Implications of Java Data Objects for Database Application Architectures

Craig E. Ward
CMSI 698 SS:Advanced Topics in Database Systems
Loyola Marymount University, Spring 2004

Abstract

This paper surveys the various technology choices regarding database connectivity using Java. After providing an overview of these technologies, summaries of previous research into efficiency issues of implementations and architectures are discussed. Conclusions regarding the balance between various architecture choices are made.

1 Introduction

The set of technologies comprising the Java programming environment provides multiple approaches to programmatic access to database systems. The number of possible choices and how they may be combined increases year-to-year. All of these approaches cannot be equally appropriate for all database systems. A full-scale, enterprise-encompassing system would be too much for a simple, low volume database while a simple application could become unacceptably slow or fail under the stresses of high volume or multiple accesses. Choosing an approach depends on the requirements of the particular database system in question.

Java technologies allow for relatively simple, direct access to the database using the *Java Database Connectivity* (JDBC) API either by direct use of the API or the direct embedding of SQL code using the SQLJ wrapper. The *Enterprise Java Beans* (EJB) of *Java 2 Enterprise Edition* (J2EE) and the more recent *Java Data Objects* (JDO) provide the ability to create *n*-tiered architectures. There are also two kinds of EJBs. *Entity* beans represent objects or records that need to persist in a database and *Session* beans that only exist during the life cycle of one transaction or session of an application. Entity beans also have a choice of methods for persisting records to the database.

This paper provides an overview of the Java technologies for web applications and database access. It next reports on the experiences of other researchers concerning the

performance of these technologies. A particular emphasis for the paper is how the newer JDO changes the balance between direct JDBC calls and a tiered EJB approach.

2 Motivation

The creation of the World Wide Web has added the problem of incorporating a database system into multi-tiered applications. Where at one time a user might use an application that ran on the same physical machine as the database itself, users now must be able to run applications and perform useful work with the applications and databases separated, sometimes by great geographic distance. Applications may run the gauntlet of a low volume of transactions, such as a library providing access to its card catalog, to high volume direct sales engines such as Amazon or eBay. Java technologies provide an excellent means for creating web applications in these ranges.

System designers in the Java realm can choose between Java *servlets* and *Java Server Pages* (JSP) for the front end of an application. A servlet is a program that runs in a container as part of an application server. Similarly, a JSP is also a servlet. The distinction comes from how they are written. A servlet could be thought of as having HTML embedded in Java code while a JSP is HTML with embedded Java code.

Servlets and JSPs can use the JDBC API directly or they can access a back end of EJBs. The designer must then choose between *Container Managed Persistence* (CMP) or *Bean Managed Persistence* (BMP) for any *Entity* EJBs. Back end containers that support the JDO API are now becoming available and provide another option. The tradeoffs and balances will be examined after each of the technologies is reviewed.

The growing number of combinations of these technologies presents an interesting problem. Which technology or combination of technologies is best for any particular application?

3 Background Work

JDO technology is a relatively recent addition to the "Java suite." Most of the material currently available focuses on the balances between using JDBC directly or EJBs in a container. The issue of CMP vs. BMP for any EJBs has also been researched. How the market will respond to JDO technology is still an open question.

(Eisenberg and Melton, 1998) describe the SQLJ standard and illustrate how it can be used to embed SQL code directly into Java code. SQLJ is a wrapper around JDBC and is similar to other embedded SQL systems.

(Salo and Hill, 2000) provide a comprehensive comparison of using EJBs in various combinations of servlets and JSP pages. (Tost, 2000) discusses the benefits of using Java Beans (an embeddable object not related to the later EJB standard) and EJBs together.

The performance and scalability of J2EE applications is addressed in (Cecchet, Marguerite, and Zwaenepoel, 2002) and (Gorton and Liu, 2003). The former implements an e-commerce application using different architectures comprised of Entity beans and Session beans with BMP and CMP. The latter focuses on a similar issue but reports on experiments using several different container systems.

One of the papers explicitly addresses JDO. In (Baldwin, 2003), the issue hinges on how well these technologies (plus some non-Java specific ones) handle the extremely large data sets at the National Climate Data Center.

