RESTful PL/SQL

Heiko Blau
Freiberuflicher Software-Entwickler
Alexander Möckel
IBYKUS AG
Erfurt

Schlüsselworte:

PL/SQL, SQL, Java, Stored Procedures, Business Logic, Datenbank, Querschnittsfunktionalität, Webservices, SOA, Unit Test

Einleitung

Anwendungslogik in PL/SQL – oder allgemein in Stored Procedures – gilt vielen als hoffnungslos veraltet und wird gerade in der Welt objektorientierter Enterprise-Plattformen häufig als grobe Verletzung des jeweiligen Designs angesehen. Wenn jedoch Oracle als Datenbank gesetzt ist, verschiedene Anwendungen auf die gleichen Daten zugreifen sollen, Massendaten verarbeitet oder komplexe Reports erstellt werden müssen bzw. ein Entwicklungsteam mehr PL/SQL- als Java/DotNet/Ruby...-Erfahrung hat, ist die Entscheidung für PL/SQL zumindest gut begründet.

Gerade daten-intensive Anwendungen profitieren von der einfachen PL/SQL-Syntax mit nahtlos integriertem SQL, der hohen Performance ohne Netzwerk-Overhead und der leichten Wartbarkeit von PL/SQL-Artefakten. Java Stored Procedures helfen, wo PL/SQL nicht weiterkommt.

Darüber hinaus bietet PL/SQL genügend Mittel, um Schnittstellen zu realisieren, die von unterschiedlichen Sprachen, Frameworks und Plattformen genauso einfach zu bedienen sind wie z. B. ein HTTP-Request:

```
// fiktive DB-API
// request/response: XML, YAML, CSV, …
DB.getConnection(url)
  .prepareCall('begin PutDoc(:req, :resp); end;')
  .bindIn('req', request)
  .bindOut('resp', response)
  .executeAndClose();
```

Unser Vortrag stellt verschiedene Ansätze für stabile, aufrufer-freundliche PL/SQL-Schnittstellen vor und beleuchtet ihre Effekte auf die mit der Datenbank kommunizierenden Programme, seien es einfache Kommandozeilentools, Rich Clients oder skalierbare Server. Wir zeigen, wie für lose Kopplungen geeignete Interfaces definiert und unterschiedliche Daten gemeinsam behandelt werden können.

Und der Nutzen? Der Application Server wandert in die Datenbank. PL/SQL-Code bleibt „lebendig“, d. h. änder- und erweiterbar, ohne die Lauffähigkeit darauf zugreifender Artefakte zu brechen. Diese Artefakte können kompakte Datenbank-Aufrufe nutzen und gegebenenfalls sogar ohne größeren Aufwand auf eine andere Persistenzschicht oder andere Kommunikationskanäle wechseln.

Architektur und Querschnittsfunktionalität

Der augenfälligste Nutzen von PL/SQL für den Software-Entwickler ist die Kombination von SQL mit den Konstrukten einer prozeduralen 4GL-Sprache. Salopp ausgedrückt tut eine kompilierbare PL/SQL-Routine schnell und sicher das, was ihr Programmierer von ihr wollte – ohne Memory-Leaks, Threading-Probleme und hohen Einarbeitungsaufwand in Klassen-Hierarchien. Die Alternativen Prozedur, Funktion oder Package sind sehr überschaubar, weiterreichende Konstrukte wie Java SP's und Trigger stehen unaufdringlich zur Verfügung.

