Wasser in der Wüste

Vom custom-made SQL-DB-Messaging Antipattern zum JMS-Standard mit Apache Camel

Niko Köbler
Freiberuflicher Berater und Software-Architekt

Schlüsselworte
Oracle Advanced Queuing, AQ, Java, Apache Camel, JMS, Messaging, Enterprise Integration Patterns, EAI, SQL, DB

Einleitung
Viele Unternehmen haben mit ihrer Oracle Datenbank und dem darin enthaltenen Oracle Advanced Queuing (AQ) eine Standard-Messaging-Lösung im Haus, wissen es aber meist nicht. Somit werden zur Anwendungsintegration (EAI) teils umständliche und meist proprietäre Eigenentwicklungen und Anti-Pattern über DB-Tabellen und/oder Text-Dateien realisiert, oftmals unter dem Gesichtspunkt "das-wird-schon-gutgehen". Wenn dazu noch die unterschiedlichen Entwicklertypen wie der DB- oder PL/SQL-Freak auf der einen und zwei verschiedene Software-Entwickler (in unserem Beispiel der Java-Geek und der Host-Fuchs) auf der anderen Seite aufeinandertreffen, fragt man sich als Architekt beim Review oft, wie das so lange so gut gehen konnte.

Dieser Vortrag erzählt aus einem Projekt, in dem aus der beschriebenen Ausgangssituation ohne die Beschaffung von kostenintensiven Lizenzen und Hardware, nur mit vorhandenen Bordmitteln (Oracle Advanced Queuing), Open Source Software (Apache Camel und ein paar weitere Java-Frameworks), ein wenig Programmierung (Custom-Java-Code) und überwiegend Konfiguration (XML, DSL), eine neue, leichtgewichtige, flexible und skalierbare JMS-Standard-Lösung ähnlich eines ESBs geschaffen werden konnte, mit der alle Beteiligten nach wie vor in ihrem eigenen Scope arbeiten können, ohne die anderen Entwicklungslager im Detail verstehen zu müssen.

Die Herausforderung
Zu Beginn des Projektes sah die Ausgangssituation und Aufgabenstellung beim Kunden wie folgt aus: Es existierten ein Host-System in Form einer IBM AS/400 (heute iSeries/System i/i5/…) mit integriertem Derivat einer DB2-Datenbank, historisch gesehen eine ISAM-DB; eine Oracle-DB mit mehreren mittels PL/SQL erstellten Anwendungen und ein JEE-Container, der ebenfalls die Oracle-DB als Persistenz-Schicht nutzt. In der angegebenen Reihenfolge sind die Systeme auch beim Kunden eingeführt/angeschafft worden

Alle Systeme haben eine eigene Datenhaltung, benötigen aber auch Daten aus den jeweils anderen Systemen.

Zu Beginn löste man das Schnittstellen- bzw. Datenaustausch-Thema relativ einfach über den Austausch von Textdateien (heute würde man von „FTP" reden. Auf der AS/400 wurden die

benötigten Daten in eine „logische Datei“ (Tabelle) geschrieben und von dort in einer Datei mit fester Satzlänge geschrieben. Die PL/SQL-Anwendung las die Daten ein, verarbeitete sie entsprechend und schrieb ihrerseits auch wieder Daten in eine Textdatei, welche die AS/400 einlas und verarbeitete. Keine Echtzeit-Kommunikation, aber die bestehende Latenz war akzeptabel.

Einziger Knackpunkt dabei war, dass die AS/400, bzw. die Entwickler beim Kunden, relativ unflexibel waren, was das Datenformat dieser Textdatei anbelangte, so mussten es immer Datensätze mit fester Spaltenlänge sein und sowieso war die AS/400 das führende System und bestimmte den gesamten Umgang mit den Daten. Was das System (oder die Entwickler) nicht konnten bzw. kannten, war keine Option.

Als dann das Java-System hinzukam, benötigte dies plötzlich auch Daten von der AS/400, aber ggf. andere als das PL/SQL-System. Aber auch Daten aus der PL/SQL-Anwendung waren interessant und die Daten die das Java-System beheimatete wurden auch von den anderen Systemen benötigt (bzw. sollten auch dort zur Verfügung stehen).

