Zuverlässiger Busverkehr

JBossESB: SOA für Faule

Jan Diller, Jörg Viola

Serviceorientierte Architekturen (SOA) kapseln Geschäftslogik in appetitliche Häppchen – die Services bzw. Dienste. Ein Dienst hat eine stabile Schnittstelle und ist möglichst interoperabel aufrufbar. Aus diesen Mosaikstückchen lassen sich schnell komplexe Abläufe beliebig zusammenstellen. Soweit die Theorie. In der Praxis ist die Extraktion von Diensten aus vorhandenen Legacy-Anwendungen und ihre Ansteuerung nicht umsonst zu haben. Hilfe bieten verschiedene Integrationsplattformen und -frameworks an, insbesondere auch die ESB-Systeme. Sie liefern einen Werkzeugkasten mit Hilfsmitteln zur Anbindung verschiedenster Technologien und Anwendungen, zum zuverlässigen Aufruf der Dienste und im Idealfall auch zur Orchestrierung ganzer Abläufe. Dann auch noch „kostenlos" wie beim JBossESB? – Da ist entweder ein Haken oder man muss dringend mal reinschauen!

Grau ist alle Theorie

Der Enterprise Service Bus (ESB) von JBoss ist, wie alle Produkte der Firma, zunächst Open Source (LGPL 2.1). Der Quelltext ist also frei zugänglich und seine Verwendung auch ohne Erwerb einer Lizenz erlaubt. Im Rahmen der SOA-Plattform von JBoss [Fink09] kann das Produkt aber auch eingekauft werden, dann genießt man besser getestete und zertifizierte Versionen mit entsprechendem Support.

Im Kern besteht der JBossESB natürlich aus dem Bus, einem zuverlässigen Medium zum Transport von Nachrichten. Die Überraschung ist aber, dass der ESB genau genommen dieses Medium nicht enthält, sondern nur Werkzeuge zum Zugriff darauf, sogenannte Provider. Oft wird man eine JMS-Queue wählen, wie z. B. JBoss Messaging, Oracle AQ oder MQ Series. Der ESB unterstützt aber, insbesondere für die Einspeisung von Nachrichten aus externen Systemen, auch E-Mail, Datenbank oder Dateisystem. Er stellt eine Schnittstelle zur Verfügung, mit der auf dieses Medium Nachrichten geschrieben und von dort gelesen werden können.

Ein Dienst besteht aus einem Listener, der auf dem Medium lauscht und relevante Nachrichten als Dienstaufruf interpretiert. Die Nachricht wird dann in eine Kette von Actions hineingeworfen, die sie verarbeiten und dabei die eigentliche Geschäftslogik implementieren. Dabei kann eine weitere Nachricht erzeugt und dadurch ein weiterer Dienst aufgerufen werden.

Weiter stellt der ESB noch eine Vielzahl an Werkzeugen (Adaptern) zum Zugriff auf externe Informationsquellen zur Verfügung: http(s), (s)ftp, Dateisystem, JMS, E-Mail, SQL, JCA, TCP/IP und SOAP sind eine Auswahl der unterstützten Technologien.

Schließlich spielt die Transformation von Nachrichten eine wesentliche Rolle, ebenso wie die inhaltsbasierte Weiterleitung (CBR: Content-Based Routing): Wird beispielsweise vom Dateisystem eine CSV-Datei mit Überweisungsdaten gelesen, so kann ein erster Dienst die Nachricht in handliches XML transformieren, ein zweiter die Höhe der Überweisung prüfen und bei großen Beträgen die Nachricht an einen Prüfdienst weiterleiten, nicht ohne vorab aus Sicherheitsgründen die Kontonummern verschleiert zu haben.

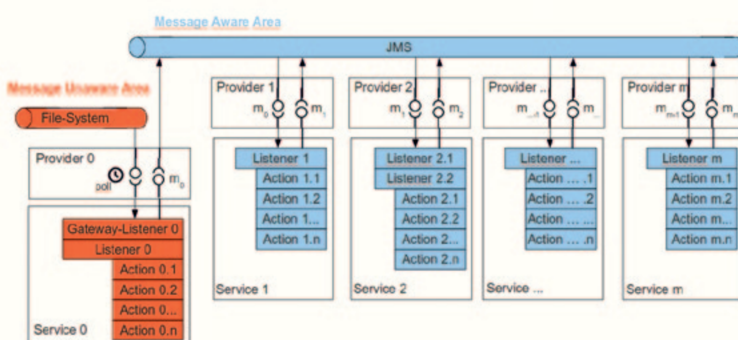
Auspacken und loslegen ...

