2008 is een interessant jaar op het gebied van open source integratie frameworks. Er zijn verschillende nieuwe frameworks die in 2008 beschikbaar zijn gekomen zoals Spring Integration (http://www.springframework.org/spring-integration) en Apache Camel (http://activemq.apache.org/camel). Verder zijn er nieuwe versies op komst van Apache ServiceMix (http://servicemix.apache.org) en Open ESB (https://open-esb.dev.java.net/) die op OSGI gebaseerd zullen zijn.

Mule 2 en jBPM: een open source SOA en BPM platform

Maar laten we ons concentreren op het onderwerp van dit artikel, de nieuwe versie van Mule ESB (http://www.mulesource.org) die aan het begin van dit jaar beschikbaar is gekomen, versie 2.0. Mule biedt een lichtgewicht en flexibel integratie platform waarmee op eenvoudige wijze integratie logica kan worden geimplementeerd. Bovendien biedt Mule ook standaard integratie met JBoss jBPM (http://www.jboss.org/jbossjbpm), een open source process engine. Het jBPM project ondersteunt op dit moment een tweetal procestalen: jPDL en WS-BPEL. jPDL is een zogenaamde 'graph-based' taal die is geimplementeerd in Java en een Java API biedt. WS-BPEL is de algemeen bekende open standaard op het gebied van procestalen die is gebaseerd op web service technologie met als belangrijkste standaarden WSDL, SOAP, en XML Schema.

In dit artikel zullen we gaan kijken naar de mogelijkheden die beide platformen afzonderlijk bieden en naar de combinatie van Mule en jBPM als een totaaloplossing als SOA en BPM platform.

Een introductie in Mule 2

Er zijn een tweetal open standaarden beschikbaar die voorschrijven hoe een integratieplatform kan worden geimplementeerd: Java Business Integration (JBI) en Service Component Architecture (SCA). Toen Ross Mason in 2003 begon met het Mule project, waren deze specificaties nog niet beschikbaar. Alleen aan de JBI werd al gewerkt. Ross startte het Mule project nadat hij had ondervonden dat er geen lichtgewicht en flexibel integratie platform beschikbaar was. Hij wilde de architectuur niet baseren op JBI (JSR 208) vanwege onder andere de sterke afhankelijkheid met XML berichten. De Mule architectuur maakt gebruik van POJOs voor componenten waarin integratie logica kan worden geschreven, en ook als *payload* van berichten die over de Mule bus worden gestuurd worden Java objecten gebruikt.

Nu is het 2008 en de JBI specificatie is al drie jaar beschikbaar zonder echt breedgedragen te worden. Daarnaast is er de SCA specificatie die wel door belangrijke integratieleveranciers wordt ondersteund. Apache Tuscany (http://tuscany.apache.org) is een open source implementatie van deze SCA specificatie. Maar de strategie van het Mule project is nog steeds om vast te houden aan de zelf ontwikkelde architectuur en niet aan te sluiten bij JBI of SCA. Dit is enerzijds de kracht van Mule, omdat het uiterst krachtige, doch op simpele Java classes gebouwde integratie oplossingen mogelijk maakt. Anderzijds is het nadeel natuurlijk dat dit een proprietary model is, dat geen mogelijkheden biedt voor hergebruik van componenten zoals bij JBI, of een eenduidig service component model zoals bij SCA.

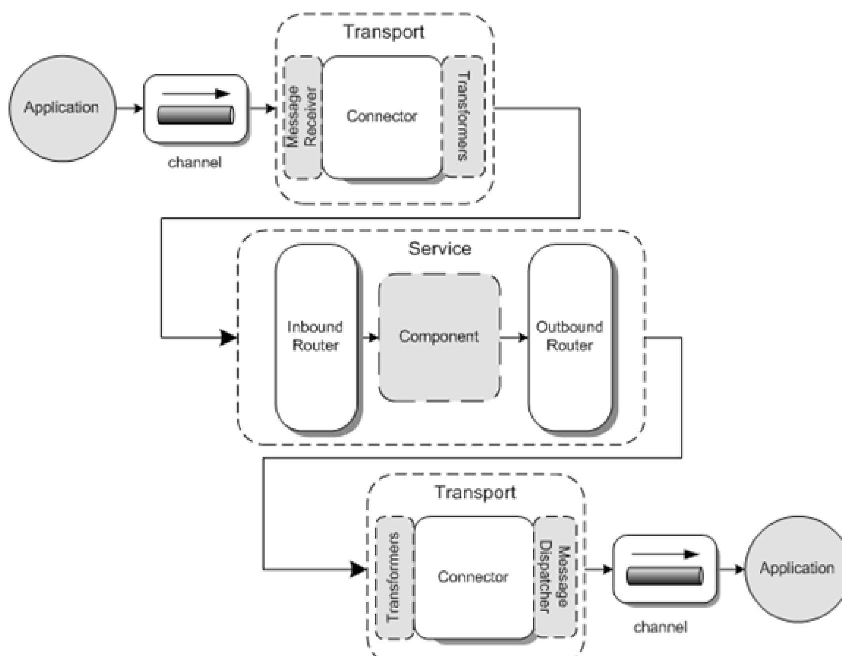
Maar laten we eens kijken naar de bouwstenen van Mule en een klein voorbeeld

Tijs Rademakers
Auteur van Open Source ESBs in Action (Manning, 2008)
Software architect Atos Origin

van een Mule configuratie file, om een beter inzicht te krijgen in de architectuur. Een goede manier om de bouwstenen van Mule te analyseren is door een standaard berichtenflow op de Mule bus te bekijken. Een applicatie stuurt een bericht via een berichtenkanaal zoals JMS of HTTP naar de Mule ESB. Dit bericht wordt opgepikt door Mule en verwerkt volgens de standaard berichtenflow en uiteindelijk via een ander berichtenkanaal naar de doelapplicatie gestuurd. Figuur 1 geeft een overzicht van de bouwstenen met de standaard berichtenflow.