Nun fing ein findiger Entwickler an, da einen JDBC-Treiber für die AS/400 gefunden hatte, den Datenaustausch über eine wilde Kombination aus verschiedenen Datenbanken und Tabellen zu realisieren:
Aus der AS/400 zu exportierende Daten wurden dort in eine Export-Tabelle geschrieben, nun aber keine Textdatei mehr erzeugt, sondern von dem neuen, kleinen Java-Programm ausgelesen, in eine Import-Tabelle der Oracle-DB geschrieben und im Quellsystem gelöscht (der Datensatz war ja nicht verloren, sondern lediglich in ein anderes System gewandert).
Die PL/SQL- und die Java-Anwendung konnten nun, da sie ja sowieso Zugriff auf die Oracle-DB hatten, diese Daten auslesen und ihrerseits verarbeiten. Nach der Verarbeitung wurden die Daten aus der Import-Tabelle gelöscht. Daten die sie untereinander austauschten, wurden ebenfalls über diese Tabelle geleitet.
Der Weg zurück, funktionierte nach dem gleichen Muster: Export-Tabelle in der Oracle-DB, Java-Programm holt die Datensätze ab, schreibt sie in eine Import-Tabelle auf der AS/400 und löscht sie aus der Oracle-DB, die AS/400 pollt auf die Import-Tabelle, verarbeitet die Daten und löscht sie aus der Import-Tabelle.

Damit dieses System des Datenaustauschs flexibel und „skalierbar“ war, existierten in den jeweiligen Tabellen im Grunde je nur zwei Spalten:

1. Ein Identifier, der die Art des Datensatzes angab und
2. Ein Varchar/LOB, der die Daten selbst beinhaltete

Die Daten selbst waren immer noch im alten Format der festen Spalten-/Satzlänge angegeben, somit bestimmte der Identifier in der ersten Spalte quasi das Format der eigentlichen Daten. Einen gemeinsamen (zentralen) Vertrag gab es nicht. Änderungen am Satzformat bedurften der Absprache.

So weit, so akzeptabel. Nur eine Sache funkte oft in den „normalen“ Ablauf ein:
Jede Partei hatte außer den notwendigen Lese- natürlich auch Schreib- und Löschrechte auf den Tabellen. Im Falle des Weges Oracle → AS/400 kein Problem, aber umgekehrt war es doch oftmals

schwieriger, denn je nachdem welches System zuerst die Daten las und verarbeitete, löschte es auch den Datensatz und das jeweils andere System bekam von den Daten nichts mehr mit.
Aus diesem Grunde hat man dann zusätzlich noch einen nächtlichen Job geschrieben, der nochmals alle Daten zwischen der AS/400 und den Oracle-Anwendungen (PL/SQL und Java) abglich – über den guten alten und bewährten Austausch von Textdateien, versteht sich.

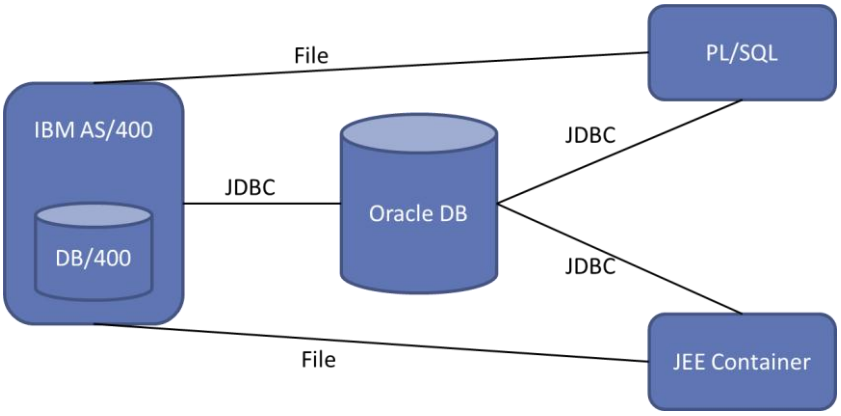

Abbildung 1:Ausgangssituation und Herausforderung

Also ein Durcheinander auf breiter Ebene, keine zentrale Steuerung, kein Durchblick und das große Hoffen auf die Richtigkeit und Konsistenz der Daten und das Ausbleiben einer großen Katastrophe.