Allerdings stellt sich bei der Arbeit mit PL/SQL schon bald heraus, dass man zum einen auf keine Architekturen zurückgreifen kann (die z. B. in der Java-Welt in Form der JEE-Spezifikationen und diverser hochentwickelter Frameworks zur Verfügung stehen). Zum anderen gestaltet sich die Bereitstellung von Querschnittsfunktionalität schwierig – wie integriere ich z. B. eine fachliche Berechtigungsprüfung jenseits der vorhandenen SQL-grant's in einen bestehenden Satz PL/SQL-Routinen? Programmierung gegen Interfaces, Filter- und Plugin-Konzepte, Aspekte sind nicht von Haus aus vorhanden. Nachbildungen solcher Mechanismen sind nur bedingt möglich und nehmen bestimmte Nachteile in Kauf, z. B. durch Overhead und Datentyp-Einschränkungen beim Einsatz dynamischer Aufrufe (execute immediate, dbms_sql).

Wenn darüber hinaus eine PL/SQL-Lösung über viele Jahre entwickelt wird – die laufenden Agrarförderungen der IBYKUS AG werden seit mittlerweile ca.10 Jahre beständig weiterentwickelt –, helfen neue Features, Packages und APIs der Oracle-Datenbank nur bedingt und sind nicht eben leicht zu integrieren.

Also ist es für PL/SQL-basierte Anwendungen wichtig, neben den üblichen Gesichtspunkten der Software-Entwicklung wie Versionsverwaltung, Test-Strategie, Dokumentation etc. auch eine Architektur und die damit verbundene Querschnittsfunktionalität zu entwerfen und zu pflegen. Einen relativ kleinen Ausschnitt der bei der IBYKUS AG eingesetzten Architektur zeigt Abb. 1.

Komponenten einer gemeinsamen Code-Basis

Oracle liefert mit diversen Packages und Typen wie den dbms- und utl-Packages, XMLTYPE usw. grundlegende Hilfsmittel für ein breites Funktionalitätsspektrum an, seien es XML- oder LOB-Verarbeitung, externe Kommunikation über TCP, HTTP, Mail und Messaging, „Kleinkram“ wie Base64-Kodierung, adminstrative Aufgaben u.a.

Leider ist die Arbeit mit einigen dieser Packages „unbequem“; z. B. liefert DBMS_LOB eine absolut unzureichende Performance, wenn man Daten nicht in größeren Blöcken (Chunks) schreibt und liest. Nutzer von DBMS_XMLDOM leiden unter Unmengen von MakeNode-, MakeElement-, MakeAttr-Aufrufen, dazu kommt der ohnehin sperrige Umgang mit der XML-Knoten-orientierten DOM-API, wenn nur Element-Knoten und die Text-Knoten von Leaf-Elementen benötigt werden.

Bei IBYKUS haben wir in den letzten Jahren eine Reihe von Packages entwickelt, die zum einen allgemein nutzbar sind, zum anderen stabile, homogene, dynamisch aufrufbare Schnittstellen ermöglichen.

