

JavaTMmagazin

Internet & Enterprise Technology

XML
extra included

J2EE Persistenz

Wie speichert man Daten? CMP vs. JDO

Business Process Management

Geschäftsprozesse analysieren, kontrollieren, optimieren

Java aus SAP aufrufen

Bi-direktionale Kommunikation

IBM WebSphere

Neuer Application Server

Peer-to-Peer

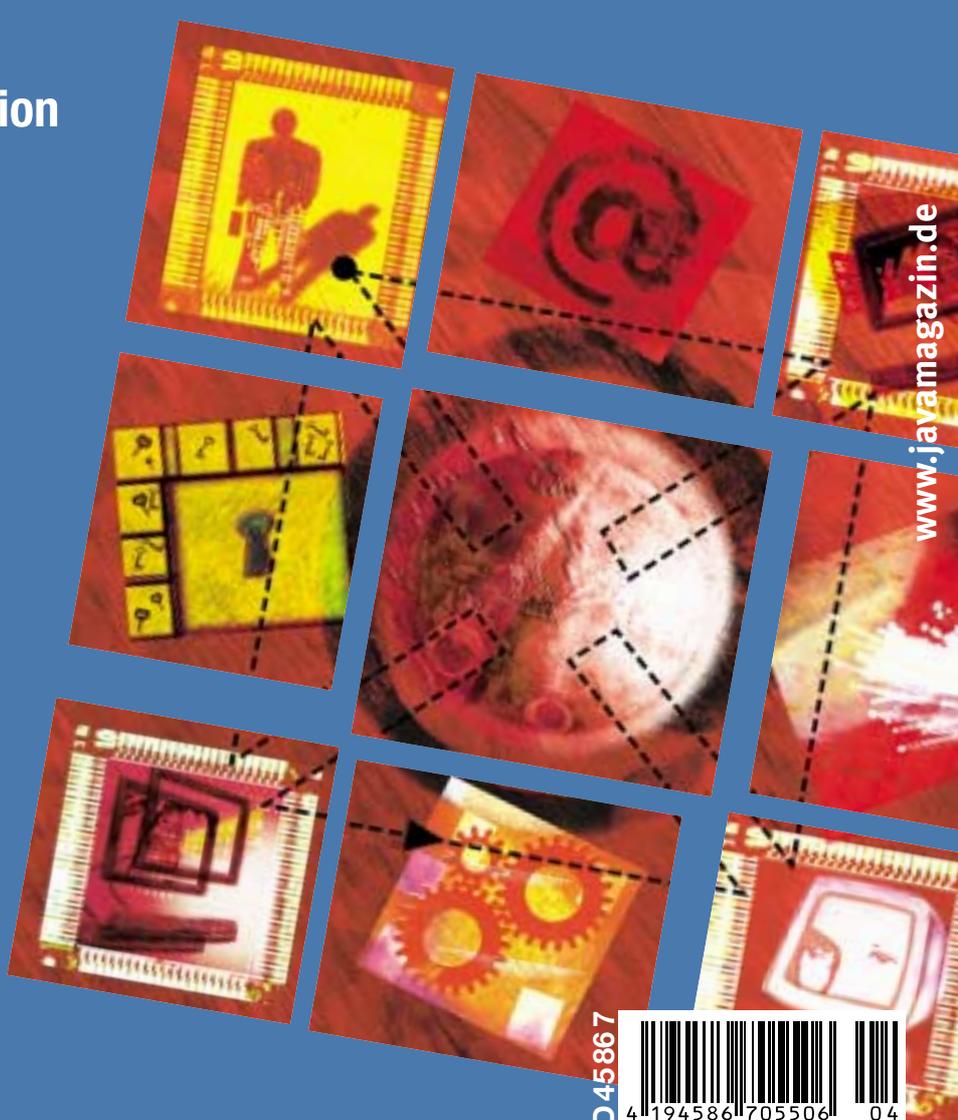
Praxis-Tutorial JXTA

Over The Air

Java auf Handys installieren

Generatoren

Massendaten erzeugen
und darstellen



www.javamagazin.de



mit CD!

