JPA implementations versus pure JDBC

Jorge Edison Lascano
Master of Science in Software Engineering student,
San Jose State University
Computer Science Department faculty staff member,
Escuela Politecnica del Ejercito
001-408-625-0469
elascano@dcc.espe.edu.ec

Abstract
In the present paper, I introduce two different approaches adopted for data persistence on Java platforms over the course of time: JDBC (Java Data Base Connectivity) and JPA (Java Persistence API). Generally speaking; for any n-tier application, data persistence is the most critical part and can be achieved through different ways, like: file systems, RDBMS (Relational Database Management System), object oriented DBMS, XML files, etc. From POJO (Plain Old Java Objects) to recent EJB3.0 (Enterprise JavaBeans) specifications and from JDBC to JPA implementations, there have been lots of changes in the context of persistence on Java platforms. In either case, the method chosen by the developer for data persistence depends mainly on the architecture of the application to be built. Furthermore, each method has its own pros and cons associated with it; and even though, not everything is black or white in database access implementations, the new tools let developers use an improved persistence routine based mainly in Object – Relational Mapping techniques; as a consequence, persistence providers as Hibernate or Toplynk have appeared to facilitate these improved persistence routines in a transparent way for Java developers.

In the last decade, we have seen many new technologies in data persistence coming up; nonetheless, the usage of old methods of data persistence is not completely demolished. Primarily, my argument is focused on the comparison between Java Persistence API and JDBC (Java Database Connectivity), mainly from the programming and mapping complexity points of view; this way, along the document, I will be giving an approach of how the new persistence technologies are more helpful and powerful in persisting data and mapping objects to entities effectively than earlier technologies.

Categories and Subject Descriptors D.2.11 [Software Engineering]: Software architectures; H.4.3 [Information Systems Applications]: Communications applications;

General Terms Programming, Database Access.

Keywords JPA, entities, persistence, JDBC, Programming, java, EJB.

1. INTRODUCTION

Since the first days of Java, JDBC API has provided the standard set of Java classes and interfaces with their respective methods for executing SQL statements against a relational database[21], namely: class.forName(…), when you want to load JDBC drivers; DriverManager.getConnection(), when you want to open connections to databases; among others. And it had been the most widely used technology to persist data under Java platforms by nearly all programmers. While Java was evolving, there have been many different versions of JDBC in the course of time, from the first available version JDBC1.0 to the latest JDBC 4.0 (JDK 6). Regardless of the version, accessing a database using JDBC API creates a lot of overhead due to the disk head movement; and the software developer has to write intense code for achieving the object relational mapping[1]. Nevertheless, some of the n-tier web applications still use JDBC API with SQL statements or with stored procedure calls for persisting objects and for Create, Read, Update and Delete operations (a.k.a. CRUD). Finally, in order to improve performance response especially in web applications, cache has to be implemented in the application tier.

In the most of cases, the fastest way to get data from a database is not to execute an enhanced query; the answer is not accessing database often, it means to access cache[2]: When a query is executed by the client, the returned value

[1] Object Relational Mapping is the process to match a database table to a programming class with its correspondent columns matched as attributes

[2] Cache is a map-style object implementation that holds key-value pairs in memory, for example ID-1, name-Edison, ID-2, name-Daniel, reducing the number of accesses to the database.

is stored as the value of a key (is cached), whether in memory or in hard drive, whether locally or in a secondary server; the main idea is to reduce direct access to the data source as much as possible. Thus, whenever the browser (or any other client) needs to access information from the server, the cache is reviewed first by the key provided; if the information was cached, then the value is retrieved from the cache; consequently, a further visit to the server is not needed.

Besides application cache; some DBMSs manage their own cache implementations, one example of this is MySQL [10], which enables query cache in an easy way; moreover, it is automatically driven by MySQL and "clients will not cache the information more than necessary" [11]; in addition to this cache, and in order to improve performance, application and client cache is essential; unfortunately, application cache is not present in traditional JDBC implementations, and must be coded by the use of some extra libraries and code; on the other side, Hibernate for example can use ehcache[3] by means of a simple configuration file and some annotations[4] [13]. Another kind of caching is connections pools cache, which lets manage a number of database connections without the necessity to open a connection each time a new query is executed, this type of cache can be managed indistinctly by JDBC or JPA (Java Persistence API). With the aim of solving the problems mentioned before (cache and object - relational mapping implementations), and consequently increasing data access performance and developer productivity, new APIs have appeared.