The other references serve as background for the JDO technology.

The goal of this research paper will be to synthesize the established results regarding direct access using JDBC in servlets or JSPs versus using the tiered architectures possible with EJBs with the new developments possible with JDO.

4 Technologies Overview

This section provides an overview of the various Java technologies available for database access.

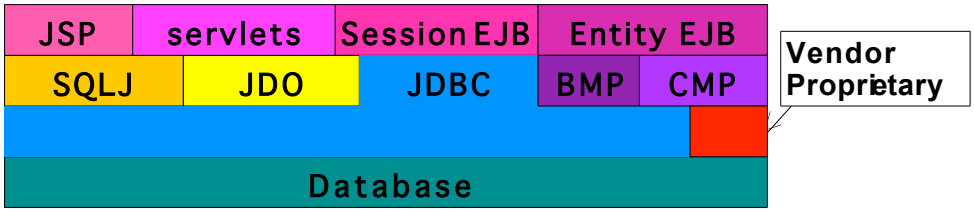

Figure 1. Relationships of Java technologies and database access. (Standalone applications and applets are not shown.)

4.1 JDBC

Java Database Connectivity is the Java base standard interface for accessing databases. JDBC encapsulates and hides the details of finding and connecting to a database. It also shields the Java code from specific vendor database implementations.

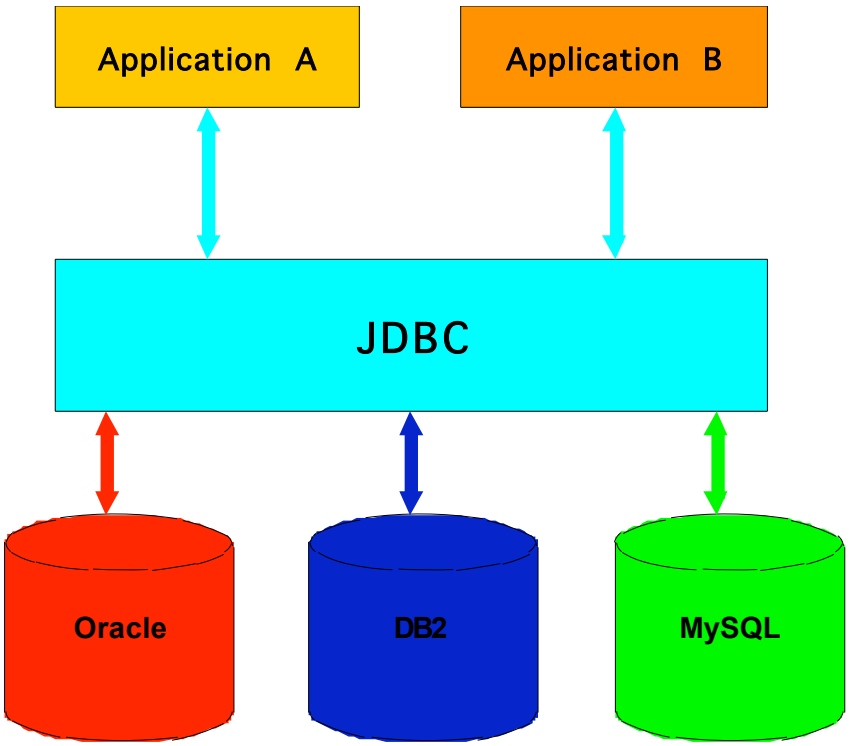

Figure 2: Role of JDBC as interface between applications and vendor database engines.

There are four types of JDBC drivers. Type 1 drivers are "bridging technology" and require another layer, such as ODBC, to do the actual access. Type 2 drivers call a host native API, such as an API written for C programs, to access the database as if they were written in the native language. Type 3 drivers use a network API to connect to another middleware product that performs the access work. Type 4 drivers, or pure Java drivers, use Java sockets to talk directly to the database engines. These will be the most efficient drivers.

The main classes of JDBC that handle connections, statements and data are Connection, Statement, and ResultSet, respectively. In the background are a Driver class and a DriverManager class.