D 45 86 7



Datenbankunabhängige Applikationsentwicklung mit Open Source Middleware

von Erik Rehrmann

Ionisierung

Was macht eine gute Middleware aus? Dass sie eine sprachübergreifende Kommunikation zwischen Client und Server erlaubt? Dass ein Client quasi-parallel mehrere Verarbeitungen auf dem Server durchführen kann, ohne dass der Client Multithreading-fähig programmiert sein muss? Dass ohne Applikationsänderungen verschiedene Datenbanken verwendet werden können, auch gleichzeitig? Dass der Sourcecode öffentlich ist?

Da in einem Projekt ein seit vielen Jahren bewährtes System wegen veränderter Lizenzbedingungen und einer neuen Versionspolitik nicht mehr eingesetzt werden konnte, musste eine neue Lösung her.

Ionic ist ein offenes System auf Basis von Open Source. Eine Vielzahl von Features werden zur Verfügung gestellt, die gleichzeitig eine flexible Nutzung zulassen. Dabei muss weder CORBA beherrscht werden, um eine sprachübergreifende Kommunikation zu ermöglichen, noch muss der Anwender Enterprise JavaBeans programmieren können, um eine Application Business Logic auf dem Server zu entwickeln. Vor allem erlaubt Ionic einen einfachen Zugriff auf unterschiedliche Datenbanken, ohne dass verschiedene SQL-Statements für die zum Einsatz kommenden DBMS programmiert werden müssen.

Ionic stellt keine reine Middleware dar. Es ist ein Verbund von aufeinander abgestimmten Modulen, die (gemeinsam eingesetzt) Anwendung finden als Middleware, Application Server oder für Web Services. Separat können sie als Bibliotheken genutzt werden, um die Portabilität und Flexibilität einer Applikation zu steigern. Die folgenden Abschnitte listen die Komponenten im Einzelnen auf.

ionic.net

Für die .NET-Komponente wurde XML-RPC als zugrunde liegendes Kommunikationsprotokoll gewählt, weil es im Gegen-

satz zu CORBA und sogar zu SOAP eine leichtgewichtige Netzwerkkommunikation auf Basis von XML-Dokumenten darstellt. Eingesetzt wurde dafür die Marquee-Bibliothek, eine Open Source-Implementierung von XML-RPC. Die Kommunikation zum Server ist bei XML-RPC auch durch Firewalls hindurch möglich, da es über das HTTP-Protokoll getunnelt wird. Nicht nur um Web Services zu unterstützen, sondern um ganze Applikationen auf Basis von XML-RPC laufen zu lassen, war es notwendig, mehr als ein einfaches Session Handling zur Verfügung zu stellen.

Dieses Problem ist gelöst, in dem die Marquee-Bibliothek durch ein so genanntes Application Handling erweitert wurde. Damit ist die Verwendung normaler XML-RPC Clients immer noch gewährleistet. Es ist sogar möglich, durch einen erweiterten Methodenaufruf dem Server mitzuteilen, dass sich die Aufrufe im Kontext einer Applikation befinden und daher alle Zustandsinformationen vorgehalten werden sollen. So wird beim Start eines Clients der Application Handler *Ionic.app* instanziiert

und die Verbindung auch über einen langen inaktiven Zeitraum hinweg vorgehalten. Die maximale Inaktivitätszeit kann definiert werden, sodass offene Sessions automatisch, aber ordnungsgemäß, vom Server selbst beendet werden.

ionic.app

Jede Applikation stellt die Businesslogik auf dem Server über einen Application Handler zur Verfügung. Eine Liste der benutzbaren Handler liest der Server beim Start ein und registriert sie. In einem zukünftigen Release soll der Server ebenso dynamisch zur Laufzeit neue Application Handler registrieren.

Wenn sich ein Client über die Startmethode einer Applikation am Server anmeldet, wird für diesen Client eine neue Instanz des Server-Objektes erzeugt. Von diesem Zeitpunkt an greift der Client bis zur Abmeldung auf diese Instanz zu. Das Verfahren ist allerdings wenig geeignet, um Hunderte oder Tausende von Sessions für eine Internet-Applikation vorzuhalten. Es ist für den Einsatz normaler Appli-



Quellcode auf CD!

