SOA@T-Mobile (2)

In diesem Teil geht es um die Laufzeitumgebung und das Routing innerhalb der SOA-Backplane. Dabei wird die konzeptionelle sowie die technische Umsetzung dargestellt und die Konfigurationsartefakte erläutert.

von Carsten Sensler und Andre Karalus

Im ersten Teil dieses dreiteiligen Artikels wurden die Grundlagen des SOA-BP-Programms der T-Mobile beschrieben. Es wurde detailliert dargestellt, wie der ESB aufgebaut ist, wofür ein zentrales Service Repository benötigt und wie es umgesetzt wird. Außerdem wurde der architekturelle Blueprint der SOA-Backplane erläutert. Im Folgenden geht es um die Laufzeitumgebung und das Routing innerhalb der SOA-Backplane.

SOA-Backplane: Runtime

Ein Service Consumer (SC) kommuniziert mit einem Service Provider (SP) über die SOA-BP im Wesentlichen völlig transparent, beide benötigen also kein Wissen über die Beschaffenheit des ESBs bzw. der ihm zugrunde liegenden Infrastruktur. Die einzige Infrastrukturkomponente, deren reale Adresse sie kennen müssen, und zu der eine Firewall-Freischaltung bestehen muss, ist der CAL (Common Access Layer). Der CAL dient damit als Gateway zum ESB.

Bei der Kommunikation über die SOA-Backplane haben sich SC und SP auf eine gemeinsame Schnittstelle in Form einer WSDL geeinigt. Die WSDL-Message ist dabei so definiert, dass die SOAP-Nachricht aus zwei Elementen besteht:

- *eiMessageContext*-Element: Es enthält die technischen Informationen für die SOA-BP-Runtime.
- Data-Element, das die fachlichen Nutzdaten (Payload) der Nachricht definiert.

Folgende Elemente des *eiMessageContext* sind besonders wichtig:

- *sender*: Name der Komponente, die die Nachricht verschickt.
- *target*: Port der Komponente, die die Nachricht verschickt (ist optional, wenn die Komponente diesen Typ Nachricht nur an genau eine Providing Component schickt, andernfalls mandatorisch).
- *timeLeft*: Zeitspanne, wie viel Zeit dem ESB verbleibt, um die Antwort zum SC zurück zu transportieren. Da die Zeit, die der ESB selbst benötigt, unbedeutend klein ist (10-100ms), hat dies Auswirkung darauf, wie lange auf die Antwort des SP gewartet wird.

Die Elemente *sender* und *target* identifizieren den SC eindeutig bezüglich des statischen Routings. Eine Beispielnachricht für einen synchronen Service zeigt Listing 1, die dazugehörige WSDL ist in Listing 2 auszugsweise abgebildet.

Der CAL nimmt einen Request vom SC entgegen und bestimmt anhand der *sender/target*-Kombination im *eiMessageContext* den logischen Ausgangsport (Using Port) der aufrufenden Komponente [1]. Dank der Konfiguration, die der CAL durch das Service Provisioning erfahren hat, kann er daraufhin die nöti-

Abkürzungen

CAL – Common Access Layer
CEISeR – Central Enterprise Intergation Service Repository
CR – Change Request
EMS – Enhanced Messaging Service
ESB – Enterprise Service Bus
GSPA – Global Service Provisioing Agent
LSPA – Local Service Provisioning Agent
MEP – Message Exchange Pattern
oAW – openArchitectureWare
Payload – fachliche Nutzdaten einer Nachricht
SOA-BP – SOA-Backplane
SP – Service Provider: serviceanbietende Komponente
SC – Service Consumer: servicenutzende Komponente
SPT – Service Provisioning Team

Quellcode auf CD

gen Informationen zur Kommunikation über den JMS-Bus (ESB) ermitteln. Die wesentliche Information ist der Name der Queue.

Damit in Problemsituationen in der Runtime schnell und effizient Kommunikationsprobleme identifiziert und analysiert werden können, schreibt der CAL bei wichtigen Punkten in der Kommunikation LogPoints in den LogStore. Insgesamt sind acht verschiedene LogPoints vorgesehen. Abbildung 2 zeigt die acht LogPoints im synchronen MEP für eine internationale Kommunikation.