The DriverManager class maintains a list of registered Driver classes. An application can create a new connection by sending the DriverManager a connection string. The following example illustrates (adapted from Reese, 2000):

```
Connection connection = null;
      try { // load the driver
       Class.forName(args[0]).newInstance();
     } catch( Exception e ) { // problem loading driver, class not exist?
       e.printStackTrace();
       return;
     }
     try {
       connection = DriverManager.getConnection(args[1],args[2],args[3]);
     } catch( SQLException e ) {
       e.printStackTrace();
     } finally {
       if( connection != null ) {
           try { connection.close(); }
           catch( SQLException e ) {
             e.printStackTrace();
           }
       }
     }
```

When the JDBC driver class is loaded (Class.forName()), it registers itself with the DriverManager so that it may be returned by the getConnection() method. Any error or problem encountered by the JDBC driver results in an SQLException.

The Statement classes are used to query and modify data. SQL code can either be "hard coded" into the creation of the statement or can be dynamically built. Data is returned from a query in a ResultSet. The following example method illustrates a PreparedStatement and the processing of a ResultSet.

```
public void printAccounts(String name) throws SQLException
    {
        DataSource ds  = null;
        Connection conn = null;
        PreparedStatement ps = null;
        ResultSet rs = null;

        try {
            ds = this.getDataSource();
            conn = ds.getConnection();
            ps = conn.prepareStatement("select account,ballance from accounts
                                       where customer_name = ?");
            ps.setString(1,name);
            rs = ps.executeQuery();

            while (rs.next()) {
                System.out.println (rs.getString("account") + "\t" +
                                    rs.getFloat("ballance"));
            }
        } catch (Exception e) {
            System.out.println(e.getMessage());
        } finally {
            if(rs!=null) rs.close();
            if(ps!=null) ps.close();
            if(conn!=null) conn.close();
        }
    }
```

The example illustrates the some of the methods of available from a PreparedStatement and ResultSet objects and how JDBC can process multiple rows of data.

JDBC is a useful interface for accessing data in a relational database. It allows the developer to ignore the details of connections and allows more focus on the needs of the application.

4.2 SQLJ

SQLJ is a call language interface (CLI) and follows a traditional pattern allowing for SQL code to be embedded directly into Java code. A pre-processor is used to translate the SQL into the corresponding JDBC calls. Although SQLJ is not an official Java standard, it is part of a suite of standards that allow database applications to use Java both in the user interface programs and within the database engine itself[1] and has been incorporated into the database products of some major database vendors. Below is a simple example:[2]

```
try {
#sql { DELETE
       FROM employee
       WHERE emp_id = 17
     };
}
catch (SQLException sqe) {
    System.out.println
    (sqe.getMessage());
}
```

The "`#sql`" indicates to the preprocessor that SQLJ code is beginning. Everything between the following matching braces will be translated into the corresponding JDBC code. The "`#sql`" serves the same purpose as "EXEC SQL" in the CLI for C. Note that the code catches a SQL exception just as in the JDBC examples.

SQLJ is designed to allow static SQL statements to be inserted directly into Java code. If dynamic SQL is needed, then direct JDBC is preferred (Eisenberg and Melton, 1998).

The details of how the database connection is established and maintained are encapsulated in connection context and execution context classes. These may be either

[1] Writing database stored procedures in Java is beyond the scope of this paper.
[2] SQLJ Code examples are from (Eisenberg and Melton,1998).

implicit or explicit and can be shared. The next SQLJ example illustrates the use of an explicit connection context:

```
#sql context EmpContext;
String url
   = "jdbc:sybase:Tds:localhost:2638";
EmpContext empCtxt =
    new EmpContext(url, "dba",
                   "sql", false);
#sql [empCtxt] { DELETE
                 FROM employee
                 WHERE emp_id = 17
               };
```

In this example, the connection context is referenced between the square brackets before the braces.

