Persistence for Large Enterprise Systems in the Java World

Bernhard Schiefer* and Christian Fecht**

*FH Kaiserslautern
Amerikastr. 1, 66482 Zweibrücken
schiefer@informatik.fh-kl.de

**SAP AG
Neurottstr. 16, 69190 Walldorf
christian.fecht@sap.com

In the following paper several persistence approaches evaluated at SAP are analyzed and a number of open issues are identified. The resulting architecture, which is currently being implemented at SAP as part of its J2EE engine to enable its application developers to produce real world J2EE applications, is presented.

1 Introduction

Development of enterprise applications has essentially two requirements: a persistent data store that enables efficient modification and retrieval of large amounts of data, and a powerful middleware infrastructure for communication in distributed systems. Today's dominating ERP systems were developed during the last twenty years using proprietary 4GL environments, such as SAP's ABAP or ORACLE's Forms. The companies developed their own data access layers, application servers and communication protocols. This has required heavy investments in technology and has hindered tight cooperation of different systems.

During the last seven years Java has become a promising candidate for a standard platform for enterprise applications. The first companies realizing the potential of the J2EE environment were middleware experts that came from the CORBA environment.

During the last few years the Java platform has attracted many of the big players in the ERP software market. Proprietary languages and infrastructure have begun to hinder their further growth in new markets. Customers have become more aware of the importance of standard application infrastructure to enable faster and cheaper modifications of standard software and have begun to prefer software based on open standards. Following their customers needs, many big ERP vendors like IBM, ORACLE or SAP decided to shift their technology basis onto this new platform. Many of the vendors of proprietary middleware have an enormous know-how in the area of building highly scalable middleware and they began to develop Java compliant middleware technology, such as J2EE application servers (IBM WEBSPHERE, ORACLE OC4J, SAP J2EE Engine).

Java is still a young technology and in many areas lacks the capabilities of the proprietary development environments. One of the important points is the data access layer. Java implies an object-oriented way of accessing data, whereas all big ERP systems run on relational data stores. Several standards for accessing data were established in the Java community: JDBC ([JDBC98]), SQLJ ([SQLJ99]), JDO (JDO02]), EJB CMP ([J2EE02]).

Each of these standards solves a different set of problems. However, some problems are not addressed at all by these standards. A very important one is portability among different database systems -- a crucial point for a vendor of DB-independent solutions. Ideally, there should be one way of accessing the database in a portable, efficient and seamless way. To determine the best strategy to access the database in the Java world was the goal of a project at SAP Walldorf. The project showed, however, that no single strategy could be identified as the best one at present. A number of technologies had to be combined to achieve the following goals:

- Portable (database-independent) data access
- Elegant, seamless data access where possible
- Efficient bulk data operations where needed
- Conformance to Java standards

To illustrate the different technologies (JDBC, SQLJ, JDO, and EJB CMP) we use a simple employee, department example:

```
EMPLOYEE   (EID, ENAME, DID)
DEPARTMENT (DID, DNAME)
```

The primary key attributes are underlined.

To demonstrate the different data access approaches, we use a read and an update task:

- The first task consists of printing all employees working for a given department.
- The second task is to move one employee from one department to another.

The excerpts of implementation files given in the following sections are only to illustrate the most important concepts in the relevant context. They are not intended to be complete, executable source code. Exception handling is always left aside, to keep the examples as small as possible. Details on how the connection to the database is established are not given here.

2 Transparent Data Access

Java software developers think in objects. Objects are usually accessed by identifying one object as a starting point and finding further objects by following references or calling methods. Relational database systems have a completely different working model based on set-oriented query and update statements.

If transient and persistent data can be accessed in the same way, algorithms can be used uniformly for transient and persistent data. Application development is simplified and less error-prone because developers concentrate on one programming paradigm. This idea led to the development of object-oriented database systems in the late 80s. Up to now these systems could not fulfill the high expectations.

