whitehorses

member of the human network

WHITEBOOK

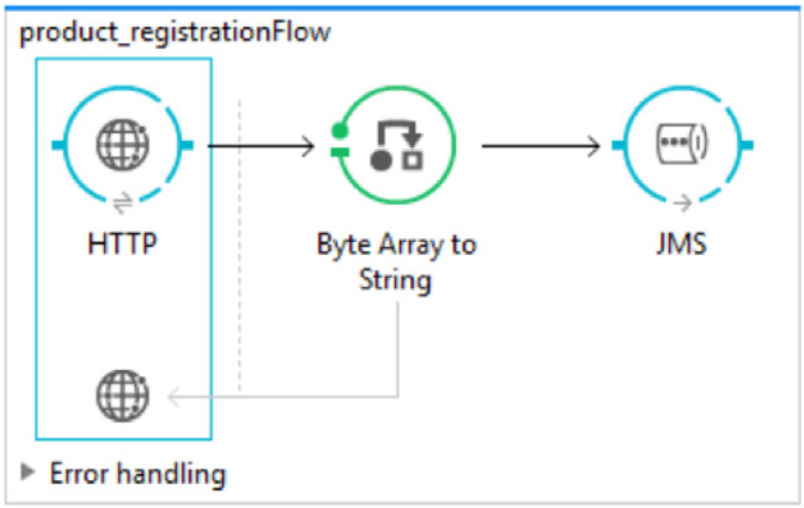
Mule Patterns

Maart 2016

Auteur:
Roger Goossens
INTEGRATIE SPECIALIST

Inleiding

Mule is een open source enterprise service bus die je in staat stelt om integratie services te ontwikkelen (zie http://www.whitehorses.nl/whitebooks/2011/mule-een-open-source-enterprise-service-bus voor een overview). Het centrale hart van een mule service is de flow file. De flow file bevat alle configuratie onderdelen (connectors, components, transformers, etc.) van een specifieke mule service en is in essentie niets meer dan een xml file.



Voorbeeld van een mule flow in Anypoint Studio

```
<mule ...>
   <jms:activemq-connector name="Active_MQ"
      specification="1.1" brokerURL="tcp://localhost:61616"
      validateConnections="true" doc:name="Active MQ" />
   <flow name="product_registrationFlow">
      <http:inbound-endpoint exchange-pattern="one-way"
     host="localhost" port="8080" path="products" doc:name="HTTP" />
  <byte-array-to-string-transformer
     doc:name="Byte Array to String" />
      <jms:outbound-endpoint queue="products"
         connector-ref="Active_MQ" doc:name="JMS" />
   </flow>
</mule>
```

De bijbehorende XML (de namespace definities zijn weggelaten)

Een volwassen Service Oriented Architecture kan uit een groot aantal van deze services bestaan. Naarmate een service landschap groeit, groeit ook de hoeveelheid (vaak verbose) xml. De kans is zeer groot dat groepen services dezelfde patronen delen (bv. SOAP services die hun berichten op een JMS queue afleveren of HTTP proxy services die services in het afgeschermde interne netwerk ontsluiten).

Dit alles kan tot een hoop herhaalde xml in de gebouwde services leiden en druist daarmee in tegen het DRY (Don't Repeat Yourself) principe. Om dit probleem op te lossen heeft Mule sinds versie 3 patterns geïntroduceerd.

De xml configuratie wordt met behulp van patterns behoorlijk versimpeld. Een bijkomend voordeel is dat overal waar op een specifieke manier wordt geïntegreerd, hetzelfde patroon gebruikt kan worden. Daarnaast spreekt iedereen dezelfde taal als het over een patroon gaat en begrijpt iedereen in een ontwikkelteam ook direct wat ermee bedoeld wordt.