Mule biedt een lijst van zogenaamde **transport connectors** die berichten kunnen versturen en ontvangen via bijvoorbeeld JMS, HTTP, SMTP, file systeem etc. De standaard berichtenflow start daarom met het ontvangen van een bericht via de transport connector, bijvoorbeeld de JMS transport.

Wanneer het bericht is ontvangen, kan het bericht worden getransformeerd, voordat deze verder wordt verwerkt via een **transformer**. Nu denk je bij transformaties natuurlijk al snel aan conversie van berichtformaat van bijvoorbeeld EDI naar XML. Dit soort functionaliteit kun je in Mule ook met een transformer implementeren, maar daarnaast biedt de transformer ook meer basale transformaties en dit is ook een voorbeeld van de nauwe verbondenheid van Mule met Java. Mule biedt standaard transformers die een binnenkomend bericht omzetten naar Java objecten die voor de integratie ontwikkelaar makkelijker zijn om mee te werken. Bijvoorbeeld, als er vanuit een JMS transport een *TextMessage* of *ObjectMessage* binnenkomt, dan zal Mule deze berichten standaard omzetten naar respectievelijk een String en een Object. Hetzelfde geldt voor het versturen van een bericht vanuit Mule naar een JMS queue of topic. Een String wordt standaard omgezet naar een *TextMessage* en een zelfde ontwikkelde POJO vergelijkbaar met een implementatie van het Transfer Object pattern wordt standaard omgezet naar een *ObjectMessage*. Dit soort handigheidjes maken de werkzaamheden voor ontwikkelaars een stuk eenvoudiger.



Figuur 1: Overzicht van de belangrijkste bouwstenen van de Mule architectuur aan de hand van een standaard flow van een bericht over de Mule bus.

Nadat een bericht door eventuele standaard transformers van een transport connector is verwerkt, is de volgende stap in de standaard berichtenflow de **inbound router**. De inbound router geeft binnen Mule aan op welk adres of welke addressen berichten binnen kunnen komen. De inbound router geeft dus aan welke transport connectors er worden gebruikt en op welke adressen de transport connector moet gaan pollen of als listener moet gaan functioneren. Daarnaast kan een inbound router ook worden geconfigureerd om bijvoorbeeld alleen specifieke berichten te accepteren, zoals het selective consumer pattern en om binnenkomende berichten die kunnen worden gecorreleerd op basis van elementen uit het bericht samen te voegen met een implementatie van het aggregator pattern.

Nadat het bericht is verwerkt door de inbound router wordt deze doorgegeven aan de **component**. De component biedt ontwikkelaars de mogelijkheid om integratie logica te schrijven met behulp van een Java class, een Spring bean of een script geschreven in bijvoorbeeld Groovy of JavaScript. Het ontvangen bericht wordt dan doorgegeven als input parameter. De component biedt dus alle vrijheid om de integratie logica te schrijven die je nodig hebt.

Outbound router

De output van de component, wat in de meeste gevallen dus het object is dat als return waarde is teruggegeven door de componenten, wordt vervolgens doorgegeven aan de **outbound router**. In de outbound router kan worden bepaald waar het bericht naar toe gestuurd moet worden en welke transport connectors hierbij moeten worden gebruikt. Ook kan in de outbound router logica worden geimplementeerd zoals bijvoorbeeld content-based routing, waarbij een bericht naar een specifiek berichtenkanaal wordt gestuurd als de inhoud overeenkomt met een zogenaamde routing rule.

We hebben nu de belangrijkste onderdelen van een Mule berichtenflow besproken en deze infor-

matie is voldoende om met Mule aan de slag te kunnen. Natuurlijk hebben we wel nog wat kennis nodig over hoe we een berichtenflow in Mule kunnen implementeren. Hiervoor biedt Mule een XML configuratie taal die gebruik maakt van Spring 2.x functionaliteit en een scripting configuratie waarmee je op basis van bijvoorbeeld een Groovy script een Mule configuratie kan ontwikkelen. De XML configuratie is verreweg het meest gebruikt en daarom gebruiken we in het volgende "hello world" voorbeeld ook deze vorm van configuratie.

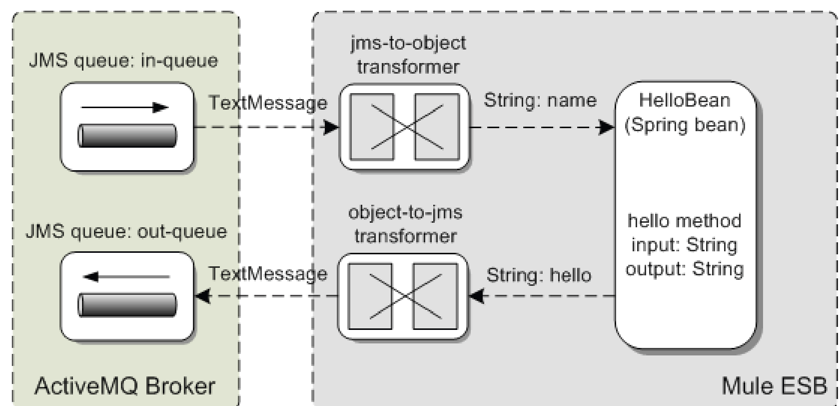
```
<mule  xmlns="http://www.mulesource.org/schema/mule/core/2.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:spring="http://www.springframework.org/schema/beans"
   xmlns:jms="http://www.mulesource.org/schema/mule/jms/2.0"
        xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
              http://www.mulesource.org/schema/mule/core/2.0
     http://www.mulesource.org/schema/mule/core/2.0/mule.xsd
              http://www.mulesource.org/schema/mule/jms/2.0
      http://www.mulesource.org/schema/mule/jms/2.0/mule-jms.xsd">

  <jms:activemq-connector name="jmsCon" brokerURL="tcp://localhost:61616"

   <spring:bean  id="helloBean"  class="esb.example.HelloBean"/>

  <model name="helloModel">
    <service name="helloService">
      <inbound>
        <jms:inbound-endpoint queue="in-queue"/>
      </inbound>
      <component>
        <spring-object bean="helloBean"/>
      </component>
      <outbound>
        <outbound-pass-through-router>
                <jms:outbound-endpoint  queue="out-queue"/>
        </outbound-pass-through-router>
      </outbound>
    </service>
  </model>
</mule>
```