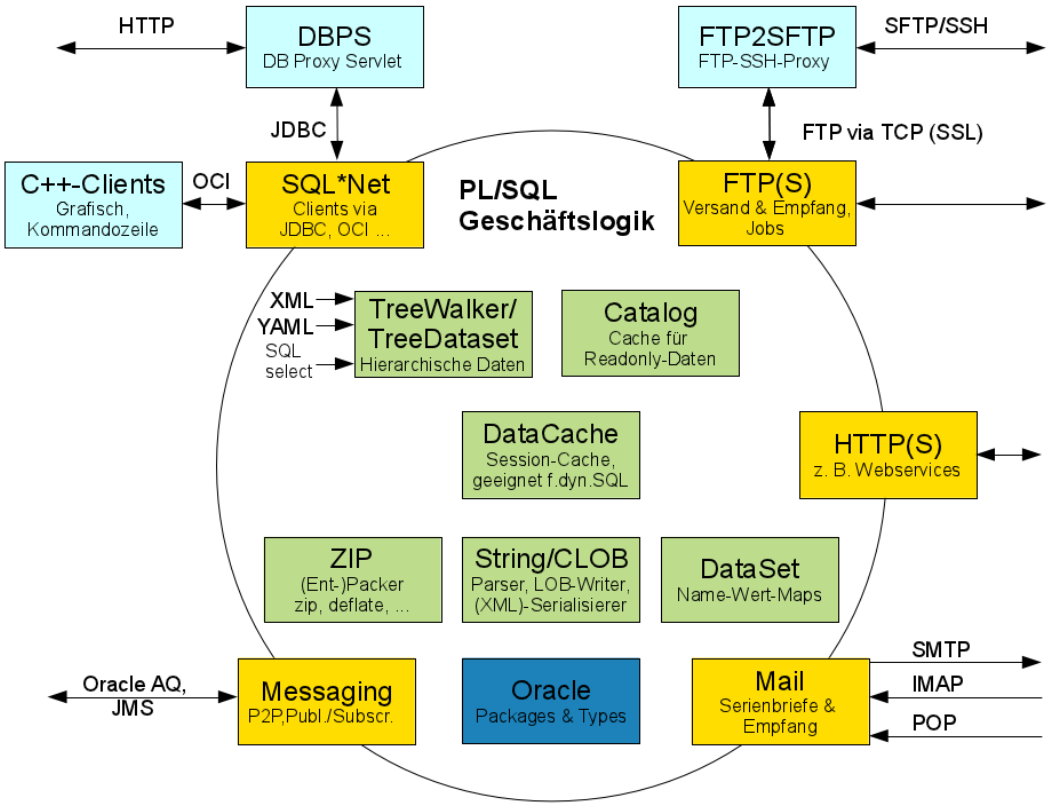

Abb. 1: PL/SQL-Komponenten und Kommunikationskanäle

Zeichenketten-Verarbeitung

Praktisch jeder PL/SQL-Programmierer wird schnell auf immer wiederkehrende Aufgaben wie das Auseinandernehmen komma-getrennter Listen, Zerlegen in Zeilen, Suchen und Ersetzen stoßen. Eine gut gepflegte und getestete String-Package vermeidet Copy/Paste und wird nach einer Weile sehr mächtige Funktionen ausprägen:

```
StringUtil.Split(
  source     in varchar2 | clob,
  tabParts   in out nocopy dbms_sql.varchar2_table,
  delimiter  in varchar2,
  flags      in pls_integer); -- Trim, IgnoreEmpty, SetOfChars, KeepCString

StringUtil.Split(
  source     in varchar2 | clob,
  tabPairs   in out nocopy StringUtil.t_tabNameValuePairs,
  pairDelim  in varchar2,
  valueDelim in varchar2,
  flags      in pls_integer);
```

Hier wird mit einem Aufruf ein Text in seine Einzelteile zerlegt, wobei über eine Reihe von Flags (hier als Bitmaske) diverse Aspekte gesteuert werden, z. B. das Überspringen leerer Elemente, die Beachtung von C/Java-Strings (ein weiteres Flag für SQL-Strings gibt's natürlich auch), die Möglichkeit, verschiedene Trennzeichen zu verwenden (SetOfChars) und anderes mehr

Die angedeuteten Split-Routinen erlauben verschiedene Implementationen: man kann mit den bekannten SQL.-Funktionen INSTR und SUBSTR, aber auch die neueren REGEXP_INSTR und REGEXP_SUBSTR für reguläre Ausdrücke arbeiten. Für die Verarbeitung von CLOBs ist unter Umständen sogar Java geeignet – hier muss der Geschwindigkeitsgewinn in Java den Overhead des PL/SQL-nach-Java-Calls und der Datenkonvertierung übertreffen oder wenigstens einigermaßen ausgleichen.

Außerordentlich wichtig und gleichzeitig besonders einfach zu implementieren sind Unit-Tests für eine String-Package. Nur so bleibt die Package leicht erweiterbar und vor allem refaktorierbar.

CLOB- und XML-Writer

Oracle erklärt in der Beschreibung der Funktion GETCHUNKSIZE der Package DBMS_LOB: „Performance is improved if you enter read/write requests using a multiple of this chunk size." (vgl. http://download.oracle.com/docs/cd/B19306_01/appdev.102/b14258/d_lob.htm#i998451).