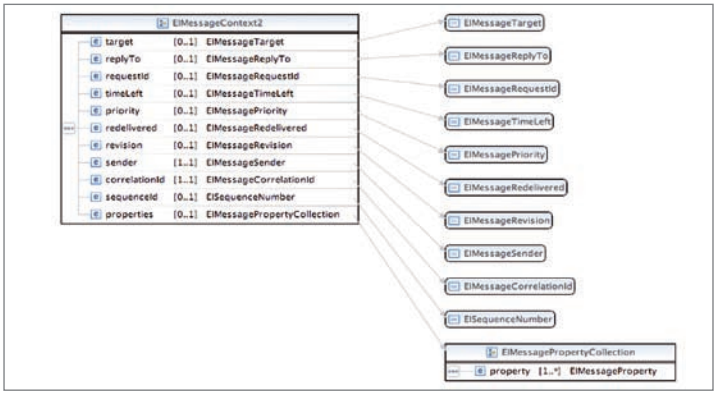


Abb. 1: SOA-Backplane: eiMessageContext2

Ab in den Supermarkt – Queues im ESB

Der JMS-Bus ist sternförmig (Hub and Spoke) aufgebaut, internationaler Traffic wird also vom entgegennehmenden JMS-Server über den EMS-Hub geleitet und beim empfangenden EMS-Server bereitgestellt. Beim EMS-Server werden dazu Remote Queues und Global Queues eingerichtet. Der entgegennehmende und der empfangende EMS-Server besitzen jeweils eine Remote Queue, die auf die globale Home Queue auf dem Hub verweist. Stellt ein JMS-Client (der CAL) die Nachricht in eine Remote Queue ein (LogPoint 2), dient diese nur

Listing 1: Beispielnachricht eines synchronen Service

```
<soapenv:Envelope xmlns:soapenv="http://schemas.
 xmlsoap.org/soap/envelope/" xmlns:tes="http:// itse.ei.
  tmobile.de/javamagazin/services/testping" xmlns:dat=
   "http://messaging.ei.tmobile.net/datatypes" xmlns:
        tes1="http://itse.ei.tmobile.de/javamagazin/
                                    datatypes/testping">

 <soapenv:Header/>
 <soapenv:Body>
  <tes:testPing>
   <tes:eiMessageContext>
    <dat:target>testPing</dat:target>
    <dat:sender>tmd.tmobile.demo.
                    jm.JavaMagazin:Demo</dat:sender>
    <dat:correlationId>CAS-002123-20080811
                                  </dat:correlationId>
   </tes:eiMessageContext>
   <tes:data>
    <tes1:someData>JavaMagazin_Demo</tes1:someData>
   </tes:data>
  </tes:testPing>
 </soapenv:Body>
</soapenv:Envelope>
```

Listing 2: SOA-Backplane-Beispiel-WSDL (auszugsweise)

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions name="testPingDefinitions"
targetNamespace="http://itse.ei.tmobile.de/javamagazin/
   services/testping" xmlns="http://schemas.xmlsoap.org/
                                                wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                                                xmlns:
tns="http://itse.ei.tmobile.de/javamagazin/services/
testping" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
...
 <xsd:element name="TechnicalExceptionElement" type=
    "ei_messaging_datatypes:SOABPException" xmlns:xsd=
                  "http://www.w3.org/2001/XMLSchema"/>
 <xsd:element name="testPing" xmlns:xsd=
                  "http://www.w3.org/2001/XMLSchema">
   <xsd:complexType>
    <xsd:sequence>
    <!-- REQUIRED structure-element „eiMessageContext" -->
    <xsd:element maxOccurs="1" minOccurs="1" name=
      "eiMessageContext" type="ei_messaging_datatypes:
                                        EIMessageContext"/>
     <!-- REQUIRED structure-element „data" -->
     <xsd:element maxOccurs="1" minOccurs="1" name=
          "data" type="de_tmobile_ei_itse_javamagazin_
                datatypes:testPing_testPingRequest"/>
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
 <xsd:element name="testPingOutput" xmlns:xsd=
                  "http://www.w3.org/2001/XMLSchema">
   <xsd:complexType>
    <xsd:sequence>
     <!-- REQUIRED structure-element "eiMessageContext" -->
     <xsd:element maxOccurs="1" minOccurs="1"
name="eiMessageContext2" type="ei_messaging_
                              datatypes:EIMessageContext"/>
     <!-- REQUIRED structure-element "data" -->
     <xsd:element maxOccurs="1" minOccurs="1" name=
          "data"type="de_tmobile_ei_itse_javamagazin_
                datatypes:testPing_testPingResponse"/>
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
</xsd:schema>