Zoals je kunt zien zijn veel reeds besproken bouwstenen van Mule rechtstreeks terug te vinden in deze XML configuratie. De *jms:activemq-connector* is een voorbeeld van een transport connector en verzorgt de connectie tussen Mule en Apache ActiveMQ. Naast ActiveMQ ondersteunt Mule alle brokers die de JMS specificatie implementeren. Daarnaast is ook te zien dat op de gebruikelijke manier Spring beans kunnen worden gedefinieerd. Omdat Spring is geïntegreerd in de Mule container kun je alle functionaliteit die het Spring framework biedt ook gebruiken in de Mule configuratie.

Om services binnen een Mule configuratie te kunnen definiëren is een model noodzakelijk. Voor een introductie in Mule kun je het model beschouwen als een container waarin een x-aantal Mule services draaien. Een Mule service bestaat uit een inbound router, met een inbound endpoint, een component en een outbound router, met een outbound endpoint. In dit voorbeeld is een inbound router gedefinieerd die een bericht van een JMS queue afhaalt. Vervolgens wordt dit bericht doorgegeven aan de component, wat bijvoorbeeld een Java class of een Groovy script kan zijn, maar in dit voorbeeld gebruiken we de volgende Spring bean.

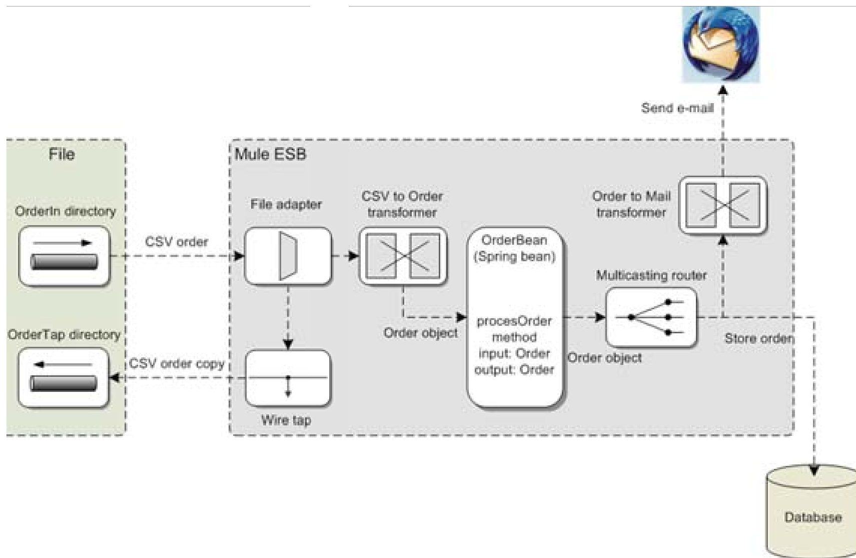
```
package esb.example;

public class HelloBean {

  public String hello(String name) {
    return "hello " + name;
  }
}
```

De *HelloBean* is een doodnormale POJO en heeft als input parameter een String en geeft ook een String terug. Zoals je weet is een tekstbericht binnen JMS een *TextMessage* object, dus er is een conversie nodig van een *TextMessage* instantie naar een String, voordat de HelloBean kan worden aangesproken door de Mule container. Hier komt de transformer van pas en zoals al eerder is beschreven in dit artikel, zorgt Mule automatisch voor de conversie van een *TextMessage* naar een String en vice versa. Het resultaat van de *HelloBean* wordt vervolgens doorgegeven aan de outbound router, die de String automatisch transformeert naar een *TextMessage* en op de JMS *out-queue* plaatst. Voor de volledigheid wordt het 'hello world' voorbeeld schematisch getoond in figuur 2.



Figuur 2: Schematisch overzicht van het 'hello world' voorbeeld waarbij een bericht van een JMS queue wordt gehaald, een Spring bean wordt aangesproken met de inhoud van dit bericht, en het resultaat weer op een JMS queue wordt gezet.

Dit ‘hello world’ voorbeeld laat zien dat het zeer eenvoudig is om in Mule te communiceren met JMS en gebruik te maken van Spring beans. Maar in de praktijk is er meer functionaliteit nodig om integratie problemen op te lossen. Laten we daarom kijken naar een complexer voorbeeld, waarbij een order verwerkt moet worden en zowel in een database moet worden opgeslagen als in een e-mail richting een inkoper moet worden gestuurd. Een overzicht van de te implementeren onderdelen is weergegeven in figuur 3.



Figuur 3: Een order komt via het filesysteem binnen met een CSV formaat. Het bericht wordt verwerkt door de order service en vervolgens zowel opgeslagen in de database en als e-mail richting de inkoper gestuurd.

