trivadis
Part of Accenture

Atomikos TransactionsEssentials
mit Spring

) Schulung)

) Beratung)

) Entwicklung)

) Artikel)

AUTOR

Martin Bengl
Orientation in Objects GmbH

Veröffentlicht am: 17.7.2008

ABSTRACT

Wird für den Einsatz verteilter Transaktionen unbedingt ein Java EE Application Server benötigt? Nicht unbedingt - mit Atomikos Transactions [1] steht ein freier Transaktionsmanager zur Verfügung, dessen Verwendung es ermöglicht, verteilte Transaktionen auf einer leichtgewichtigen Laufzeitumgebung wie z.B. Apache Tomcat zu nutzen. In diesem Artikel / Tutorial wird der Einsatz von Atomikos Transactions in Verbindung mit einer Spring Anwendung demonstriert. Diese Anwendung wird um eine zusätzliche Datenquelle erweitert und hierbei das transaktionale Verhalten so ausgebaut, dass die Datenbankzugriffe über mehrere Datenquellen in einer verteilten Transaktion ablaufen.

Trivadis Germany GmbH

Weinheimer Str. 68
D-68309 Mannheim

Tel. +49 (0) 6 21 - 7 18 39 - 0
Fax +49 (0) 6 21 - 7 18 39 - 50

www.oio.dekontakt@trivadis.com

DIE THEORIE...

Die Java EE 5.0 Spezifikation [2] beschreibt, welche Aufgaben ein zertifizierter Application Server bewältigen muss. Dazu zählt neben der Bereitstellung von Web- und EJB-Container Implementierungen auch das Mitliefern eines Transaktionsmanagers. Dieser wird benötigt, um Zugriffe auf verschiedene Datenquellen innerhalb einer verteilten Transaktion zu kontrollieren. Die Implementierung des Transaktionsmanagers muss der Java Transaction API (JTA) Spezifikation [3] in der Version 1.1 genügen.

LOKALE ODER VERTEILTE TRANSAKTION

Durch den Einsatz eines JTA Transaktionsmanagers ist es möglich, verteilte Transaktionen auf verschiedenen Ressourcen auszuführen. Was aber unterscheidet lokale von verteilten Transaktionen genau?

Lokale Transaktionen spielen sich nur innerhalb eines Ressourcen-Managers ab. Dies kann beispielsweise ein relationales Datenbanksystem (z.B. DB2, Oracle) oder eine Message Oriented Middleware [4] (z.B. WebsphereMQ , ActiveMQ) sein. Die Anwendung steuert die Transaktion direkt über die API eines Ressource-Adapters, wie z.B. JDBC oder JMS (siehe Abbildung 1) über die Methoden "commit" und "rollback". Lokale Transaktionen stellen die gebräuchlichste und schnellste Variante für transaktionale Zugriffe dar, sind jedoch auf den Einsatz eines Ressourcen-Managers beschränkt.

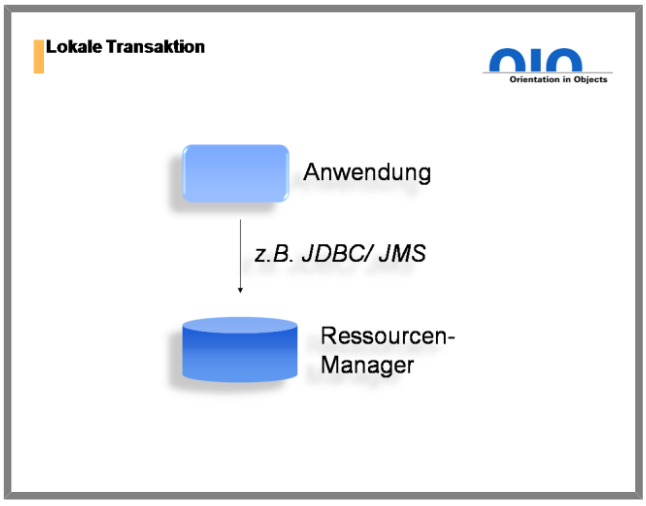

Abbildung 1: Lokale Transaktion

Besteht die Anforderung, Zugriffe auf mehrere Ressourcen-Manager in einer Transaktion durchzuführen, so kommen verteilte Transaktionen zum Einsatz. Voraussetzung hierfür sind mit dem XOpen/XA Protokoll [5] konforme Ressourcen-Manager und ein Transaktionsmanager, der die Koordination der Transaktion übernimmt (siehe Abbildung 2).