Herunterladen, auspacken, starten. So einfach ist es wirklich, wenn man sich entscheidet, das ESB-Server-Paket zu nutzen. Man erhält dann einen JBossAS (ein Applikationsserver, der grundsätzlich die Laufzeitumgebung bereit stellt) mit darin installiertem ESB. Dieser bleibt jedoch aufs Default-Profil beschränkt.

Ambitionierter ist die Erweiterung eines existierenden JBossAS, falls ein solcher bereits produktiv ist. Zu diesem Zweck kann der ESB separat heruntergeladen werden. Per Ant wird der ESB dann in diesen Server installiert. Hierzu sollte der Applikationsserver dann mindestens Version 4.2.3 bzw. 5.1.0 haben.

Schematische Arbeitsweise JBossESB

Zentrales Informationselement des JBossESB ist die zwischen den Services gehandelte Nachricht. Zu Beginn der Verarbeitungskette muss diese zunächst erzeugt werden, dies geschieht in einem sogenannten Gateway-Listener.



Hier können bereits erste Transformationsschritte vorgenommen werden. Von diesem Message-Unaware-Bereich geht es nun in den Message-Aware-Bereich. Der Nachrichtenaustausch findet über den Bus statt.

Welches Medium den Bus realisiert, ist nicht näher definiert. Theoretisch könnte der Bus auf dem Dateisystem oder in der Datenbank realisiert werden, praktisch wird dies keiner tun. Hier kommt typischerweise JMS zum Einsatz.

Das Lauschen auf dem Bus erledigen die Provider, welche ihrerseits die registrierten Listener aufrufen. Der Listener seinerseits stößt dann die eigentliche Verarbeitungskette an. Hierbei kann eine Verarbeitungskette durchaus an mehrere Listener gebunden sein, was in der Praxis bedeutet, dass es mehrere Auslöser für ein und dieselbe Verarbeitungskette gibt.

Anatomie eines Dienstes

Für seine ESB-Dienste definiert JBoss eine Verzeichnisstruktur. Typische Inhalte dieser Struktur sind:

- */META-INF/jboss-esb.xml*: Dies ist der zentrale Deskriptor für einen Dienst. Er wird unten noch ausführlich erläutert.
- */META-INF/deployment.xml*: Optional, zeigt Abhängigkeiten zu anderen Diensten auf.
- *Java-Klassen:* Selbst codierte Actions, Listener und andere Komponenten.
- *JARs:* Benutzte Bibliotheken können innerhalb des ESB-Archivs liegen.
- *queue-service.xml:* Benötigte JMS-Queues können hier deklariert werden. Dies ist der einfachste Weg, einen self-contained Dienst auszuliefern. Die Queues werden dann beim Deployment des Dienstes automatisch angelegt.

Mit dem JAR-Algorithmus gepackt und mit der Endung .esb versehen, wird diese Komponente auf dem Applikationsserver bereitgestellt.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<jbossesb xmlns=
"http://anonsvn.labs.jboss.com/labs/jbossesb/ trunk/product/etc/schemas/
xml/jbossesb-1.0.1.xsd">
  <providers>
    <jms-jca-provider name="JBossMQ" connection-
factory="ConnectionFactory">
      <jms-bus busid="shoeStoreJMSGateway">
        <jms-message-filter dest-type="QUEUE"
          dest-name="queue/shoeStore"/>
      </jms-bus>
    </jms-jca-provider>
  </providers>
  <services>
    <service category="Retail" name="ShoeStore">
      <listeners>
        <jms-listener name="shoeStore"
          busidref="shoeStore"/>
      </listeners>
      <actions>
        <action name="println"
          class="org.jboss.soa.esb.actions.SystemPrintln" />
      </actions>
    </service>
  </services>
</jbossesb>
```

Listing 1: Beispiel eines typischen Dienst-Deskriptors

Listing 1 zeigt einen typischen Dienst-Deskriptor. Wie man sieht, werden in einem Deskriptor Provider und Services deklariert. Im Beispiel gibt es nur einen JMS-Provider zur Queue **`queue/shoeStoreJMSGateway`**. Damit ist ein Bus deklariert, an dem die Dienste „hängen". (Tatsächlich wird hier eine JCA-JMS-Provider verwendet. Die Queue-Zugriffe erfolgen hier transaktionell. Fehler in der Actions-Kette führen zum Rollback und erneutem Aufruf des Service.)

Diese Queues können am einfachsten durch einen *-service.xml-Deskriptor erzeugt werden. Ein Beispiel zeigt Listing 2. In dieser exemplarischen jboss-esb.xml ist nur ein Dienst beschrieben. Dieser Dienst besteht aus Listener und Actions. Der Listener stellt die Verbindung zum Provider her. Soll ein Dienst über mehrere Provider angesprochen werden, reicht die Definition weiterer verknüpfter Listener an den Dienst aus.

