

Relational Database Persistence Practices

Draft - Work in Progress

By Alexandre Polozoff and Wayne Beaton

Persisting application data is a common application requirement; the ability to work with Java objects instead of raw data and to transparently save/retrieve application state and data. A variety of ways exist to implement persistence and selecting the best method is not always easily determined.

Various programmatic methodologies exist to persist data in a Java application. Java serialization is a generally accepted methodology for simple object persistence but is limited, provides no relational capabilities and entails a high price for the simplicity it provides. EJBs are good for the specific scenarios where they provide an advantage. Though in many cases simple JDBC/SQL persistence is not only more practical but also the simplest solution.

Relational databases provide fast access to data, handle read-write accessing issues and are also easily distributed. A number of programming solutions to access relational databases include Java's own JDBC, VisualAge's Persistence Builder and third-party mapping tools for both Java objects and EJB entity beans. But which solution is right for you? In this paper, we discuss the various options, present insight into when a particular solution is (and is not) appropriate, and present some best practices for object persistence.

Solution One: Direct JDBC

JDBC is the Java standard for accessing relational databases. JDBC provides Java programmers with a common API for accessing relational databases. It is the responsibility of the individual database vendors to provide JDBC drivers to abstract access to their particular implementation. In essence, JDBC provides a mechanism for delivering SQL statements to the relational database and tools for making use of the results of those statements.

One of the most common mistakes made by first time Java developers is the direct use of JDBC in servlets and other application code as shown (abstractly) below:

```
public void doGet(HttpServletRequest request, HttpServletResponse response) {
    PrintWriter out = response.getWriter();
    out.println("<html><body>");
    out.println("<ul>");
    generateList(out);
    out.println("</ul>");
    out.println("</body></html>");
}

private void generateList(PrintWriter out) throws SQLException {
    Connection connection = null;
    PreparedStatement statement = null;
    try {
        connection = getConnection();
        statement = connection.prepareStatement("SELECT first, last FROM EMP");
        ResultSet results = statement.executeQuery();
    }
```

```

        while (results.next()) {
            out.println("<li>");
            out.println(results.get(1)); // "first" name field
            out.println(results.get(2)); // "last" name field
        }
    } catch (SQLException e) {
        throw e;
    } finally {
        try { statement.close(); } catch (SQLException e) {}
        try { connection.close(); } catch (SQLException e) {}
    }

    private Connection getConnection() throws SQLException {
        getDataSource().getConnection(); // dataSource is populated from a JNDI lookup.
    }

```

It is not necessarily wrong to use JDBC directly in application code, but it is generally not an appropriate solution. As the application code grows in bulk, the JDBC code can get difficult to manage. Further, as the number of locations using JDBC in your code increases, management approaches a point where it is nearly impossible to maintain¹. The code presented above is reasonably well factored, but will likely still suffer over the long term as additional application complexity is added.

One obvious effect of using direct JDBC is that it precludes the use of JavaServer Pages (JSP). It is possible to include JDBC code in a JSP, but this practice is discouraged.

In a recent effort to migrate code from a competitive application server and Oracle to WebSphere Application Server and DB2, we encountered an application that made extensive use of direct JDBC calls. Very little modification was required to get the application to run in WebSphere; however extensive modifications were required to make the code work with DB2.

The application code made use of the classic JDBC DriverManager class to obtain connections and made extensive use of proprietary Oracle SQL extensions. None of the database code was centralized, so migration was a tiring mechanical effort of finding and modifying the offending code. Of course, a large amount of refactoring took place in this effort.

An interesting observation was made during the migration effort. It seems that, in the absence of platform support for connection pooling in the competitive application server, the client had actually built their own connection pooling mechanism. That mechanism was used in parts of the application, but not consistently. Had they at least refactored the way that connections are obtained, introduction of their connection pooling mechanism would have been relatively easy.

The complexity of direct JDBC is amplified when the database's proprietary extensions are employed. If you must make use of proprietary database extensions, consider hiding them behind an abstraction.

