**De afgelopen jaren zijn er verschillende pogingen gedaan om de problematiek rondom Enterprise Application Integration (EAI) en Business-to-Business (B2B) integratie aan te pakken. De grootste uitdagingen zitten in het beheersen van de kosten en de doorlooptijd van dergelijke projecten. De nieuwste ontwikkeling is de Enterprise Service Bus (ESB) gebaseerd op een service-oriented architectuur (SOA). De producten die op dit gebied voorhanden zijn, zijn weliswaar gebaseerd op standaard technologieën, maar zijn niet onderling uitwisselbaar.**



# Java Business Integration

## *Standaardisatie in ESB-land?*

Een bedrijf zal dus het product kiezen dat het best past bij zijn integratie vraagstuk. Grote kans dat dit product niet voor alle systemen out-of-the-box een oplossing biedt. Geen enkele aanbieder kan namelijk de gehele scope van EAI en B2B dekken aangezien deze dan duizenden applicaties en protocollen moet ondersteunen. Het bedrijf zal hierdoor moeten investeren in een (vaak dure) maatwerk oplossing.

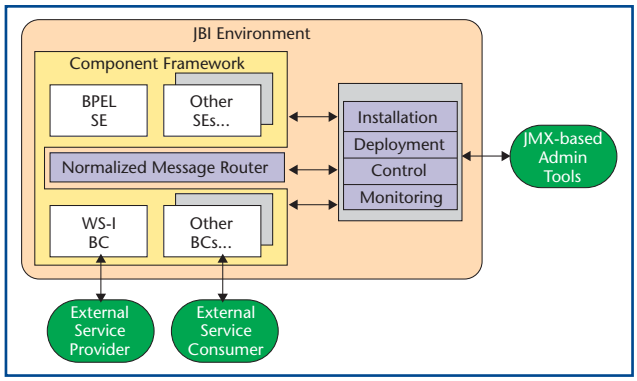
Java Business Integration (JBI) tracht hier een oplossing voor te vinden door een standaard architectuur te definiëren voor alle integratie vraagstukken. Deze architectuur biedt een omgeving waarin third-party componenten kunnen worden 'ingeplugd', zodat deze componenten op een voorspelbare en betrouwbare manier met elkaar kunnen communiceren ondanks dat ze door verschillende aanbieders ontwikkeld zijn.

**JBI Environment** De Java Business Integration specificatie (JSR 208) beschrijft een architectuur voor integratie systemen, waarin componenten met elkaar communiceren door het uitwisselen van berichten via een message bus in plaats van via rechtstreekse koppelingen. De message bus wordt binnen de JBI specificatie de Normalized Message Router (NMR) genoemd. De NMR routeert genormaliseerde berichten tussen componenten die services aanbieden en componenten die deze services consumeren.

Deze ontkoppeling verhoogt de flexibiliteit omdat elk component slechts hoeft te weten hoe het moet communiceren met de bus in plaats van met elk ander component.

Er zijn twee soorten componenten welke in de JBI omgeving kunnen worden geplugged (zie ook Figuur 1):

- *Service Engines (SE):* Deze componenten zijn verantwoordelijk voor het implementeren van business logica. Voorbeelden zijn low-level services zoals data transformatie, caching, routering, maar ook een WS-BPEL instantie waarin ingewikkelde business processen gemodelleerd kunnen worden. SE componenten kunnen services aanbieden, consumeren of beide.
- *Binding Components (BC):* Deze componenten zorgen voor de communicatie tussen de JBI omgeving en de buitenwereld via een specifiek protocol of transport. Aan de ene kant ontsluiten ze externe services voor gebruik binnen de JBI omgeving, aan de andere kant



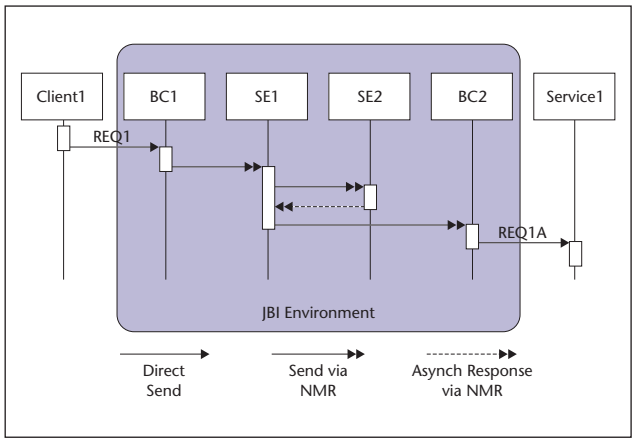

FIGUUR 1. Overzicht JBI Omgeving

bieden zij een ingang voor services die in de JBI omgeving draaien. Een voorbeeld is een HTTP binding component voor het versturen van een bericht naar een SE via HTTP (al dan niet ingepakt in een SOAP bericht). Andere voorbeelden zijn file transfer, JMS, RSS, SMTP, SMS en legacy-applicaties.