Wir haben einen CLOB-Writer realisiert, der eine intern verwaltete PL/SQL-long-Variable bis zum Anschlag auffüllt und bei Überläufen in den Ziel-CLOB schreibt. Zusätzlich gewinnt man die Möglichkeiten, Zeichensatz-Konvertierungen durchzuführen (u. a. bei der XML-Erstellung wichtig, um den DB-Zeichensatz in UTF-8 zu überführen) und einen Mark-Rewind-Mechanismus zu implementieren.

Mit dem CLOB-Writer als Basis läßt sich recht einfach ein XML-Writer implementieren, der ein XML-Dokument seriell zusammenstellt und dem Nutzer z. B. die Konvertierung in den Zielzeichensatz, das Escaping und die relativ aufwändige Verwaltung der Element-Hierarchie und der Einrückung abnimmt.

Datasets – Associative Arrays als universeller Datentyp

In unserem DOAG-Konferenz-Beitrag 2006 „Komponenten-basierte, dynamische PL/SQL-Anwendungen" haben wir einen Associative-Array-Typ vorgestellt, der Parameternamen auf Werte in den für uns wichtigen Datentypen mappt (varchar2, number, date/timestamp, boolean, LOB's). Diesen Typ bezeichnen wir als Dataset.

Mit Datasets kann man ein Pattern analog zum J2EE Data Access Object-Pattern realisieren. Spezialisierte oder dynamische Lese- (select) und Schreib-Routinen (insert/update) transferieren physische Datenbank-Daten in und aus Dataset-Instanzen, auf die die Geschäftslogik mittels Get- und Put-Routinen zugreift.

```
-- Get- und Put-Routinen
if DataSet.GetDate(recDS, 'CREATED') is null then
  DataSet.Put(recDS, 'CREATED', sysdate);
end if;

-- oder einfacher
DataSet.PutIfNull(recDS, 'CREATED', sysdate);
```

Gleichzeitig bilden die Datasets einen universell einsetzbaren PL/SQL-Datentyp, der sich als Alternative zu unzähligen RECORD-Definitionen bzw. %ROWTYPE-Variablen anbietet. Der Nachteil, dass dabei die statischen Prüfungen durch den PL/SQL-Compiler umgangen werden, hat sich in unserer Praxis insgesamt als unerheblich erwiesen.

Universeller Session-Cache

Ebenfalls im oben erwähnten DOAG-Konferenz-Beitrag haben wir die Möglichkeiten einer allgemeinen PL/SQL-Caching-Lösung dargestellt. Dazu haben wir eine Mapping-Struktur definiert, in der Zeichenketten-Schlüssel auf Datasets verweisen – den Datacache.

Der Datacache ermöglicht wie auch die Datasets eine Trennung zwischen den SQL-CRUD-Operationen und der Geschäftslogik. Anstatt Daten(-teile) ad hoc per select zu lesen bzw. per insert oder update zu schreiben, kann ein READ-PROCESS-WRITE-Pattern angewendet werden, wobei im PROCESS-Schritt die Geschäftslogik stattfindet. Da der Cache – im Gegensatz zu den Datasets – für dynamische Aufrufe geeignet ist, kann das Pattern z. B. durch Aspekte bzw. Filter rund um den PROCESS-Schritt erweitert werden. Die SQL-basierten READ- und WRITE-Operationen wiederum können gegen andere IO-Kanäle ausgetauscht werden.