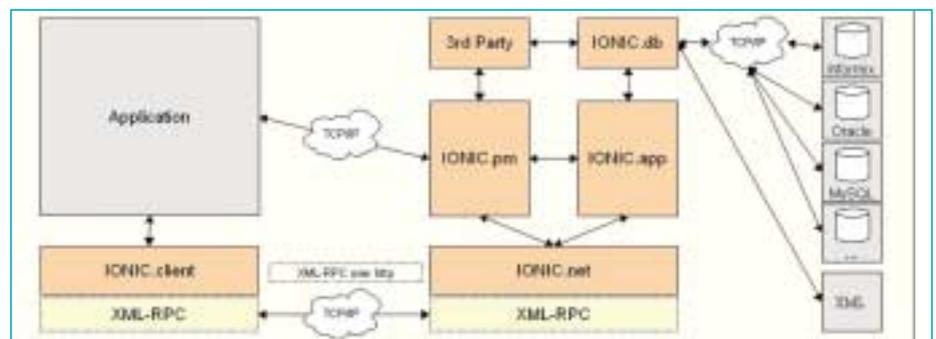


Abb. 1: Übersicht der Ionic-Komponenten

kationen erstellt, d.h. in einer Umgebung, in der die maximale Anzahl von gleichzeitigen Benutzern annähernd bekannt ist.

Für ein Projekt im Umfeld von Web Services werden zur Zeit Untersuchungen zur Skalierbarkeit der Komponenten durchgeführt. Die gewünschte Größenordnung soll bei 8.000 – 10.000 gleichzeitigen Zugriffen liegen.

Ionic.client

Der *Ionic.client* wird benötigt, um C++ Clients die Kommunikation zum Server zu ermöglichen. Die in dieser Bibliothek enthaltenen Klassen erlauben einen schnellen Austausch mit den serverseitigen Datenobjekten, die durch *Ionic.db* zur Verfügung stehen. Der Aufruf der Businesslogik von *Ionic.app* ist vollständig in diesen Objekten gekapselt. Für den Datenaustausch mit dem Server wird die XML-RPC Implementierung von Eric Kidd verwendet. Als Transportschicht wird seit kurzem *cURL* eingesetzt, da die Verwendung von *wininet* wegen der Abhängigkeit von Einstellungen des Internet Explorers zu Problemen geführt hat.

Ionic.com

Für MS-Windows-Plattformen wurde der *Ionic.client* in einem COM-Objekt gekapselt, um auch anderen Sprachen wie Delphi und Visual Basic einen einfachen Zugriff auf den Server zu ermöglichen.

Ionic.pm

Die Prozessmanager-Komponente übernimmt die Annahme, Verwaltung, Ausführung und Protokollierung von Prozessen. Ursprünglich ist die Komponente entwickelt, um die Ausführung von „Langläufern“ vom Client zu starten, ohne dass der Client auf die Beendigung der Ausführung warten muss. Langläufer können langwierige Berechnungen, Druckaufbereitungen oder Analysen historischer Daten sein.

Der Prozessmanager kann so konfiguriert werden, dass für unterschiedliche Applikationen verschiedene Prozesstypen verarbeitet werden. So kann die Applikation <A> die Ausführung von Shell-Skripten und 4GL-Code von Datenbanken wünschen, während die Applikation Java und C++ Prozesse ausführen möchte. Die Konfiguration dafür ist zweigeteilt. Auf der

Serverseite gibt es einen statischen Konfigurationsteil, der Informationen über die zu unterstützenden Applikationen gibt. Dieser ist in der Datei *pm.call* definiert und besteht aus einem Rufzeichen (Callsign), dem Namen der Applikation sowie einem Pfad auf ein Shell-Script-File, das beim Aufbau des Prozesses als Basis verwendet werden soll, wie beispielsweise.:

```
# CallSign, Application, Path to environment file
JAVA,HAV-RX,.../etc/pm.jhav-rx.app
```

Der dynamische Konfigurationsteil wird vom Client mit dem Prozessauftrag mitgeschickt und kann sowohl Erweiterungen zur Laufzeitumgebung enthalten als auch Programmparameter. Als der Prozessmanager entwickelt wurde, gab es für den Informationsaustausch zwischen Client und Server das *Ionic.net* noch nicht. Daher besitzt der Prozessmanager einen eigenen Socket-Server, der auf einem konfigurierbaren Port lauscht und Aufträge entgegennimmt. Auch das Austauschformat ist noch nicht auf XML standardisiert, sondern basiert auf einem einfachen ASCII-Format. Ein Client-Request sieht z.B. wie folgt aus:

```
[CALL_SIGN],JAVA
[USERID],8
[SERVERID],1
[APPLICATION],HAV-RX
[FUNC],kapitel
[MODULE],de.fus.havrx.tools.PrintPreparation
[DESCRIPTION],Kapiteldruckaufbereitung
[ENV],hpJahr,2002
[ENV],hpVersion,099003
[ENV],hpEpKap,05010+05070+0571
[EOF]
```