Laten we het concept illustreren aan de hand van een voorbeeld. Stel een bedrijf heeft een fulfillment systeem voor de afhandeling van alle communicatie naar zijn klanten. Wanneer een brief of email moet worden verstuurd dient een SOAP service te worden aangeroepen (in Figuur 2 aangeduid als Service1). Datzelfde bedrijf heeft een standaard pakket ingekocht voor de registratie van nieuwe klanten. Dit pakket exporteert dagelijks alle nieuwe klantregistraties naar een directory (aangeduid als Client1). Graag wil het bedrijf deze twee applicaties integreren. Het liefst zonder één van beide applicaties aan te passen, gezien de kosten en de hechte koppeling die dan ontstaat tussen de applicaties.

In Figuur 2 is een oplossing middels JBI gegeven. Een binding component (BC1) controleert periodiek of er een bericht in de export directory staat. Dit bericht wordt doorgestuurd naar de service engine SE1. SE1 laat allereerst het bericht transformeren naar het juiste output formaat (door SE2) en vervolgens wordt het bericht naar een HTTP/SOAP binding component (BC2) gestuurd. Deze verpakt het bericht in een SOAP envelop en stuurt het naar Service1.

Het loskoppelen van de communicatie logica (BC's) van de business en processing logica (SE's) maakt de componenten veel eenvoudiger en flexibeler. Immers de SE hoeft geen rekening te houden met protocol specifieke zaken. Het enige dat de SE moet doen is een protocol-onafhankelijk bericht te genereren en op de bus te zetten. De rol van de BC is om dit protocol-onafhankelijke bericht (het zogenaamde genormaliseerde of neutrale bericht) te versturen over het juiste transport




**FIGUUR 2. Message Adapter**

en om binnenkomende berichten te ontdoen van transport afhankelijke zaken. Wanneer wordt besloten om de service via een ander protocol beschikbaar te stellen, hoeft alleen een nieuwe BC te worden geïnstalleerd.

Tot slot biedt de JBI omgeving een aantal management extensions (JMX) voor het installeren en configureren van de componenten, beheren van de lifecycle en monitoring.

**NORMALIZED MESSAGE ROUTER** Een van de belangrijkste componenten in de JBI architectuur is de Normalized Message Router (NMR). De NMR ontvangt

> Elke component heeft een connectie naar de NMR via het zogenaamde Delivery Channel

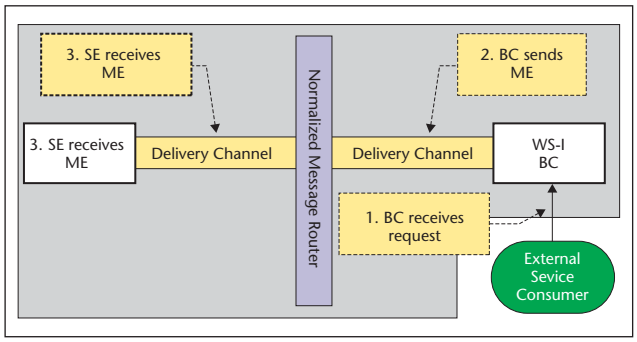
genormaliseerde berichten van JBI componenten (service engines en bindings) en routeert deze berichten naar het juiste component. Dit zogenaamde mediator model ontkoppelt service consumenten van service aanbieders.

JBI gebruikt het concept "genormaliseerd bericht" (normalized message) voor de interactie tussen consumenten en aanbieders. Een genormaliseerd bericht bestaat uit drie delen:

- *De "payload", of neutraal bericht:* Een XML document dat voldoet aan een abstracte WSDL berichttype, zonder protocol specifieke encodering of formattering.
- *Properties (of metadata):* Extra data dat geassocieerd is met het bericht. De properties kunnen security informatie bevatten, of de transactionele context en component specifieke data.
- *Attachments:* Een deel van het bericht kan bestaan uit attachments (met een referentie hier naartoe in de payload). De attachments hoeven geen XML te zijn (en zijn dit meestal ook niet).