¹ In general, application code spends far more time in maintenance than in initial construction.

When is this appropriate

This style is appropriate if there is no real need for objects and the application itself is a very small, one-function tool. A reporting application is a good example of this.

When is this not appropriate

This style is not appropriate when objects are required. Additionally, this solution does not adhere to the Model-View-Controller (MVC) paradigm and closely links the data model to the servlet controller logic. As an application grows in scope and size this solution is not recommended.

Recommendation: Factor your code

If you must use the direct JDBC pattern, factor your code into the smallest pieces possible. Remember that factoring is more than an effort to break code into smaller bits, but is also an effort to avoid duplication, where possible, of application logic.

Obtaining the connection is one obvious piece of code that can be refactored from application code into a single reusable part.

Recommendation: Use DataSources

DataSources are the preferred way to access JDBC resources. DataSources provide connection pooling which is absolutely necessary in an enterprise environment. Connection pooling reduces the overhead of opening and closing a connection. Individual database connections require large amounts of memory and system resources. Opening and closing database connections are expensive (time and memory consuming) operations; a pool of open connections minimizes this overhead. For some database drivers, there are limitations beyond available memory on the number of connections allowed.

In the early days of JDBC, the DriverManager class was used to obtain connections (this mechanism is still available). When building applets and standalone applications, single connections are the norm. However, in the world of enterprise applications where a single system may require hundreds or thousands of connections, maintaining a single connection per process is unrealistic at best. Connection pooling via DataSources is absolutely required.

Connection pooling may or may not be available at the JDBC driver level. Some vendor implementations do not provide native connection pooling support. When utilizing a datasource, WebSphere Application Server provides connection pooling regardless of the native driver support. Adjustments to the number of connections in the pool is, again, configured in the WebSphere Application Server and does not require application code changes nor driver specific configuration knowledge.

DataSources are most commonly obtained through JNDI. This allows for name spaces configured in the WebSphere Application Server providing the ability to change the JDBC referenced database without application code changes. This facilitates administration of testing and staging environments.

Recommendation: Use Prepared Statements

Prepared Statements build SQL statements that the database caches and executes many times over without building the statement again and again. As long as the SQL statement does not change, and only the parameter values change, you will experience dramatic improvement of SQL execution. This requires that the SQL string is written as:

```
sql = "SELECT project, tasknumber, date FROM tasks WHERE project=?"
```

Utilizing the parameterized SQL statement allows for dynamic values and enables DB2 to cache the statement. In most DB2 implementations the prepared statement cache is system-wide meaning that any user on any connection has access to all prepared statements.

By using prepared statements, instead of stored procedures, you also avoid hiding business logic inside the database. Additionally, stored procedures present a problem when moving/upgrading a database if they break in the process. It is also difficult to modify a stored procedure if the business logic changes at some later date.

Additionally, on DB2 for OS/390 there is the concept of the *dynamic statement cache* that supports similar features to prepared statements as long as the dynamic statement executed is loaded with host variables. Using the dynamic statement cache should be done with a competent DBA at your side (see "DB2 Magazine" Quarter 2, 2001 edition.) More detail on the DB2 prepared statement cache is available in the "DB2 Administration and Programming" guide for your operating system.

For performance tuning of the prepared statements cache in WebSphere v3.5 see http://www-4.ibm.com/software/webservers/appserv/doc/v35/ae/infocenter/was/08010301_v35.html#b152

Best practice: Use Data Abstraction

Some databases provide useful extensions such as automatic sequence counters. Features such as these are not governed by any standards; their use has the effect of tightening your dependency on that database implementation and thereby making the code difficult to change should you need to swap databases in the future.

Lessen the impact of a particular extension on your code by hiding the use of an extension behind an abstraction. When done in a generic enough way your abstraction can change to adapt to a different database keeping the impact of that change isolated from the bulk of your code.

An issue observed with sequence number generation internal to the database is that, in most vendor implementations, the database does not reuse the number when an SQL error occurs during the insert of a row with an autogenerated column. Therefore, if 10 SQL inserts fail in succession a gap of 10 numbers is created. This may or may not be a side effect that is acceptable.