Das Callsign muss zusammen mit der Application-ID in der Datei *pm.call* enthalten sein. Diese Kombination liefert das Basis Shell-Script zum Starten des eigentlichen Prozesses. Je nach Callsign werden die anderen Parameter ausgewertet und in das konkrete Prozess-Shell-Script integriert. Wann ein Prozess tatsächlich ausgeführt wird, hängt von einigen Bedingungen ab. Neue Prozesse werden ans Ende der Prozessqueue eingereiht, können aber durch höhere Prioritäten weiter nach vorne gelangen. In die Prozessqueue eingereihte Prozesse, die noch nicht gestartet

Anzeige

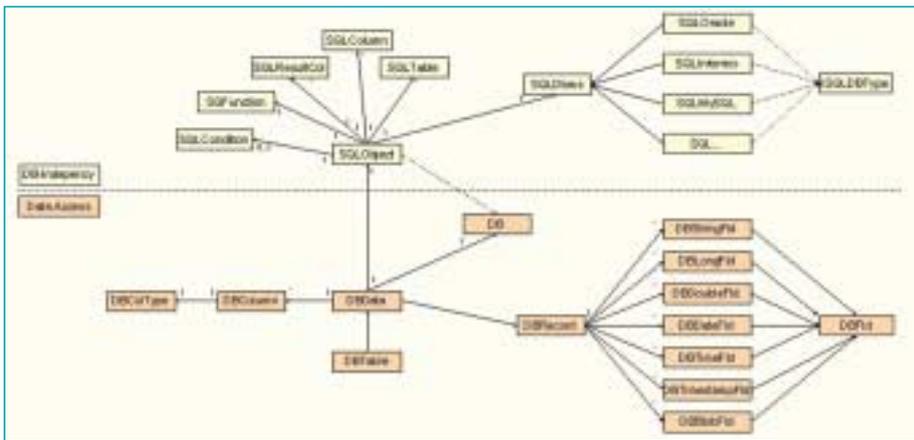


Abb. 2: Eine vereinfachte Darstellung der Klassenstruktur von *Ionic.db*

sind, können in der Priorität auch nachträglich verändert oder sogar ganz gelöscht werden. Sollte ein Prozess nicht gestartet werden können, gibt es eine konfigurierbare Anzahl von Wiederholungsversuchen für einen Restart, mit einer, auch konfigurierbaren, Mindestwartezeit zwischen den Versuchen. Weiterhin können für einen Prozess Ausschlusskriterien definiert werden, d.h. Prozesse werden nicht gestartet und zurückgestellt, wenn bereits Prozesse aktiv sind, auf die eine andere Prozessausführung negative Auswirkungen haben kann. So können z.B. Prozesse für einen Datentransfer zurückgestellt werden, weil Datensynchronisationsprozesse oder Berechnungen noch nicht abgeschlossen sind.

Die Ausführung des Prozesses geschieht durch das Starten eines konkreten Shell-Skriptes, in das vom Prozessmanager alle mit dem Clientrequest erhaltenen Information eingearbeitet worden sind. Die *.pm*-Komponente überwacht und protokolliert jeden aktiven Prozess. Für jeden Client-Request wird ein Prozessobjekt mit seinen Zustandsinformationen persistent gehalten. Dadurch hat der Prozessmanager nach einem Serverabsturz die Möglichkeit, die gleichen Zustandsinformationen wiederherzustellen. Alle bereits gestarteten, aber noch nicht abgeschlossenen Prozesse werden erneut gestartet.

Alle Daten, die von einem aktiven Prozess nach `<stdout>` und `<stderr>` geschrieben worden sind, werden mit dem Prozess verbunden und können zusätzlich zu den Prozessparametern abgerufen und auf Anfrage zum Client übertragen werden. Bereits abgeschlossene Prozesse werden eine

einstellbare Anzahl von Tagen vorgehalten, bis sie automatisch vom Server gelöscht werden. Der Client kann, solange seine Prozessinformationen auf dem Server vorgehalten werden, die Ergebnisse erneut abrufen.