Im zweiten großen Anwendungsszenario arbeiten andere Packages mit dem Datacache und können im Idealfall auf eigene Package-Variablen (Package State) verzichten. Hoch konfigurierbare Dienste wie z. B. FTP, HTTP, Logger usw. lassen sich über einen Kontext initialisieren und liefern ihre Ergebnisse – z. B. Mail mit Adressaten, Betreff, Inhalt und Anhängen – sehr kompakt über einen Dataset oder einen äquivalenten Cache-Eintrag zurück; Erweiterungen bei den Ergebnissen verändern dabei nicht die technischen PL/SQL-Schnittstellen.

```
catalogEntry := Catalog.Get('GEMEINDEN', 'ERFURT');
nZipCode     := DataCache.GetNumber(catalogEntry, 'ZIP_CODE');
strState     := DataCache.GetString(catalogEntry, 'STATE');
dtFounded    := DataCache.GetDate  (catalogEntry, 'FOUNDED');
```

Hier holt die CATALOG-Package den Eintrag „Erfurt“ aus dem Katalog „GEMEINDEN“; dabei kann sie den bereits gelesenen Eintrag aus dem Cache verwenden oder ihn aus der Datenbank einlesen. Die Package ist ein Beispiel dafür, wie mittels des Caches Querschnittsfunktionalität gebaut werden kann.

Datensatz-Bäume statt XML

XML wird heute in allen Plattformen umfassend unterstützt. Alternativen wie JSON, YAML oder auch DSL's verdeutlichen aber, dass XML nicht unbedingt für jede Situation ideal ist – insbesondere ist die Daten-(De)Serialisierung mittels XML nicht trivial.

Java, DOT.NET und andere Plattformen unterstützen XML Data Binding – wie aber kann man XML mit den oben vorgestellten Datasets verknüpfen? Eine sehr intuitive Variante sind hierarchisch organisierte Schlüssel:

```
DataSet.Put(recDS, '/s:Envelope/s:Body/i:GetCustomers/i:year', 2011);
```

In diesem Beispiel steckt ein bis auf die fehlenden Namespace-Attribute kompletter SOAP-Request. Datasets mit einer derartigen hierarchischen Struktur nennen wir Tree-Datasets. Die Schlüssel lassen sich gut mit den oben angedeuteten Split-Routinen parsen, während der XML-Writer als Serialisierer

dient. Das Ergebnis sähe so aus (und ist wegen der fehlenden xmlns-Attribute noch nicht valides XML):

```
<?xml version="1.0" encoding="UTF-8"?>
<s:Envelope>
  <s:Body>
    <i:GetCustomers>
      <i:year>2011</i:year>
    </i:GetCustomers>
  </s:Body>
</s:Envelope>
```

Ebenso leicht können Tree-Dataset-Einträge in JSON, YAML oder in anderen Formaten der Wahl dargestellt werden. Also wird hier nicht mehr XML-, sondern PL/SQL-zentriert gearbeitet, was eine spürbare Erleichterung für den Entwickler darstellt. Die Navigation durch Tree-Datasets ist wesentlich einfacher als z. B. über eine DOM-Schnittstelle.

Wie kommen umgekehrt hierarchische Daten, z. B. ein XML-Dokument, in einen Tree-Dataset? Wir haben dazu die Package TreeWalker implementiert, die einen Baum in Form von Datacache-Einträgen bildet, wobei jeder Eintrag einen Knoten mit seinen Attributen (Properties) bildet, eine Beziehung zu einem Elternknoten und eine Reihenfolge innerhalb seiner Geschwister-Ebene hat. Die Attribute der Knoten werden über dynamisch gerufene Handler eingelesen, eine Handler-Gruppe behandelt bspw. XML-Daten, eine andere liest die über diverse Relationen verknüpften Tabellendaten von IBYKUS AP/ ein. Die Handler werden an Events geknüpft, die beim Durchlaufen eines Baums auftreten: GET_ROOT, GET_FIRST_CHILD, GET_NEXT_SIBLING u. a. Die Navigation auf einem solchen Baum kann

1. DOM-artig (GetFirstChild, GetNextSibling),
2. über Xpath-Ausdrücke und
3. in einer Kombination aus DOM- und Xpath

erfolgen. Die Knoten des Baums können vier verschiedene Zustände haben:

1. Noch nicht gelesen.
2. Teilweise gelesen; wenigstens der Knotenname ist gebildet (z. B. XML-Elementname).
3. Vollständig eingelesen; Zugriff auf die Attribute (Properties) des Knotens z. B. in Xpath-Ausdrücken ist möglich.
4. In sequentiellen TreeWalkern: Knoten aus dem Baum entfernt.