Zoals te zien in figuur 3 wordt de *OrderIn* directory gepolled op nieuwe orders. Vervolgens wordt met een wire tap het binnenkomende bericht gekopieerd naar de *OrderTap* directory. Een wire tap is een optie om binnenkomende berichten te kunnen bekijken, wat vooral in een test omgeving handige functionaliteit is. Vervolgens wordt het CSV order bericht getransformeerd naar een *Order* Java object met een transformer. Nadat de *OrderBean* de order heeft verwerkt (in dit voorbeeld loggen we alleen de inhoud van de order naar de console), wordt het order bericht met een router naar een database en als e-mail gestuurd. Het volgende XML voorbeeld toont de Mule configuratie die deze functionaliteit implementeert.

```
<mule>
  <spring:bean name="OrderBean" class="esb.example.
OrderBean" />

  <spring:bean name="datasource"
                class="org.enhydra.jdbc.standard.
StandardDataSource">
     <spring:property name="driverName" value="org.
hsqldb.jdbcDriver" />
    <spring:property name="url"
value="jdbc:hsqldb:hsql://localhost/orderdb" />
    <spring:property name="user" value="sa" />
  </spring:bean>

  <file:connector name="fileConnector"
streaming="false" />
    <jdbc:connector  name="hsqldb-connector"  data-
Source-ref="datasource">
    <jdbc:query key="orderInsert"
      value="INSERT into orders (id, clientname,
product, quantity) VALUES
        (${bean:id},
        ${bean:clientName},
        ${bean:product},
        ${bean:quantity})" />
  </jdbc:connector>

  <model name="OrderModel">
    <service name="OrderService">
      <inbound>
        <file:inbound-endpoint path="OrderIn" poll-
ingFrequency="5000">
          <file:file-to-string-transformer/>
            <custom-transformer class="esb.example.
OrderTransformer"/>
        </file:inbound-endpoint>
        <wire-tap-router>
          <file:outbound-endpoint path="OrderTap"/>
        </wire-tap-router>
      </inbound>
      <component>
        <spring-object bean="OrderBean"/>
      </component>
      <outbound>
        <multicasting-router>
                          <jdbc:outbound-endpoint
queryKey="orderInsert"/>
          <smtp:outbound-endpoint to="johndoe@local-
host"
                from="order@localhost"
                subject="A new order has arrived"
                host="localhost"
                port="25"
                user="order"
                password="order">
            <custom-transformer class="esb.example.
OrderToEmailTransformer"/>
            <email:string-to-email-transformer/>
          </smtp:outbound-endpoint>
        </multicasting-router>
      </outbound>
    </service>
  </model>
</mule>
```

De namespace definitie is in dit voorbeeld weggelaten, maar dit was al getoond in het ‘hello world’ voorbeeld. In de configuratie van de inbound router van de *OrderService* zie je een voorbeeld van het zogenaamde *chainen* van transformers. Je kunt meerdere transformers definiëren waarbij de output van de eerste transformer als input van de tweede transformer wordt doorgegeven. In dit voorbeeld wordt de inhoud van de order CSV omgezet van een *File* object naar een String. Vervolgens wordt deze CSV String getransformeerd naar een *Order* object instantie zoals hieronder getoond.

```
package esb.example;

import org.mule.api.transformer.
TransformerException;
import org.mule.transformer.AbstractTransformer;

public class OrderTransformer extends
AbstractTransformer {
```

```
  protected Object doTransform
(Object payload, String encoding)
          throws TransformerException {

    if(!(payload instanceof String)) {
      throw new TransformerException(this, new
        IllegalArgumentException("expected String as
input"));
    }
    String strPayload = (String) payload;
    String[] payloadArray = strPayload.split(",");
    Order order = new Order();
    order.setId(payloadArray[0]);
    order.setClientName(payloadArray[1]);
    order.setProduct(payloadArray[2]);
    order.setQuantity(Integer.
valueOf(payloadArray[3]));
    return order;
  }
}
```

Zoals je kunt zien is een transformer binnen Mule eigenlijk een implementatie van een *doTransform* methode als onderdeel van een *AbstractTransformer* class.

Een ander onderdeel van de Mule configuratie dat uitleg behoeft, is hoe het *Order* object wordt opgeslagen in een database. De Mule JDBC connector die in dit voorbeeld wordt gebruikt, is geconfigureerd met een standaard *datasource* definitie. Als onderdeel van een JDBC connector kunnen verschillende queries worden gedefinieerd. In dit geval is er één insert query gedefinieerd, die gebruik maakt van een zogenaamde *bean* expressie. Mule biedt verschillende expressietalen om waardes uit onder andere de payload van een bericht of een header van een bericht te halen. In dit voorbeeld is de *bean* expressietaal gebruikt waarmee uit het *Order* object instantie de properties kunnen worden gehaald.

Natuurlijk is dit slechts een zeer korte introductie in Mule, maar de voorbeelden geven een goede indicatie van de mogelijkheden en de manier van configuratie en het verwerken van berichten binnen Mule. Laten we nu gaan kijken naar een mogelijke open source invulling van een BPM platform: jBPM.

Een introductie in jBPM

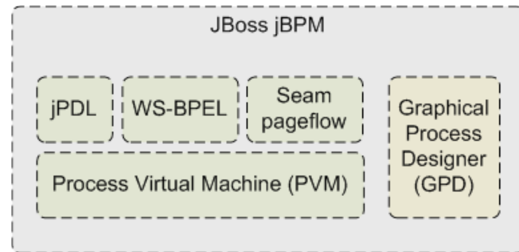
Mule leent zich zeer goed om berichten te verwerken en door te sturen naar de juiste services. De afhandeling van deze berichten wordt veelal in *request-response* of *one-way* iteractie patronen uitgevoerd waarbij één externe service invocatie wordt gedaan. In het geval van één service invocatie is er dus geen service orchestratie nodig en is er ook geen state management. Wanneer er meerdere service invocaties nodig zijn, kan er wel een behoefte zijn aan een manier om deze

Je wilt naast je werk tijd voor jezelf overhouden?
Wij bieden je 13 adv-dagen bovenop je 24 vakantiedagen.