Ionic.db

Um diese Komponente dreht sich (fast) alles. Sie stellt den Zugriff auf die bisher unterstützten DB-Systeme zur Verfügung. Zur Zeit sind das Oracle, Informix und MySQL. Ganz oben auf der Liste der noch zu implementierenden DB-Schnittstellen stehen der Microsoft SQL Server und PostgreSQL. Als Grundlage wird JDBC verwendet, jedoch hat der Anwender der *Ionic.db*-Komponenten in der Regel nichts mehr mit der JDBC-Funktionalität zu tun. Die Auswertung und Bestückung von Resultsets, Absetzen von Datenbank-Befehlen, etc. ist in einer Menge von Klassen in *Ionic.db* gekapselt worden. Abbildung 2 zeigt eine Übersicht der wichtigsten Klassen dieser Komponente und wie sie miteinander im Zusammenhang stehen.

Die datenbankspezifischen Klassen implementieren das Interface *ISQLDbType*. Dort sind die Methoden definiert, die für die Unterstützung eines zusätzlichen DBMS auf jeden Fall implementiert werden müssen. Der überwiegende Teil der Funktionalität wird von der abstrakten Klasse *SQLDbms* zur Verfügung gestellt.

Bei den zu *Ionic.db* gehörenden Klassen fällt eine Zweiteilung in der Benennung auf. Dabei sind alle Klassen, die mit SQL beginnen, eine Abstraktion von SQL-typischen Bestandteilen. Alle Klassen, die mit dem Prefix DB beginnen, sind

entweder für die Abbildung von Ergebnissen ausgeführter SQL-Statements vorhanden oder enthalten Zusatzinformationen für Tabellen, Spalten oder Ergebnisse.

Die wechselnde Verwendung von DBMS wie auch die gleichzeitige Verwendung verschiedener DBMS lässt sich durch Instanzieren einer oder mehrerer DB-Objekte realisieren. Eine besondere Berücksichtigung findet dabei das Data-Type-Mapping. Damit wird sicher gestellt, dass bei nicht exakt gleichen Datentypen verschiedener DBMS eine automatische Verwendung des bestmöglich passenden Typs erfolgt. Bei einer Datenübertragung von einer DB in eine andere unterschiedlichen Typs, wie es bei Datawarehouse-Anwendungen üblich ist, kann die Zieltabelle mit dem für dieses DBMS korrekten Datentyp erzeugt werden, ohne dass der Entwickler dies explizit berücksichtigen muss. Innerhalb der *Ionic.db* Komponenten wird dafür mit den *ION DataTypes* gearbeitet. *DBData* und damit auch *DBTable* halten die Spalteninformationen in einer Menge von *DBCColumn*-Objekten vor. Dort werden zum *ION DataType* auch die JDBC- und DBMS-Typen vorgehalten.

Indexinformationen können automatisch aus der Datenbank ausgelesen und bei Übertragungen berücksichtigt werden, sodass auch diese Informationen im Ziel vorhanden sind. Die Daten müssen nicht 1:1 übertragen, sondern können auch vor der Erstellung im Ziel manipuliert werden, entweder durch Datenveränderungen oder es werden nur geänderte Daten ins Ziel übertragen. Möglich ist auch, dass Zeitstempel erstellt werden und geänderte Daten mit einem Gültigkeitszeitraum historisiert sind. Dies sind nur wenige Beispiele, die aber alle im Datawarehouse-Umfeld häufig zum Einsatz kommen und oft mit teuren ETL (Extraction Transforming Loading)-Tools realisiert werden. Aber gerade hier wird häufig mit Kanonen auf Spatzen geschossen, denn viele ETL-Tools sind mit Funktionen ausgestattet, die nicht benötigt, aber trotzdem mitbezahlt werden müssen.