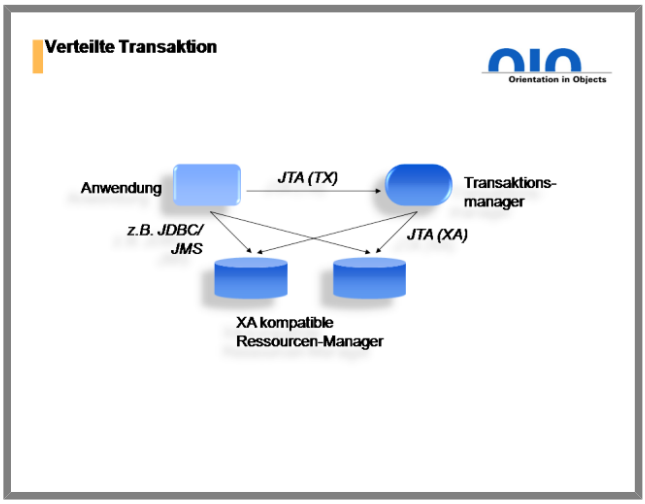

Abbildung 2: Verteilte Transaktion

Eine Spezialisierung von verteilten Transaktionen stellen Transaktionen dar, bei denen die Anforderung besteht, dass mehrere auf unterschiedlichen Servern bereitgestellte Komponenten an einer verteilten Transaktion teilnehmen müssen (siehe Abbildung 3).

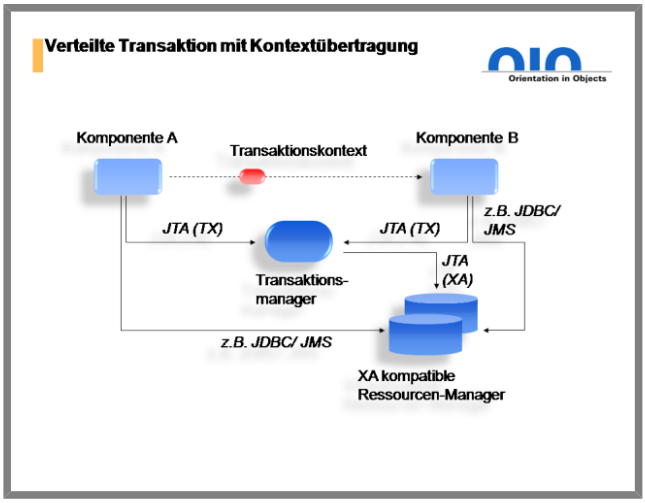

Abbildung 3: Verteilte Transaktion mit Kontextübertragung

So ist es zum Beispiel vorstellbar, dass Komponente A die Transaktion startet und Komponente B diese beendet. Die Koordination wird hierbei von einem Transaktionsmanager übernommen. Die Herausforderung bei dieser Art von verteilten Transaktionen besteht darin, den Transaktionskontext über VM Grenzen hinweg zu transportieren. Das zum Zuge kommende Verfahren nennt man "Piggy Backing" und könnte wie folgt unterhaltsam illustriert werden.

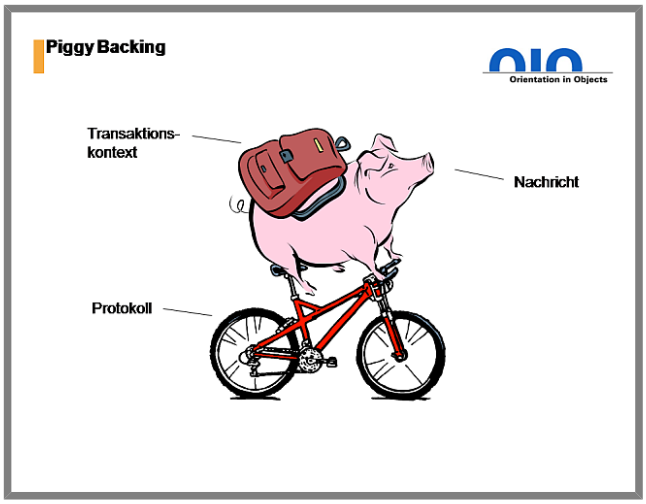

Abbildung 4: Piggy Backing

Besteht beispielsweise die Anforderung, die Transaktionssicherheit während eines Aufrufs einer EJB auf eine entfernte EJB zu gewährleisten, so würde der rufende Application Server den Transaktionskontext an die Nachricht hängen und der aufgerufene Application Server diesen herausnehmen und mit dem Thread der Ausführung assoziieren. Als Protokoll kommt in Java EE Umgebungen üblicherweise RMI over IIOP zum Einsatz.