Als Java Specialist werk je hard. Maar naast je werk heb je ook een privéleven. Dat begrijpen wij en daarom belonen we je met 37 vrije dagen per jaar. Dit geeft je tijd voor een lange vakantie of kwaliteitstijd met je kinderen. ADV dagen zijn naast de leaseauto, winstdeling en pensioenregeling slechts een van de uitstekende secundaire arbeidsvoorwaarden. Wil je weten wat SPIE nog meer te bieden heeft, kijk op www. SPIE-ICT.nl

service invocaties te orchestreren en om de state van de orchestratie te bewaren. Dit is nu precies wat jBPM biedt.

jBPM is een open source process engine van JBoss en is reeds enkele jaren beschikbaar. jBPM bestaat uit een aantal sub-projecten, zoals getoond in figuur 4.



Figuur 4: De sub-projecten van JBoss jBPM, waarbij in dit artikel wordt ingegaan op de jBPM Process Definition Language (jPDL) engine en de Graphical Process Desiginer (GPD).

Het jBPM project is gestart met de jPDL en de GPD onderdelen en dit zijn dus ook de meest volwassen onderdelen. WS-BPEL is de bekende industriestandaard van OASIS voor het definiëren van processen. jBPM biedt een implementatie van een WS-BPEL engine, maar na contact met Mark Little van JBoss blijkt dat dit sub-project niet verder zal worden ontwikkeld. Seam pageflow maakt gebruik van jPDL en is gericht op het definiëren van scherm interacties, waarmee een extra scope wordt geïntroduceerd, de zogenaamde conversations. Het laatste sub-project is de PVM, dat een simpele Java library biedt om processen uit te voeren. De PVM kan dus in elke Java applicatie worden gebruikt om procesgedrag te implementeren, echter dit project is nog in alpha fase. Dit artikel richt zich op jPDL om processen te implementeren.

Het grote verschil tussen WS-BPEL en jPDL is dat WS-BPEL gebaseerd is op de web service standaarden (SOAP, WSDL, XSD) en jPDL op Java. In een Java omgeving is jPDL dus een eenvoudige en flexibele taal waarmee door middel van Java objecten kan worden gecommuniceerd. WS-BPEL heeft een sterke focus op web service communicatie en dus XML berichten. Verder biedt jPDL ook workflow ondersteuning en deze is standaard in WS-BPEL niet beschikbaar zonder toevoeging van BPEL4People.

jPDL is een eenvoudige manier van proces definitie die sterk overeenkomt met het UML state diagram. Door states met zogenaamde transities

UW ICT-PROJECT GEGARANDEERD OP TIJD OPGELEVERD!

SOMMIGEN BELOVEN HET. WIJ GARANDEREN HET!

De Caesar Groep is een ICT-dienstverlener die oplossingen biedt met rendement. Wij nemen daarbij de doelstellingen van de klant als uitgangspunt. In de vorm van TimeValue-projecten realiseren wij gegarandeerd op tijd opgeleverde ICT-oplossingen met korte doorlooptijden en aantoonbare waarde voor onze klanten. Indien wij toch te laat opleveren, leggen wij onszelf een aanzienlijke boete op. We betalen onze klant dan tot 50% terug! De technologie die wij gebruiken is onder andere gebaseerd op Oracle, Java, Progress en Microsoft.

Wat is de waarde van elk van uw ICT-projecten? Hoe kunt u ervoor zorgen dat uw ICT-project op tijd wordt opgeleverd? En waarom durft Caesar die opleverdatum keihard te garanderen?
Graag beantwoorden wij deze vragen in onze workshop TimeValue.
Voor meer informatie en aanmelding kijk op **www.timevalue.nl**

GRAAG NODIGEN WIJ U UIT VOOR EEN INSPIRERENDE TIMEVALUE WORKSHOP

Caesar Groep - Zonnebaan 9 - 3542 EA Utrecht - tel. 030 - 240 42 00 - www.caesar.nl - info@caesar.nl

ICT OPTIMA FORMA CAESAR GROEP

te verbinden, kun je een proces definitie maken, waarvan een simpel voorbeeld te zien is in figuur 5. Dit voorbeeld is gemodelleerd met de GPD Eclipse plugin van het jBPM project.

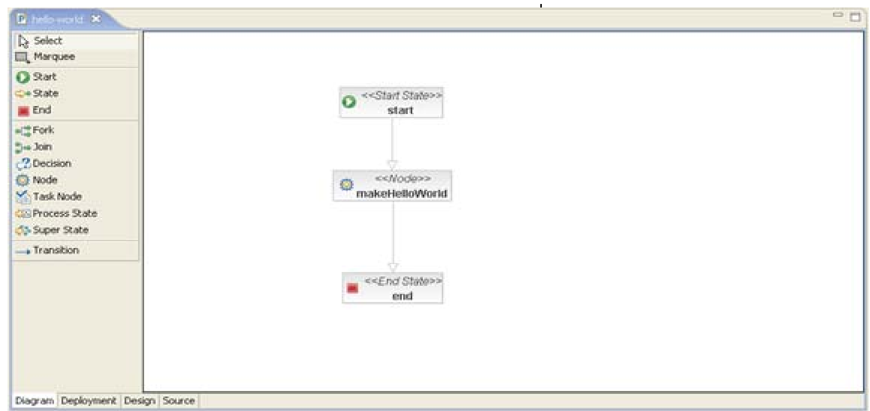
In het voorbeeld zoals weergegeven in figuur 5, is een proces met slechts één extra node naast de start en end nodes getoond. In de *makeHelloWorld* node kun je een actie definiëren die wordt uitgevoerd wanneer de node actief wordt. Naast de node is er de state, wat een specieke implementatie van een node is. Een state wordt pas verlaten nadat er expliciet een *leaveNode* methode is aangeroepen op een proces instantie. Voor een node geldt dat deze wordt uitgevoerd en daarna automatisch naar de volgende toestand wordt verplaatst als er geen custom actie is geconfigureerd. In het “hello world” proces van figuur 5 wordt nadat de actie in de node is uitgevoerd dus de end state bereikt, waarmee de process instantie wordt beëindigd.