Die XML-Schnittstelle von Ionic.db

Bei einem Datentransfer ohne direkte Verbindung zur Zieldatenbank gibt es die

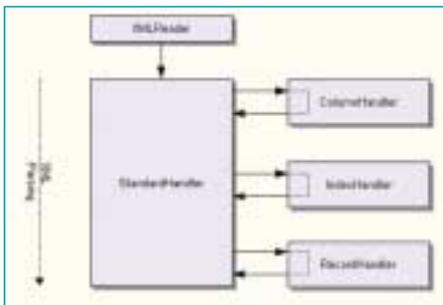


Abb. 2: Die XML-Import Handler

Möglichkeit, die Daten über eine XML-Schnittstelle auszutauschen. Bei der Standardschnittstelle werden alle relevanten Daten der Quelltable, wie Spalteninformation und Indexinformationen, zusätzlich zu den eigentlichen Daten abgelegt. Im Kasten „XML-Datenaustausch“ ist ein Codebeispiel für den XML-Export und -Import aufgeführt.

Die im XML-Format exportierten Daten können abhängig von der selektierten Menge sehr umfangreich werden. Bei der Entwicklung der Schnittstelle wurde Wert darauf gelegt, dass die zu exportierende Datenmenge nicht durch die Ionic-Komponenten begrenzt werden darf. Dadurch wird die Möglichkeit aus, mittels JDOM oder einer anderen API das XML-Dokument im Speicher aufzubauen. Der eigentliche Export der Daten erfolgt daher über das direkte Schreiben der Daten in einen *OutputStream*. Bei der Verwendung eines *FileOutputStreams* kann die entstandene Datei aufgenommen und transportiert werden. Am Zielsystem angekommen, können mit Hilfe des XML-Imports die Daten in eine andere Datenbank wieder eingespielt werden.

Da in der XML-Datei keine speziellen Datentypen, sondern die *ION DataTypes* enthalten sind, können auch aus einer Informix-Datenbank exportierte Daten in ein Oracle-System importiert werden. Auch hier kommt es darauf an, große Datenmengen verarbeiten zu können. Der Import erfolgt Event-gesteuert auf Basis von SAX. Verschiedene *ContentHandler* kümmern sich um die Analyse der eingehenden XML-Daten und erstellen daraus für ein *DBTable*-Objekt der Reihe nach die Spalten- und Indexinformationen. Anschließend werden die eigentlichen Datensätze in die Tabelle eingefügt.

Die Datenbankunabhängigkeit von Ionic.db

Ein wichtiges Merkmal der *Ionic.db*-Komponente ist ihre Datenbankunabhängigkeit. Unterschiedliche DBMS unterstützen zu können ist sehr hilfreich, aber eine richtige Unabhängigkeit wird erst geschaffen, wenn man sich nicht mehr um die Besonderheiten der einzelnen DBMS kümmern muss. Viele wichtige SQL-Funktionen sind bereits in *Ionic.db* datenbankunabhängig verfügbar. Dazu gehören Funktionen aus dem Bereich von *DB*, *Table*, *Index*, *View*, *Procedure* und *Synonym*. Es werden

immer diejenigen Funktionen als Nächstes entwickelt, die gerade benötigt werden.

Natürlich kann die Programmierung eines datenbankunabhängigen SQL-Statements nicht so kompakt ausfallen wie ein konkretes Statement. Alle Bestandteile eines Statements müssen über Objekte abgebildet und angesprochen werden. Für viele Funktionen ist die Verwendung der SQL-Objekte vor dem Anwender verborgen. Zum Beispiel bei den Tabellenbefehlen: Statements wie *Insert*, *Update*, *Delete* oder *Drop* sind durch das *DBTable*-Objekt gekapselt und werden durch mehrere,

XML-Datenaustausch

In den folgenden beiden Methoden wird der Datenaustausch über die Standard XML-Schnittstelle demonstriert. Durch den Aufruf der *xmlExport()*-Methode wird eine Datei mit dem Namen *data.xml* erzeugt. Sie enthält alle zur Tabelle gehörenden Informationen wie Spaltendaten, Indexinformationen und natürlich die selektierte Datenmenge. Der Einfachheit halber wurden hier alle Datensätze exportiert.

Die Methode macht vor allem dort Sinn, wo keine direkte Verbindung zur Zieldatenbank vorhanden ist. Die resultierende XML-Datei kann einfach zum Zielsystem übertragen und dort mit der *xmlImport()*-Methode wieder eingelesen werden.