Best practice: Use Larger-Grained Objects

Triple negative in the following sentence. Any ideas on how to clean it up? -Alex

Not requiring an object instance to represent each row does not imply objects should not be used. In the reporting example, it is excessive to convert row data into individual object instances to generate a report. Especially when you consider that these object instances are short-lived imposing additional work on the garbage collector.

A large grained object implementation provides:

- Simplified code. The servlet/JSP does not contain JDBC code making it easier to read, extend and support.
- Better semantic meaning. By encapsulating JDBC code within an object, you apply domain-specific language, making the code easier to read, extend and support.
- Decoupling. The code is less dependent on the particular JDBC implementation.

Solution Two: Customized Object Mapping

Java is an object-oriented programming language and it is only natural to want to use objects. Objects need to be persisted somehow; this is the domain of a customized object mapping strategy.

Some might argue that this is the domain of EJB entity beans and container-managed persistence. Entity beans are not generally appropriate for small-grained objects. Even the various versions of the EJB specification indicate that so-called “dependent objects” must fill in the gaps left by entity beans. Dependent objects are defined by regular Java classes. No part of the EJB specification (up to version 1.1) makes any mention of how these dependent objects are made persistent.

A well-established best practice employs an EJB session bean as a Façade for the business logic of the application. The business logic is implemented as objects behind the Façade. An application using this pattern is immediately distributable (the Façade and everything behind it can run on a physically separate machine) and includes transactional support. Still, a mechanism is required to make these dependent objects persistent.

The notion of a broker is vital to this solution. The premise is that the broker governs the work of reading and writing object instances to/from the database. The broker presents a high-level interface to the outside world, encapsulating the gritty bits of JDBC code that turns rows into object instances and object instances into rows.

A business process interacts with the broker using a high-level interface as demonstrated by the following code segment.

```
EmployeeBroker broker = getEmployeeBroker();
Employee employee = broker.findEmployeeByName("Wayne", "Beaton");
employee.setSalary(employee.getSalary() + 25000.0);
broker.update(employee);
```

In the best of all worlds domain objects know nothing about the persistence mechanism nor about the broker. Of course, that's the best of all worlds and we live in a world of compromise. The trick is to understand the compromises and their consequences.

In this simple example the Employee object is a standard Java class that extends `Java.lang.Object`. Instances have standard get and set methods to modify their state. In short, there is nothing interesting about the implementation of the Employee object so far as

persistence is concerned. Even the primary key for the corresponding database table is not represented. However, to keep things relatively simple, it's easiest if the state of the object approximates the state of a row in the database table.

The broker is responsible for implementing object persistence, handling such tasks as reading, writing, updating and removing database rows representing instances of the classes it manages. The implementation of a broker can become complex. It is possible, for example, to build a broker that keeps track of all changes made to the object instances it manages; the broker can automatically decide when an object instance needs to be inserted or updated in the database. A more sophisticated broker can determine which object instances need to be removed from the database. However, added sophistication begets added complexity and future maintenance. Provide such sophistication when only absolutely necessary or a business requirement.

In the simplest case, the broker implements basic object-relational mappings. The Employee class has a corresponding EMPLOYEE table in the database. The broker's implementation of findEmployeeByName looks like the following:

```
public Employee findEmployeeByName(String first, String last) throws BrokerException {
    Connection connection = null;
    PreparedStatement statement = null;
    try {
        connection = getConnection();
        statement = connection
            .prepareStatement("Select id, first, last, dob, salary from EMPLOYEE where first=? and
last=?");
        statement.setString(1, first);
        statement.setString(2, last);
        ResultSet results = statement.executeQuery();
        if (!results.next()) throw new NotFoundException();
        return buildEmployeeFrom(results);
    } catch (SQLException e) {
        throw new BrokerException(e);
    } finally {
        closeStatement(statement);
        closeConnection(connection);
    }
}

private Employee buildEmployeeFrom(ResultSet results) throws SQLException {
    Employee employee = new Employee();
    employee.setId(results.getInt(1));
    employee.setFirst(results.getString(2));
    employee.setLast(results.getString(3));
    employee.setSalary(results.getDouble(4));
    employee.setDateOfBirth(new Date(results.getDate(5).getTime()));
    return employee;
}
```