Wanneer je processen modelleert met de GPD Eclipse plugin wordt er op de achtergrond een XML file aangemaakt en gevuld. Een jPDL deployable is uiteindelijk een XML file met eventuele Java classes die in het proces worden gebruikt die zijn gepackaged in een par file, wat overeenkomt met de bekende jar files. In het volgende code voorbeeld is de XML definitie van het ‘hello-world’ proces te lezen.

```
<process-definition  xmlns=”urn:jbpm.org:jpdl-3.2”
name=”hello-world”>

  <start-state name=”start”>
      <transition to=”makeHelloWorld”/>
  </start-state>

  <node name=”makeHelloWorld”>
      <action name=”helloWorldAction”
                                  class=”esb.chapter11.
HelloWorldHandler”/>
      <transition to=”end”/>
  </node>

  <end-state name=”end”/>
</process-definition>
```

De XML element namen van een jPDL proces definitie komen overeen met de graphische notatie van de processen in de GPD. Zoals je kunt zien is de *HelloWorldHandler* geconfigureerd als de actie die wordt uitgevoerd in de *makeHelloWorld* node. De implementatie van deze handler ziet er als volgt uit:

```
public class HelloWorldHandler implements
ActionHandler {


  public void execute(ExecutionContext execContext) {
    String name = (String) execContext.getContextIn-
stance().getVariable(“name”);
```



Figuur 5: Screenshot van een een ‘hello world’ proces in jPDL, gemodelleerd met de process designer Eclipse plug-in.

```
     execContext.getContextInstance().setVariable(
         “output”, “hello “ + name);
     execContext.leaveNode();
   }
}
```

In de *HelloWorldHandler* wordt verwacht dat het proces is gestart met een variabele *name*. Het starten van een proces is mogelijk via de jPDL API.

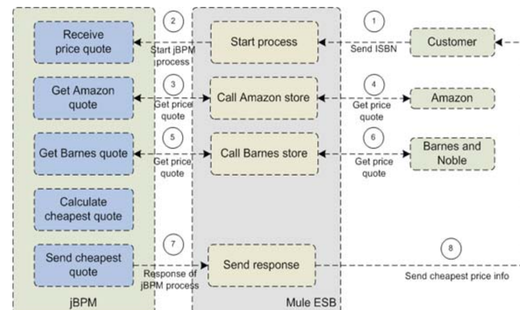
```
// jbpmConfig wordt geinjecteerd en laat de jBPM
configuratie in
JbpmContext ctx = jbpmConfig.createJbpmContext();
ProcessInstance processInstance = ctx.
newProcessInstance(“hello-world”);
processInstance.getContextInstance().
createVariable(“name”, “Tijs”);
processInstance.signal();
jbpmContext.save(processInstance);
```

In de *HelloWorldHandler* wordt de *name* variabele opgevraagd uit de proces instantie context en daarna wordt een nieuwe variabele *output* aangemaakt in diezelfde context. Vervolgens wordt de node verlaten met de *leaveNode* methode en wordt het proces beëindigd via de end state.
Nu we inzicht hebben in de functionaliteit en het gebruik van Mule en jBPM kunnen we ingaan op de samenwerking van deze twee producten. Mule biedt een redumentaire implementatie van een integratie met jBPM, en dit geeft een goed beeld van de mogelijkheden van een open source SOA en BPM platform.

Integratie van Mule 2 met jBPM

Mule heeft verschillende andere open source frameworks geïntegreerd om een volledige set van ESB functionaliteit te kunnen bieden. Een van de frameworks die in de Mule oplossing is geïntegreerd is jBPM. Mule kan zo worden geconfigureerd dat een jBPM process engine binnen de Mule container wordt geïnstantieerd, zodat jBPM binnen dezelfde Java virtual machine draait als Mule. Hierdoor kan vanuit Mule een jBPM proces worden gestart en kan vanuit een jBPM proces de Mule container worden gebruikt om de benodigde connectiviteit,

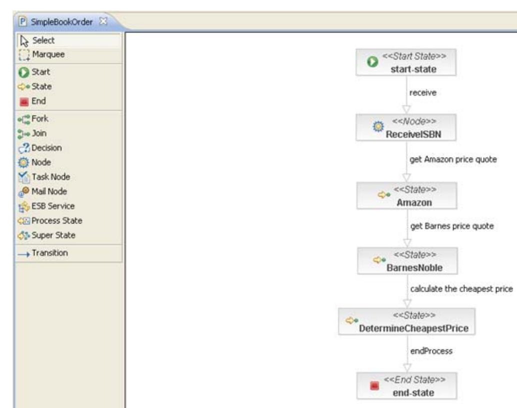
routing en transformatie functionaliteit te implementeren. Door middel van een simpel voorbeeld gaan we als slot van dit artikel de combinatie van Mule en jBPM bekijken. Het gaat hier om een opvraag van de laagste prijs van een boek op basis van een isbn nummer. In figuur 6 is dit voorbeeld op een schematische wijze uitgewerkt.



Figuur 6: Overzicht van de interactie tussen Mule en jBPM tijdens het doorlopen van een prijsaanvraag proces op basis van een isbn nummer.

Voor eenvoudige implementaties voldoet de integratie prima

Wanneer een klant een prijs opvraagt op een website of via een web service wordt in de Mule ESB een nieuwe instantie aangemaakt van het prijsopvraag proces dat we zodadelijk gaan bekijken. Op dat moment is het jBPM proces sturend in de communicatie die plaatsvindt. Vanuit het proces wordt via de Mule ESB de Amazon en de Barnes and Noble prijzen opgevraagd. Door het gebruik van Mule is eventuele transformatie logica en de aanroep van de web service of REST service losgekoppeld van de proces implementatie. Dit heeft als grote voordeel dat het proces geen technische details hoeft te bevatten. Nadat de laagste prijs is bepaald wordt het resultaat teruggegeven aan de klant, wederom via de Mule ESB. Laten we nu kijken naar de implementatie van het proces in de jBPM process designer, zoals getoond in figuur 7.