But the attractiveness of uniform access to data in memory and in persistent stores still remains. This has led to the development of object-oriented wrappers for non-object-oriented data stores like powerful and proven robust relational databases for example. In the Java area two different approaches to access these data stores have been standardized: Entity beans with container managed persistence (CMP) and Java data objects (JDO). Both address the same key issues:

- Object-oriented view of persistent data
- Transparent loading and storing
- Transparent navigation
- Transparent object-relational mapping
- Automatic track of changes
- Automatic bundling of changes
- Separation of business logic and persistence logic
- Support of complex relationships

2.1 Entity Beans with container managed persistence (CMP)

The first approach, EJB entity beans with CMP, was developed in the context of the J2EE standard ([J2EE02]). In EJB 2.0 one declaratively specifies an abstract persistence schema consisting of entity beans that can be related to each other by complex relationships. This abstract persistence schema has to be mapped to a database schema. However, the persistence logic is completely separated from the business logic. The application accesses persistent data only by means of enterprise beans and does not have to know anything about the underlying data store.

The steps to a CMP-based implementation are:

- Provide the entity enterprise Java bean according to the J2EE standard as an abstract class – with abstract get-/set-access methods for the persistent fields. Implementation of the methods will be generated in a subclass at deploy time.
- Write a standard XML deployment descriptor that contains information about the persistent fields, relationship fields, and EJB queries.
- Specify the database, tables, etc. in an application server-specific deploy tool.

In the following figures we present excerpts of the files that must be provided for the CMP solution.

```
public abstract class EmployeeBean implements javax.ejb.EntityBean {
   private javax.ejb.EntityContext context;

   // get and set methods for the cmp fields
   public abstract String getEmpID();
   public abstract void setEmpID (String eid);

   public abstract int getName();
   public abstract void setName (String name);

   // get and set methods for the relationship fields
   public abstract Department getDepartment();
   public abstract void setDepartment(Department d);

   // Other methods required for the entity bean interface
   // and optionally own business methods
   . . .
}
```

Figure 2.1.1: Bean class for the employee bean

```
public abstract class DepartmentBean implements javax.ejb.EntityBean {
   private javax.ejb.EntityContext context;

   // get and set methods for the cmp fields
   public abstract String getDepID();
   public abstract void setDepID (String did);

   public abstract int getName();
   public abstract void setName (String name);

   // get and set methods for the relationship fields
   public abstract Collection getEmployees();
   public abstract void setEmployees (Collection employees);

   // Other methods required for the entity bean interface
   // and optionally own business methods
   . . .
}
```

Figure 2.1.2: Bean class for the department bean

```
<entity>
   <ejb-name> EmployeeEJB </ejb-name>
    <persistence-type> Container </persistence-type>
    <prim-key-class> java.lang.String </prim-key-class>
    <cmp-field><field-name> EmpID </field-name></cmp-field>
    <cmp-field><field-name> Name </field-name></cmp-field>
    <primkey-field> EmpID </primkey-field>
    <query>
     <query-method>
        <method-name> findByName </method-name>
        <method-param> name </method-param>
     </query-method>
     <ejb-ql>
        SELECT object(e) FROM Employee e WHERE Name LIKE '?1%'
     </ejb-ql>
    </query>
</entity>
```

Figure 2.1.3: Part of the <enterprise-beans> deployment descriptor (ejb-jar.xml) for the employee bean

To demonstrate the way in which queries can be specified in a deployment descriptor, a method findByName is shown above. This method searches for employees whose name starts with a given prefix.

```
<ejb-relation>
  <ejb-relation-name> Emp-Dep </ejb-relation-name>
  <ejb-relationship-role>
    <ejb-relationship-role-name> emp-has-dep </ejb-relationship-role-name>
    <multiplicity> Many </multiplicity>
    <relationship-role-source>
      <ejb-name> EmployeeEJB </ejb-name>
    </relationship-role-source>
    <cmr-field>
      <cmr-field-name> Department </cmr-field-name>
    </cmr-field>
  </ejb-relationship-role>

  <ejb-relationship-role>
    <ejb-relationship-role-name>dep-employs-emp</ejb-relationship-role-name>
    <multiplicity> One </multiplicity>
    <cascade-delete/>
    <relationship-role-source>
      <ejb-name> DepartmentEJB </ejb-name>
    </relationship-role-source>
    <cmr-field>
      <cmr-field-name> Employees </cmr-field-name>
      <cmr-field-type> java.util.Collection </cmr-field-type>
    </cmr-field>
  </ejb-relationship-role>
</ejb-relation>
```