SQLJ also has the ability to access Java variables and can create iterators for processing sets of data by rows. The iterators may bind by name or by column number. This next example illustrates the use of a named iterator:

```
#sql iterator Employee
               (int emp_id,
                String emp_Iname,
                java.sql.Date start date
               );
Employee emp;
#sql emp = { SELECT emp_iname, emp_id, start_date
             FROM employee
             WHERE emp_fname LIKE 'C%'
           };
while (emp.next()) {
   System.out.println
          (emp.start date() + ", "
           + emp.emp id() + ", "
           + emp.emp iname() .trim()
          );
}
emp.close ( ) ;
```

Vendors that implement SQLJ can do a lot to tailor the resulting JDBC code to enhance its performance without making the code non-portable. A vendor-specific customizer can enhance the generated code to allow it to access special vendor features when it runs on the vendor's platform. On other platforms, the standard JDBC directed code would run just as before.

SQLJ provides an interesting approach to database connectivity. While SQLJ is not considered "object oriented," it can still have benefits. The use of SQL directly in code could increase the level of communication between a Java developer and a DBA. Such an enhancement to communication could make the difference between a failed project and a successful project.

4.3 JDO

Java Data Objects is a alternative to JDBC and SQLJ. In addition to encapsulating the details of connecting to a database, it also allows for the developer to see the data as Java objects. It could be thought of as a an object-oriented wrapper around JDBC.

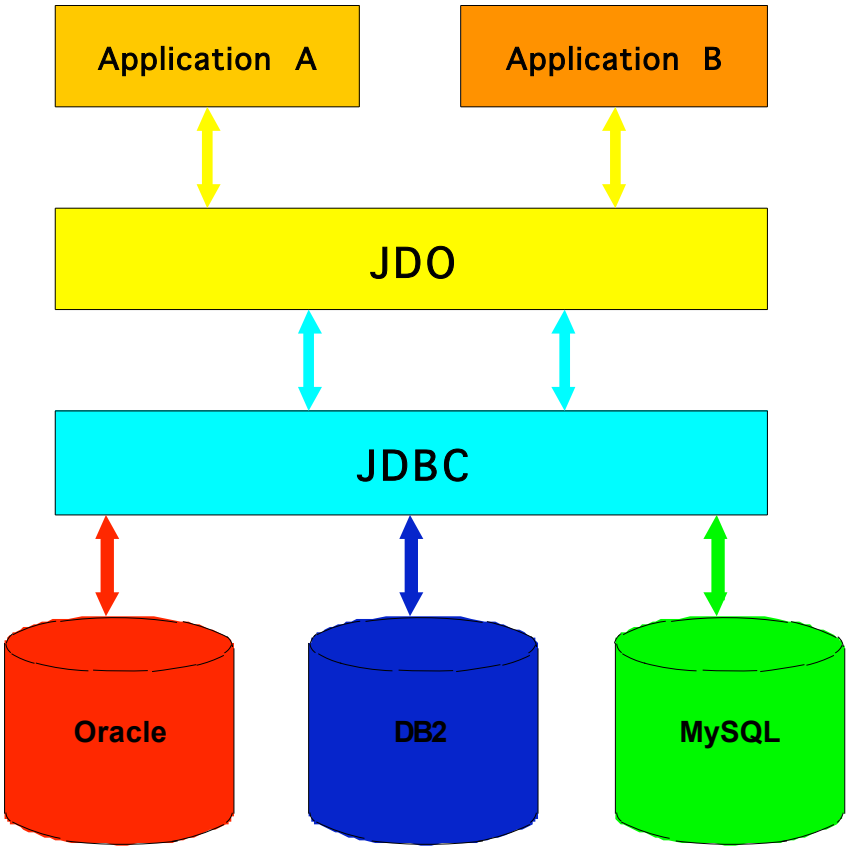

Figure 3: The relationship of JDO to JDBC and applications.

A Java application that uses JDO for database access does not need to explicitly setup connections or explicitly give the commands to persist an option in the database. Classes need only follow a few standard conventions to be able to make use of JDO. The fields of an object must be accessible to the JDO classes. This is basically the definition of a Java Bean. All fields of the class are accessible using setters and getters. Finally, the fields must be compatible with a JDO-supported data type. (JDO does not support unserializable types like Thread or Socket.)