Elke component heeft een connectie naar de NMR via het zogenaamde Delivery Channel. Via dit kanaal kunnen berichten zowel worden verstuurd als ontvangen. Een end-to-end interactie tussen een service aanbieder en een consument wordt een MessageExchange (ME) genoemd (zie Figuur 3).

De NMR ondersteunt verschillende Message Exchange Patterns (MEP). De JBI specificatie schrijft voor dat een JBI implementatie in ieder geval onderstaande MEP's moet ondersteunen. De implementatie is vrij om er meer aan te bieden:




**FIGUUR 3. Communicatie tussen componenten en NMR**

- In-Only: consument stuurt een request naar een service aanbieder. De consument weet niet of de aanbieder de aanvraag heeft kunnen verwerken.
- Robust In-Only: consument stuurt een request naar een service aanbieder. Als de aanbieder de aanvraag niet kan verwerken stuurt de aanbieder een fout terug.
- In-Out: consument stuurt een request naar de service aanbieder, met de verwachting een antwoord terug te krijgen. De aanbieder stuurt het antwoord of een fout als er iets mis is gegaan.
- In Optional-Out: consument stuurt een request naar de service aanbieder. Als hier iets misgaat stuurt de aanbieder een fout terug. Optioneel krijgt de consument een antwoord terug. Hierop kan de consument met een fout reageren als er iets mis is gegaan.

**COMPONENT FRAMEWORK** Componenten en de manier waarop deze berichten met elkaar uitwisselen vormen de basis voor de oplossing van een specifiek integratie probleem. De componenten worden als plug-ins in de JBI container geïnstalleerd. Vergelijk het met een Java EE container die Enterprise JavaBeans (EJB) componenten beheert, of een portal waarin portlets zijn gehangen. De JBI container beheert integratie componenten. Je kun je eigen componenten schrijven of componenten van derden gebruiken. Op dezelfde manier als dat je portlets zelf bouwt of hergebruikt.

Om een component te installeren wordt gebruik gemaakt van de management API

Zoals al eerder aangegeven kan de JBI container twee soorten componenten beheren: Service engines (SE) en binding components (BC). Deze componenten verschillen echter alleen qua definitie. Voor het schrijven van je eigen componenten dienen voor beiden dezelfde interfaces geïmplementeerd te worden, te weten:

- `Component`, wordt gebruikt door de container om component specifieke informatie op te vragen.
- `ComponentLifeCycle`, wordt gebruikt door de container om de lifecycle van het component te beheren.
- `Bootstrap`, biedt de mogelijkheid om tijdens installatie en de-installatie extra handelingen uit te voeren.

De volgende interface is optioneel:

- `ServiceUnitManager`, wordt gebruikt om runtime artifacts naar het component te sturen (zie Management).

Daarnaast stelt de container de `ComponentContext` ter beschikking aan het component voor communicatie naar de JBI omgeving. Deze interfaces zijn allemaal te vinden in de package `javax.jbi.component`.

Een component wordt ingepakt in een zip- of jar-archive. Naast de Java-klassen, bevat dit archief tevens alle benodigde libraries en een file descriptor met meta-informatie over het component (jbi.xml). Om een component te installeren wordt gebruik gemaakt van de management API (zie Management).

**VOORBEELD** Als voorbeeld schrijven we een quote service component. Deze service krijgt als parameter de naam van een beroemd persoon mee en retourneert een random uitspraak van de opgegeven persoon (niet echt een service die veel integratie vraagstukken zal oplossen, maar het gaat om het idee). Dit is een typisch voorbeeld van een In-Out message exchange pattern. Immers de consument van deze service verwacht een resultaat (de uitspraak) terug van de service.

Aangezien er maar een paar regels code nodig zijn voor zowel de `Component` als de `ComponentLifeCycle` zullen beide interfaces door een enkele klasse worden geïmplementeerd. De `QuoteServiceEngine`-klasse staat hieronder en spreekt voor zich. Alleen de `start()` methode behoeft wat uitleg. Om ervoor te zorgen dat het `QuoteServiceEngine` object niet blokkeert tijdens het wachten op binnenkomende berichten wordt in een nieuwe thread een `QuoteServiceListener` opgestart. JBI schrijft hier geen standaard manier voor. De `QuoteServiceListener` krijgt als parameter een `DeliveryChannel` mee, het kanaal waarover berichten binnenkomen en uitgaan. Vervolgens wordt het component bekend gemaakt en geactiveerd binnen de JBI Container.

```
package nl.iprofs.jbi.se;

import javax.jbi.*;
```

```
public class QuoteServiceEngine implements
Component, ComponentLifeCycle {

    private ComponentContext context = null;
    private QuoteServiceListener listener;

    private Map<String, String[]> quotes =
new HashMap<String, String[]>();

    public void init(ComponentContext
    context) throws JBIException {
        this.context = context;
    }

    public void start() throws JBIException {
        // Start de QuoteServiceListener om
berichten te verwerken
        listener = new QuoteServiceListener(
context.getDeliveryChannel(), quotes);
        (new Thread(listener)).start();

        // Registreer de component endpoint
bij de JBI environment
        context.activateEndpoint(new
QName("QuoteService"), "QSEndpoint");
    }

    public void stop() throws JBIException {
        // Stop de QuoteServiceListener
        listener.stopProcessing();
    }

    public void shutDown() throws
JBIException {
    }

    public ComponentLifeCycle getLifeCycle()
{
        return this;
    }
...
}
```