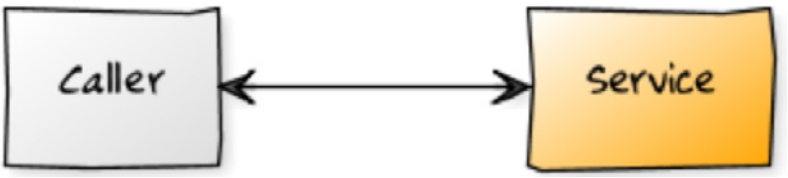
Mule beschikt momenteel over een vijftal patterns:

- Simple Service: endpoint dat een simpele service ontsluit (jax-ws, jax-rs of een mule component);
- Bridge: integratie tussen een inbound en een outbound endpoint;
- Validator: validatie van een bericht in een inbound endpoint;
- HTTP proxy: ontsluit een HTTP service via een alternatief endpoint;
- WS proxy: ontsluit een soap service via een alternatief endpoint.

Deze patterns zullen in dit Whitebook worden toegelicht en een aantal ervan worden uitgewerkt aan de hand van eenvoudige code voorbeelden: De code voorbeelden zijn te downloaden via GitHub en zijn ontwikkeld met behulp van de Anypoint Studio IDE (https://www.mulesoft.com/platform/studio).

Tenslotte zal een aantal van deze patronen vergeleken worden met de standaard mule flow oplossing. Hieruit moet blijken of patterns een welkome toevoeging zijn aan de mule toolkit.

Simple service



Simple service pattern

Het simple service pattern kan gebruikt worden om een mule component, jax-ws of jax-rs service te ontsluiten via HTTP (de mule component laten we in dit Whitebook buiten beschouwing). Mule maakt het dus mogelijk om zonder gebruik van een aparte JEE of Webserver een jax-ws of jax-rs service te ontsluiten. Dit kan interessant zijn als je je server landschap niet te complex wilt laten worden.

JAX-RS JSON service

Het eerste en tevens eenvoudigste voorbeeld dat we uitwerken, is een jax-rs service die json als communicatiemiddel gebruikt. Deze service is opgenomen in het *wb_mule_rest* project binnen de GitHub code (https://github.com/rphgoossens/whitehorses/tree/master/whitebook_mule/wb_mule_rest).
De jax-rs service ontsluit een simpele Product POJO via een HTTP GET operatie:

```
@Path("/product")
public class ProductServiceImpl implements ProductService {
   @Override
   @GET
   @Path("/get")
   @Produces("application/json")
   public Product getProduct() {
      Product prod = new Product();
      prod.setId(1);
      prod.setName("Mattress");
      prod.setDescription("Queen size mattress");
      prod.setPrice(500);

      return prod;
   }
}
```

Het ontsluiten van deze service vindt plaats via een simple-service pattern en vergt de volgende regels code in de wb_mule_rest.xml flow file:

```
<pattern:simple-service name="ProductRESTJson"
   address="http://localhost:8089/rest-json" component-
class="nl.whitehorses.mule.rest.services.json.ProductServiceImpl"
  type="jax-rs">
</pattern:simple-service>
```

Dit is alle benodigde configuratie! Je geeft aan welke klasse je wilt ontsluiten en van welk type de service is, in dit geval jax-rs. Het *address* attribute geeft vervolgens het base address aan waarop je de service wilt aanbieden.

Het *address* attribuut in de getoonde configuratie bevat een lokaal gedefinieerde url. Het is ook mogelijk (en dit geldt voor alle patterns) om met globale referenties te werken. In plaats van het address attribuut wordt dan gebruik gemaakt van het *endpoint-ref* attribuut.

Het is ook mogelijk om transformers in de configuratie van een simple service op te nemen om het request of response bericht te transformeren.

Het voorbeeld project bevat een maven pom welke vanuit Anypoint Studio eenvoudig gestart kan worden. Hierbij wordt een embedded mule server gestart, waarop de service wordt uitgerold en getest kan worden. Voor jax-rs ondersteuning moet een extra dependency toegevoegd worden aan de default gegenereerde maven pom file:

```
<dependency>
<groupId>javax.ws.rs</groupId>
   <artifactId>javax.ws.rs-api</artifactId>
   <version>2.0.1</version>
</dependency>
```

Een HTTP GET op de url http://localhost:8089/rest-json/product /get levert uiteindelijk het verwachte Json antwoord op:

```
{"id":1,"name":"Mattress","description":"Queen size mattress","price":500}
```