Nach dieser kurzen Gegenüberstellung von lokalen und verteilten Transaktionen soll nun im folgenden Absatz das Zusammenspiel zwischen einer Anwendung, dem JTA Transaktionsmanager und den Ressourcen-Managern am Beispiel eines verteilten transaktionalen Zugriffs auf zwei JDBC Datenbanksysteme veranschaulicht werden.

VERTEILTE TRANSAKTIONEN MIT JDBC DATENQUELLEN

```
DataSource flightDS =
   (DataSource)jndiContext.lookup("java:env/jdbc/flightDataSource");
DataSource hotelDS =
   (DataSource)jndiContext.lookup("java:env/jdbc/hotelDataSource");
UserTransaction transaction =
   (UserTransaction)jndiContext.lookup("java:env/UserTransaction");
transaction.begin();
Connection flightConnection =
   flightBookingDS.getConnection();
doSomethingWithFlightConnection(flightConnection);
Connection hotelConnection = hotelBookingDS.getConnection();
doSomethingWithHotelConnection(hotelConnection);
transaction.commit();
```

Beispiel 1: Manuelle verteilte Transaktionsbehandlung

Zuerst benötigt die Anwendung die jeweiligen Datenquellen (Implementierungen von javax.sql.DataSource). In Java EE Umgebungen werden diese üblicherweise über JNDI (Java Naming and Directory Interface) vom Application Server ermittelt (siehe Listing 1). Die DataSource Implementierungen stellen hierbei Fassaden dar, welche beim Anfordern einer JDBC Verbindung einen Proxy zurückliefern. Dieser Proxy sorgt bei Ausführung von Datenbankoperationen (z.B. Absenden von SQL Operationen) dafür, dass die aktuelle Verbindung beim Transaktionsmanager auf der globalen Transaktion registriert wird (falls noch nicht geschehen). Dadurch weiß der Transaktionsmanager, welche Ressourcen an der globalen Transaktion teilnehmen. Dieser Vorgang wird als "Resource Enlistment" bezeichnet.

Neben den an der globalen Transaktion beteiligten Datenquellen, muss auch eine Implementierung der JTA UserTransaction Schnittstelle ermittelt werden. Auch diese werden in Java EE Umgebungen üblicherweise über JNDI verwaltet. Mit Hilfe dieser Schnittstelle kann die Anwendung Transaktionen starten, abschließen und zurückrollen. Gestartet wird eine globale Transaktion über die Methode "begin". Anschließend können über die zuvor ermittelten Datenquellen beliebige Datenbankoperationen ausgeführt werden. Diese werden nun transaktional ausgeführt und erst durch den Aufruf der UserTransaction Methode "commit" im Datenbanksystem abgeschlossen.

Der Einsatz eines JTA kompatiblen Transaktionsmanagers ist nicht auf die Nutzung innerhalb der Java EE Plattform beschränkt. Durch die klar spezifizierte Aufgabenteilung zwischen Anwendung, Transaktionsmanager, Application Server und Ressourcen-Manager ist es möglich, globale Transaktionen auch innerhalb von Java SE Umgebungen ausführen zu können. Die Aufgaben des Application Servers (z.B. Resource Enlistment) können z.B. direkt vom Transaktionsmanager übernommen werden.

Kommt in solchen Umgebungen das Spring Framework zum Einsatz, so kann Dank der Abstraktionsschicht transaktionales Verhalten konfiguriert werden, und zwar unabhängig davon, ob nun lokale Transaktionen auf einer Datenquelle oder globale Transaktionen auf mehreren Datenquellen zum Zuge kommen. Da lokale Transaktionen komplett innerhalb eines Ressourcen-Managers ablaufen, wird im ersten Fall auch keine zusätzliche Infrastruktur benötigt. Möchte man hingegen verteilte Transaktionen einsetzen, so ist ein wie oben erwähnter JTA kompatibler Transaktionsmanager notwendig. Dieser wird zusätzlich in die Spring Konfigurationsdatei aufgenommen. Atomikos Transactions bietet einen in Verbindung mit Spring einfach integrier- und konfigurierbaren JTA kompatiblen Transaktionmanager zur freien Verfügung an.

In den folgenden Abschnitten wird die Integration von Atomikos Transactions in eine bestehende Spring Anwendung gezeigt. Als demonstratives Beispiel soll ein fiktives Buchungssystem dienen.

TUTORIAL

Die Firma HebAb! GmbH hat sich auf Buchungen von besonders günstigen Flügen spezialisiert und unterhält mehrere Reisebüros in verschiedenen Städten. Die einzelnen Zweigstellen greifen zur Buchung auf einen Server in der Firmenzentrale in Frankfurt zu. Auf diesem Server ist das nun folgende vorgestellte Buchungssystem produktiv im Einsatz.