The code uses JDBC to fetch row data matching the criteria specified in the method arguments. Data from the resulting row (ResultSet) is extracted and an instance of the Employee class is built. The findEmployeeByName(String, String) method gets the required resources and initiates the query. The buildEmployeeFrom(ResultSet) method is responsible for extracting a single employee instance from the results. One benefit of this factoring is that the code building the Employee instances is instantly reusable. Perhaps more important, if the Employee class (or

table) changes, knowledge of the shape of that class is isolated and can be easily modified (consider the impact of such a change if the employee building code was duplicated across a number of methods). By isolating this code, change is easier and can be made with confidence.

Additional “find” method can be added with relative simplicity:

```
public List findByLastName(String last) throws BrokerException {
    List all = new ArrayList();
    Connection connection = null;
    PreparedStatement statement = null;
    try {
        connection = getConnection();
        statement = connection
            .prepareStatement("Select id, first, last, dob, salary from EMPLOYEE where last=?");
        statement.setString(1, first);
        statement.setString(2, last);
        ResultSet results = statement.executeQuery();
        while (results.next()) {
            all.add(buildEmployeeFrom(results));
        }
        return all;
    } catch (SQLException e) {
        throw new BrokerException(e);
    } finally {
        closeStatement(statement);
        closeConnection(connection);
    }
}
```

The astute reader will observe that there is duplicated code in this example. It is difficult to make this code any simpler without utilizing metadata.

The corresponding write method is also relatively simple:

```
public void write(Employee employee) throws BrokerException {
    Connection connection = null;
    PreparedStatement statement = null;
    try {
        connection = getConnection();
        statement = connection.prepareStatement("insert into EMPLOYEE values(?, ?, ?, ?, ?)");
        statement.setInt(1, employee.getId());
        statement.setString(2, employee.getFirst());
        statement.setString(3, employee.getLast());
        statement.setDouble(4, employee.getSalary());
        statement.setDate(5, new java.sql.Date(employee.getDateOfBirth().getTime()));
        statement.execute();
    } catch (SQLException e) {
        throw new BrokerException(e);
    } finally {
        try { closeStatement(statement); } catch (Exception ce) { }
        try { closeConnection(connection); } catch (Exception ce) { }
    }
}
```

There are some interesting things happening in these methods that warrants discussion.

SQLExceptions must be caught and processed. Additionally, both the statement and the database connection must be closed (the ResultSet is automatically closed with the statement). When an SQLException occurs, we replaced it with a broker-specific exception and retain the original exception for logging and/or troubleshooting at a later date.

Additionally, the instance of Java.sql.Date produced from the ResultSet is converted into a Java.util.Date providing loose coupling between the raw data and the object instance's value.

Coupling is a measure of how much knowledge each part of your code has about the other parts. The broker is an abstraction to disguise the use of JDBC. To complete the abstraction we hide as many details of the implementation as possible. This includes masking JDBC-specific exceptions and types.

Why bother with providing loose coupling of the data? The main reason is to reduce the effect of change on the code or to the backend database. By disguising how the broker is implemented we reduce the effect of any future changes. This is important when you consider that as the complexity of your object model increases so does the broker. At some point it makes sense to replace the customized broker technology with a different solution, possibly EJB or some other third party solution. With loose coupling of the data the effect of such a change is minimized though not completely eliminated.