In the last few years many object relational mapping frameworks (a.k.a. data persistence providers), such as: Bea Systems "KODO"; Oracle "TopLink"; JBoss "Hibernate"; and Apache "OpenJPA" have come to appear in order to improve performance and make data persistence and Object - Relational Mapping simpler for the developer and faster for the user. They relieve developers from writing intensive Java code to manage connections, queries, cache and Object – Relational Mapping; as a result, developers will be able to concentrate on business logic when developing software. Moreover, in the attempt to solve those challenges on data persistence, Sun has implemented Java modern standards for enterprise applications.

[3] Ehcache: a cache provider for Hibernate 2. and 3.x

[4] "Annotation metadata: it is a language feature that allows attaching source code with structured and typed metadata" e.g. @Secure [22].

Sun implemented EJB 2.x container - managed persistence (CMP) entity beans to manage data persistence; nevertheless, it cannot be considered a successful solution at all, because of its complexity in development and deployment. Since then, there have been recent developments coming up in the field of Java persistence and object relational mapping.

Lastly, with the launch of the EJB3.0 - JPA specification, Sun Microsystems has standardized the persistence to a large extent on Java platforms; thus, EJB 3.0 specifications [16] utilize Object -Relational Mapping contributions from JDO, Hibernate and TopLink communities [24] because they have many technical vantages over the pure use of the standard JDBC API. Concluding, the EJB3 specification offers a compelling option for building the persistence tier of Enterprise and Standard Java Applications with the least difficulties for the programmer, with transparent Object - Relational Mapping implementation and with improvements related to data access and administration. This way, JPA has many advantages over the traditional and pure JDBC; advantages that make developers move from JDBC to JPA; especially in the development of new pure object oriented enterprise systems; nevertheless, JDBC code is still being used by developers and supported by Sun, all over the world.

2. JDBC - Java Database Connectivity

JDBC API makes possible for Java developers to connect to relational databases and perform database transactions using SQL queries; furthermore, it allows Java developers to access database table data from Java applications using database specific JDBC drivers and connection pools. Explicitly, there are four types of JDBC drivers [15]: (1) JDBC-ODBC bridge, which converts Java syntax to native ODBC calls; (2) Native API, which converts Java to native DB API (client); (3) Network Protocol (Pure Java), a database driver implementation which makes use of a middle-tier between the calling program and the database; and, (4) Native Protocol, a database driver implementation that converts JDBC calls directly into the vendor - specific protocol. More to the point, by using JDBC to manage connections and to access to database tables; lots of lines of code must be written repetitively; additionally, coding the Object to Relational Mapping is also code intensive and non – trivial, it means the programmer should create the classes to match the tables in the database with corresponding possible mistakes or omissions and by hand checking (the impedance mismatch). In figure 1, the code to retrieve all PERSON objects using basic JDBC is shown (code for connecting to database and loading the driver is excluded in this

example and object relational mapping is programmed in the function mapPerson ()).

```
1.  public ArrayList getAllPersons()
2.    {
3.       ArrayList all = new ArrayList();
4.       PooledConnection connPool = null;
5.       Connection conn = null;
6.       PreparedStatement stmt = null;
7.       ResultSet rs = null;
8.       try
9.       {
10.        connPool = getConnection();
11.        conn = connPool.getConnection();
12.
13.        stmt = conn.prepareStatement
14.            ("select * from PERSON");
12.        rs = stmt.executeQuery();
13.        while ( rs.next() )
14.        {
15.          person p = mapPerson(rs, conn);
16.          all.add(p);
17.        }
18.      }
19.      catch( Exception e )
20.      {
21.        e.printStackTrace();
22.      }
23.       finally
24.      {
25.        if ( rs != null )
26.           try { rs.close(); }
27.              catch(Exception ex) {}
28.        if ( stmt != null )
29.           try { stmt.close(); }
30.               catch(Exception ex) {}
31.        if ( connPool != null )
32.           try { returnConnection(connPool); }
33.             catch(Exception ex) {}
34.      }
35.
36.      return all;
37.   }
38. public Person mapPerson(ResultSet rs,
39.     Connection conn)
40.      throws Exception
41.   {
42.      Person p = new Person();
43.
44.      p.setId(rs.getString(1));
45.      p.setLastName(rs.getString(2));
46.      p.setFirstName(rs.getString(3));
47.
48.      return p;
49.    }
```

Figure 1: Java code to retrieve all objects using JDBC [9]

Although JDBC has some drawbacks, developers are still using it; consequently, some enhancements of JDBC have taken place in the version 4.0 to help especially on Object - Relational Mapping and ease of connection; among others, the new features include: annotation based SQL (DataSet implementation of SQL using annotations); simplified connection management; auto loading of JDBC driver classes [12]; SQL XML support; and, dataset-connected and disconnected view of a result set; nevertheless, these features are not enough to consider JDBC a complete solution to systems performance and developer productivity.