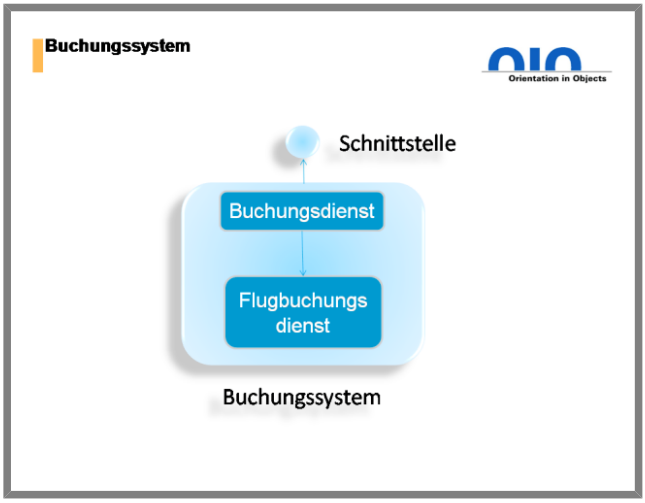

Abbildung 5: Das Buchungssystem

Mit Hilfe einer öffentlichen Schnittstelle können Flüge reserviert werden. Da der Artikel den Fokus auf die Verwendung von verteilten Transaktionen legt, ist das Buchungssystem bewußt einfach gehalten. Die Schnittstelle des Buchungssystems ermöglicht das Buchen von Flügen. Hierzu wird die Methode book() bereitgestellt (siehe Listing 2). Der Methode werden beim Aufruf die Informationen über den zu buchenden Flug übergeben. Als Rückgabewert wird eine Vorgangsnummer zurückgeliefert.

```
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;
public interface IBookingService {
   @TransactionAttribute(TransactionAttributeType.REQUIRED)
   public IssueId book(BookingData bookingData);
}
```

Beispiel 2: Die Schnittstelle des Buchungsdienstes

Durch die Angabe der Annotation @TransactionAttribute wird das transaktionale Verhalten der Methode beschrieben. In diesem Fall wird mittels TransactionAttributeType.Required festgelegt, dass der Methodenaufruf immer innerhalb einer Transaktion geschehen muss. Ist noch keine Transaktion vorhanden, so muss eine neue erstellt und nach Abschluss des Methodenaufrufs beendet werden. Ist eine Transaktion vorhanden, so wird der Methodenaufruf innerhalb dieser Transaktion durchgeführt. Diese Annotationen können z.B. von einem EJB 3.0 Container und vom Spring Framework interpretiert werden.

In Listing 3 ist eine mögliche Implementierung dieser Schnittstelle zu sehen. In der Methode book() werden die vom Aufrufer übergebenen Buchungsdaten mit Hilfe eines Data Transfer Object (DTO) Assemblers [6] in ein für den Flugbuchungsservice benötigtes Domainobjekt umgewandelt. Anschließend wird der Flugbuchungsservice aufgerufen und die erzeugte Vorgangsnummer nach Konvertierung durch den DTO-Assembler als Rückgabewert zurückgeliefert.

```
public IssueId book(BookingData bookingData) {
// convert DTO input into domain objects:
Flight flight = assembler.convertToFlight(bookingData);
// book the flight:
String issueId = flightBookingService.bookFlight(flight);
// convert domain output to DTO:
return assembler.createIssueId(issueId); }
```

Beispiel 3: Implementierung des Buchungsdienstes

Das in Listing 3 verwendete Flugbuchungssystem implementiert die Schnittstelle aus Listing 4. Mittels Angabe von TransactionATtributeType.MANDATORY als Wert für die @TransactionAttribute Annotation wird festgelegt, dass bei Aufruf der Methode "bookFlight" bereits eine Transaktion vorhanden sein muss.

```
public interface IFlightBookingService {
   @TransactionAttribute(TransactionAttributeType.MANDATORY)
   public String bookFlight(Flight flight);
}
```

Beispiel 4: Schnittstelle des Flugbuchungsdienstes

Die verwendete Implementierung des Flugbuchungssystems speichert die übergebenen Flugdaten in einer Datenbank, erstellt eine eindeutige Buchungsbezeichnung und stellt diese als Rückgabewert bereit.

Hier können Sie die Beispiele des Atomikos-Tutorials herunter laden.

DIE TECHNISCHE INFRASTRUKTUR

Die Laufzeitumgebung dieser Beispielanwendung stellt ein Apache Tomcat Server der Version 6.0 dar. Um die reservierten Flüge persistent zu halten, wird eine Apache Derby Datenbank in der Version 10.3.2.1 verwendet.