Figuur 7: Screenshot van het prijsopvraag proces in de jBPM process designer.

Het prijsopvraag proces, zoals te zien in figuur 7, is opgebouwd uit verschillende logische processtappen. Na de ontvangst van het ISBN nummer wordt een prijsopvraag gedaan bij Amazon en Barnes and Noble, waarna de laagste prijs wordt bepaald. De interactie met de Mule ESB is in de details van de processtappen verborgen door het gebruik van events en daaraan verbonden acties. Zo is het opslaan van het binnenkomende ISBN nummer als actie aan de *ReceiveISBN* node verbonden, zoals te zien in figuur 8.

De *StoreIncomingData* class is onderdeel van de Mule integratie met jBPM en kan worden gebruikt om de inhoud van het binnenkomende bericht op te slaan in een variable in de proces context, zoals in dit geval de *isbn* variabele. Deze *ActionHandler* wordt uitgevoerd zoda de proces instantie de node *ReceiveISBN* gaat uitvoeren, zoals geconfigureerd met de *node-enter* event. Na de *ReceiveISBN* processtap is het ISBN nummer dus beschikbaar als proces variabele en deze wordt ook automatisch gepersisteerd. In de volgende stap vindt de communicatie plaats met de Amazon boekwinkel en de daaraan verbonden acties zijn te zien in figuur 9.

Voor de *Amazon* state zijn er twee acties gedefinieerd bij twee verschillende event types. De eerste is de *SendMuleEvent* class die een bericht vanuit het proces naar Mule stuurt. In dit voorbeeld wordt de *isbn* variabele als bericht naar het *AmazonRequest* endpoint gestuurd binnen Mule. In de Mule configuratie wordt zodadelijk duidelijk hoe dit endpoint binnen Mule is gedefinieerd. In de andere actie is wederom een *StoreIncomingData* class gebruikt om de response van de Amazon prijsopvraag op te slaan als proces variabele. De *Barnes* state is bijna identiek aan deze *Amazon* state, behalve de endpoint configuratie. De *DetermineCheapestPrice* state berekent de laagste prijs en stuurt deze vervolgens naar een endpoint binnen Mule. De action handler die de laagste prijs berekent is hieronder weergegeven.

```
public class CalculateCheapestPriceHandler imple-
ments ActionHandler {
    public void execute(ExecutionContext context)
throws Exception {
            PriceQuote priceQuote = new
PriceQuote();
            priceQuote.setIsbn((String) context.
getVariable(“isbn”));
            Float amazonPrice = (Float) context.
getVariable(“amazonPrice”);
            Float barnesPrice = (Float) context.
getVariable(“barnesPrice”);
            if(amazonPrice <= barnesPrice) {
                priceQuote.
setCheapestStore(“Amazon”);
                priceQuote.
setCheapestPrice(amazonPrice);

priceQuote.setPriceDifference(barnesPrice - amazon-
Price);
            } else {
```

```
priceQuote.setCheapestStore("Barnes and Noble");
                priceQuote.
setCheapestPrice(barnesPrice);

priceQuote.setPriceDifference(amazonPrice - bar-
nesPrice);
           }
           context.setVariable("cheapest", price-
Quote);
     }
}
```

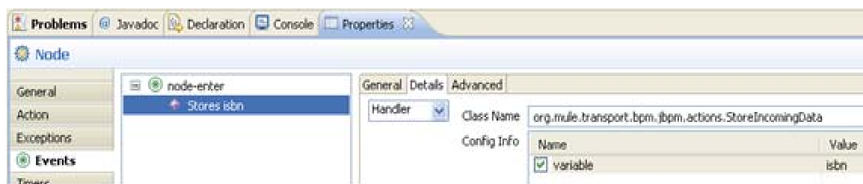
De *cheapest* proces variabele wordt als eindresultaat van het prijsopvraag proces naar de Mule ESB gestuurd. Voor het complete plaatje van de implementatie missen we nog een belangrijk onderdeel en dat is de Mule configuratie.

```
<?xml version="1.0" encoding="UTF-8"?>
<mule
  <spring:beans>
    <spring:import resource="jbpm-beans.xml"/>
    <spring:import resource="bookorder-beans.xml"/>
  </spring:beans>

  <spring:bean id="jbpm" class="org.mule.transport.
bpm.jbpm.Jbpm" destroy-method="destroy">
    <spring:property name="jbpmConfiguration">
      <spring:ref bean="jbpmConfig" />
    </spring:property>
  </spring:bean>


  <bpm:connector name="jBpmConnector" bpms-
ref="jbpm" />


  <jms:activemq-connector name="activeMQConnector"
brokerURL="tcp://localhost:61616"/>

  <bpm:endpoint name="ProcessEngine"
process="SimpleBookOrder"/>
  <jms:endpoint name="BookOrderRequest"
queue="bookorder.in" />
  <jms:endpoint name="AmazonRequest" queue="amazon.
in" />
  <jms:endpoint name="BarnesRequest" queue="barnes.
in" />
  <jms:endpoint name="AmazonResponse" queue="amazon.
out" />
  <jms:endpoint name="BarnesResponse" queue="barnes.
out" />
  <jms:endpoint name="BookOrderResponse"
queue="bookorder.out"/>

  <model>
    <service name="ToBPMS">
      <inbound>
        <inbound-endpoint ref="BookOrderRequest"/>
        <inbound-endpoint ref="AmazonResponse"/>
        <inbound-endpoint ref="BarnesResponse"/>
      </inbound>
      <outbound>
        <filtering-router>
          <outbound-endpoint ref="ProcessEngine"
synchronous="false" />
        </filtering-router>
      </outbound>
    </service>

    <service name="FromBPMS">
      <inbound>
        <inbound-endpoint ref="ProcessEngine"/>
      </inbound>
```

Jij wilt het beste uit jezelf halen?