The broker is now extended as an example of a custom persistence mapping framework implementing relationships to reference an entity bean. The appropriate field is added to the Employee object class, and the broker method responsible for creating the instance of Employee handles converting the foreign key to an entity bean reference.

```
private Employee buildEmployeeFrom(ResultSet results)
throws SQLException, FinderException, RemoteException {
    Employee employee = new Employee();
    employee.setId(results.getInt(1));
    employee.setFirst(results.getString(2));
    employee.setLast(results.getString(3));
    employee.setSalary(results.getDouble(4));
    employee.setDateOfBirth(new Date(results.getDate(5).getTime()));
    employee.setProject(getProjectHome().findByld(results.getInt(6)));
    return employee;
}
```

When writing an Employee instance back to the database, the process is reversed:

```
public void write(Employee employee) throws BrokerException, RemoteException {
    Connection connection = null;
    PreparedStatement statement = null;
    try {
        connection = getConnection();
        statement = connection.prepareStatement("insert into EMPLOYEE values(?, ?, ?, ?, ?, ?)");
        statement.setInt(1, employee.getId());
        statement.setString(2, employee.getFirst());
        statement.setString(3, employee.getLast());
        statement.setDouble(4, employee.getSalary());
        statement.setDate(5, new Java.sql.Date(employee.getDateOfBirth().getTime()));
        statement.setInt(6, getProject().getId());
    }
```

```

        statement.execute();
    } catch (SQLException e) {
        throw new BrokerException();
    } finally {
        closeStatement(statement);
        closeConnection(connection);
    }
}

```

The broker is just as easily used by an EJB (entity or session bean). A Project bean, for example, uses the broker directly to find all dependent objects:

```

    public List getStaff() throws RemoteException, BrokerException {
        getEmployeeBroker.findAllEmployeesOnProject(getId());
    }

```

Of course these examples represent a simple broker implementation, EJB and dependent objects.

The broker implementation evolves as the needs arise. The simple broker presented here does little more than convert row data to object instances and object instances back to row data. Adding relationships to the object instances calls for, but does not require, a caching mechanism. A well designed caching mechanism assists with how and when objects are read from the database and prevents multiple representations of the same object instance in memory. The caching framework tracks changes to object instances and decides which instances need persisting. Without caching the business process layer determines when the instance is persisted.

A full discussion of building customized brokers is out of the scope of this discussion. However, if you do choose the route of building a custom framework keep things simple. Unnecessary sophistication is a maintenance issue.

When is this appropriate

A custom persistence mapping framework is appropriate where object mapping requirements are relatively simple. This framework is scalable to handle about any requirement. This process runs dangerously close to implementing the functionality of a number of commercially available products that provide OR mapping persistence.

A prudent approach is to implement a custom framework and work "consideration" milestones into the development plan. At each milestone, review the framework and determine if the complexity warrants replacing the framework with a commercial product.

When is this not appropriate

When you have a large number of short-lived data-hold objects, you might consider a Direct JDBC type solution.

As the complexity of this solution increases, you might consider purchasing a commercially available framework.

Best Practice: Build only what you need

This best practice actually has very little to do with persistence; it is a generally applicable practice.

It is possible, quite easy in fact, to build very sophisticated brokers that provide sophisticated functionality. There are dangers associated with such an approach. Pragmatically, as your broker becomes more sophisticated, it also becomes more complex. As the complexity increases, so does the likelihood that it will break. Complexity is also a maintenance issue.

The approach to use when building a custom persistence mapping framework is to build what you need. As the requirements become more complex, the framework can evolve. In the end, you build a framework that is only as complicated as it needs to be and generally easier to maintain.

As the complexity of your custom solution increases, consider purchasing a commercially available solution. A number of commercially-available solutions are available, including Persistence Builder (included with VisualAge for Java), TopLink and CocoBase. In the long run, adoption of a commercially supported product is less expensive than supporting a custom solution. Remember that for each broker you need to write module code AND unit test cases and everyone knows 50% of the work easily goes to writing unit test cases.

Best Practice: Maximize Decoupling

This best practice also has little to do with persistence and is generally useful.

As a general rule, different parts of the application should know as little as possible about the other parts. Individual methods should know as little as possible about other methods in the same type. A good example of this is the process of obtaining a connection to the database.