```
<?xml version="1.0" encoding="UTF-8"?>
<server>
  <mbean code="org.jboss.jms.server.destination.QueueService"
    name="jboss.esb.quickstart.destination:service=Queue,
      name=shoeStore"
    xmbean-dd="xmdesc/Queue-xmbean.xml">
    <depends optional-attribute-name="ServerPeer">
      jboss.messaging:service=ServerPeer
    </depends>
    <depends>
      jboss.messaging:service=PostOffice
    </depends>
  </mbean>
</server>
```

Listing 2: *-service.xml-Deskriptor zur Erzeugung einer Queue

Schließlich wird eine Liste von Actions aufgeführt. Eine Action ist im Wesentlichen eine Klasse, welche folgende Methode implementiert:

```
public Message process(final Message message)
    throws ActionProcessingException ;
```

Wenn der Listener vom Provider eine Nachricht erhält, so erzeugt er eine Instanz vom Typ **`Message`** und ruft die Methode **`process()`** der ersten Action damit auf. Die zweite Action wird mit dem Ergebnis der ersten aufgerufen und so weiter.

Actions können selbst entwickelt werden, dazu wird man von einer abstrakten Basisklasse ableiten und einige wenige Methoden implementieren, die in Listing 3 angegeben sind. Das **`Message`**-Interface erlaubt dabei die volle Kontrolle über die ESB-Nachricht, wie im Beispiel gezeigt.