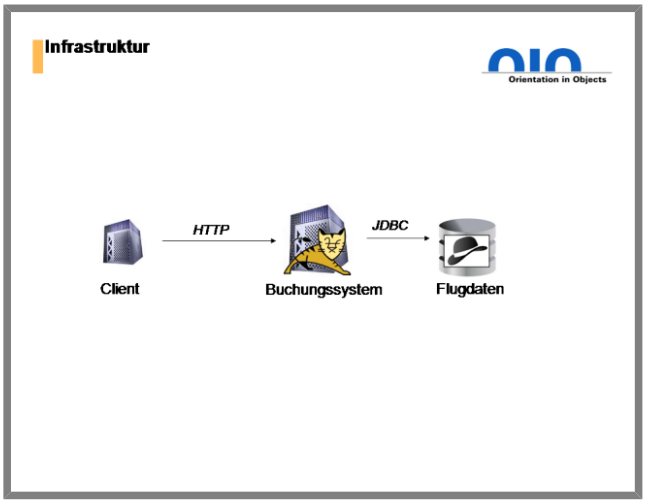

Abbildung 6: Die technische Infrastruktur

Listing 5 zeigt die zugehörige Spring Konfiguration. Der Buchungsdienst ("BookingService") besitzt eine Abhängigkeit auf eine JDBC Implementierung des Flugbuchungsdienstes ("FlightBookingService"). Diese benötigt eine JDBC DataSource, welche eine Verbindung zur lokal installierten Derby Datenbank konfiguriert. Da in diesem Szenario nur lokale Transaktionen benötigt werden, genügt hier die Konfiguration eines DataSource-TransactionManagers.

```
<!-- External Service -->
<bean id="bookingService"
    class="de.oio.bookingsystem.api.impl.BookingServiceImpl">
  <property name="flightBookingService"
         ref="flightBookingService" />
  <property name="assembler">
    <bean class="de.oio.bookingsystem.api.impl.
           BookingAssemblerImpl" />
  </property>
</bean>
<!-- Business Service -->
<bean id="flightBookingService"
    class="de.oio.bookingsystem.business.flight.impl.
    JdbcFlightBookingService">
  <property name="dataSource" ref="dataSource" />
</bean>
<!-- Resources -->
<bean id="dataSource"
    class="org.springframework.jdbc.datasource.
         DriverManagerDataSource">
  <property name="driverClassName"
         value="org.apache.derby.jdbc.ClientDataSource" />
  <property name="url"
         value="jdbc:derby://localhost/flightDB" />
  <property name="username" value="root" />
</bean>
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.
         DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource" />
</bean>
```

Beispiel 5: Konfiguration

NICHTS BLEIBT FÜR DIE EWIGKEIT...

Durch den Zusammenschluß der Firmen HebAb! und ZimmerFrei24 zur BudgetReisen GmbH besteht die Anforderung, das zum Einsatz kommende Buchungssystem zu erweitern. Hierbei soll neben der Buchung von Flügen auch das Reservieren von Unterkünften am Urlaubsort möglich sein. Daher wird in das Buchungssystem eine weitere Komponente aufgenommen. Das Hotelbuchungssystem übernimmt die Aufgabe der Reservierung von Übernachtungsmöglichkeiten.