Die Standard XML-Schnittstelle wird die Tabelle mit allen in der Datei enthaltenen Daten erzeugen, inklusive der Indexinformationen. Das Einlesen einer Teilmenge der Daten, oder auch das Einlesen auf eine Tabelle, die nur eine Teilmenge gleicher Spalten hat, ist über eine Anpassung des Standard- und RecordHandlers einfach möglich:

```
protected void xmlExport() {
    try {
        DBTable table = new DBTable(sourceDB, (String)appParameters.get("testTableName"));
        table.selectAll();
        XmlOutput xmlOut = new XmlOutput(new java.io.FileOutputStream("data.xml"));
        XmlWriter xmlWriter = new XmlWriter(table, xmlOut);
        xmlWriter.generateData();
        xmlOut.close();
        CAT.info("XML file <data.xml> successfully created");
    }
    catch (Exception e) {
        CAT.error("Error exporting data to xml file: "+e.getMessage());
    }
}

protected void xmlImport() {
    try {
        DBTable mytable = new DBTable(sourceDB, "xmltest");
        XmlReader xmlReader = new XmlReader(new java.io.FileInputStream("data.xml"));
        StandardHandler standardHandler = new StandardHandler(null);
        standardHandler.setData("xmltable", mytable);
        xmlReader.setContentHandler(standardHandler);
        xmlReader.parse();
        CAT.info("XML file <data.xml> successfully imported");
    }
    catch (java.io.FileNotFoundException e) {
        CAT.error(e);
    }
    catch (Exception e) {
        CAT.error("Error import data from xml file <data.xml>: "+e.getMessage());
    }
}
```

teils überladene Methoden zur Verfügung gestellt. Das Anfügen eines neuen Datensatzes an eine Tabelle erfolgt, indem über ein *DBTable*-Objekt die Verbindung zu einer Tabelle in der Datenbank hergestellt (oder eine neue Tabelle angelegt), die Recordstruktur für den neuen Datensatz bestückt und anschließend eine der *DBTable.insert()*-Methode aufgerufen wird. Je nach zugrunde liegender Datenbank wird ein passendes SQL-Insert Statement generiert und zur Datenbank geschickt. Die schwierigste Aufgabe war die Abbildung von SELECT-Statements, da hier viele Funktionen zur Verfügung stehen müssen.

Ein SELECT-Statement kann die folgenden logischen Bereiche enthalten:

- Funktion,
- resultierende Spalten mit eigenen Funktionen,
- Quelle, ggf. mit *join*-Informationen,
- Bedingungen,
- Gruppierung,
- Sortierung,
- Having.

Die ersten vier Bereiche werden über eigene Klassen abgebildet (*SQLFunction*, *SQLResultCol*, *SQLTable*, *SQLCondition*). Zusammengesetzt wird die Gesamtanfrage über das Verbindungsobjekt *SQLObject*, das auch die Container für die letzten drei Bereiche zur Verfügung stellt. Die Abbildung eines SELECT-Statements wirkt zwar recht komplex, ist aber durch ihren logischen Aufbau einfach nachzuvollziehen. Um den Aufbau zu vereinfachen, werden *XMLConditionReader* und *XMLConditionWriter*-Klassen erstellt. Die Klassen erlauben das Einlesen von fertigen Conditions zur Laufzeit. Eine in der Entwicklung befindliche Applikation mit grafischer Benutzeroberfläche wird ein einfaches Erstellen von Conditions mit Mausclick ermöglichen.

Weitere Beispiele für die Verwendung von Ionic.db

Die Möglichkeiten für den Einsatz der Komponenten sind sehr vielfältig und umfassen u.a. die häufig benötigte Funktion für das Übertragen von Daten aus einer Tabelle in eine andere. Die nachfolgend

aufgeführten Variationen wurden bereits in Projekten mit *Ionic.db* realisiert:

- Nur eine selektierte Menge übertragen,
- Daten vor Eintrag in Zieltabelle modifizieren,
- Prüfung auf (Teil-)Übereinstimmung zwischen Quell- und Zielrecord,
- Tabellen liegen in verschiedenen Datenbanken.