 </wsdl:types>
...
 </wsdl:message>
 <!-- ************************************** -->
 <wsdl:portType name="testPing">
  <wsdl:operation name="testPing">
   <wsdl:input message="tns:testPingInput"/>
   <wsdl:output message="tns:testPingOutput"/>
   <wsdl:fault message="tns:TechnicalExceptionMessage"
                        name="TechnicalExceptionFault
"/>
  </wsdl:operation>
 </wsdl:portType>
 <!-- ************************************** -->
 <wsdl:binding name="testPingBinding"
type="tns:testPing">
  <soap:binding style="document" transport=
                        "http://schemas.xmlsoap.org/
soap/http"/>
  <wsdl:operation name="testPing">
   <soap:operation/>
   <wsdl:input>
    <soap:body use="literal"/>
   </wsdl:input>
   <wsdl:output>
    <soap:body use="literal"/>
   </wsdl:output>
   <wsdl:fault name="TechnicalExceptionFault">
    <soap:fault name="TechnicalExceptionFault" use=
                                        "literal"/>
   </wsdl:fault>
  </wsdl:operation>
 </wsdl:binding>
 <!-- ************************************** -->
 <wsdl:service name="testPingService">
  <wsdl:port binding="tns:testPingBinding"
name="testPing">
   <soap:address location="http://localhost:8080/
       TestInfrastructure1/de-tmobile-ei-itse-javamagazin-
            services-testping-testPing.soap2jms/1.0"/>
  </wsdl:port>
 </wsdl:service>
</wsdl:definitions>
```

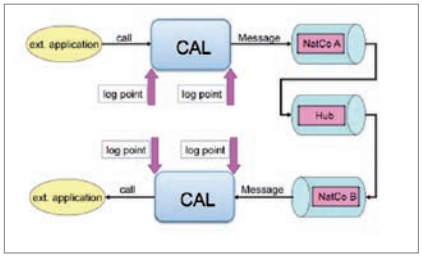


Abb. 2: Acht LogPoints des CALs

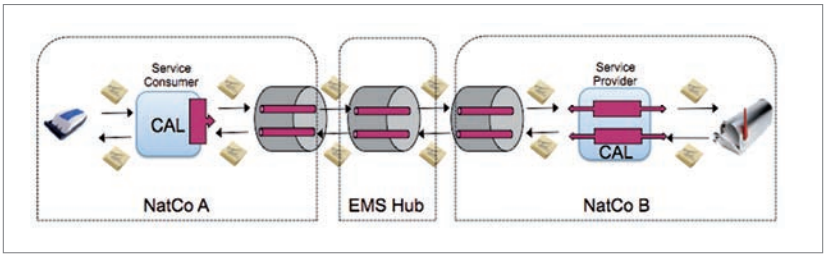


Abb. 3: Nachrichtenfluss bei internationalem Nachrichtenaustausch

als Proxy, tatsächlich wird die JMS-Message in die globale Home Queue auf dem Hub gestellt. Ein anderer JMS-Client (CAL), der sich an eine andere Remote Queue (eines anderen EMS-Servers im Spoke) auf diese globale Queue als Message Consumer angemeldet hat, kann die JMS-Message nun von dem Hub empfangen (LogPoint 3).

Um den Traffic innerhalb einer Landesgesellschaft effizient zu gestalten, kommunizieren SC und SP (bzw. deren CALs) bei einem nationalen Kommunikationsszenario nur über eine „normale" JMS-Queue an dem der Landesgesellschaft zugeordneten EMS-Server (Abb. 3). Für den CAL selbst als JMS-Client macht das im Prinzip keinen Unterschied. Schließlich müssen die unterschiedlichen Arten von Queues auf den entsprechenden EMS-Servern durch die Serviceprovisionierung angelegt werden, damit überhaupt über den ESB kommuniziert werden kann.

Das SOA-Backplane-Konzept für Queues sieht vor, dass (abhängig vom MEP) ein Paar JMS-Queues (für den Request und den Response) pro Provider angelegt werden. Für den CAL ist die grundsätzliche Arbeit beim Routing der Request Message des SC damit einfach: Nachdem er aus seiner Konfiguration den Namen der JMS-Queue für den SP, mit dem der SC statisch verbunden ist, gefunden hat, erzeugt er eine JMS-Message, die die Payload des Requests und die JMS-Header-Properties trägt, die grundsätzliche Informationen des *eiMessageContext* und der Nachricht tragen. Auf der Gegenseite ist ein CAL (oder dieselbe CAL-Instanz bei nationalem Nachrichtenaustausch) aufgrund der Konfiguration als Listener zu dieser Queue registriert und nimmt die Nachricht an. Diese Nachricht wird dem technischen Endpunkt des SP zugestellt (Outer-bound der SOA-BP) – beim synchronen MEP wird die Response des SP dann auf die Response-JMS-Queue gelegt und vom CAL auf der Seite des SC wieder empfangen und an den wartenden SC ausgeliefert (Abb. 4).

Die Korrelation des Requests mit dem zugehörigen Response geschieht dabei über die Request-ID des *eiMessageContext*. Die Konfigurationsdatei-

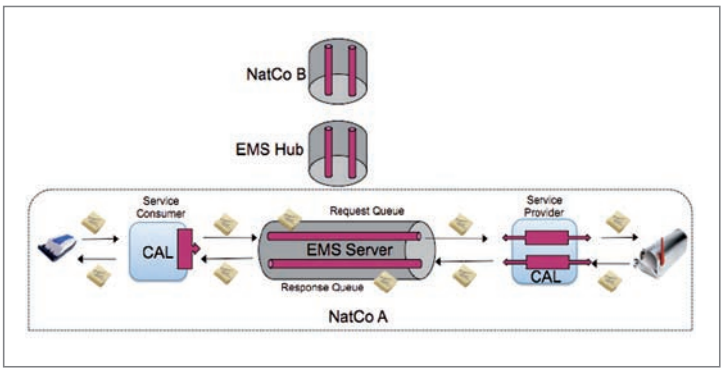
Listing 3: Beispiel „adapter.config.xml" eines Service Consumers