Figure 2.1.4: Part of the <relationships> deployment descriptor

Additionally J2EE servers need a proprietary description how to map objects to database structures. For relational databases this file contains a list of table names and fields in these tables (e.g. EMPLOYEE.EID) and the information to which member variables (e.g. Employee.EmpID) this data is mapped. This file is not listed here. A similar file can be found in the next section for JDO.

```
public printDepartment (Department d)
{
  Iterator i = d.getEmployees().iterator();
  while (i.hasNext()) {
    Employee e  = (Employee)i.next();
    System.out.println(e.getEmpID() + ": " + e.getName());
  }
}
```

Figure 2.1.5: Application using CMP to print all employees of a given department

```
public updateEmployee (Employee e, Department d)
{
  e.setDepartment (d);
}
```

Figure 2.1.6: Application using CMP to update the department of an employee

2.2 Java Data Objects (JDO)

JDO is a very new standard ([JDO02]), but there are already a number of implementations available and the number of vendors is growing fast. As with CMP, a mapping from data in the data store to Java objects must be provided. Java applications based on JDO are intended to be independent of the JDO vendor's implementation. But the mapping specification itself is vendor-specific to support arbitrary data stores. JDO is not part of the J2EE standard, and JDO-based programs can be used as stand-alone programs without an application server. The examples given here are based on SAP's JDO implementation.

Writing a JDO-based implementation consists of the following steps:

- Write a mapping file to specify which tables and columns belong to which Java classes and attributes. Normally, JDO vendors provide OR-mapping tools to generate these files.
- Implement the persistent capable classes.
- Implement the persistent aware classes (– classes that use persistent capable classes).
- Run the enhancer tool to modify the byte code of your persistent capable classes so that data in the data store is read and modified according to the methods called by the application.

```
<jdo>
  <class name="Employee" identity-type="application"
         objectid-class="EmployeePK">
    <field name="EmpId"       persistence-modifier="persistent"
           primary-key="true"/>
    <field name="Name"        persistence-modifier="persistent" />
    <field name="Department"  persistence-modifier="persistent"
           embedded="false" default-fetch-group="false"/>
  </class>
  <class name="Department" identity-type="application"
           objectid-class="DepartmentPK">
    <field name="DepId"       persistence-modifier="persistent"
           primary-key="true"/>
    <field name="Name"        persistence-modifier="persistent" />
    <field name="Employees"   persistence-modifier="persistent"
           embedded="false" default-fetch-group="false">
      <collection element-type="Employee" embedded-element="false"/>
    </field>
  </class>
</jdo>
```

Figure 2.2.1: Part of standard conform JDO declaration: EmpDep.jdo

As can be seen here, JDO requires separate key classes for the values of the primary key.

```
<map>
  <class name="Employee">
    <field name="EmpId">  <column name="EID"   table="EMPLOYEE"/></field>
    <field name="Name">   <column name="ENAME" table="EMPLOYEE"/></field>
    <relationship-field name="Department" multiplicity="one">
        <single-foreign-key>
          <foreign-key name="Emp_to_Dep"
              foreign-key-table="EMPLOYEE" primary-key-table="DEPARTMENT">
            <column-pair foreign-key-column="DID" primary-key-column="DID"/>
          </foreignKey>
        </single-foreign-key>
      </relationship-field>
  </class>
  <class name="Department">
    <field name="DepId">  <column name="DID"   table="DEPARTMENT"/></field>
    <relationship-field name="Employees" multiplicity="many">
        <single-foreign-key>
          <foreign-key name="Emp_to_Dep"
              foreign-key-table="EMPLOYEE" primary-key-table="DEPARTMENT">
            <column-pair foreign-key-column="DID" primary-key-column="DID"/>
          </foreignKey>
        </single-foreign-key>
      </relationship-field>
  </class>
</map>
```