Die Aufgabenstellung war eindeutig:
Lieber Berater, mach den Datenaustausch vernünftig, so dass wir

a) Keine Daten mehr verlieren,
b) Keine Daten redundant durch die Gegend schieben,
c) Schneller und flexibler bei Änderungen werden,
d) Wissen, was überhaupt passiert
e) Und natürlich so wenig wie möglich Kosten und Aufwand damit haben werden

Der schwere Weg
Die Anforderungen a) bis d) stellen erst mal keine große Herausforderung dar, gibt es doch genügend Werkzeuge und Plattformen, die hier helfen können. In der IBM-Familie fällt einem gleich Websphere MQ ein, welches sogar auf einer AS/400 läuft. Und im Oracle-Umfeld, da haben wir ja die Fusion Middleware mit der SOA-Suite und dem Oracle Service Bus mit vielfältigen Möglichkeiten, andere Systeme anzubinden. Die Verbindung zu einem Websphere MQ wäre da ja ein Kinderspiel.

Diese, evtl. mit-Kanonen-auf-Spatzen-, Lösung ist aber weder kostengünstig, noch hat der Kunde wenig Aufwand damit. Er hätte vielmehr plötzlich zwei, drei Systeme mehr zu administrieren, Lizenzen zu bezahlen, Know-how aufzubauen, Mitarbeiter zu schulen, evtl. sogar neue Mitarbeiter einzustellen, usw. Also eher eine suboptimale Lösung.

Also schauten wir uns die Dinge an, die uns beim Kunden zur Verfügung standen, sowie ein paar Frameworks, die wir aus anderen Projekten kannten und die uns „irgendwie“ unterstützen könnten.

Oracle Advanced Queuing[1]

Grundsätzlich fanden wir die Idee, die Kommunikation mittels JMS[2] zu realisieren, eine gute Sache, bis auf die Tatsache, dass die AS/400 mit JMS erst mal nicht so viel anfangen konnte. Mit der vorhandenen Oracle-DB hatten wir aber bereits eine Standard JMS-Lösung vor unserer Nase, die nur nicht genutzt wurde, da der Kunde nicht wusste, was er außer der Datenbank noch so mit dem Produkt anfangen konnte (bei der Gelegenheit erklärten wir ihm auch gleich die Volltext-Indizierung von Inhalten mit Oracle-Text, ohne ein zusätzliches Indizierungs-Tool zu verwenden! Klappt super!). Oracle hat seit der Version 9i (Release 2) das „Advanced Queuing" eingebaut und liefert es mit jeder (auch mit XE!) Datenbank aus, es steht also jedem Datenbank-Kunden out-of-the-box zur Verfügung. AQ arbeitet intern mit den etablierten und robusten Technologien wie Datenbanktabellen und Datenbankusern. State-of-the-art! Zudem ist das System über die Standard-DB Ports verfügbar und bedarf so noch nicht mal eine weitere Freischaltung in internen Proxies und/oder Firewalls. Die Admins mussten so gut wie nichts dazulernen, die AQ-Funktionalitäten mussten nur aktiviert werden.

Außer per JMS-Standard kann auf AQ auch noch mittels PL/SQL-Packages zugegriffen werden, somit hätten wir die Java- und die PL/SQL-Fraktion schon mal bedient. Bleibt immer noch die AS/400 – und mit neuen Technologien tun sich die Entwickler dort immer noch schwer (obwohl es mittlerweile auch hier Möglichkeiten, Bibliotheken und Tools gibt).