One important feature to mention is the use of annotations from JDBC 3.0. With this new characteristic, object - relational mapping will be done transparently inside the Java code with the use of annotation metadata; as a consequence, the use of deployment descriptors is not needed anymore.

Other important feature to mention is that besides SQL statements, the use of prepared statements feature, annotation-based SQL[5] can be used: "With JDBC 4.0, Java developers are able to specify SQL queries by using annotations, taking the advantage of metadata support available with Java release SE 5.0. This way, the developer does not have to look in two different files for JDBC code and the database query it is calling. For example, if a method called getActiveLoans () to get a list of the active loans in a loan processing database must be implemented, it can be decorated with a @Query (sql='SELECT * FROM LoanApplicationDetails WHERE LoanStatus = 'A'') annotation" [12]. In the next example (figure 2), the use of a base query is shown, in this case, the query will be accessed by g.getAll () instead of writing the SQL query. Before annotations were used to avoid repetitive and complex SQL statements a data base feature called stored procedures was implemented by some DBMSs.

```
interface PersonQueries extends BaseQuery {
     @Select (sql="SELECT * FROM person")
     DataSet<Person> getAll ();
}
PersonQueries    q    =    con.createQueryObject
(PersonQueries.class);
DataSet<Person> ds = q.getAll ();
```

Figure 2: Java 6 JDBC 4.0 annotation based SQL code sample

The use of SQL was the solution for the most of programmers while trying to access data and many SQL sentences had to be written inside business logic code. But, since executing multiple SQL calls from a Java application by using JDBC API, creates countless overhead and affects back - end - tier performance to a large extent due to increasing disk head movement; the

[5] Annotation-based SQL queries let us specify the SQL query string right within the code using an Annotation keyword, and them the programmer can use it as if it were any other method of the class.

use of stored procedures to improve performance was introduced; in this sense, this is more efficient, rather than executing multiple SQL calls from the application.

A DBMS usually accepts SQL sentences requests, and responds accordingly with rows returned or with some kind of acknowledgement depending on the command; in some cases that can be enough to access data; nevertheless, this causes many exchanges between the application and the database server, and also the program grows in complexity because of SQL sentences are included inside the code. As a consequence, a feature called stored procedures[6] embedded in most of DBMS's has to be used. Inside a stored procedure the programmer can write the necessary lines of code (SQL commands) to execute complex queries or even some business rules and save it as part of the database schema; after that, the application programmer accesses them from the code as if they were simple methods with their correspondent parameters.

Stored procedures can simplify complex database operations, whether standard SQL or a language like PL/SQL is used. Ultimately, by having SQL in a stored procedure is typically easier to maintain than SQL queries embedded in Java code. Concluding, any n-tier application with a moderate amount of data to be managed, can typically use JDBC API along with stored procedures in the back - end – tier as shown in the web service architecture diagram (Figure 3). Nevertheless, for a system with millions of users, new facets should be implemented in respect to database access. One of the most important characteristics nowadays is cache.

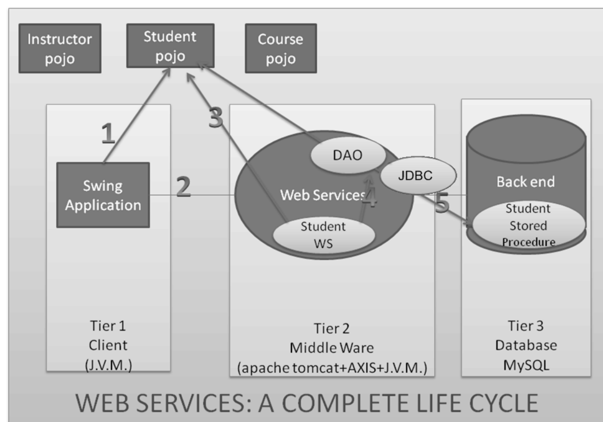

Figure 3: Web service architecture using JDBC+SP

[6] A stored procedure is a subroutine stored in the database dictionary that lest applications access a relational database system

In respect to performance aspects some level of cache should be implemented in any database connection API; consequently, one action taken in JDBC 4.0 specification in order to not overhead opening, initializing and closing of physical database connections for each client session; that is why connection pooling is supported. A connection pool solves the previous overhead problem by maintaining a cache of physical DB connections that are available to be reused across client sessions. This feature specially helps improve scalability and performance especially in three - tier environment through the implementation of ConnectionPoolDataSource, which is used by the application server to build and manage the connection pool [23]. In this way, JDBC avoids physical connections each time an application needs to access a database. But also data should be cache to avoid continuous read actions directly from the database to be used.