Statt den Baum in Form vieler Datacache-Einträge bereitzustellen, kann die TreeWalker-Package ihn auch in einen einzigen Dataset packen – das Gegenstück zur oben dargestellten Serialisierung:

```
-- XML einlesen, Namespaces entfernen, nur relevanten Teilbaum liefern
TreeWalker.GetTree(
  recTreeDS,
  'DIALECT=XML, TRIM_NAMESPACES, ROOT=/Envelope/Body',
  clobXML);
strYear := DataSet.GetString(recTreeDS, '/GetCustomers/year');
```

Damit wird aus dem auf Einzeldatensätze zugeschnittenen READ-PROCESS-WRITE-Pattern ein umfassenderes Muster, bei dem ein ganzer Datensatz-Baum bzw. -Graph an die Geschäftslogik übergeben wird – ohne dass diese wissen muss, wie und wann die konkreten Daten gelesen werden. Damit wird sowohl eine Trennung zwischen fachlichen Daten und physischer Ablage (Tabellen, XML

…) als auch der jeweiligen Repräsentation (konkrete Tabellenstruktur, XML-Dialekt …) erreicht. Das sind Eigenschaften, wie sie auch bei Object-Relational Mappern (ORM) auftreten.

ZIP, JAR, DEFLATE und COMPRESS

Eine verlustfreie Datenkompression für PL/SQL lässt sich mit einer Kombination aus Oracles UTL_COMPRESS und den Java-Klassen in java.util.zip bzw. java.util.jar recht einfach implementieren. Besonders nützlich ist die Daten(de)kompression für

- die Archivierung von „alten" Daten, insbesondere großer (C)LOBs, die z. B. beim Logging, Datenübernahmen, FTP- und Mail-Kommunikationen und dem allgegenwärtigen XML-Austausch anfallen,
- für die Übertragung größerer Datenmengen über langsame Netzwerkverbindungen,
- für das Laden von Quellcode-Archiven (PL/SQL, SQL, Java) in die Datenbank bzw den Export solcher Archive.

In der Praxis setzt sich die Benutzung der Java-Klassen mit den deflate- und zip-Algorithmen durch, da sie bei vergleichbarem Zeitaufwand wesentlich besser komprimieren.

Kommunikationskanäle

Neben der klassischen SQL-Kommunikation mit der Datenbank über OCI und JDBC haben wir es in steigendem Umfang mit Anforderungen zu tun, aus der Datenbank heraus auf Web-, FTP- und Mail-Server zuzugreifen oder Nachrichten mit Messaging-Systemen auszutauschen (Oracle AQ, IBM Websphere MQ). Zumindest Verbindungen „nach außen", zunehmend aber auch im Intranet, müssen sichere Protokolle (HTTPS, SSL/TLS-Verbindungen) verwenden.

Für diese Verbindungen stehen einerseits Oracle-Packages (z. B. utl_http, utl_mail, utl_tcp), andererseits Java SE- und EE-APIs (z. B. java.net, javax.net.ssl, JavaMail) zur Verfügung. Beide Varianten werden bei uns genutzt, für HTTP und FTP können die jeweiligen Netzwerk-APIs (utl_http und utl_tcp einerseits, java.net.* und java.net.ssl.* andererseits) über Konfigurationsparameter gewählt werden:

```
-- Funktionen geben eine Verbindungs-ID zurück.
connWithUtlTcp := Messaging.Open('TYPE=FTP,USE_UTL_TCP,HOST=...');
connWithJava   := Messaging.Open('TYPE=FTP,USE_UTL_TCP=false,USE_SSL,...');
```