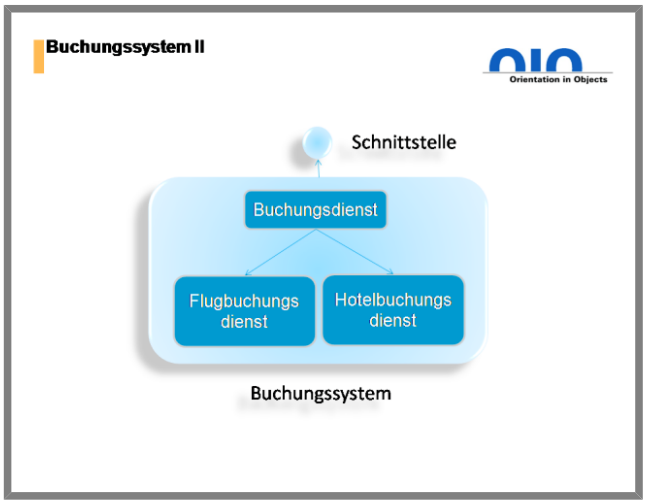

Abbildung 7: Das Buchungssystem 2.0

Die Schnittstelle des Buchungssystems muss so angepasst werden, dass die Verwender neben den Flugdaten nun auch Hotelreservierungsinformationen übergeben können. Hier wird die Datencontainerklasse BookingData um entsprechende Felder ergänzt. Anschließend kann die Implementierung des Buchungssystems um den Aufruf des Hotelbuchungsdienstes erweitert werden (siehe Listing 6).

```
public IssueId book(BookingData bookingData) {
   // convert DTO input into domain objects:
   Flight flight = assembler.convertToFlight(bookingData);
   HotelReservation hotel;
   hotel = assembler.convertToHotelReservation(bookingData);
   // book the flight:
   String flightId =
      flightBookingService.bookFlight(flight);
   // book the hotel:
   String hotelId =
      hotelBookingService.bookHotel(hotel);
   // convert domain output to DTO:
   return assembler.createIssueId(flightId, hotelId);
}
```

Beispiel 6: Erweiterte Implementierung des Buchungsdienstes

Die Buchung von Flug und Hotel soll innerhalb einer Transaktion stattfinden. Die Transaktionsbegrenzung für Flugbuchung- und Hotelreservierungsdienst wird daher auf "Mandatory" gestellt. Dies bedeutet, dass vor dem Aufruf beider Systeme eine Transaktion bereits vorhanden sein muss (siehe Listing 7 und 8).

```
public interface IFlightBookingService {
   @TransactionAttribute(TransactionAttributeType.MANDATORY)
   public String bookFlight(Flight flight);
}
```

Beispiel 7: Schnittstelle des Flugbuchungsdienstes

```
public interface IHotelBookingService {
   @TransactionAttribute(TransactionAttributeType.MANDATORY)
   public String bookHotel(HotelReservation hotel);
}
```

Beispiel 8: Schnittstelle des Hotelbuchungsdienstes

Analog zum Flugbuchungsdienst wird der Hotelreservierungsdienst die übergebenen Daten mit Hilfe einer Datenbank persistieren (siehe Abbildung 4).

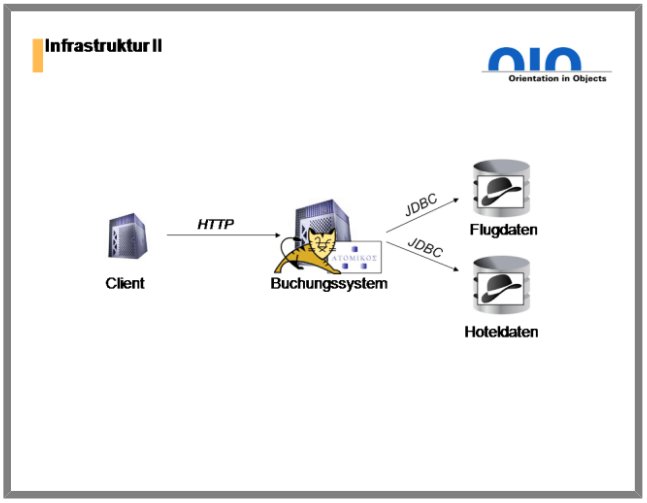

Abbildung 8: Die erweiterte technische Infrastruktur

Da der Buchungsdienst nun eine zusätzliche Abhängigkeit auf den Hotelbuchungsdienst besitzt, muss die Konfiguration entsprechend angepasst werden (siehe Listing 9).

```
<!-- External Service -->
<bean id="bookingService"
    class="de.oio.bookingsystem.api.impl.BookingServiceImpl">
  <property name="flightBookingService"
  ref="flightBookingService" />
  <property name="hotelBookingService"
  ref="hotelBookingService" />
  <property name="assembler">
    <bean
    class="de.oio.bookingsystem.api.impl.BookingAssemblerImpl"
    />
  </property>
</bean>
```

Beispiel 9: Angepasste Konfiguration des Buchungsdienstes

Der Hotelbuchungsdienst wird analog zum Flugbuchungsdienst konfiguriert und benötigt eine eigene JDBC DataSource. Da hier unterschiedliche Datenquellen (verschiedene Datenbanksysteme) zum Einsatz kommen, genügt die Verwendung einer lokalen Transaktion, wie sie im obigen Szenario gezeigt wurde, nicht mehr aus. Um das System auf verteilte Transaktionen umzustellen, muss lediglich die Konfiguration geändert werden (siehe Listing 10).

```
<bean id="flightDataSource" init-method="init"
    class="com.atomikos.jdbc.SimpleDataSourceBean">
  <property name="uniqueResourceName" value="flightXADBMS"
  />
  <property name="xaDataSourceClassName"
        value="org.apache.derby.jdbc.ClientXADataSource" />
  <property name="xaDataSourceProperties"
    value="databaseName=flightDB;serverName=localhost;portNumber=1527"
    />
  <property name="exclusiveConnectionMode" value="true" />
</bean>
```