JAX-RS XML service

Het ontsluiten van een jax-rs service die xml in plaats van json oplevert, heeft dezelfde mule configuratie.

```
<pattern:simple-service name="ProductRESTXml"
address="http://localhost:8089/rest-xml" component-class="nl.whitehorses.
mule.rest.services.xml.ProductServiceImpl"
   type="jax-rs">
</pattern:simple-service>
```

Het bouwen van de service implementatie heeft echter iets meer voeten in de aarde. Aan de hand van een Product.xsd file wordt m.b.v. JAXB's xjc commando (zie: https://jaxb.java.net) een Product POJO en een ObjectFactory gegenereerd (zie ook: http://docs.oracle.com/javaee/6/tutorial/doc/gkknj.html). Deze klassen worden vervolgens in de service implementatie gebruikt om een Product als JAXBElement te retourneren.

```
@Path("/product")
public class ProductServiceImpl implements ProductService {
   @Override
   @GET
   @Path("/get")
   @Produces("application/json")
   public Product getProduct() {
      Product prod = new Product();
      prod.setId(1);
      prod.setName("Mattress");
      prod.setDescription("Queen size mattress");
      prod.setPrice(500);

      return prod;
   }
}
```

Een HTTP GET op de url http://localhost:8089/rest-xml/product /get levert nu een xml bericht op:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<product xmlns="rest.mule.whitehorses.nl">
   <id>1</id>
   <name>Mattress</name>
   <description>Queen size mattress</description>
   <price>500</price>
</product>
```

JAX-WS service

De code voor soap services is opgenomen in het *wb_mule_soap* project in de voorbeeld code (https://github.com/rphgoossens/whitehorses/tree/master/whitebook_mule/wb_mule_soap). Het voorbeeld project bevat een service die een lijst met Belgische steden produceert. De service is contract-first gebouwd. Dit wil zeggen dat eerst het gewenste contract wordt opgesteld waarna door middel van apache cxf's **wsdl2java** tool (https://cxf.apache.org/index.html) java stubs worden gegenereerd.
De *CityServiceImpl* klasse die de gegenereerde *FindAllCities* @WebService interface implementeert, bevat de custom service code:

```
@WebService(targetNamespace = "http://services.mulewb.whitehorses.nl/
CityService", name = "FindAllCities")
public class CityServiceImpl implements FindAllCities {
   @Override
   public CityServiceResponse findAllCities(Object in) {
       return makeCityServiceResponse();
   }
}
```

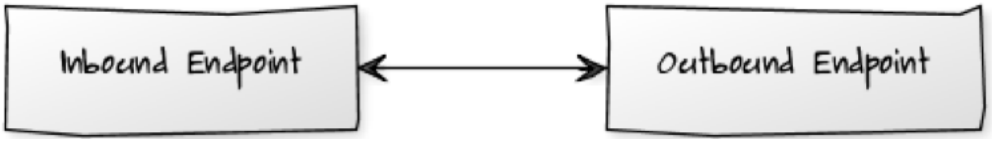
Het ontsluiten van een jax-ws service m.b.v. de simple pattern is, net als bij de jax-rs service, buitengewoon eenvoudig:

```
<pattern:simple-service name="citySOAP"
address="http://localhost:8088/soap/CityService" component-class="nl.
whitehorses.mulewb.services.soap.CityServiceImpl"
type="jax-ws">
</pattern:simple-service>
```

De wsdl voor deze service is na het runnen van het project te bereiken op de url http://localhost:8088/soap/CityService?wsdl. Het sturen van een verzoekbericht naar de *findAllCities* operatie levert het volgende antwoordbericht op:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
      <CityServiceResponse xmlns="http://services.mulewb.whitehorses.nl/
CityService">
            <Cities>
                  <City>
                        <Name>Antwerpen</Name>
                  </City>
                  <City>
                        <Name>Brussel</Name>
                  </City>
                  <City>
                        <Name>Charleroi</Name>
                  </City>
                  <City>
                        <Name>Gent</Name>
                  </City>
            </Cities>
      </CityServiceResponse>
   </soap:Body>
</soap:Envelope>
```

Bridge



Bridge pattern

Een bridge is een patroon om twee endpoints aan elkaar te koppelen. Het bridge pattern ondersteunt transformaties en transacties en kan ingezet worden voor zowel request-response als one-way communicatie. Het bridge pattern stelt je daarnaast in staat om endpoints van verschillende transport-protocollen te koppelen.