Data has to be cached by the application server in any n - tier architecture with the aim of improving the performance of a n Million-user system; furthermore, sometimes it is not just a matter of performance, it is a fact that data should be cached to let the system run in its basic functionality. For this reason, programmers have to think the way of liberating the database from fetching data from the physical database. Even though JDBC doesn't support data cache by itself, many frameworks have been developed to solve this problem, and one of them is LiveStore.

3. JPA

The EJB3 Java Persistence API (JPA) is standardizing the use of persistence in the Java platform; it provides standard mechanisms to using Object - Relational Mapping and caching tools. Moreover, annotations are used in EJB3 JPA to define objects, their attributes, their methods, Object - Relational Mappings and injection of persistent contexts. Besides, there is an EntityManager API ready to perform CRUD operations, and EJB - QL is designed to retrieve entities also. Concluding, annotations reduce the work of heavy coding and ensure a consistent mapping between entities (data base) and classes (application) as shown in figure 4. New IDE tools like Eclipse, Netbeans or JBuilder allow programmers to map directly from database by means of inheritance, delegation or other object oriented feature. The need to implement the code by hand should be discarded with the purpose of keeping consistency between the dbschema and the classes generated (POJOs).

Java class (Entity)

```
@Entity
public class Book{
 @Id
   private long bookID;
  @Column(name = "bookName")
  private String name;
  private String author;
  private String kind;
  private String numberOfPages;
 @ManyToOne
 private Shelf shelf;
}
```

Database Schema (Table)

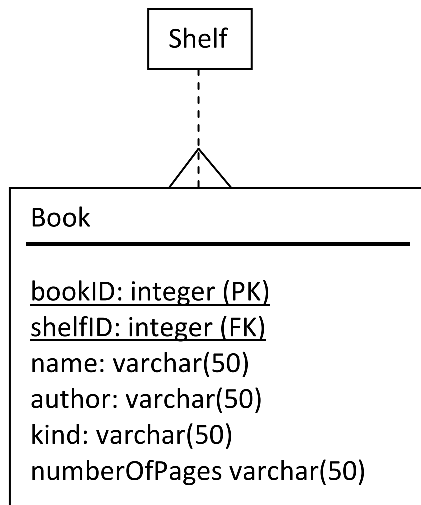

Figure 4: Use of annotations in a POJO file and the correspondent DB table

In the above example, four annotations are used: @*Entity*, which is required to identify the class Book as an 'entity' book when it is deployed; @*Id* which is to identify that *BookID* is the primary key of book; @Column(author="authorName"), which is used to change the name of the column from 'name' to 'bookName' when the table is created from the entity; and, @ManyToOne which is used to declare a one-to-many relationship to the entity (shelf 1 ←→ … books). Above and beyond, there are many annotations like @*table,* @*Transient,* @*EmbeddedID*, etc., that allow developers to map objects to relational data before executing CRUD operations by means of JPQL (Java Persistence Query Language).

EJB defines its own query language called JPQL, which supports entities along with bulk operations like delete and update. Besides, JPQL shares much in common with SQL (figure 7 and figure 8) with some key differences like: in JPQL, primary structures are entities and fields instead of tables and columns. Likewise, JPA implementations also have their own Query Language, for instance, Hibernate has HQL (Hibernate Query Language). The next line of code (figure 5) serves to retrieve PERSON objects using HQL; in contrast with JDBC code (figure 1), this is completely transparent to the Object Oriented Programmer; besides, it is saving time and resources, and solving the impedance mismatch problem along with the use of POJOs.

```
List all = s.createQuery ("from Person").list ();
```

Figure 5: Java code to retrieve all objects using Hibernate HQL [9]

In order to explain the way that this new architecture works, the "JPA, hibernate - TopLink, EJB 3 Quick Diagram" taken from "Bruno's Blog" [2] is shown. In this diagram it is clearly seen that JPA is used by the programmer to implement the components (EJBs) and the JPA is using Hibernate to access MySQL; in this sense, the programmer doesn't need to worry about the drivers and doesn't need to write JDBC code. Instead, the programmer can use JPQL with an Object Oriented approach by means of Object – Relational Mapping.