Beispiel 10: Angepasste Konfiguration des Flugdatenquelle

Die JDBC Datenquellen werden mit Hilfe des von Atomikos bereitgestellten SimpleDataSourceBean beschrieben. Durch die Eigenschaft "uniqueResourceName" wird die Datenquelle beim Transaktionsmanager mit einer eindeutigen Identifikationsbezeichnung registriert. Die Eigenschaft "xaDataSourceClassName" gibt die verwendete XADataSource-Implementierung an und kann durch die Eigenschaft "xaDataSourceProperties" entsprechend konfiguriert werden (siehe Listing 11).

```
<!-- Configure the Spring framework to use JTA transactions
from Atomikos -->
<bean id="transactionManager"
    class="org.springframework.transaction.jta.JtaTransactionManager">
  <property name="transactionManager">
    <ref bean="atomikosTransactionManager" />
  </property>
  <property name="userTransaction">
    <ref bean="atomikosUserTransaction" />
  </property>
</bean>
<!-- Construct Atomikos UserTransactionManager, -->
<!-- needed to configure Spring -->
<bean id="atomikosTransactionManager"
    class="com.atomikos.icatch.jta.UserTransactionManager"
    init-method="init" destroy-method="close">
  <property name="forceShutdown" value="false" />
  <property name="transactionTimeout" value="10" />
</bean>
<bean id="atomikosUserTransaction"
    class="com.atomikos.icatch.jta.UserTransactionImp">
  <property name="transactionTimeout" value="10" />
</bean>
```

Beispiel 11: Konfiguration des JTA Transaktionsmanagers

Damit Springs Abstraktionsschicht funktioniert, wird zur Laufzeit eine Implementierung der PlatformTransactionManager Schnittstelle benötigt. Hier liefert Spring selbst Implementierungen für verschiedene Ressourcen-Adapter (z.B. Die DataSourceTransactionManager Implementierung für JDBC), verschiedene Java EE Application Server und einen generischen JTA Transaktionsmanager (JTATransactionManager). Bei Verwendung von Atomikos Transaction wird dieser wie in Listing 12 gezeigt konfiguriert. Der JTATransactionManager delegiert die Arbeit an die Implementierung der JTA Schnittstellen UserTransaction und TransactionManager. Diese werden von Atomikos Transactions implementiert und können über Spring wie gezeigt konfiguriert werden. Somit ist die Konfiguration der Anwendung abgeschlossen. In den folgenden Abschnitten wird der Download, die Installation und Konfiguration von Atomikos Transactions erläutert.

BESCHAFFUNG UND INSTALLATION

Nach Registrierung auf www.atomikos.com erhält man einen Link per Mail, der zum Download der aktuellsten Atomikos Transactions Version führt. Zum Zeitpunkt der Artikelerstellung war dies die Version 3.3.1. Die heruntergeladene ZIP-Datei enthält den Transaktionsmanager in Form mehrerer Java Archive im Verzeichnis "dist". Zur Laufzeit werden die Java Archive transactions-api.jar, transactions-jta.jar, transactions-jdbc.jar und transactions.jar benötigt. Je nachdem welche Features verwendet werden, kommen weitere JAR-Dateien zum Einsatz. In Verbindung mit dem Apache Tomcat Server lässt sich Atomikos Transactions grundsätzlich in zwei Varianten betreiben: Zum einen eingebettet in einer Java EE Webanwendung, zum anderen als gemeinsam nutzbare Bibliothek im "lib" Ordner des Tomcat-Installationsverzeichnisses.

Besteht die Anforderung Atomikos isoliert in einer Webanwendung laufen zu lassen, so gilt es die so eben erwähnten Dateien in das "WEB-INF/lib" Verzeichnis der entsprechenden Webanwendung zu kopieren. Unter "WEB-INF/lib" befinden sich üblicherweise auch das Spring-Framework (spring.jar) und die zum Einsatz kommenden Ressourcen-Adapter (im Falle der Beispielanwendung derbyclient.jar). Soll Atomikos als gemeinsam nutzbare Bibliothek in der Tomcat-Laufzeitumgebung installiert werden, so müssen neben den Atomikos Bibliotheken auch die eingesetzten Ressourcen-Adapter in den "lib" Ordner des Tomcat-Installationsverzeichnisses kopiert werden.

Falls die Java Transaction API 1.1 in der Laufzeitumgebung noch nicht vorhanden ist, gilt es in beiden Fällen die notwendigen Java Klassen in Form der Datei jta_1-1_classes.zip von der Seite http://java.sun.com/javaee/technologies/jta/ herunterzuladen und in die Laufzeitumgebung (oder in das WEB-INF/lib Verzeichnis der Webanwendung) zu kopieren.