```
<?xml version="1.0" encoding="utf-8"?>
<clientAdapter name="de-tmobile-ei-itse-javamagazin-
        services-testping-testPing" xmlns:xsi="http://
            www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation= "clientAdapter.xsd">
 <serviceInfo messageExchangePattern="InOut" name=
                                        "testPing">
  <wsdlFileName>testPing.wsdl</wsdlFileName>
  <operationInfo name="ping"
operationType="requestReply">
   <serviceContract>
    <timeoutDefault>20000</timeoutDefault>
    <compress>true</compress>
    <priorityDefault>4</priorityDefault>
   </serviceContract>
  </operationInfo>
  <operationInfo name="testPing" operationType=
                                        "requestReply">
   <serviceContract>
    <timeoutDefault>20000</timeoutDefault>
    <compress>true</compress>
    <priorityDefault>4</priorityDefault>
   </serviceContract>
  </operationInfo>
 </serviceInfo>
 <sender name="tmd.tmobile.demo.
                        jm.JavaMagazin:Demo">
  <targetDestinationMapping target="testPing">
  <requestQueue>tmd.tmobile.demo.jm.JavaMagazin.Demo.
                    testPing.loc.req</requestQueue>
   <replyQueue>tmd.tmobile.demo.jm.JavaMagazin.Demo.
                    testPing.loc.resp</replyQueue>
  <timeout>
   <operation name="testPing" value="4000"/>
  </timeout>
    <messageSizeForCompression>5000</
                    messageSizeForCompression>
  </targetDestinationMapping>
 </sender>
</clientAdapter>
```

Listing 4: Beispiel „adapter.config.xml" eines Service Providers