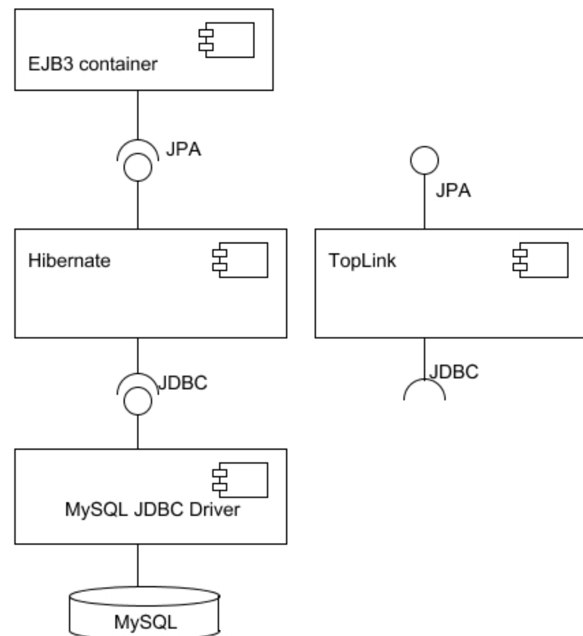

Figure 6: Source: "JPA, hibernate - TopLink, EJB 3 Quick Diagram" [2]

```
SELECT Object (P) FROM anydomain.Person P INNER JOIN P.bestFriend AS B
```

Figure 7: JPQL sentence [17]

```
"SELECT P.ID FROM PERSON P INNERJOIN PERSON B ON
B.ID = P.BESTFRIEND_ID" [17]
```

Figure 8: SQL sentence [17]

There are two ways to mapping relational data to objects in a proper manner and automatically by using IDE tools like Netbeans for example: One of them is developing the entities (classes) with the use of annotations and then building the dbschema (db script); the other approach is building the database first, and then by using reverse engineering, the entities can be generated and mapped through annotations; any way, consistence between Entity – Relational (E-R) and Object Oriented (OO) models is carried out in JPA because annotations track all kind of characteristics of a relational database to a class diagram (Figure 4). Furthermore, not only the mapping feature is present in JPA implementations, but also some performance improvement characteristics.

JPA implementations like Hibernate or TopLink are responsible for query optimization, in this way, the programmer just thinks about the fetching action, not in algorithms to improve queries. Besides, they can implement caching services by means of distributed caches like ehcache, memcached, cacheman or their own implementations; hence, the developer doesn't have to spend time in implementing cache services since they are easily coded with a few lines of programming or configuring when a session bean is being implemented.

The Persistence provider is key in cache management, especially for a session bean implementation (whether it is stateless or stateful). According to Enterprise JavaBeans Specification: EJB Core Contracts and Requirements [25]: A session bean that doesn't make use of the JPA should manage cached database data explicitly; it means, a session bean instance has to write cached database updates prior to any transaction completion, and it has to refresh any potentially stale database data copy at the beginning of the next transaction, plus a session bean has to refresh any java.sql Statement object before it is used in a new transaction context [25]. On the other hand, The use of the Java Persistence API by a session bean provides automatic management of database cache, "including the automatic flushing of ached database updates upon transaction commit" [25]. Resuming, the use of the Java Persistence API frees the programmer of some aspects related to data cache. In addition, and in order to facilitate the coding of business rules that depend on SQL statements execution with an object oriented approach inside the application server, the use of stored procedures is replaced by the use of namedQueries.

In spite of the fact that stored procedures can be called by using native queries (createNativeQuery("name_stored_procedure")), as for example in Hibernate (see figure 10), where a stored procedure (figure 9) is trying to be called, but it is not successfully running, returning an exception related to the return type and generates an exception (figure 11), some programmers still need to use some stored procedures to support legacy systems or simply some SP with logic inside; for this reason stored procedures is still supported by JPA implementations. TopLink implements SP calls by means of annotations like @NamedStoredProcedureQuery [20]. Concluding, the developer must manage all mapping result issues when using stored procedures, and they depend on the JPA implementation provider.

```
"CREATE OR REPLACE PROCEDURE HIBERNATETESTINSERT
(
  valStr IN VARCHAR2,
  valNum IN NUMBER,
  valDate IN DATE,
  valClob IN CLOB,
  result OUT VARCHAR2
) AS
--
-- NAME        :
-- DESCRIPTION :
-- AUTHOR      :

BEGIN
  insert into HIBERNATE_TEST values(valStr,
valNum, valDate, valClob);
  result := 'SUCCESS';
END;
 org.hibernate.HibernateException: Could not
extract result set metadata" [19]
```