Zie hieronder een voorbeeld van een asynchrone bridge tussen een vm en een jms endpoint.

```
<pattern:bridge name="one-way-bridge"
exchange-pattern="one-way"
   inboundAddress="vm://asynchronous-bridge.in"
   outboundAddress="jms://outputQueue" />
```

Validator

Validator pattern

De validator is de meest uitgebreide van de mule patterns. Het validator pattern maakt het mogelijk om validatie uit te voeren op een inkomend bericht voordat het aangeboden wordt aan de daadwerkelijke service.
Validatie vindt plaats door middel van een payload filter. Indien het bericht aan het filter voldoet, wordt het asynchroon doorgestuurd, zo niet dan volgt enkel een foutbericht.
Het doorsturen kan ook synchroon gebeuren, waarna de caller door middel van een *errorExpression* geïnformeerd kan worden, mocht het bericht niet succesvol zijn afgeleverd.

Zie hieronder een voorbeeld van een validator die controleert of het inkomende bericht op een vm queue een Integer betreft. Indien dit zo is, wordt het doorgestuurd naar de vm queue van de achterliggende service:

```
<pattern:validator name="integer-validator"
   inboundAddress="vm://service.in"
   ackExpression="#['Message accepted.']"
   nackExpression="#['Message rejected.']"
     outboundAddress="vm://test-service.in">
   <payload-type-filter expectedType="java.lang.Integer"/>
</pattern:validator>
```

HTTP proxy

HTTP proxy pattern

Een HTTP proxy wordt toegepast om een HTTP service middels een alternatief adres te ontsluiten (zie het *wb_mule_rest* project voor de code voorbeelden). In het simpelste geval wordt het verzoekbericht vanuit de proxy gerouteerd naar de achterliggende service waarna het antwoordbericht van deze service wordt terug gegeven naar de proxy.
De eerder ontsloten simple service die JSON oplevert, wordt met behulp van een mule pattern als volgt geproxied:

```
<pattern:http-proxy name="ProductRESTJsonProxy"
      inboundAddress="http://localhost:8090/rest-json"
      outboundAddress="http://localhost:8089/rest-json"/>
```

Niets meer en niets minder. Een GET op de proxy url http://localhost:8090/rest-json/product/get levert hetzelfde antwoordbericht op als een GET op de originele url.

Het is ook mogelijk om transformaties toe te passen op het inkomende of uitgaande bericht. Zo is het met behulp van wat extra configuratie mogelijk om de eerder gebouwde simple service die xml oplevert, te ontsluiten d.m.v. een proxy die json berichten levert.

```
<mulexml:jaxb-context name="myJaxbContext"
  packageNames="nl.whitehorses.mule.rest" />
<mulexml:jaxb-xml-to-object-transformer
  name="xmlToObjectTransformer" jaxbContext-ref="myJaxbContext" />
<json:object-to-json-transformer name="objectToJsonTransformer" />

<message-properties-transformer name="responseMessageJsonTransformer">
  <add-message-property key="Content-Type" value="application/json" />
</message-properties-transformer>

<pattern:http-proxy name="ProductRESTXmlToJsonProxy"
  inboundAddress="http://localhost:8091/rest-json"
      outboundAddress="http://localhost:8089/rest-xml"
  responseTransformer-refs="xmlToObjectTransformer
  objectToJsonTransformer responseMessageJsonTransformer" />
```

Hier zijn drie transformers voor nodig

- jaxb-xml-to-object-transformer die het jaxb element omzet in een POJO. De transformer heeft een verwijziging nodig naar de java package waar de JAXB ObjectFactory zich in bevindt;
- object-to-json-transformer die de POJO vervolgens omzet in json;
- message-properties-transformer die de Content-Type in de HTTP header omzet van application/xml naar application/json.

Door middel van het *cachingStrategy-ref* attribuut kan tenslotte ook nog caching worden toegepast op de HTTP proxy.