Figure 2.2.2: Part of SAP-specific mapping specification: EmpDep.map

```
public printDepartment (Department d)
{
  Iterator i = d.getEmployees().iterator();
  while (i.hasNext()) {
    Employee e  = (Employee)i.next();
    System.out.println(e.getEmpID() + ": " + e.getName());
  }
}
```

Figure 2.2.3: Application using JDO to print all employees of a given department

```
public updateEmployee (Employee e, Department d)
{
  e.setDepartment (d);
}
```

Figure 2.2.4: Application using JDO to update the department of an employee

2.3 Summary: Transparent Data Access

Enterprise beans, as well as Java data objects, are homogeneous, object-oriented ways to access the data. Both offer client side caching of data from the database and reduce database calls to a minimum by performing data modifications in main memory and writing modified objects to the database only when unavoidable.

JDO is a lean solution to provide Java persistence. The implementer can concentrate on the business logic. The access to the persistent objects does not have to follow special rules and looks the same as the access to normal Java objects. However, the query language is far from being perfect. Basic features known from SQL like aggregate functions or the LIKE-operator are still missing.

The complex model for **enterprise beans** leads to a lot of overhead: Bean class, home interface, remote interface and often additionally local interfaces have to be provided.

Another disadvantage is that CMP requires that the application server generates all data access. Unfortunately, in the case of flexible queries that were unknown before runtime, CMP is not applicable. If a CMP-based bean must later be modified to support one dynamic query, it has to be completely reimplemented as a BMP bean. This holds as well if features are necessary that go beyond the capabilities of the EJB QL.

To use two different beans for the same table is not a recommendable solution and will very quickly lead to lost updates, because beans usually use their own locking and, for efficiency reasons, do not rely on database locks.

Both transparent persistence strategies have a conceptual problem when updating many data records. Consider for example increasing the salary of all employees by 3%. Transparent persistence follows the record-oriented programming paradigm of the Java language. Therefore one record after another has to be read from the database, updated in memory and written out at commit time – an extremely inefficient solution for this task.

3 Relational Data Access

Relational databases with relational data retrieval and manipulation languages like SQL have been on the market since the 1970s. Since 1989 different embeddings of SQL into imperative programming languages have been standardized.

The drawback of all these embeddings is the so-called "impedance mismatch", which stems from the use of record-oriented processing in the host programming languages and the set-oriented data processing of SQL.

3.1 JDBC

The first standard way to use relational databases from Java was a call level interface named JDBC.

JDBC offers functions to pass SQL statements to the underlying database at runtime. No special class to hold the persistent data returned by a query is necessary – opposed to the transparent access methods described before.

```
public printDepartment (Connection con, int eid)
{
  PreparedStatement stmt = con.prepareStatement (
                           "SELECT EID, ENAME FROM Employee WHERE EID = ?");
  stmt.setInt (1, eid);

  ResultSet rs = stmt.executeQuery();
  while (rs.next()) {
    System.out.println(rs.getInt(1) + ": " + rs.getString(2));
  }
  rs.close();
  stmt.close();
}
```

Figure 3.1.1: Application using JDBC to print all employees of a given department

```
public updateEmployee (Connection con, int eid, int did)
{
  PreparedStatement stmt = con.prepareStatement (
    "UPDATE Employee SET DID = ? WHERE EID = ?");

  stmt.setInt (1, did);
  stmt.setInt (2, eid);

  stmt.executeUpdate();
}
```

Figure 3.1.2: Application using JDBC to update the department of an employee record

3.2 SQLJ

SQLJ is the standard for embedded SQL in Java. The SQLJ translator is a preprocessor. It extracts embedded SQL statements from *SQLJ-files* and translates the SQLJ-files to *Java-files*. The generated Java files contain calls to the SQLJ- runtime. The SQL statements are kept in serialized form in an additional *Profile file*.