Die fachlichen Muster für diese Kommunikationen sind:

- Versand von Daten aus der Datenbank an Kunden, Behörden und Drittsysteme als csv-, xml- und pdf-Dateien, Serienbriefe usw.
- Empfang von Daten für Batch-Verarbeitungen; häufig von Drittsystemen
- Request-Response-Kommunikationen wie Webservices, Dokument-Rendering (Formular-Server und Reporting-Systeme) und Zugriff auf Geo-Informationssysteme.

Innerhalb der Fachlogik werden dabei Messages aus Fachdaten zusammengestellt bzw. in solche überführt. Sie bestehen je nach Kommunikationstyp aus einem oder mehreren Text- oder Binär-Teilen und deren Meta-Informationen (Dateiname, Mimetype u. a.). Die Formate dieser Messages sind nicht datenbank-spezifisch, sondern Standards (HTTP-Formulare, SOAP-Messages), durch Drittsysteme vorgegeben (XML-Dialekte von Formularservern) oder werden zwischen den jeweiligen Kommunikationspartnern ausgehandelt (XML-Austauschformate, PDF-Dokumente).

Neuerdings werden umgekehrt Webservice-Requests an unsere PL/SQL-Fachlogik durchgereicht. Dazu – und in anderen Fällen wie z. B. einer FTP-SFTP/SSH-Bridge – verwenden wir von der Fachlogik unabhängige Agenten bzw. Proxys, die ein nicht direkt in der Datenbank unterstütztes Protokoll (ssh) oder Kommunikationsrolle (HTTP-Server) entsprechend umsetzen.

Für eingehende HTTP-Requests benutzen wir als Alternative zu Oracles beiden Lösungen „mod_plsql“ und „Embedded PL/SQL Gateway“ ein Servlet, das den Request als Datacache-Eintrag an einen PL/SQL-Handler übergibt, der seinerseits die Response über den gleichen Datacache-Eintrag an das Servlet zur Weiterleitung an den Aufrufer zurückreicht. Dieses Servlet lässt sich ohne größeren Aufwand durch andere Lösungen wie z. B. einen C++-HTTP-Server mit OCI/OCCI-Schnittstelle zur Datenbank ersetzen.

Kommunikation über LOBs

Die Kommunikation des Servlets über den Datacache geschieht über simple, aber recht umfangreiche PL/SQL-Aufrufe. Es müssen alle Request-Daten wie CGI-Variablen und HTTP-Formulare übergeben und die Response-Daten inklusive diverser HTTP-Header übernommen werden.

Als Alternative haben wir die Request/Response-Übergabe auf Basis von Objekt-Serialisierung z. B. mittels YAML oder XML realisisiert – daraus ergibt sich der in der Einleitung angedeutete einfache Datenbank-Aufruf. In Groovy sieht dieser Aufruf so aus (Groovy-Experten werden den Code-Schnipsel sicherlich eleganter formulieren können):

```
import groovy.sql.Sql

def doRequest = { request ->
  def sql = Sql.newInstance('jdbc:oracle:thin:@db-server:1521:MYDB',
                            'me','secret', 'oracle.jdbc.OracleDriver')
  try {
    sql.call('begin DoRequest(?, ?); end;', [ request, Sql.CLOB ])
    { response ->
      println response.getAsciiStream().getText()
    }
  } finally {
    sql.close()
  }
}
```

Hier wird eine Datenbank-Verbindung angelegt, eine Stored Procedure mit einem in- (request) und einem out-Parameter (Datentyp Sql.CLOB) gerufen und das Ergebnis (response) über eine Closure auf stdout ausgegeben.