WS proxy

WS proxy pattern

Een WS Proxy is de SOAP equivalent van de HTTP proxy en wordt gebruikt om web services te proxien. Zie voor de code het *wb_mule_soap* project.
In de simpelste vorm bevat het pattern analoog aan de HTTP proxy niets meer en minder dan een name, een *inboundAddress* en een *outboundAddress* attribuut. Ook is het mogelijk om middels het *wsdlFile* of *wsdlLocation* attribuut te verwijzen naar de wsdl van de te proxien service als deze niet aanwezig is op het standaard ?wsdl adres.
In het voorbeeld project is een response transformer toegevoegd in de vorm van een xslt file die de stadsnamen uitbreidt met ‘, Belgie’:

```
<mulexml:xslt-transformer name="cityTransformer" xsl-file="CityName.xslt" />

<pattern:web-service-proxy name="citySOAPProxy"
    inboundAddress="http://localhost:8089/soap/CityService"
  outboundAddress="http://localhost:8088/soap/CityService"
  responseTransformer-refs="cityTransformer">
</pattern:web-service-proxy>
```

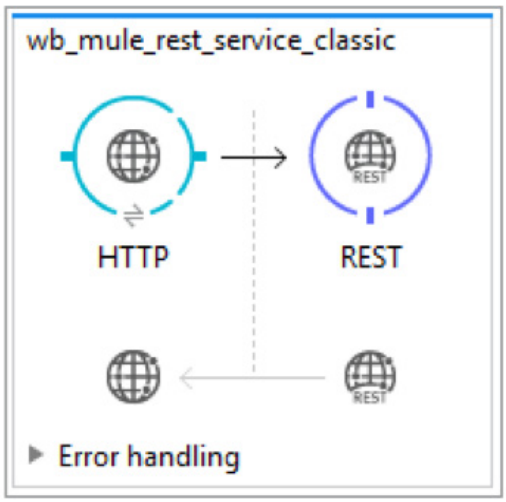
Ook hier is goed te zien dat de configuratie heel eenvoudig is en zich beperkt tot het meest noodzakelijke.
Een beperking in de WS proxy met betrekking tot de transformaties is dat je geen invloed hebt op de WSDL die geserveerd wordt. Die is, met uitzondering van de endpoints, identiek aan de WSDL van de oorspronkelijke service. De transformaties die je kan toepassen, zijn dus eigenlijk beperkt in het feit dat ze een bericht moeten opleveren dat nog steeds aan het oorspronkelijke contract voldoet.
Stel dat je dus in plaats van een proxy een presentatieservice nodig hebt, die de berichten van en naar de originele service wezenlijk verandert, dan voldoet dit patroon niet meer en zul je je heil moeten zoeken in standaard mule flows.

Patterns vs flows

De mule patronen zijn inmiddels de revue gepasseerd. Het is heel interessant om ze te vergelijken met de configuratie die nodig zou zijn geweest als we met standaard mule flows hadden gewerkt. Zie de projecten wb_mule_rest_classic en wb_mule_soap_classic voor de code.

Simple service

Het configureren van een simple jax-rs service zou er met behulp van classic mule flows als volgt uit kunnen zien:



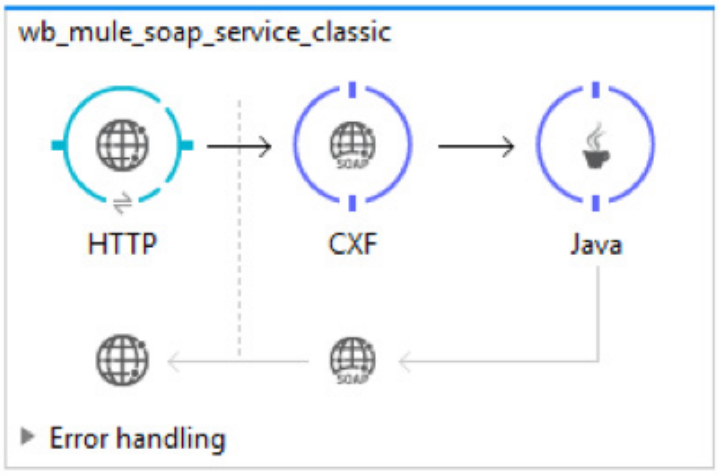
jax-rs service

De bijbehorende xml:

```
<mule ...>
   <http:listener-config name="HTTP_Listener_Configuration"
       host="localhost" port="8088" doc:name="HTTP Listener Configuration"
       basePath="/rest-json" />

   <flow name="wb_mule_rest_service_classic">
       <http:listener config-ref="HTTP_Listener_Configuration"
             path="/*" doc:name="HTTP" />
       <jersey:resources doc:name="REST">
             <component class="nl.whitehorses.mule.rest.service.
ProductServiceImpl" />
       </jersey:resources>
   </flow>
</mule>
```

Het valt meteen op dat de pattern configuratie veel meer to the point is en ook veel leesbaarder is dan de bijbehorende configuratie d.m.v. een mule flow. Dit verschil is nog groter indien we gaan kijken naar de jax-ws variant.



jax-ws service

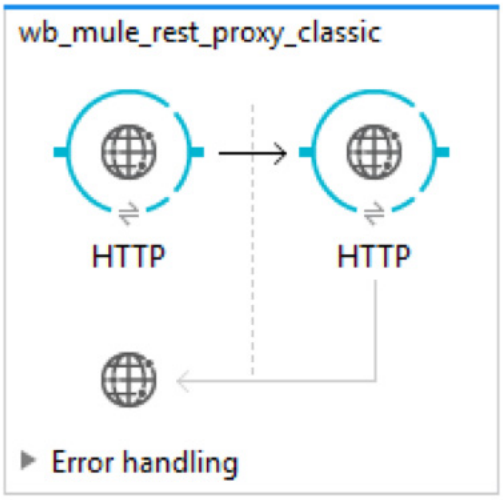
De bijbehorende xml:

```
<mule ...>
   <http:listener-config name="HTTP_Listener_Configuration"
       host="localhost" port="8088" basePath="/soap/CityService"
doc:name="HTTP Listener Configuration" />
   <flow name="wb_mule_soap_service_classic">
       <http:listener config-ref="HTTP_Listener_Configuration"
             path="/" doc:name="HTTP" />
       <cxf:jaxws-service
             serviceClass="nl.whitehorses.mulewb.services.cityservice.
FindAllCities"
             doc:name="CXF" />
       <component class="nl.whitehorses.mulewb.services.soap.
CityServiceImpl"
             doc:name="Java" />
   </flow>
</mule>
```

Hier valt vooral de overbodige configuratie op van de @Service interface. Het lijkt misschien een detail maar het is wel boilerplating die in elke service terugkeert. En ook hier is de pattern variant veel leesbaarder.

HTTP proxy

Een HTTP proxy met behulp van een standaard mule flow ziet er zo uit:



HTTP proxy

De xml:

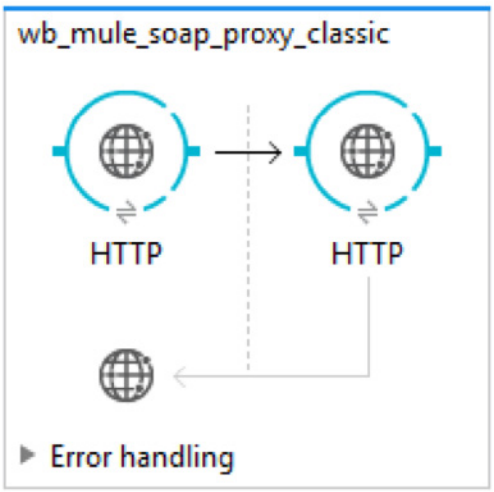
```
<mule ...>
   <http:listener-config name="HTTP_Proxy_Configuration"
       host="localhost" port="8089" basePath="/rest-json" doc:name="HTTP
Listener Configuration" />
   <http:request-config name="HTTP_Request_Configuration"
       host="localhost" port="8088" doc:name="HTTP Request Configuration" />

   <flow name="wb_mule_rest_proxy_classic">
       <http:listener config-ref="HTTP_Proxy_Configuration"
             path="/*" doc:name="HTTP">
       </http:listener>
       <http:request config-ref="HTTP_Request_Configuration"
             path="#[message.inboundProperties['http.request.path']]"
method="#[message.inboundProperties['http.method']]"
             doc:name="HTTP">
       </http:request>
   </flow>
</mule>
```

Hier valt vooral de boilerplate code op die nodig is om het pad en de methode van de proxy door te geven naar de achterliggende service.