Figure9: stored procedure: hibernatetestinsert [19]

```
"...
final String queryString = " call
hibernatetestinsert(?1, 5, null, null, ?2)";
Query query =
getEntityManager().createNativeQuery(queryString
);
String test = new String();
query.setParameter(1, "HelloParameter");
query.setParameter(2, test);
return query.getSingleResult();
..."[19]
```

Figure10: Hibernate implementation to call a SP using createNativeQuery [19]

```
"org.hibernate.HibernateException: Could not
extract result set metadata" [19]
```

Figure10: Exception message from SP call using createNativeQuery [19]

4. COMPARISON

In the following paragraphs, some ideas that contrast, evaluate and are common for both approaches (JPA or JDBC) are described. They pretend to give an overview

from the programmer and performance points of view, namely: cache, object – relational mapping, query optimizations, stored procedures support, Query Languages supported, efficiency in accessing data, ease of implementation (RAD), ease of data persistence, independence of JDBC driver, ease of connection, scalability and applicability.

Object – Relational Mapping: JPQL queries are executed by entity manager on persistence context, it is also independent of underlying schema and has nothing to do with the database that is used, which in this example is MySQL (see figure 6); whereas, in JDBC it is necessary to change queries format depending on the database that is being used and it is dependent on the schema; in spite of the fact that in JDBC 4.0, the driver discovery issue is implemented.

Cache: The implementation of data access through cache is done in the Java Persistence API, and the programmer doesn't worry about updating cache or similar activities related to caching, but if no Java Persistence API is used, i.e. pure JDBC is utilized to manage cache; the programmer should be concerned about all cache characteristics.

Query optimizations: The JPA implementations (e.g. Hibernate or TopLink) are responsible for query optimizations and can provide some caching services as a manner of aid to improve performance; XML configuration files and Java annotations provide the metadata used by the JPA implementation to generate and optimize the SQL sentences on the fly; whereas, in JDBC the developer has to implement hand coding to manage cache or to optimize SQL sentences.

Stored procedures support: Stored procedures do nothing to really help the application developer in relation to cache; they are procedure - based and cannot be cached (except for what the database server may provide in respect to pool and data cache management). Nevertheless, as well as in JDBC, stored procedures are also supported in the Java Persistence API. Above and beyond, in respect to stored procedures calls, JDBC manages SP calls transparently by means of ResultSets as a simple SQL statement execution; while in JPA, the developer needs to do some programming to deal with result set metadata, and object relational issues appear as a problem to solve inside JPA implementations.

Query Languages supported: JPA implementations as Hibernate, which uses HQL (figure 5), also supports SQL; in this way, JPA implementations give more flexibility to the programmer in building data access programs, by means of its own query language, and also supporting the classical Structured Query Language.

Efficiency in accessing data: JDBC only supports native SQL, then, the developer has to take care of maximizing the queries; while, in a JPA implementation a better query language is available to get more efficient database manipulation tasks, and all possible improvement is done inside Hibernate, TopLink, etc.

Ease of implementation (RAD): JPA relieves the programmer from manual handling of persistent data in CRUD operations, hence reducing the development time and maintenance cost, it is enough to provide the data base schema to generate automatically the most of the CRUD operations needed to develop the system.

Ease of data persistence: In JPA, JPQL queries may take bind parameters and results may be returned as entities or ordinary Java objects; whereas, in JDBC the developer should develop lengthy code to manage objects that are to be mapped from database, see figure 1 and figure 5, where it is easily seen that 49 lines using JDBC can be implemented by 1 line using Hibernate. The developer has to deal with the declaration of classes according to the tables in the Entity Relational model, and he/she must be careful about the programming of Ids, master-slave relationships, cascade implementations, one to many, many to many or one to one relationships; it means, all related to referential integrity. Additionally, annotations can help JDBC applications to improve data persistence.

Independence of JDBC driver: In JPA, the access information like database name, server, user and password are usually in a configuration file of Object – Relational Mapping (e.g.: *hibernateconfig.xml*), while in JDBC, the programmer has to write a bulk of code to establish and close the connection with database, and for every operation, the developer needs to call methods (*e.g.: getDBConnector(…)*). Although, in the new version of JDBC (4.0) automatic driver discovery is implemented.

Ease of connection: JPA scales very well in any environment, no matter if it is used in in-house Intranet that serves hundreds of users or for mission - critical applications that serve hundreds of thousands of users. JDBC cannot be scaled easily.