A class that will be getting or putting objects into a database uses a PersistenceManager. The PersistenceManager class is supplied by the JDO vendor. This

class handles all of the interaction with JDBC. The following example illustrates how an application might use a JDO persistence manager (adapted from Almaer, 2002):

```
package addressbook;
import java.util.*;
import javax.jdo.*;

public class PersonPersist
{
      private final static int SIZE         = 3;
      private PersistenceManagerFactory pmf = null;
      private PersistenceManager pm         = null;
      private Transaction transaction       = null;

      private Person[] people;              // Array of people which we will persist
      private Vector id = new Vector(SIZE); // Vector of current object identifiers

      public PersonPersist() {
       System.out.println("Initializing JDO PersistenceManagerFactory.....");

       try {
              // -- these properties can be changed to work with other databases
          Properties props = new Properties();
        props.setProperty("javax.jdo.PersistenceManagerFactoryClass",
                          "com.prismt.j2ee.jdo.PersistenceManagerFactoryImpl");

          // -- HSQL DB
          props.setProperty("javax.jdo.option.ConnectionUserName", "sa");
          props.setProperty("javax.jdo.option.ConnectionPassword", "");
          props.setProperty("javax.jdo.option.ConnectionURL",
                            "jdbc:hsqldb:hsql://localhost");
          props.setProperty("javax.jdo.option.ConnectionDriverName",
                            "org.hsqldb.jdbcDriver");
          pmf = JDOHelper.getPersistenceManagerFactory(props);
       } catch(Exception ex) {
          ex.printStackTrace();
          System.exit(1);
       }
      }

      public static void main(String[] args) {
       System.out.println("Create PersonPersist");
       PersonPersist personPersist = new PersonPersist();

       System.out.println("Setup and persist a group of people");
       personPersist.persistPeople();

       System.out.println("Display the people that we persisted");
       personPersist.display(SIZE);

       System.out.println("Change a name, and then view the people again");
       personPersist.change();
       personPersist.display(SIZE);

       System.out.println("Delete a person, and then view the people again");
       personPersist.delete();
       personPersist.display(SIZE - 1);
      }

      public void persistPeople() {
             System.out.println("Creating three people to persist");

       // create an array of Person's
          people = new Person[SIZE];

          people[0] = new Person("Gary Segal", "123 Foobar Lane", "123-123-1234",
                                 "gary@segal.com", "(608) 294-0192", "(608) 029-4059");
          people[1] = new Person("Michael Owen", "222 Bazza Lane, Liverpool, MN",
                                 "111-222-3333", "michael@owen.com",
                                 "(720) 111-2222", "(303) 222-3333");
```

```
        people[2] = new Person("Roy Keane", "222 Trafford Ave, Manchester, MN",
                               "234-235-3830", "roy@keane.com", "(720) 940-9049",
                               "(303) 309-7599)");
      System.out.println("Persisting people");
      pm = pmf.getPersistenceManager();
      transaction = pm.currentTransaction();
      transaction.begin();
      // make all of the objects in the graph persistent
      pm.makePersistentAll(people);
      transaction.commit();
      System.out.println("Finished persisting objects.");

      // retrieve the object ids for the persisted objects
      for(int i = 0; i < people.length; i++) {
         id.add(pm.getObjectId(people[i]));
         System.out.println("Object id is: " + id.elementAt(i));
      }

      // close current persistence manager to ensure that objects are
      // read from the datastore rather than the persistence manager's memory cache.
      pm.close();
     }
```

Like SQLJ, JDO uses an extra step before an application using JDO can run. A JDO implementation provides an enhancer that adds the necessary extra code to allow the application to access the database. The input into the enhancer is an XML file that maps the object to the database. The enhancer can either modify the Java code before it is compiled or it may modify the byte code that a Java compiler generates. Most vendors choose the latter. Below is an example of an XML file configuring an object for use by JDO:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "file:/c:/Apps/Java/OpenFusionJDO/xml/schema/jdo.dtd">
<jdo>
   <package name="addressbook">
      <class name="Person" identity-type="datastore">
      </class>
   </package>
</jdo>
```

A distinction of JDO is that the objects that are persisted in a database do not need to correspond to a single row of a relational database table. The same JDO methods and services will work with object-oriented databases as well as relational or object-relational. JDO does not require that Java objects be decomposed for insertion into a database. Both JDBC and SQLJ follow a relational paradigm and are therefore more bound to that type of database.

4.4 Java Servlets

Servlets are small programs that run in a container as part of a server. This is in contrast to the applet, which runs on a client, usually a web browser with Java enabled.

A common use of servlets is to provide interactive, dynamic web applications. Servlets are one of the responses to scalability problems with Common Gateway Interface (CGI), an early technology used for dynamic web applications.

The distinction between a CGI program and a servlet is that CGI programs run in their own address space from the server. That is, the server must create a new process to run the CGI program. Creating such processes is expensive. For sites that must handle large numbers of requests, this expense is prohibitive. Because servlets run in a container, they do not require a separate process and can share the same address space and resources as the server. Below is a sample of a simple servlet:

```
package first;

import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;
import java.util.*;

public class MyServlet extends HttpServlet
{
    public void doGet (HttpServletRequest request, HttpServletResponse response)
            throws IOException
    {
            response.setContentType("text/html");
            PrintWriter out = response.getWriter();
            out.println("<h1>Craig's First Servlet</h1>");
            out.println("<p>hello!</p>");
            out.println(new Date());

    }
}
```

Servlets extend the HttpServlet class to be able to communicate with the container. The HTML code that is sent back to a browser is coded into calls to a PrinterWriter object.

XML is used to configure servlets for a web server. The XML properties are used to map locations to URLs and provide the configuration data needed by servlets to access resources such as JDBC drivers and connection strings for those drivers. Below is an example of one such XML configuration:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
 <display-name>JBlog</display-name>
    <description>
      Simple Servlet Web log Example
    </description>
    <context-param>
      <param-name>webmaster</param-name>
      <param-value>cewcew@mac.com</param-value>
      <description>
        The EMAIL address of the administrator to whom questions
```

```
        and comments about this application should be addressed.
      </description>
    </context-param>
    <servlet>
       <servlet-name>DBtest</servlet-name>
       <description>Testing Connection to MySQL</description>
       <servlet-class>DBtest</servlet-class>
    </servlet>
    <servlet>
      <servlet-name>Write</servlet-name>
      <description>
      Write web log data
      </description>
      <servlet-class>jblog.Write</servlet-class>
    </servlet>
    <servlet>
      <servlet-name>Show</servlet-name>
      <description>
      Show Web log
      </description>
      <servlet-class>jblog.Show</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>DBtest</servlet-name>
        <url-pattern>/DBtest</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>Write</servlet-name>
        <url-pattern>/wa/Write</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>Show</servlet-name>
        <url-pattern>/wa/Show</url-pattern>
    </servlet-mapping>
    <error-page>
       <exception-type>java.lang.IllegalArgumentException</exception-type>
              <location>/MyGlobalErrorPage.jsp</location>
    </error-page>
</web-app>
```

Servlets are a useful tool for developing web applications. They may or may not need database access. For applications that do need such access, basically any of the access layers can be used.

4.5 Java Server Pages

Java Server Pages are an alternate way to write servlets. As shown above, in a Java servlet, the HTML that is sent to a client for rendering is encoded in the calls to a method of a PrinterWriter object, for a JSP, the HTML is simply written into the file and the Java code is encapsulated in special tags.

This alternative is often cited as a way to allow non-programmers to develop servlets. The Java coding can be completely encapsulated in special purpose tag libraries allowing for Java code to be added without the JSP writer seeing any of the syntax of Java. Below is a simple JSP example that would display the current date:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
       "http://www.w3.org/TR/1999/REC-html401-19991224/loose.dtd">
```