Apache Camel[3]

Wir schauten uns weiter bei den uns bekannten Frameworks um. Spring Integration[4], Mule ESB[5] und Apache Camel standen da zur Auswahl. Spring Integration, altbekannt und wohl-erprobt, aber irgendwie umständlich in der Konfiguration und nicht sofort lesbar. Wäre aber eine Alternative. Was Mule ESB nicht war, da wir hier zu wenig Know-how hatten und den Kunden das „ESB" gestört hat, obwohl man nicht den ganzen ESB einsetzen muss, sondern auch rein mit der Bibliothek die Integrationsmechanismen in sein Projekt einbinden kann. Blieb Apache Camel.

Tolle Dokumentation, viele Komponenten, viele unterstützte Connectoren bzw. Protokolle, einfach eigene Komponenten zu konfigurieren, baut auf den bekannten Enterprise Integration Patterns[6] (nach Gregor Hohpe & Bobby Woolf) auf, wird als Basis-Framework in Produkten wie Apache ServiceMix und Apache ActiveMQ genutzt.
Und: sexy DSL! Nicht eine, nicht zwei, sondern gleich drei Möglichkeiten bringt Apache Camel mit, die Integrationsaufgaben bzw. Nachrichtenflüsse und -modifizierung („Routes") zu beschreiben:

- altbekanntes XML (auch in Verbindung mit Spring Konfiguration)
- eine Java-DSL und
- eine Scala-DSL (noch nicht so leistungsfähig wie XML und Java, aber groß im Kommen)

---

[1] http://en.wikipedia.org/wiki/Oracle_Advanced_Queuing
[2] http://de.wikipedia.org/wiki/Java_Message_Service
[3] http://camel.apache.org
[4] http://www.springsource.org/spring-integration
[5] http://www.mulesoft.org
[6] http://www.eaipatterns.com

Für alle Technikfreaks also etwas dabei. Sicherlich also ein Tool, welches uns bei unserem Integrationsvorhaben sehr gut unterstützen kann.

Doch was tun mit der AS/400?
Na gut, in Apache Camel können eigene Komponenten eingehängt werden, also schreiben wir uns halt eine Komponente, die auf der IBM Java Toolbox JT400[7] (Open Source!) aufbaut und sich mit den AS/400 Kanälen verbindet… *Doch was ist das??* Bei genauerem Studium der sehr langen Komponenten-Liste stellt sich heraus, dass Apache Camel eine Komponente namens „JT400" mitbringt. Kann das sein? Ja, es kann! Das gute, schlanke Camel-Framework kann sich out-of-the-box mit einem Legacy-Host-System verbinden. Klasse!

Welche Möglichkeiten bietet die JT400 Komponente?

- Zugriff direkt auf Tabellen
- Lesen/Schreiben von DTAQs
  (das sind auch Message-Queues, wie man sie aus „modernen" Message Oriented Middlewares kennt, in die jeglicher (Text-)Inhalt gesteckt werden kann, jedoch sind sie nicht persistent, d.h. wenn das System mal runtergefahren wird, geht der Inhalt verloren – aber wann wird eine AS/400 mal (ungeplant) runtergefahren? Und das gab's schon Mitte der 80er Jahre!)
- Ansprechen von Programmen (ähnlich RPCs)
- Usw. (theoretisch alles, was mit der JT400 Library möglich ist)

Die Lösung
Wir hatten die Bausteine gefunden, die zum Einsatz kommen sollten, jetzt galt es, diese „nur noch" zusammenzufügen und in die Produktion zu bekommen. Dies war letztendlich wirklich „nur noch" Fleißarbeit, da die Systeme und Frameworks sehr robust und zuverlässig ihre Arbeit verrichten.