Werden Tabellen von einer Datenbank in eine andere übertragen, kann angegeben werden, ob die Zieltabelle lediglich erzeugt oder auch gefüllt werden soll und ob die gleichen Indizes wie bei der Quelltable erstellt werden sollen. Nachdem man Quell- und Zieltabelle angegeben hat, wird der eigentliche Datentransfer mit der Methode *sourceTable.copyTo(targetTable)* durchgeführt. Sollen auch die Indexinformationen übertragen werden, werden sie aus der Quelltable ermittelt und der Zieltabelle bekannt gemacht *targetTable.setIndices(sourceTable.getIndices(false))*. Zur tatsächlichen Anlage der Indizes kommt es, wenn zum Schluss *targetTable.createIndex()* aufgerufen wird.

Viel weniger Arbeit kann man sich für ein datenbankübergreifendes Übertragen von Tabellendaten wohl nicht machen.

Fazit

Die Open Source verfügbaren Ionic-Komponenten sind kein Produkt grauer Theorie, sondern im evolutionären Prozess verschiedener IT-Projekte entwickelt und erprobt. Die Komponenten sind modular aufgebaut und einzeln nutzbar. Somit sind sie auch nicht als Middleware im eigentlichen Sinne zu verstehen. Der Komponenten-Aufbau ermöglicht vielmehr die Integration einzelner Teile von Ionic in eine bereits implementierte Middleware. Dennoch ist Ionic eine Alternative zu den auf dem Markt befindlichen Tools, weil sie in der Gesamtheit aller Module den Leistungen einer Middleware entspricht. ■

Links & Literatur

- IONIC-Komponenten: ionic.ion.ag/ionic/
- XML-RPC: www.xmlrpc.com/
- SOAP: www.w3.org/tr/soap/
- CORBA: www.omg.org/
- JDBC: java.sun.com/jdbc/

Wie ist es zu Ionic gekommen?

Die IOn AG hat seit dem Beginn der neunziger Jahre für eigene Applikationen eine Middleware eingesetzt, die folgende Kriterien erfüllte: Für viele Betriebssystem-Plattformen verfügbar, unterstützte sie mehr als 30 verschiedene Datenbanksysteme mit einem einheitlichen Zugriffslayer. Die Erstellung der Geschäftslogik erfolgte objektorientiert mit einer 4GL-Sprache und war plattformübergreifend gültig. Clients konnten über eine C++ basierte Zugriffsbibliothek die Serverlogik ansprechen. Die Middleware war skalier- und verteilbar.

Leider wurde der Hersteller der Middleware aufgekauft. Die Versionspolitik sowie das Lizenzmodell änderten sich. Die Middleware war für die IOn AG nicht mehr einsetzbar.

Als sich die Entwicklung abzeichnete, begann die IOn AG auf der Basis von Java-Komponenten mit der Erweiterung bestehender Server-Funktionalitäten.

Anfang 2000 wurde *ionic.pm*, der Prozessmanager, als erste Komponente fertig gestellt und in zwei Applikationen eingesetzt. Damit war ein Nachteil der alten Middleware (der Kernel war nicht Multithreading-fähig) ausgemerzt. So konnten lang laufende Funktionen der Applikationen auf den Server ausgelagert werden.

Parallel erfolgte die schwierigste Aufgabe: Die Erstellung der Datenbankkomponente *ionic.db*. Dabei lag der Fokus auf der Erstellung einer soliden, erweiterungsfähigen Basis. Im Frühjahr 2001 wurden erste Projekte zum Befüllen eines Datawarehouses mit Hilfe der *ionic.db*-Komponente erfolgreich realisiert.

Seit Herbst 2001 waren Tests mit den ersten Prototypen von *ionic.client* und *ionic.net* gestartet, nachdem schon einige Applikationen im Einsatz waren, die *ionic.pm* und *ionic.db* einsetzen. Die IOn AG erhielt einen Auftrag zur Umstellung und Erweiterung einer Applikation, die bis dahin auf der alten Middleware aufbaute. Vorgabe war, mindestens die gleiche Performance wie bisher zu erreichen. Tatsächlich wurden große Teile der Applikation wesentlich schneller.

Im Herbst 2002 wurde die Applikation, die nun alle Ionic-Komponenten verwendet, freigegeben und in den produktiven Betrieb genommen. Weitere Applikationen werden im ersten Halbjahr 2003 folgen. Die IOn AG entschied sich im Spätsommer 2002 die Komponenten als Open Source unter der LGPL zu veröffentlichen.