```
<?xml version="1.0" encoding="utf-8"?>
<providerAdapter messageExchangePattern="InOut"
 name="tmd.tmobile.demo.jm.JavaMagazin.Demo.testPing.
                                        loc.req"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="providerAdapter.xsd">
 <requestQueue>
  tmd.tmobile.demo.jm.JavaMagazin.Demo.testPing.loc.req
 </requestQueue>
 <responseQueue>
  tmd.tmobile.demo.jm.JavaMagazin.Demo.testPing.loc.resp
 </responseQueue>
 <local-jndi-name>
  local/tmd.tmobile.demo.jm.JavaMagazin.Demo.testPing.
                                        loc.req
 </local-jndi-name>
 <provider name="tmd.tmobile.demo.jm.JavaMagazin:Demo">
  <endpoints>
   <address url="http://localhost:9999/ping" weight="0">
   </address>
  </endpoints>
  <xsltAdapter
   requestStylesheet="tmd.tmobile.demo.jm.JavaMagazin.
                    Demo.testPing-REQUEST.xsl"
   responseStylesheet="tmd.tmobile.demo.jm.JavaMagazin.
                    Demo.testPing-RESPONSE.xsl" />
 </provider>
</providerAdapter>
```

Abb. 4: Nachrichtenfluss bei nationalem Nachrichtenaustausch



en, die im CAL provisioniert werden, sind in XML formuliert und Listing 3 zeigt beispielhaft eine solche Konfigurationsdatei.

Neben den Informationen, um welchen Service es sich handelt (WSDL), sind auch operationsspezifische QoS-Eigenschaften konfiguriert, beispielsweise können unterschiedliche Operationen der Schnittstelle einen unterschiedlichen *timeout*-Default haben oder Nachrichten können komprimiert werden. Beim Mapping des SC – identifiziert durch die *sender/target*-Kombination – auf die zu benutzenden JMS-Queues ist es möglich, weitere QoS-Eigenschaften speziell für diese Kommunikationsstrecke zu konfigurieren. In Listing 3 werden Messages ab einer Größe von 5000 Bytes komprimiert und der Default-Timeout für die Operation *testPing* wird auf 4000ms festgelegt.

Bei den Konfigurationsinformationen für den SP (Listing 4) wird im Wesentlichen festgelegt, welches die Request und Response Queue ist, für die der CAL verantwortlich zeichnet und wie der technische Endpunkt des SP lautet (wohin der CAL die eingehende Nachricht schicken muss). In diesem Beispiel kann man zusätzlich erkennen, wie ein Plug-in [2] konfiguriert wird – hier ist es ein XSLT-Adapter. Das Request Stylesheet sowie das Response Stylesheet werden zur Laufzeit im CAL benutzt, um jeweils den Request und Response zu transformieren. Man benötigt diese Möglichkeit der Transformationen von Nachrichten, um z.B. einen SP anzubinden, der seine Services nicht SOA-Backplane-konform anbietet. Diese Möglichkeit unterstützt enorm die Migration bestehender Kommunikationspartner auf die SOA-Backplane.

Um die unterbrechungsfreie Rekonfiguration der Runtime zu gewährleisten, verlassen wir uns auf die im

Providing Port – Using Port: Knapp erläutert

Ports

Ports im CEISeR-Kontext sind logische Endpunkte zur Kommunikation zwischen Komponenten. Dabei geht jede Kommunikationsverbindung von einem Using Port zu einem Providing Port. Damit ein Routing für eine solche Verbindung provisioniert werden kann, müssen beide Ports vollständig definiert sein und in der Umgebung, in die provisioniert werden soll, gebunden (Binding) sein.

Providing Port

Der Providing Port kann mehreren Using Ports einen Service bereitstellen, dazu benötigt er beim Binding in der Regel einen technischen Endpunkt (z.B. eine http-URL im Falle eines Web Service).

Using Port

Ein Using Port hat eine Beziehung zu genau einem Providing Port. Auch er muss gebunden werden, benötigt aber nur einen technischen Endpunkt für das asynchrone MEP (dort wird dann die asynchrone Response empfangen).

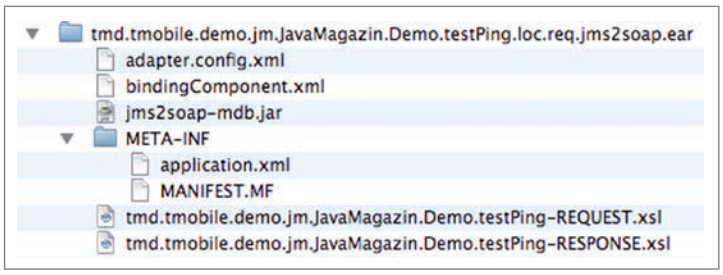


Abb. 5: Inhalte eines EAR-Files

JEE-Umfeld übliche Technik des Hot Deployments. Für jeden angebundenen Service werden aus dem zentralen Service Repository ein WAR-File und ein EAR-File erzeugt. Die WAR-Files enthalten Informationen über die Consumer und die EAR-Files über den entsprechenden Provider. Sie enthalten in unserem Fall keinen ausführbaren Code, sondern in der Regel nur die Konfigurationsdatei *adapter.config.xml*. Es können aber noch weitere Artefakte wie XSLT-Skripte oder bei Verwendung eines Java-Adapters eine *jar*-Datei in den EAR- oder WAR-Files enthalten sein.