WS proxy

Tot slot de WS proxy. Die ziet er met behulp van een standaard flow als volgt uit:



web service proxy

De bijbehorende xml:

```
<mule ...>
   <http:listener-config name="HTTP_Proxy_Configuration"
       host="localhost" port="8089" doc:name="HTTP Listener Configuration" />
   <http:request-config name="HTTP_Request_Configuration"
       host="localhost" port="8088" doc:name="HTTP Request Configuration" />

   <flow name="wb_mule_soap_proxy_classic">
       <http:listener config-ref="HTTP_Proxy_Configuration"
             path="/soap/CityService" doc:name="HTTP" />
       <http:request config-ref="HTTP_Request_Configuration"
             path="#[message.inboundProperties.'http.listener.path']"
method="#[message.inboundProperties.'http.method']"
             doc:name="HTTP" />
   </flow>
</mule>
```

Analoog aan de HTTP proxy valt ook hier de configuratie op die nodig is om het pad en de methode door te geven. Configuratie die niet nodig is indien gebruik wordt gemaakt van de WS proxy pattern.

Conclusie

In dit Whitebook zijn mule patterns behandeld. Mule patterns zijn een alternatief voor standaard mule flows en kunnen worden toegepast voor specifieke integratie patronen. Zoals we hebben laten zien, is een groot voordeel van mule patterns dat er minder (boilerplate) xml nodig is vergeleken met standaard mule flows. Ook zijn patterns simpeler en leesbaarder dan de vergelijkbare oplossingen met flows. Dat zal zeker bij de Validator pattern het geval zijn. Patterns zijn een goed voorbeeld van de toepassing van DRY bij het bouwen van integratie services.
Een groot nadeel van patterns momenteel is dat er geen IDE ondersteuning voor is, iets wat de ontwikkelaars van AnyStudio in de toekomst hopelijk gaan veranderen. Daarnaast ben je ook beperkt in wat de patterns je bieden. Als je bijvoorbeeld logging componenten wilt inpassen, dan voldoen patterns al niet meer, tenzij je je heil zoekt in java code. Je zou kunnen zeggen dat patterns out-of-the-box alleen datgene doen waar ze voor bedoeld zijn.
Vergeleken met bijvoorbeeld de Oracle Service Bus heeft mule voor een hele andere DRY oplossing gekozen. Oracle Service Bus biedt hiervoor sinds 12c templating aan. In een template is een standaard patroon uitgewerkt (bijvoorbeeld een proxy met een validatie en een logging component), waarop services gebaseerd kunnen worden. Dit biedt mijns inziens meer flexibiliteit dan de patterns van mule. Met templates kun je als het ware je eigen patterns bouwen.
Een onderdeel dat we niet behandeld hebben, is pattern inheritance. Patterns kunnen een abstract pattern extenden, waarmee ook een vorm van templating bereikt kan worden.
Het is te hopen en te verwachten dat er in de toekomst meer en meer patterns bij gaan komen om standaard integratie patronen te tacklen. Voor de mule ontwikkelaar is dit alleen maar toe te juichen.

Referenties

Mule in action: https://www.manning.com/books/mule-in-action-second-edition
Mule een open source Enterprise Service Bus: http://www.whitehorses.nl/whitebooks/2011/mule-een-open-source-enterprise-service-bus
Code: https://github.com/rphgoossens/whitehorses/tree/master/whitebook_mule
Mule Configuration Patterns: https://docs.mulesoft.com/mule-user-guide/v/3.7/using-mule-configuration-patterns
Anypoint Studio: https://www.mulesoft.com/lp/dl/studio
Jaxb: https://jaxb.java.net