Wij zorgen dat je de beste opleidingen volgt.

Als Java specialist ben je nooit uitgeleerd. Ontwikkelingen op de voet volgen is een drive die wij verlangen. Daarom zorgen wij ervoor dat je maximaal gebruik maakt van onze uitstekende opleidingstrajecten, waarbij certificeren van belang is. Kernwoorden die we gebruiken zijn J2EE, JSF, JBoss Seam, EJB3.0, Java Portlets, AJAX, Spring en Hibernate. Tools die daarbij gebruikt worden zijn IBM Websphere/Eclipse en Oracle. Wil je weten wat SPIE nog meer te bieden heeft, kijk op www. SPIE-ICT.nl

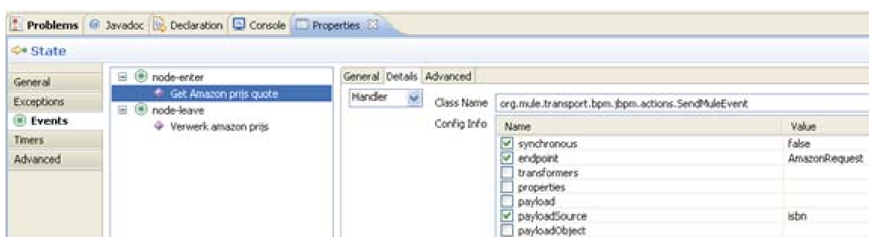


Figuur 8: Screenshot van de properties view van de ReceiveISBN node.

```
        <outbound>
          <bpm:outbound-router>
            <outbound-endpoint ref=”AmazonRequest”/>
            <outbound-endpoint ref=”BarnesRequest”/>
            <outbound-endpoint
ref=”BookOrderResponse”/>
          </bpm:outbound-router>
        </outbound>
      </service>


      <service name=”AmazonStore”>
        <inbound>
          <inbound-endpoint ref=”AmazonRequest”/>
        </inbound>
        <component>
          <method-entry-point-resolver>
            <include-entry-point
method=”getPriceQuote”/>
          </method-entry-point-resolver>
          <spring-object bean=”amazonStoreBean”/>
        </component>
        <outbound>
          <outbound-pass-through-router>
            <outbound-endpoint ref=”AmazonResponse”/>
          </outbound-pass-through-router>
        </outbound>
      </service>


      <service name=”BarnesStore”>
        <inbound>
          <inbound-endpoint ref=”BarnesRequest”/>
        </inbound>
        <component>
          <method-entry-point-resolver>
            <include-entry-point
method=”getPriceQuote”/>
          </method-entry-point-resolver>
          <spring-object bean=”barnesStoreBean”/>
        </component>
        <outbound>
          <outbound-pass-through-router>
            <outbound-endpoint ref=”BarnesResponse”/>
          </outbound-pass-through-router>
        </outbound>
      </service>

    </model>
</mule>
```

De integratie tussen Mule en jBPM wordt gerealiseerd met behulp van de *bpm:connector*. Deze connector is geconfigureerd met een Spring bean waarin de jBPM configuratie is gedefinieerd. Het voert hier te ver om naar de jBPM configuratie, in de jbpm-beans.xml file te bekijken. Maar de code is te downloaden via de zip file die bij de links van dit artikel wordt genoemd. Er zijn verschillende endpoints gedefinieerd, zoals bijvoorbeeld het *AmazonRequest* endpoint, die vanuit het prijsopvraag proces worden gebruikt. In dit voorbeeld zijn alle endpoints JMS queues behalve het *ProcessEngine* bpm endpoint. Natuurlijk zou dit voorbeeld ook gebruik kunnen maken van web service endpoints, via Apache CXF, of bijvoorbeeld http endpoints. Voor dit voorbeeld zijn de Amazon en Barnes and Noble services geïmplementeerd via Spring beans die als stub fungeren. De *amazonStoreBean* en de *barnesStoreBean* implementeren de *getPriceQuote* methode die een ISBN nummer als input verwacht en daar een prijs voor teruggeeft. Zoals je ziet kan met behulp van de method entry point resolver worden aangegeven welke methode moet worden aangeroepen op de Spring bean. Een laatste onderdeel van dit voorbeeld dat specifiek is voor de jBPM integratie zijn de *ToBPMS* en *FromBPMS* services. In de *ToBPMS* service zijn de endpoints geconfigueerd die als input voor het jBPM proces worden gebruikt en in de *FromBPMS* service staan de endpoints die vanuit het proces worden aangesproken.

Figuur 9: Screenshot van de properties view van de Amazon state.



De integratie tussen Mule en jBPM is basaal en kan in complexere implementaties niet voldoende blijken. Voor eenvoudige implementaties voldoet de integratie echter prima en geeft het een goed beeld van de mogelijkheden voor een open source SOA en BPM platform. Voor meer informatie over Mule en jBPM en open source ESBs in zijn algemeenheid is er het boek Open Source ESBs in Action van uitgever Manning. Ook op de website van het boek, http://www.esbinaction.com, is er meer informatie te vinden. «

Referenties

1. Mule – http://mule.mulesource.org
2. Enterprise Integration Patterns – http://www.enterprise-integrationpatterns.com
3. Mule 2.0.2 download – http://mule.mulesource.org/display/MULE/Download
4. Code download – http://www.esbinaction.com/file/java-magazine_mulejbpm.zip
5. Website boek – http://www.esbinaction.com