Mit diesem Aufruf kann z. B. realisiert werden:

- Eine SOAP-Webservice-Kommunikation.
- Eine Massendatenübergabe per csv-Datei.
- Eine HTTP-Kommunikation über einen HTTP-JDBC-Umsetzer mit Übergabe der CGI-Variablen und Ergebnisse im Property-Format.

Testbarkeit und Unabhängigkeit

Die LOB-Kommunikation hat zwei weitere Stärken. Zum einen ist sie für beide Kommunikationspartner – DB-Client und aufgerufene PL/SQL-Logik – ausgesprochen gut testbar. Der PL/SQL-Routine ist es egal, ob der Request tatsächlich über HTTP transportiert wurde oder ob ein einfacher PL/SQL-Block als Test-Mock den Request als einfachen String übergibt:

```
declare
  response    clob;
begin
  DoRequest(
    '<?xml version="1.0" encoding="UTF-8"?>
     <soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
       <soap:Body>
         <d:document xmlns:d="http://blau-it.de/doag">
           <d:DOC_IDENT>BLOG_2011-09-14_13:17</d:DOC_IDENT>
           <d:DOC_TITLE>Blog entry from Sep. 14th, 2011</d:DOC_TITLE>
           <d:DOC_MIMETYPE>text/plain; charset="utf-8"</d:DOC_MIMETYPE>
           <d:DOC_TEXT_CONTENT>My first blog entry</d:DOC_TEXT_CONTENT>
         </d:document>
       </soap:Body>
     </soap:Envelope>', response);
  dbms_output.put_line(dbms_lob.substr(response));
end;
```

Zum anderen ist die Schnittstelle der Routine DoRequest komplett unabhängig von Nicht-Oracle-Packages oder -Typen. Diese Unabhängigkeit entkoppelt auch den Aufrufer und ermöglicht analog zum Test-Mock für die PL/SQL-Routine auch die Bereitstellung einer DoRequest-Attrappe für den Aufrufer.

Fazit

Wir haben einige Beispiele von PL/SQL-Packages gezeigt, die innerhalb der IBYKUS-Architektur vor allem für externe Schnittstellen und als Querschnittsfunktionalität Anwendung finden.

Die konsequente Kommunikation über PL/SQL statt SQL, sei es über Datasets, den Datacache oder insbesondere den Austausch von LOBs ermöglicht sehr einfache Datenbank-Aufrufe, die anderen Request/Response-Szenarien ähnlich sind.

Erstellung und Verarbeitung plattform-unabhängiger, nicht unbedingt auf XML beschränkter „Dokumente“, die als LOBs transportiert werden, sind in aktuellen Sprachen und Umgebungen wesentlich leichter als der Umgang mit SQL oder Object Relational Mappern – und sie ermöglicht im Idealfall die Umsetzung des REST-Prinzips mit seiner zustandslosen Client-Server-Kommunikation, unterschiedlichen Ressourcen-Repräsentation und Adressierbarkeit.

Verweise

- Pro und Contra Stored Procedures – Diskussion auf Stackoverflow: http://stackoverflow.com/questions/484516/arguments-for-against-business-logic-in-stored-procedures
- „Developing PL/SQL Web Applications“ (mod_plsql, Embedded PL/SQL Gateway): http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28424/adfns_web.htm
- Data Access Object (DAO) Pattern: http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html

Kontaktadressen

Heiko Blau
Freiberuflicher Software-Entwickler
Hütergasse 13
D-99084 Erfurt

Telefon: +49 (0) 361– 644 584 60
Fax: +49 (0) 361– 644 14 49
E-Mail heiko.blau@blau-it.de
Internet: http://www.blau-it.de

Alexander Möckel
IBYKUS AG
Herman-Hollerith-Str. 1
D-99099 Erfurt

Telefon: +49 (0) 361– 44 10 355
Fax: +49 (0) 361– 44 10 110
E-Mail alexander.moeckel@ibykus.de
Internet: http://www.ibykus.de