Und so sieht der Datenfluss jetzt aus:

- Die AS/400 legt die Daten, die anderen Systemen zur Verfügung gestellt werden sollen, in eine DTAQ, ggf. auch in unterschiedliche DTAQs, wenn dies fachlich und/oder technisch notwendig sein sollte.
- Camel greift die Daten aus den DTAQs ab, und transformiert sie aus dem AS/400-Quellformat (feste Satzlänge!) in ein XML-Format. Diese Transformationsregeln sind zentral in der Camel-Anwendung abgelegt.
- Je nach Nachrichtentyp wird die Nachricht selbst per JMS an die AQ-Queue für die PL/SQL-Anwendung und/oder direkt per JPA in die Zieltabelle des Java-Systems geleitet. Auch für das Java-System wurden Queues in AQ angelegt, um spezielle Events damit abzufangen/abzubilden.
- Im Falle von JPA ist klar, dass die entsprechende Ziel-Entität erzeugt werden muss. Im Falle von JMS werden die Nachrichten in das jeweilige vom Ziel-System geforderte XML-Format transformiert, was wiederum auch nur zentral in Apache Camel geschieht – *Single Point of Truth / Single Point of Failure*!

[7] http://jt400.sourceforge.net

- Der gesamte Datenfluss ist transaktional, so dass Daten bei einem evtl. Fehler nicht verloren gehen, zusätzlich wurde ein zentrales „Error-Hospital“, ähnlich wie bei einem ESB, eingerichtet, welches auch wiederum per Listener überwacht wird und ein Admin ggf. benachrichtigt wird.
- Der Rückweg ist analog:
  Die PL/SQL-Anwendung legt ihre Daten im eigenen Format in ihre Queue, die Java-Anwendung ebenso (hier wurde auf die JPA-Möglichkeit verzichtet), Apache Camel holt die Nachrichten ab, transformiert diese zunächst in ein gemeinsames XML-Datenformat, persistiert diese in AQ zwischen, bevor die konsolidierten Nachrichten dann über den JT400-Konnektor wieder in die DTAQ der AS/400 geschrieben werden.

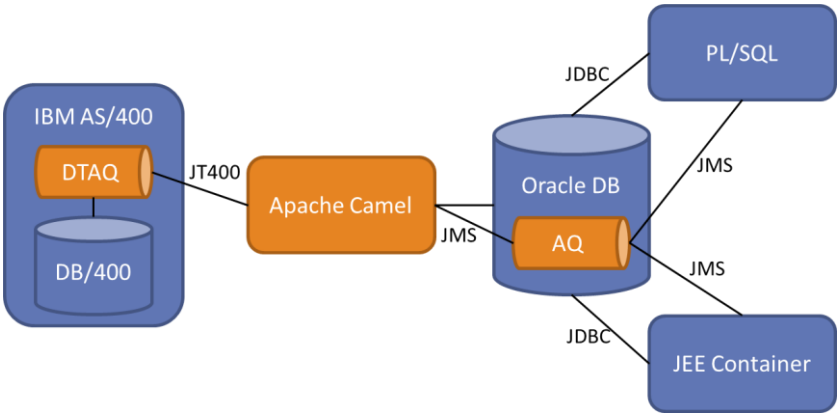

Abbildung 2: Lösungsarchitektur

Alle Anforderungen des Kunden konnten wir mit dieser Lösung erfüllen, alle Beteiligten sind rundherum glücklich:

- Das Management, da Geld gespart wurde und kein neues, teures und übergroßes System angeschafft werden musste.
- Die Admins, weil sie kein neues System zu betreuen haben.
- Die Entwickler, weil sie sich auf ihre eigene Umgebung konzentrieren können und sich nicht mit den Umständen eines anderen Systems auseinandersetzen müssen.
- Wir – weil der Kunde glücklich ist ☺

Der initiale Aufwand, die Camel-Routes zu erstellen, wurde im Rahmen des Projektes geleistet. Der Wartungsaufwand beläuft sich auf minimale Anforderungen und Tätigkeiten, die von der Java-Crew mit übernommen werden können. Da die Java-Abteilung sowieso expandiert, konnten hier problemlos die Ressourcen für die Wartung eingeplant und beschafft werden.

Kontaktadresse:
Niko Köbler IT-Beratung
Weingartenstr. 48a
D-65795 Hattersheim am Main
Telefon: +49 (0) 172-6714839
E-Mail info@n-k.de
Internet: http://www.n-k.de