In the methods presented here, a connection is obtained using the `getConnection()` method. The manner in which this method is implemented is not relevant to the users of the method. An initial implementation of the broker implements the method as follows:

```
private Connection getConnection() throws SQLException {
    DriverManager.registerDriver(new jdbc.idbDriver());
    return DriverManager.getConnection("jdbc:ldb:c:/ldb/test/ns.prp");
}
```

As the implementation matures, the hard-coded values are replaced with customizable values. Ultimately however, this method should utilize connection pooling:

```
private Connection getConnection() throws SQLException {
    getDataSource().getConnection();
}
```

The point is that the users of the `getConnection` method have no specific knowledge of how the connection is acquired.

Another good example of where decoupling applies is in the construction of SQL statements. In the `find` method shown above, the SQL `SELECT` statement explicitly lists the names of the columns to read. As the number of `find` methods increases, so does the use of such statements; as the use of these statements increases, the consequences of a change to the database table or object model has far-reaching effects on the code (the `SELECT` statements have to be changed). At very least the column names should be provided by either constants or a separate method.

Best Practice: Re-throw SQL Exceptions as Broker-specific exceptions

The purpose of the broker is to shield application code from the specific implementation details of the persistence layer. SQLException is a specific exception thrown by JDBC, and as such should be considered a detail of a JDBC-based solution. Hiding of exceptions specific to the back end layers allows the application code to catch only one type of exception instead of several types. This further simplifies the application logic by not having to catch three or four unrelated and disparate exceptions..

Burying the original exception allows the servlet controller logic to log why it decided to send the user to a particular page. Simply reporting that a BrokerException occurred is pretty useless without the root cause problem. In a high volume, multi-node, multi-clone web site trying to correlate logged errors with previously logged errors may be next to impossible. Therefore, when the servlet sends the user to an error page it logs precisely the root cause error that occurred..

Best Practice: Do the Math

The problem with customized persistence mapping solutions is that they only become more complicated. For simple needs a customized solution works very well and as those needs grow more complicated the cost of maintaining and extending a customized solution grows in proportion. Extending a customized persistence solution is multiplied by the number of Brokers developed. Each change affecting the Brokers has to be implemented, tested and deployed.

How complex is complex? A good question that is hard to quantify. Complexity depends on many factors, not the least of which is how experienced the development team is in building persistence frameworks. Here are some examples of complexity:

- Maintaining object identity;
- Complex relationships;
- Many-to-many relationships;
- Write ordering/stale data/referential integrity issues;
- Query framework

None of these features are particularly difficult to implement, but add complexity to your solution that must be architected, programmed and maintained over the entire life of the application. Keep in mind that your best and brightest people will likely move on to the next project when this application is in maintenance.

This is where the math comes in. As new complexity is introduced, you have to calculate the cost of implementing and supporting that complexity. The point at which a third-party product becomes more cost effective comes very quickly.

Keep in mind that you will underestimate the amount of complexity required and therefore how much time building the custom solution will take.

Solution: Third Party Persistence Mapping Tools

Third party solutions exist with both commercial and open source solutions.

Toplink - <http://www.webgain.com/products/toplink/>

CocoBase - <http://www.thoughtinc.com>

Open Source Efforts - a number exist though no formal testing has been conducted. See ObjectBridge at <http://sfads.osdn.com/1.html> and Castor at <http://www.castor.org> and COBRA at <http://www.kimble.easynet.co.uk/cobra/index.htm>

Solution: EJB Entity Beans

EJB Entity Beans are a good solution when used in conjunction with EJB Session Beans for scenarios that involve multi-phased transactions, multiple datasources or a high transaction volume.

Drawbacks to using the EJB solution include the serialization of objects even if utilized only locally on the same server.

Additional References:

Sun has started JDO (Java Data Objects)

http://Java.sun.com/aboutJava/communityprocess/jsr/jsr_012_dataobj.html

The OMDG has participated in the JDO persistence effort, see <http://www.odmg.org/>