JMS-Queues mit EMS – Begriffsdefinition

Bei der Implementierung von Queues in EMS gibt es Features, die über den JMS-Standard hinausgehen.

- *Remote Queue*: Ist ein Proxy für eine Home Queue, die auf einem anderen EMS-Server liegt, zu dem ein Routing definiert ist. Ist der Server, der die Home Queue bereitstellt, nicht verfügbar, ist es nicht möglich, über die Remote Queue Nachrichten zu empfangen oder zu versenden.
- *Home Queue*: Ist eine normale Queue – der Begriff wird häufig da verwendet, wo es Remote Queues gibt, die auf diese Queue verweisen.
- *Bridge*: Kopiert Nachrichten von einer JMS-Destination (Queue oder Topic) in eine andere JMS-Destination, z.B. von einem Topic in eine Queue.
- *selector*: Eine Bridge kann einen JMS-selector besitzen – ein Ausdruck (spezifiziert mit einem Subset von ANSI-SQL), der einschränkt, welche Nachrichten „gebridged" werden. Der selector bezieht sich nur auf JMS-Header-Properties.

Auswirkung der QoS-Profiles

Neben Informationen, die letztendlich das statische Routing über den ESB definieren, sind im Service Repository auch QoS-Informationen hinterlegt, die ebenfalls in die Konfigurationsartefakte einfließen. So lässt sich z.B. die Anzahl der Worker für einen Providing Port und damit technisch die Anzahl der MDB-Instanzen, die parallel Requests an einen SP ausliefern, festlegen.

Einige Konfigurationsinformationen werden zur Laufzeit vom CAL auf JMS-Header-Properties abgebildet, und damit wird z.B. das Verhalten des LMS gesteuert. So kann man für bestimmte Routings festlegen, dass die Payload einer Nachricht nicht im MessageStore gespeichert werden darf oder dass für einen Message Exchange keine LogPoints geschrieben werden sollen.

Die Konfiguration basiert auf den Informationen zu Services, Architektur, Binding, QoS und Infrastruktur, die in dem Service Repository *CEISeR* abgelegt sind. Zur Design-Time wird modelliert, welche Services generell vorhanden sind (WSDLs und XSDs). In der Architektur wird festgelegt, welche Applikationen mit welchen Komponenten an Ports Services nutzen oder anbieten und welche Using Ports mit welchen Providing Ports verbunden sind. In der Infrastruktur im Repository wird festgelegt, in welcher Landesgesellschaft welche CALs existieren, mit welchen JMS-Servern diese verbunden sind und auf welchen Maschinen in welchem Netzwerk diese laufen. Im Binding wird spezifiziert, welche Komponenten mit einem CAL in der Runtime verbunden sein sollen.

Ausblick

Nachdem wir nun im ersten Teil die Konzepte der SOA-Backplane dargestellt und in diesem Artikel die Umsetzung der Laufzeitumgebung detailliert erläutert haben, werden wir im dritten und letzten Teil dieser Artikelserie die automatische Serviceprovisionierung auf dem ESB und ihre Vorteile aufzeigen. ■

Dipl.-Ing. Carsten Sensler ist Angestellter der T-Mobile Deutschland GmbH und füllt dort die Rolle des Teamleiters eines internationalen Teams zur Serviceprovisionierung aus. Zudem arbeitet er an den Konzepten der SOA-Backplane mit und gibt interne, internationale Trainings bezüglich SOA-Backplane. Kontakt: http://www.sensler.de

Dipl.-Inform. Andre Karalus ist freiberuflicher Softwarearchitekt mit den Schwerpunkten Architektur, modellgetriebene Entwicklung und Integrations-Frameworks. Er arbeitet zurzeit für die T-Mobile Deutschland GmbH; dort gestaltet er am Design der SOA-Backplane mit und unterstützt in der Umsetzung.

Links & Literatur

[1] Dirk Mögenburg: Der alternative Web Service, in Java Magazin 12.2006, S. 44–48.

[2] Carsten Sensler, Andre Karalus: SOA@T-Mobile – Vollautomatische Serviceprovisionierung auf dem ESB, in Java Magazin 10.2008, S. 97–102.

[3] Thomas Grimm, Michael Kunz: OOP 2007, SOA-Backplane – MDSD-basierte SOA der T-Mobile.

[4] Carsten Sensler, Andre Karalus: Subconf 2007, Integration eines Service Repositories mit Subversion zur Anbindung an den ESB: http://2007.subconf.de/fileadmin/PDF_Dateien/SubConf_2007/Vortraege/Integration_eines_SOA_Repositories_Sensler_Karalus.pdf