Scalability and applicability: For smaller applications, it could not be necessary to use JPA implementation, since it may increase complexity in applications that has single database or very few tables; in this case, the developer could prefer to go for JDBC plus stored procedures. In case of any small application that is expecting to grow in

business in future, the best option is JPA, in view of the fact that it affords more scalability. But, for Million - user applications, the best solution should be the use of the Java Persistence API since its support of cache, query optimization, scalability, among other characteristics.

Usage of the Java Persistence API with Hibernate/TopLink creates overhead for applications:

- That are simple and use one database that never change.
- That need to put data to database tables, no further SQL queries.
- Where there are no objects to be mapped to two different tables.

As a result, for these types of applications JDBC with stored procedures is the best choice.

The next table resumes comparison aspects between JPA implementations and JDBC:

Characteristic	JPA implementations	pure JDBC
Object - Relational Mapping	Supported (by using annotations)	The developer must do it (using annotations)
Caching	Has to be activated by the developer, and can be used inside code transparently.	Must be implemented by the developer.
Query optimization	Implemented by JPA implementations	The developer must do it
Ease of connection	XML configuration file	Driver discovery since JDBC 4.0, before that version, the developer has to write the code
Ease of data persistence	POJOs: getters and setters	Manually done, although POJOs can also be used
Annotations	Supported	Supported from JDBC 3.0
SQL	Supported	Supported
Precompiled SQL statements	preparedStatements	namedQueries
Query Languages	SQL and Object Oriented Query Languages: JPQL, HQL, etc.	SQL oriented: SQL
Return type	Returns entities	Returns rows
Relational concerns	Driven by annotations (POJOs)	The developer must implement it, can be done by simple classes and also by annotations (POJOs)
Database independent	Queries are executed by entity manager	Change queries depending on the schema
Scaling	Ease	Not ease
Stored procedures support	Can be implemented, but needs some programming skills	Transparently implemented and used as any other SQL statement.
Container needed	Yes	No

Table 1: comparison between JPA and JDBC

5. CONCLUSIONS

Current n-tier web applications are being developed using Java, some of them use only JDBC API for database access in which the developer has to write intensive code to implement the business logic along with SQL sentences (the use of stored procedures can help, but that code will remain on the database server), additionally, connection pooling and data caching should be programmed to optimize the JDBC performance. On the other hand, when using JPA implementations, the programmer writes less and more understandable lines of code to achieve the data persistence; besides, JPA also reduces the disk head movement and network resource consumption because ORM makes a binding for the first time when the data is mapped; furthermore, that binding will be utilized for next accesses.

For applications like a University Library System that needs large scalability and better performance, usage of JPA in the back end tier is preferable. For applications that don't need much scalability like a faculty department which has a few users and accesses, usage of JDBC API with stored procedures affords a decent performance, nonetheless, not advisable at all.

JPA is really geared towards application development. Like stored procedures, they abstract away the SQL from the developer. The abstraction however is more tuned to the application development methodology and this is where Object - Relational Mapping (O/RM) comes into play.

JDBC implements pool cache in order for the applications don't open connections to the database each time they run a query, while data cache is implemented by persistence providers (JPA) in order to improve performance by reducing accesses to the database data. When millions of users pretend to access the same row of a table they usually went to open a connection to the database and then the data was retrieved from the table, in this sense the database became a bottleneck for many applications; that is why data caching is of primary importance for new Web 2.0 applications; in this sense, the Java Persistence API offers many possibilities to increase performance by means of using many distributed cache frameworks.

In spite of the fact that both APIs share lots in common, they implement each feature in a different way, and the developer is the one who will decide on what API to use for the development of future applications; nevertheless, it is clearly seen that the Java Persistence API has many better features than JDBC to be considered. Above all, when an enterprise application is going to be developed and this application will be used by millions of users, the development team has to consider not only the ease of implementation, but also the performance, in this sense, JDBC lacks of many features that the programmer will have to take care of, e.g. caching. Concluding, the Java Persistence API is the best option to have an efficient system and to develop consistently and rapidly, as the new demands of the software market require.

In conclusion, the use of a persistence provider can save time to the developer in the whole implementation process (more productivity) and time to the user when accessing any application (maximum performance).

