                                                  <property name="
                                                    names" value="
                                                    recType,
                                                    customerName,
                                                    orderNumber" />
19                                              </bean>
20                                          </entry>
21                                          <entry key="ITEM*">
22                                              <bean
23                                                  class="org.
                                                    springframework.
                                                    batch.item.file.
                                                    transform.
                                                    DelimitedLineTokenizer
                                                    ">
24                                                  <property name="
                                                    delimiter" value
                                                    =";" />
25                                                  <property name="
                                                    names" value="
                                                    recType,
                                                    itemType,name,
                                                    number" />
26                                              </bean>
27                                          </entry>
28                                      </map>
29                                  </property>
30                              </bean>
31                          </property>
32                          <property name="fieldSetMapper">
33                              <bean
34                                  class="org.springframework.batch.
                                    item.file.mapping.
                                    PassThroughFieldSetMapper" />
35                          </property>
36                      </bean>
37                  </property>
38              </bean>
39          </constructor-arg>
40  </bean>
```

- DelimitedLineTokenizer: teilt jede Zeile in einzelne Felder.
- PatternMatchingCompositeLineTokenizer: entscheidet auf Grund des Names(ORDER

oder ITEM) welcher DelimitedLineTokenizer zu verwenden ist.

- FlatFileItemReader: liest die CSV Datei zeilenweise.

Spring Batch benötigt ein wenig Hilfe da es sich um einen so genannten Multi-Line Records handelt. Die Implementierung dafür ist in OrderFlatFileItemReaderDelegate

```
public Order read() throws ... {
    FieldSet fieldSet = delegate.read();
    Order order = null;
    while (fieldSet != null) {
        if (nextOrder != null)
            order = nextOrder;
        String prefix = fieldSet.readString(0);
        if (prefix.equals("ORDER"))
        {
            if (order != null)
            {
                nextOrder = new Order(fieldSet.readString("
                    customerName"), fieldSet.readString("
                    orderNumber"));
                return order;
            }
            else
                order = new Order(fieldSet.readString("
                    customerName"), fieldSet.readString("
                    orderNumber"));
        }
        else if (prefix.equals("ITEM"))
        {
            Assert.notNull(order, "order must not be null");
            ItemType itemType;
            // Map ItemType excluded here
            Item item = new Item(itemType, fieldSet.readString
                ("name"), fieldSet.readString("number"));
            order.getOrderItems().add(new OrderItem(item));
        }
        else
            throw new ParseException("No record matching "+
                prefix);
        fieldSet = delegate.read();
    }
    return order;
}
```

Da jetzt Spring Batch so weit konfiguriert ist, folgt nun Spring Integration:

```
<int-file:inbound-channel-adapter id="orderChannelAdapter"
    directory="file:../eip-common/src/main/resources/orders"
        channel="csvOrderChannel">
    <int:poller fixed-rate="1000"/>
```

```
4 </int-file:inbound-channel-adapter>
5
6 <int:channel id="csvOrderChannel" />
7
8 <int:service-activator input-channel="csvOrderChannel"
9       ref="orderCsvImport" />
10
11 <bean id="orderCsvImport" class="eip.spring.integration.
      OrderCsvImport">
12      <constructor-arg name="reader" ref="orderCsvReader" />
13      <constructor-arg name="channel" ref="orderServiceChannel"
          />
14 </bean>
15
16 <int:channel id="orderServiceChannel" />
17 <int:chain input-channel="orderServiceChannel">
18      <int:service-activator ref="orderService" method="
          handleOrder"/>
19      <int:service-activator ref="backlogService" method="
          saveBacklogItems"/>
20 </int:chain>
```

- inbound-channel-adapter: überwacht ein Verzeichnis. Wenn eine Datei entdeckt wird, wird sie in den *csvOrderChannel* gesteckt.
- service-activator: nimmt einen Nachricht - hier eine Datei - aus *csvOrderChannel* und verarbeitet sie zeilenweise mittels den o.g. *orderCsvReader*. Das vom *orderCsvReader* erzeugte *Order* Objekt wird als Message ins *orderServiceChannel* übergeben
- chain: eine Kette wird hier verwendet um die Anzahl von expliziten input-/output-channels zu reduzieren. Eine *Order* wird aus *orderServiceChannel* genommen und verarbeitet und als Ergebnis wird ein *Backlog* Objekt erzeugt und den *BacklogService* weitergegeben.

Es ist zwar einiges an Konfiguration vorzunehmen, jedoch ist die Flexibilität gegenüber einer klassischen Java-Implementierung wesentlich höher.

Umsetzung mit Camel

Camel hat eine hohe Anzahl von Komponenten(Components). Diese werden in Form von URIs konfiguriert:

```
1 <camel:camelContext id="orderImport">
2      <camel:route>
3           <camel:from
4                uri="file://../eip-common/src/main/resources/orders
                     ?consumer.delay=1000&noop=true" />
5           <camel:split streaming="true">
```

```
            <camel:tokenize token="ORDER" xml="false" />
            <camel:unmarshal>
                <camel:csv delimiter=";" />
            </camel:unmarshal>
            <camel:process ref="csvToOrderProcessor"/>
            <camel:bean ref="orderService" />
            <camel:bean ref="backlogService"/>
        </camel:split>
    </camel:route>
</camel:camelContext>
```

- from: die File-Komponente liest vom Verzeichnis eine Datei.
- split: die Datei wird aufgeteilt in ORDER mit ITEMs.
- unmarshal: die CSV Komponente wird hier verwendet.
- process: der *CsvToOrderProcessor* erzeugt aus ORDER/ITEM Zeilen ein *Order* Objekt
- bean: *OrderService* verarbeitet die *Order* und erzeugt ein *Backlog* Objekt welches dann den *BacklogService* übergeben wird.

Die Einzige noch notwendige Implementierung ist der CsvToOrderProcessor:

```
public void process(Exchange exchange) throws Exception {
    String csvString = exchange.getIn().getBody(String.class);
    //[[, Bike support, 1], [ITEM, FRAME, Road bike frame 60 cm
       , 19351823 66], [ITEM, DRIVE, Shimano HG LX, 1935182439]]
    List<String> recs = Arrays.asList(csvString.split("\\],"));
    String orderString = recs.get(0).replace("[", "");
    orderString = orderString.replace("]", "");
    List<String> orderStrings = Arrays.asList(orderString.split
       (","));
    String customerName = orderStrings.get(1).trim();
    String orderNumber = orderStrings.get(2).trim();

    Set<OrderItem> orderItems = new HashSet<OrderItem>();
    for (int i = 1; i < recs.size(); i++) {
        orderString = recs.get(i).replace("[", "");
        orderString = orderString.replace("]", "");
        orderStrings = Arrays.asList(orderString.split(","));
        Assert.isTrue(orderStrings.size() == 4);
        ItemType itemType = ItemType.OTHER;
        ItemType itemType;
        // Map ItemType excluded here
        OrderItem orderItem = new OrderItem(new Item(itemType,
           orderStrings.get(2).trim(), orderStrings.get(3).trim
           ()));
        orderItems.add(orderItem);
```

```
22      }
23      Order order = new Order(customerName, orderNumber,
            orderItems);
24      exchange.getIn().setBody(order);
25  }
```

Auch hier gelingt es mit Spring Konfiguration und wenig Implementierung die Services zu verdrahten.

Entkopplung Bestellbestätigung mittels JMS

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:

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 (notification.getCustomer().equals("customerWithSms"))
5          outNotification = new SmsNotification(notification.
               getCustomer(),
6                  notification.getMessage(), "smsNumber");
7      else
8          outNotification = new MailNotification(notification.
               getCustomer(),
9                  "mailAddress", "mailSubject", notification.
                       getMessage());
10     return MessageBuilder.withPayload(outNotification).build();
11 }
```

Die Konfiguration ob der Kunde SMS oder Mail erhalten soll, ist hier der Einfachheit halber im Namen des Kunden enthalten.

Die Spring Konfiguration schaut wie folgt aus:

```
1 <int:channel id="transformerChannel" />
```

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

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

```
<int:channel id="routingChannel" />
<int:payload-type-router input-channel="routingChannel">
      <int:mapping type="eip.common.services.SmsNotification"
            channel="smsOutQueue" />
      <int:mapping type="eip.common.services.MailNotification"
            channel="mailOutQueue" />
</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:channel id="smsOutQueue" />
<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:poller id="poller" default="true" fixed-delay="1000" />

<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*.
- poller: definiert wie oft der Empfänger, der *jms:message-driven-channel-adapter* pollen soll.
- 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.getCustomer().equals("customerWithSms"))
        exchange.getIn().setBody(
                new SmsNotification(notification.getCustomer(),
                        notification.getMessage(), "smsNumber")
                        );
    else
        exchange.getIn().setBody(
                new MailNotification(notification.getCustomer()
                    ,
                        "mailAddress", "mailSubject",
                            notification
                                .getMessage()));
}
```