De `QuoteServiceListener` verwerkt de `MessageExchanges` die door de NMR naar de QuoteService zijn gestuurd. De listener blokkeert en wacht op binnenkomende berichten door middel van de accept() methode van de `DeliveryChannel`. Wanneer een bericht binnenkomt wordt gecontroleerd of de juiste methode is aangeroepen en of de MEP van het juiste type is (InOut). Vervolgens wordt de naam uit het binnenkomende bericht gehaald, een random uitspraak gegenereerd en er wordt een uitgaand bericht gegenereerd met daarin de uitspraak. De `MessageExchange` wordt vervolgens met de `send()` methode teruggestuurd naar de NMR.

```
package nl.iprofs.jbi.se;

...
public class QuoteServiceListener implements
Runnable {
    private DeliveryChannel channel;
    private Map<String, String[]> quotes;

    public QuoteServiceListener
    (DeliveryChannel channel, Map quotes){
        this.channel = channel;
        this.quotes = quotes;
    }

    public void run() {
        running= true;

        while(running) {
            try {
                MessageExchange me =
                channel.accept();
                process(me);
            } catch (Exception e) { }
        }
    }

    private void process(MessageExchange msg)
throws Exception {

        if (new QName("GetQuote").equals(msg.
getOperation())&& msg instanceof InOut) {

            // We hebben een InOut MEP ont-
vangen, met de "GetQuote" operatie
            InOut inOut = (InOut)msg;

            // Haal naam op
            String from =
getPersonName(inOut.getInMessage().
getContent());

            // Genereer een random quote van
deze persoon
            String quote =
getRandomQuote(from);

            // Creeer een out-message met
            hierin de quote
            NormalizedMessage out = inOut.
            createMessage();
            out.setContent(createOutMessage(
            quote));
            inOut.setOutMessage(out);

            // Stuur de quote terug naar de
NMR
            channel.send(inOut);
        }
    }
}
```

Wanneer we dit component vervolgens installeren in een JBI container (bijvoorbeeld ServiceMix) en we configureren een SOAP binding component dan kunnen we het component buiten de JBI Container testen door onderstaand SOAP bericht te sturen naar het externe endpoint (URL) van dit BC.

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://
schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
     <GetQuote>
        <From>Johan Cruijff</From>
     </GetQuote>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Als antwoord zouden we onderstaande uitspraak kunnen krijgen. Als we in plaats van een SOAP BC een SMS BC hadden geconfigureerd, dan hadden we de uitspraak via SMS kunnen opvragen en ontvangen.

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://
schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
     <GetQuoteResponse>
        <Quote>Als wij de bal hebben kunnen
zij niet scoren.</Quote>
     </GetQuoteResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

**Management Framework** JBI definieert een management structuur, gebaseerd op Java Management eXtensions (JMX). JBI biedt standaard mechanismen voor:

- Installatie van componenten
- Beheer van de lifecycle van een component (stop/start etc.)
- Deployen van artifacts naar componenten
- Monitoring

Om het mogelijk te maken dat componenten beheert kunnen worden dienen zij een aantal JBI interfaces te implementeren. Eén zo'n interface hebben we al gezien, `ComponentLifeCycle`, om de lifecycle van een component te beheren.

Verder is de JBI container verplicht om een aantal JMX MBeans aan te bieden om beheer mogelijk te maken. Daarnaast mogen componenten, optioneel, hun eigen MBeans beschikbaar stellen voor component specifieke management functies, zoals bijvoorbeeld voor het instellen van het loglevel.