6. REFERENCES

[1] JPX Java Persistent Objects (2008). "*Comparison of JDO and JPA*", retrieved April 10, 2008, from www.jpox.org website: http://www.jpox.org/docs/1_2/jdovsjpa.html

[2] Bruno's blog (2008). "*JPA, Hibernate, EJB3, and TopLink quick diagram*", retrieved April 10, 2008, from http://brunovernay.wordpress.com website: http://brunovernay.wordpress.com/2008/01/22/jpa-hibernate-ejb3-and-toplink-quick-diagram/

[3] jblog (2007). "*Performance Comparison: JSP+JDBC vs. JSF+JPA*", retrieved April 10, 2008, from anydoby.com website: http://anydoby.com/jblog/article.htm?id=20

[4] TheServerSide.COM (2007). "JPA versus JDBC and Stored Procs", retrieved April 12, 2008, from www.theserverside.com website: http://www.theserverside.com/discussions/thread.tss?thread_id=45074

[5] Kodali R., and Wetherbee J. with Zadrozny P. "Beginning EJB 3 Application Development: From Novice to Professional", Berkeley, CA: Apress, 2006, pg. 86-87

[6] (2008)."Hibernate versus JDBC", retrieved April 12, 2008, from www.mindfiresolutions.com website: http://209.85.173.104/search?q=cache:m41hdKBJRgEJ:www.mindfiresolutions.com/download/Java%3DHibernate_JDBC.pdf+why+ORM+is+better+than+JDBC&hl=en&ct=clnk&cd=3&gl=us

[7] Hibernate (2008). "Java Persistence with Hibernate", retrieved April 12, 2008, from www.hibernate.org website: http://www.hibernate.org/397.html

[8] Sun Microsystems (2008). "Java Persistence API FAQ", retrieved April 10, 2008, from Java.sun.com website: http://Java.sun.com/Javaee/overview/faq/persistence.jsp

[9] Gash, J. "Relational Database servers", September 2007

[10] Freitag, P. (2005). "The MySQL Query Cache", Retrieved April 12, 2008, from : http://www.petefreitag.com/item/390.cfm

[11] Zawodny, J. and Balling D. "High Performance MySQL optimization, back up, implementation & load balancing", Sabastopol, CA: O Reilly Media, Inc., 2004, pg. 131

[12] Penchikala, S., O Reilly on Java.com: The independent source for Enterprise Java (2006), "JDBC 4.0 Enhancements in Java SE 6", Retrieved April 14, 2008, from www.onjava.com website: http://www.onjava.com/pub/a/onjava/2006/08/02/jjdbc-4-enhancements-in-java-se-6.html

[13] Luck G. 2007 JavaOne Conference (2007). "Distributed Caching Using the JCACHE API and ehcache, Including a case study in Wotif.com", Retrieved April 14, 2008, from developers.sun.com website: http://developers.sun.com/learning/javaoneonline/2007/pdf/TS-6175.pdf

[14] Kumar, A., Lascano, E., and Jayakumaran, S. "JPA implementations versus JDBC + Stored Procedures", 2008

[15] Keogh J. “Java demystified”, Emeryville, CA: McGraw-Hill/Osborne, 2004, pg. 257

[16] Burke B., Monson-Haefel, R. “Enterprise Java Beans 3.0”, Sebastopol, CA: O Reilly, 5th Edition, 2006.

[17] JPOX Java Persistent Objects (2008). “RDBMS JPQL Queries”, retrieved April 14, 2008, from www.jpox.org website: http://www.jpox.org/docs/1_2/rdbms/jpql.html

[18] Avaje, “JPA API issues”, Retrieved April 14, 2008, from www.avaje.org website: http://www.avaje.org/jpaapi.html

[19] My Eclipse infinite possibilities: MyEclipse forum (2007), “Stored Procedures and JPA”, Retrieved April 14, 2008, from www.myeclipseide.com website: http://www.myeclipseide.com/PNphpBB2-viewtopic-t-19196.html

[20] Burns, D., Regular Average Java Programmer RAJP (2008). “Why I choose O/R Mapping (JPA) over Stored Procedures for CRUD”, Retrieved April 20, 2008, from www.myeclipseide.com website: http://davidwburns.wordpress.com/2008/03/10/why-i-chose-or-mapping-jpa-over-stored-procedures-for-crud/

[21] Liang, D. “Rapid Java Application Development Using Sun ONE ™ Studio 4”, New Jersey: Prentice Hall, 2003,1st edition, pg. 514

[22] Keith, M., and Schincariol, M. “Pro EJB 3: Java Persistence API”, New York, NY: Apress, 2006, pg. 19

[23] Andersen L. “JDBC ™ 4.0 Specification JSR 221”, November 2007, pg. 19

[24] DeMichiel, L., and Keith, M. Sun Microsystems. “JSR 220: Enterprise Java Beans ™, version 3.0 Java Persistence API”, May 2006, pg. 15

[25] DeMichiel, L., and Keith, M. Sun Microsystems, “JSR 220: Enterprise Java Beans ™, version 3.0 EJB Core Contracts and Requirements”, May 2006, pg. 62