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

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

Die Spring Konfiguration dazu:

```
<camel:camelContext id="activeMqTest">
    <camel:proxy id="notificationService" serviceInterface="eip
        .common.services.NotificationService"
        serviceUrl="direct:notification" />

    <camel:route>
        <camel:from uri="direct:notification"/>
        <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>
```

Lieferantenbestellung 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.

```
1  <build>
2      <plugins>
3          <plugin>
4              <groupId>org.jvnet.jaxb2.maven2</groupId>
```

```
5               <artifactId>maven-jaxb2-plugin</artifactId>
6               <executions>
7                   <execution>
8                       <goals>
9                           <goal>generate</goal>
10                      </goals>
11                  </execution>
12              </executions>
13              <configuration>
14                  <schemaLanguage>WSDL</schemaLanguage>
15                  <generatePackage>parts.eip</generatePackage>
16                  <forceRegenerate>true</forceRegenerate>
17                  <schemas>
18                      <schema>
19                          <fileset>
20                              <directory>${basedir}/src/main/
                                    resources/</directory>
21                              <includes>
22                                  <include>partsorder.wsdl</
                                        include>
23                              </includes>
24                          </fileset>
25                      </schema>
26                  </schemas>
27              </configuration>
28          </plugin>
29      </plugins>
30  </build>
```

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 C [cxf] zur Verwendung. Analog zu Spring Integration wird hier ein Maven Plugin für die Codegenerierung verwendet.

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.cxf</groupId>
            <artifactId>cxf-codegen-plugin</artifactId>
            <version>3.0.1</version>
            <executions>
                <execution>
                    <id>generate-sources</id>
                    <phase>generate-sources</phase>
                    <configuration>
                        <sourceRoot>${project.build.directory}/
                            generated/cxf</sourceRoot>
                        <wsdlOptions>
                            <wsdlOption>
                                <wsdl>${basedir}/src/main/
                                    resources/partsorder.wsdl</
                                    wsdl>
                            </wsdlOption>
                        </wsdlOptions>
                    </configuration>
                    <goals>
                        <goal>wsdl2java</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
```

```
26  </build>
```

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.

```
1  @WebService(targetNamespace = "http://eip.parts", name =
       "PartsOrder")
2  @XmlSeeAlso({ObjectFactory.class})
3  @SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
4  public interface PartsOrder {
5
6      @WebResult(name = "OrderResponse", targetNamespace = "http
           ://eip.parts", partName = "OrderResponse")
7      @WebMethod(operationName = "Order")
8      public OrderResponse order(
9          @WebParam(partName = "OrderRequest", name =
               "OrderRequest", targetNamespace = "http://eip.parts
               ")
10         OrderRequest orderRequest
11     );
12 }
```

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.

```
1  <jaxws:client id="partsOrderServiceClient" serviceName="
       partsOrderService" endpointName="partsOrderEndpoint"
       address="http://localhost:8080/eip-webservice/partsOrder"
       serviceClass="parts.eip.PartsOrder">
2  </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.

```
1  <task:annotation-driven scheduler="myScheduler"/>
2  <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.

```
1  @Scheduled(fixedDelay = 1000)
2  public void processBacklog() {
3          if (getBacklogItems().size() > 0) {
4              BacklogItem backlogItem = getBacklogItems().get(0)
                   ;
5              OrderResponse orderResponse =
                   getSOAPWebServiceClient().order(toOrderRequest(
                   backlogItem));
6              if (orderResponse != null && isValidOrderNumber(
                   orderResponse.getOrderNumberUuid())) {
7                list.remove(backlogItem);
8              }
9          }
10 }
```

Fazit

Die Enterprise Integration Patterns definieren einen Katalog von Mustern für ein erweiterbare Architektur im Sinne Event Driven Architecture (EDA). 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 mittlere Systemgröße zu empfehlen, einmal vorhanden und verstanden, werden Sie eine Vielzahl von Anwendungsfälle und Möglichkeiten entdecken und schätzen.

Referenzen

[eip]: http://www.enterpriseintegrationpatterns.com/ “Enterprise Integration Patterns, Gregor Hohpe & Bobby Woolf” [camel]: http://camel.apache.org/ “Apache Camel” [si]: http://projects.spring.io/spring-integration/ “Spring Integration” [sb]: http://projects.spring.io/spring-batch/ “Spring Batch” [cxf]: http://cxf.apache.org/ “Apache CXF: An Open-Source Services Framework”