Als letzter Schritt muß noch die Konfiguration von Atomikos Transactions durchgeführt werden.

KONFIGURATION

Atomikos Transactions wird mittels der Datei transactions.properties konfiguriert. Sie enthält alle notwendigen Werte für Initialisierung und Betrieb des Transaktionsmanagers. Wie in Listing 13 gezeigt, muss der Wert für die Eigenschaft "com.atomikos.icatch.service" explizit gesetzt werden. Die Konfigurationsdatei muss für den Transaktionsmanager im Klassenpfad erreichbar sein. Je nach Betriebsart variiert daher der Ablageort dieser Datei. Wird Atomikos innerhalb einer Webanwendung betrieben, so kann die Datei direkt im Source-Folder der Applikation abgelegt werden. Wird Atomikos als "Shared Library" innerhalb von Apache Tomcat betrieben, so muss die Datei transactions.properties in den Ordner "conf" unterhalb des Tomcat-Installationsverzeichnisses abgelegt werden.

```
#Required: factory implementation class of the transaction
core.
#NOTE: there is no default for this, so it MUST be specified!
#
com.atomikos.icatch.service=
   com.atomikos.icatch.standalone.UserTransactionServiceFactory
#Set base name of file where messages are output
#(also known as the 'console file').
#
com.atomikos.icatch.console_file_name =
   atomikos.out
#Size limit (in bytes) for the console file;
#negative means unlimited.
com.atomikos.icatch.console_file_limit=1024000
#For size-limited console files, this option
#specifies a number of rotating files to
#maintain.
# com.atomikos.icatch.console_file_count=5
#Set output directory where console file and other files
#are to be put
#make sure this directory exists!
#
com.atomikos.icatch.output_dir = ./atomikos
#Set directory of log files; make sure this directory exists!
#
com.atomikos.icatch.log_base_dir = ./atomikos
#Set the max timeout (in milliseconds) for transactions
#
com.atomikos.icatch.max_timeout = 300000
#The globally unique name of this transaction manager process
#override this value with a globally unique name
#
com.atomikos.icatch.tm_unique_name =
   de.oio.bookingsystem.txmanager
```

Beispiel 12: Konfiguration von Atomikos Transactions

Die Entwickler von Atomikos Transactions empfehlen die Standardwerte für folgende Eigenschaften mit für die Umgebung sinnvollen Werten zu überschreiben.

com.atomikos.icatch.tm_unique_name	Eindeutiger Name des Transaktionsmanagers
com.atomikos.icatch.max_timeout	Das maximal setzbare Timeout für Transaktionen
com.atomikos.icatch.console_file_limit	Maximale Größe der Logdateien in Bytes
com.atomikos.icatch.console_file_count	Maximale Anzahl der Logdateien

Damit ist die Konfiguration von Atomikos Transactions abgeschlossen und die Anwendung kann in Betrieb genommen werden.

FAZIT

Die Konfiguration von Atomikos Transactions ist in wenigen Minuten erledigt - kommt in der Anwendung auch das Spring Framework zum Einsatz, so ist die Konfiguration der Datenquellen ebenso einfach wie der gezeigte Wechsel von lokalen auf globale Transaktionen. Wer aus administrativen Gründen JNDI zur zentralen Konfiguration der Datenquellen und der UserTransaction-Implementierung verwenden möchte, wird davon auch nicht abgehalten. Beim Betrieb im Apache Tomcat gilt es hierbei zu beachten, dass alle im JNDI registrierten Klassen für die Tomcat Laufzeitumgebung erreichbar sind. Verteilte Transaktionen auf einem Tomcat laufen zu lassen ist wohl eines der wenigen Dinge, die in der Theorie komplizierter erscheinen als sie in der Praxis tatsächlich sind.

REFERENZEN

- [1] Atomikos Transactions
  (http://www.atomikos.com)
- [2] Java Enterprise Edition Specification 5.0
  (http://jcp.org/en/jsr/detail?id=244)
- [3] Java Transaction API (JTA) Spezifikation
  (http://java.sun.com/javaee/technologies/jta/index.jsp)
- [4] Wikipedia: Message Oriented Middleware
  (http://de.wikipedia.org/wiki/Message_Oriented_Middleware)
- [5] XA Specifikation
  (http://www.opengroup.org/publications/catalog/c193.htm)
- [6] Transfer Object Assembler
  (http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObjectAssembler.html)
- Download: Beispiele des Tutorials
  (/public/opensource/atomikos/atomikos-tutorial.zip)