```
public class ActionXXXProcessor
    extends AbstractActionPipelineProcessor {
  public ActionXXXProcessor(ConfigTree config)
      throws ConfigurationException {
    String reqOpt = config.getParent().getRequiredAttribute("reqOpt");
    String obtOpt = config.getParent().getAttribute("obtOpt");
    String defOpt = config.getParent().getAttribute("defOpt", "defVal");
  }
  public Message process(final Message message)
      throws ActionProcessingException {
    String beispiel = (String)message.getBody().get("beispiel");
    String upper = beispiel.toUppercase();
    message.getBody().add("gross", upper);
    return message;
  }
}
```

Listing 3: Actions

Abschließend muss noch entschieden werden, wie der Dienst aufgerufen wird. Dies geschieht grundsätzlich asynchron, jedoch gibt es verschiedene Verfahren:

- Beim One-way-Aufruf wird der Dienst (zuverlässig) aufgerufen, das Ergebnis jedoch nicht benötigt („fire and forget"). Dies geschieht im Prinzip, indem eine Nachricht auf den Bus zu diesem Service gelegt wird.
- Beim request/reply-Aufruf wird ein Bus angegeben, auf den eine Ergebnisnachricht gelegt wird. Nach Abarbeitung des Dienstes erfolgt also eine Art Callback zu einem vorher festgelegten Dienst, per default zum aufrufenden Dienst.

Helferlein

Gerade wurde gezeigt, wie der ESB prinzipiell verwendet wird. Der Charme des Systems rührt aber hauptsächlich von dem großen Werkzeugkasten mit vielen Actions her, der sofort genutzt werden kann, um externe Systeme anzusprechen, Nachrichten zu transformieren und erneut zu versenden und somit ganze Abläufe zu choreografieren. Tabelle 1 reißt die Möglichkeiten an.

`ByteArrayToString`, `LongToDate` usw.	Nichttriviale Typumwandlungen
`ObjectInvoke`	Ruft eine beliebige Methode auf
`ObjectToCSV`, `ObjectToXStream`, `XStreamToObject`	CSV- und XML-Konvertierung
`XsltAction`	Nachrichtentransformation mittels XSLT
`SmooksAction`	Nachrichtentransformation mittels Smooks
`BpmProcessor`	Erzeugen, Starten und Stoppen von jBPM-Prozessen
`GroovyActionProcessor`	Groovy Scripting
`ScriptingAction`	BSF Scripting
`EJBProcessor`	Aufruf einer Stateless Session-Bean aus EJB 2 oder 3
`Aggregator`	Aufsammeln mehrere Nachrichten
`EchoRouter`	Ausgabe der Nachricht im Log
`HttpRouter`	HTTP-Aufruf mit der Nachricht als Parameter
`JMSRouter`	Nachricht per JMS versenden
`EmailRouter`	Nachricht per E-Mail versenden
`ContentBasedRouter`	Nachricht per XPath oder Drools regelbasiert weiterleiten
`StaticRouter`	Nachricht an einen Dienst weiterleiten, Action-Liste beenden
`SyncServiceInvoker`	Synchroner Dienstaufruf
`StaticWireTap`	Nachricht an einen Dienst weiterleiten, mit Action-Liste fortfahren
`EMailWireTap`	Nachricht per E-Mail versenden, mit Action-Liste fortfahren
`Notifier`	Zusammen mit verschiedenen Targets ein eigenes Event-System
`SOAPProcessor`, `SOAPClient`	SOAP-Dienst aufrufen
`SchemaValidation`	Validierung einer Nachricht gegen ein XSD

Tabelle 1: JBossESB eigene Transformations- und Router-Klassen

Außerdem gibt es viele Erweiterungen, zum Teil als Open Source. Das soatools-Projekt auf github etwa wartet auf mit:
- einem Gateway zur Microsoft Message Queue (MSMQ),
- einem LogStore zur Analyse von Abläufen im Problemfall und
- einem Framework zur komfortablen und sicheren Manipulation von Nachrichteninhalten (MFM).

Unterstützung bei der Entwicklung von ESB-Projekten liefert das entsprechende Eclipse-Plug-in von JBoss.

Fazit: Geht's oder geht's nicht?

Wir setzen den JBossESB für Projekte in der Telekommunikation (Geräte-Provisionierung) und bei Web-Portalen (Ablage von Immobilien-Angeboten in verschiedenen Portalen) erfolgreich ein.

Das skizzierte Modell des einfachen Plug'n'Play lässt einen schnellen Einstieg zu. Wer sorgfältig vorgeht, erhält eine übersichtliche Dienstelandschaft. Dies, gepaart mit dem Message Queuing führt zu einer vergleichsweise einfachen betrieblichen Handhabbarkeit.

Jedoch sind Wartbarkeit, Skalierbarkeit und Ausfallsicherheit keine Selbstläufer und erfordern einiges an Know-how und Disziplin.

Literatur Links

[Chap04] D. A Chappell, Enterprise Service Bus, O'Reilly, 2004, `http://oreilly.com/catalog/9780596006754`

[Fink09] T. Fink, Die SOA-Plattform von JBoss, in: JavaSPEKTRUM, 1/2009

[JBossESBPrgG] JBossESB Programmers Guide, `http://docs.jboss.org/jbossesb/docs/4.9/manuals/html/Programmers_Guide/index.html`

[OcESB] ObjectCode GmbH ESB Referenz, `http://www.objectcode.de/esb`

[RaDi08] T. Rademakers, J. Dirksen, Open-Source ESBs in Action, Manning Publications Co., 2008, `http://www.manning.com/rademakers/`

[SoaTools] `http://github.com/untoldwind/soatools`

Jan Diller arbeitet als Softwarearchitekt, Anwendungsentwickler und Coach bei der ObjectCode GmbH. Schwerpunktmäßig beschäftigt er sich hier mit der Realisierung von Middleware-Projekten im JEE-Umfeld. Die vergangenen zwei Jahre gehörte der ESB von JBoss zu seinen Aufgabenschwerpunkten.
E-Mail: diller@objectcode.de

Jörg Viola, Gesellschafter und Geschäftsführer der ObjectCode GmbH, beschäftigt sich mit Internet-Software- und Business-Development. Mit Begeisterung formt er aus modernsten Technologien sinnvolle Anwendungen für Internet, Mobile und Social.
E-Mail: viola@objectcode.de