Een van de beheerfuncties is het deployen van artifacts naar componenten. JBI componenten fungeren vaak als een soort container, waarbij artifacts kunnen worden toegevoegd om extra logica te bieden. Denk bijvoorbeeld aan een transformatie service waaraan on-the-fly extra XSLT stylesheets kunnen worden toegevoegd of een BPEL service engine waar nieuwe business processen kunnen worden toegevoegd. Een ander voorbeeld is de SOAP BC uit ons voorbeeld waar we moesten configureren dat het binnenkomende bericht naar de QuoteService moest worden gestuurd. Een enkele deployment package, bestemd voor één component wordt een Service Unit genoemd. Verschillende Service Units kunnen gelijktijdig gedeployed worden door deze te packagen in een Service Assembly.

Om het voor een component mogelijk te maken om runtime artifacts te ontvangen dient deze via de `Component.getServiceUnitManager()` een implementatie van de `ServiceUnitManager` te retourneren. Onze quote service wordt pas herbruikbaar wanneer we de uitspraken kunnen configureren. Nieuwe uitspraken kunnen dan heel gemakkelijk gedeployed worden. Voorbeeld implementatie staat hieronder afgebeeld.

```
package nl.iprofs.jbi.se;

...
public class QuoteServiceUnitManager
implements ServiceUnitManager {
    private QuoteServiceEngine se = null;

    public QuoteServiceUnitManager
    (QuoteServiceEngine se) {
        this.se = se;
    }

    public String deploy(String service
    UnitName, String serviceUnitRootPath) {

        File rootDir = new File(serviceUnit
        RootPath);
        File[] quoteFiles =
        rootDir.listFiles();

        for (int i = 0; i
        < quoteFiles.length; i++) {
            // Lees quote file in
            newQuotes = ...
```

```
            se.putQuotesFrom(quoteFile.
getName(), newQuotes.toArray(new String[0]));
        }

        return createComponentTaskResult
        Message("deploy", "SUCCESS");
    }

...
}
```

**TOOLING** Naast de verplichte Ant tasks die elke container moet aanbieden om gemakkelijk componenten en artifacts te installeren/deployen, bieden de meeste JBI containers extra tooling. Zo komen ServiceMix en Petals met een Maven JBI plugin waarmee nieuwe project types, zoals Component (Service Engine, Binding Component), Service Unit en Service Assembly kunnen worden geconfigureerd zodat de archive file automatisch wordt gegenereerd.

**CONCLUSIE** De tendens naar open standaarden en specificaties binnen de hele industrie is er één die moet worden toegejuigd. Het voordeel voor de klant is duidelijk. Een standaard als JBI kan hem in de toekomst veel geld besparen. Hij hoeft geen product aan te schaffen waar hij de helft niet van gebruikt en die hij daarnaast moet uitbreiden met maatwerk (en dus dure) oplossingen omdat het pakket net niet ondersteunt wat hij nodig heeft. In plaats daarvan kiest hij een JBI container tezamen met alleen die standaard (en dus goedkopere) componenten die voor zijn integratie vraagstuk nodig zijn. Doordat JBI gebruik maakt van een service-oriented architectuur en ESB infrastructuur biedt het de flexibiliteit om componenten te laten samenwerken zonder dat ze specifieke kennis van elkaar moeten hebben. Hierdoor zijn individuele componenten makkelijker te vervangen door nieuwe of verbeterde implementaties.

Zowel commerciële aanbieders als opensource projecten hebben de JBI standaard geadopteerd binnen hun eigen ESB platform. Laten we hopen dat ze geleerd hebben van de JEE specificatie en zorgen dat componenten ook daadwerkelijk uitwisselbaar zijn tussen JBI implementaties, zonder container specifieke zaken te hoeven configureren. Alleen dan zal het beoogde doel van JBI gehaald worden en dat is alleen maar in het voordeel van de klant.

## Bronnen


- JSR-208 Java Business Integration (JBI) - http://jcp.or/en/jsr/detail?id=208
- David Chappell, Enterprise Service Bus, O'Reilly, 2004.
- Open ESB - https://open-esb.dev.java.net/
- Mule Open Source ESB - http://mule.codehaus.org/
- ServiceMix Open Source ESB - http://servicemix.org/
- Petals Open Source JBI Implementatie - http://petals.object-web.org



*Mario Klaver (mklaver@iprofs.nl) is werkzaam als software architect bij IPROFS. Hij is bijna 10 jaar werkzaam in de ICT en heeft veel ervaring op het gebied van EAI en B2B. De laatste jaren heeft hij veel succesvolle projecten gedaan gebaseerd op een service-oriented architectuur.*