Semantics checking is performed against a schema description, which might be read online accessing a database catalog or in the SAP environment alternatively offline from an XML description of the database catalog.

```
#sql public iterator EmpIterator (int eid, String ename);

public printDepartment (int did)
{
  EmpIterator eit;

  #sql eit = { SELECT EID, ENAME FROM Employee WHERE EID = :IN did};

  while (eit.next()) {
    System.out.println(eit.eid() + ": " + eit.ename());
  }
  eit.close();
}
```

Figure 3.2.1: Application using SQLJ to print all employees of a given department

```
public updateEmployee (int eid, int did)
{
  #sql { UPDATE Employee SET EID = :IN did WHERE EID = :IN eid};
}
```

Figure 3.2.2: Application using SQLJ to update the department of an employee record

3.3 Summary: Relational Data Access

Developers have to program different code for transient and persistent objects. Additionally, these program parts have to follow different data access paradigms and different languages are used to read and modify values of persistent and transient data.

JDBC database statements are passed to the database at runtime, therefore syntax checking of SQL queries cannot happen at compile time. This leads to the following disadvantages:

- Time has to be spent on analyzing the statements
- Syntactical errors may occur at runtime. Error handling in the program is additionally required (but not shown in the examples above).

In addition, The JDBC interface is not very comfortable to use.

SQLJ is currently restricted to static SQL. Syntax checking of SQL queries is done at compile time. This leads to some major advantages over JDBC:

- No runtime syntax errors
- Easier to use – more comfortable syntax
- Better runtime performance possible: statements can be analyzed and optimized at compile time

4 Integrating it all – The SAP Approach

In this paper, standard ways of accessing relational databases have been briefly introduced. In summary, none of them seems to be a perfect candidate for all cases:

- Transparent access methods fail especially when mass updates are required. The capabilities of their poor query languages are also sometimes a problem. CMP is part of the EJB standard and must be supported by a server to be J2EE compliant. JDO is more attractive to use, because it is more flexible and can also be used stand-alone in a non-managed environment..
- Relational access methods are more complicated to use and inefficient if a transactions performs multiple updates on the same object. SQLJ is safer and more efficient than JDBC: The statement is known at compile time and can be checked and optimized then. Up to now, standard SQLJ unfortunately does not support dynamic SQL.

The approach taken at SAP is to develop of a J2EE compliant application server (SAP J2EE Engine), which offers amongst other needed services a combination of all the described persistence strategies. The complete architecture is shown in figure 5.1.

All the goals introduced in Section 1 are addressed by this architecture:

- Portable (database-independent) data access
- Elegant seamless data access where possible
- Efficient bulk data operations where needed
- Conformance to Java standards

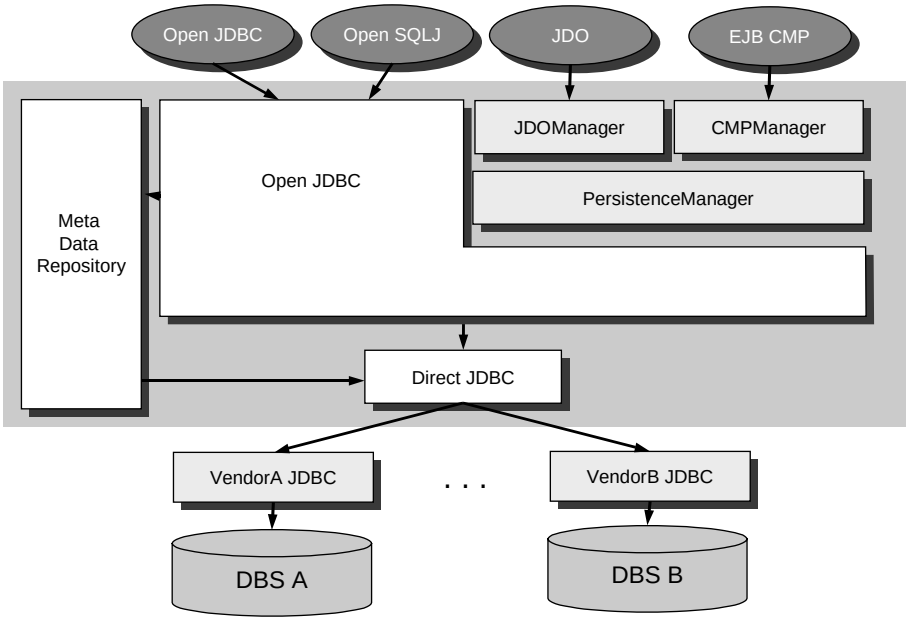


Figure 5.1: Architecture of SAP J2EE engine persistence layer

Key features of this architecture are:

Open SQL for Java

One of the problems mentioned before are the many implementation-dependent parts of the SQL standard. Open SQL for Java is a common set of SQL statements that is available for the developers using the SAP J2EE engine. The database layer does not accept SQL statements that do not conform to Open SQL for Java. Thus, it is guaranteed that all applications are portable to all database systems supported and all these applications behave identically on all platforms, i.e. they lead to the same results. Naturally, there may be differences in response time.

This component not only leads to some restrictions concerning the SQL statements that may be used, but also restricts the types of Java variables that can be used as in-/output parameters because some of the conversions also depend on the database driver used.

A similar approach was taken at SAP in the ABAP environment with *Open SQL for ABAP*, but there were a lot of extensions to standard SQL. *Open SQL for Java* currently does not contain proprietary extensions, so that applications written against this module also work in non-SAP environments with SQL'92 compliant databases.

Open JDBC

Open JDBC is a JDBC API implementation that restricts the SQL language to Open SQL for Java. Open JDBC also contains a buffer to avoid repeated calls to the database. Buffering happens only for the database tables if requested in the Meta Data Repository.

Open SQLJ

Open SQLJ restricts the SQLJ statements to the portable Open SQL for Java

Meta Data Repository

To achieve platform independency, Open SQL is capable of performing semantics checks against a platform-independent logical database catalog. This catalog is mapped to the true database catalog in a platform-specific way.

Persistence Manger

The persistence manager component is the core engine of the transparent Java database access. Its responsibility is the management of cached instances representing data records.

JDO Persistence Manger

The JDO Persistence Manager implements the protocol defined in [JDO02].

CMP Persistence Manger

The CMP Persistence Manger is responsible for the EJB compliance of the J2EE engine with respect to container-managed persistence as defined in [J2EE02].

Direct JDBC

Direct JDBC is a JDBC driver manager that adds the following services to the JDBC drivers provided by the database vendors:

- Caching of prepared statements. If the same statement is called several times, the cursor to this statement is not closed, so that reparsing on the database side is avoided.
- Logging of statements as a basis for SQL analyzers to identify performance bottlenecks easily

The functions added by the direct JDBC driver are identical to the services that SAP developed for the ABAP environment. These services have been used in this environment for many years and have led to very good performance and excellent analysis possibilities.

5 Conclusion

The main standards to access databases from Java have been analyzed in this paper, and the main components of the architecture currently being implemented at SAP have been presented. At the start of the project at SAP there was the expectation that one of these techniques could become the new standard technique for application developers at SAP. As it turned out, none of these alone is currently able to be the basis for the development of large enterprise systems. The reasons for this are pointed out in section two and three.

This has led to the architecture where all these standards and additionally some necessary extensions are now available to application developers. The main focus is currently on SQLJ and JDO.

This complex approach is far from an optimal solution. Research and standardization need to do a lot more work to make application development in the Java environment as simple as it was with proprietary 4GL languages. Transparent object-oriented data access still needs to be combined with means to update large amounts of data efficiently in one consistent model.

References

[J2EE02] *Enterprise JavaBeans Specification*, Final Version, Version 2.0, SUN Microsystems Inc., August 2001

[JDBC98] JDBC™ 2.0 API, Sun Microsystems Inc., May 30, 1998

[JDO02] Java Data Objects, JSR 112, Version 1.0, SUN Microsystems Inc., 2002

[SQLJ99] ANSI NCITS 331-1:1999, SQLJ - Part 1: SQL Routines using the Java(TM), 1999