```
<html>
<head>
    <title>My First JSP</title>
    <meta name="generator" content="BBEdit 6.5.2">
</head>
<body>
<h1>My First JSP</h1>
<p>Hello!</p>
<h2>Scriptlet Tag</h2>
<p>
Can contain one or more valid java statements separated by semi-colons. You can use
jsp "implicit objects," i.e. built-in, global objects that have been pre-initialized
and instantiated for you.
</p>
<%
out.println( new java.util.Date() );
%>
<h2>Expression Tag</h2>
<p>Saves you from typing out.println.</p>
<%= new Date() %>
<h2>Page Directive</h2>
<p>
Usually at the top.
</p>
<%@ page import="java.util.*"%>
</body>
</html>
```

The distinction between Java servlets and JSPs is in how they are created. From the view point of the container, they are all servlets.

4.6 Enterprise Java Beans

Enterprise Java Beans are part of the *Java 2 Enterprise Edition* (J2EE) specification. The intent of J2EE to provide a platform for building large scale, efficient, distributed applications. J2EE is very complex. (EJBs are not to be confused with the older specification of Java Beans. Simple Java Beans only need to provide accessor and mutator methods for their attributes.)

The types of EJBs of concern here are Session and Entity EJBs.

Session beans are designed to provide non-persistent, temporary services while Entity beans are designed to represent objects in a database. It takes three Java source code files to create one bean. Two of the files define interface methods that the container will implement and the third provides the actual bean logic.

EJBs declare a Home interface which provides life cycle methods for the bean. The Remote interface provides methods that provide the services of the bean. The final class is a class that implements the services of the bean. Below is a simple example. Note the correspondence between the methods of the Remote interface and the implementation.

Home interface:

```
package J2EEApp;

import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import java.rmi.RemoteException;

public interface BankMgrHome extends EJBHome
{
    public BankMgr create() throws CreateException, RemoteException;
} // end class
```

Remote Interface:

```
package J2EEApp;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface BankMgr extends EJBObject
{
    // Returns UserID after validating Username + Password in database
    public String loginUser(String pUsername, String pPassword) throws RemoteException;

    // Returns data model object of a Customer
    public CustomerData getCustomerData(String customerID) throws RemoteException;

} // end class
```

Bean implementation:

```
package J2EEApp;

import java.sql.*;

import javax.sql.*;
import javax.naming.*;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class BankMgrEJB implements SessionBean
{
    private static final String loginQuery =
       "SELECT USERID FROM J2EEAPPUSER WHERE USERNAME = ? AND PASSWORD = ?";
    private javax.sql.DataSource jdbcFactory;
    private CustomerHome myCustomerHome;

    // SessionBean implementation
    public void ejbActivate() { }
    public void ejbPassivate() { }
    public void ejbCreate()
    {
     try {
         Context initialContext = new InitialContext();
         jdbcFactory =
               (DataSource)initialContext.lookup("java:comp/env/jdbc/BankMgrDB");
         myCustomerHome = (CustomerHome)initialContext.lookup("customerServer");
     } catch (NamingException ne) {
         System.out.println("NamingException:\n" + ne.getMessage());
     }
    }
    public void ejbPostCreate() { }
    public void ejbRemove() { }
    public void setSessionContext(SessionContext ctx) { }

    // Bean methods
    public String loginUser(String pUsername, String pPassword)
    {
```

```
      System.out.println("loginUser: " + pUsername + ", " + pPassword + "\n");

      String userID = null;
      Connection conn = null;
      PreparedStatement ps = null;
      ResultSet rs = null;
      try {
          conn = jdbcFactory.getConnection();
          ps = conn.prepareStatement(loginQuery);
          ps.setString(1, pUsername);
          ps.setString(2, pPassword);
          rs = ps.executeQuery();
          if (rs !=  null && rs.next()) {
              userID = rs.getString("USERID").trim();

          } else {
              System.out.println("loginUser: No USERID found\n");
          }
      } catch (SQLException se) {
          System.out.println("SQL Exception: " + se.getMessage());
          se.printStackTrace();
      } catch (Exception e) {
          System.out.println("Exception: " + e.getMessage());
          e.printStackTrace();
      } finally {
          try {
              if (rs != null) rs.close();
              if (ps != null) ps.close();
              if (conn != null) conn.close();
          } catch (SQLException se) { } // ignore here
      }
      return userID;
    } // loginUser

    public CustomerData getCustomerData(String customerID)
    {
      Customer customer = null;
      CustomerData cust = null;
      try {
          customer = myCustomerHome.findByPrimaryKey(customerID);
          System.out.println("Got Customer from findByPrimaryKey()");
          cust = customer.getCustomerData();
      } catch (Exception e) {
          System.out.println("Bad ID " + customerID + ": " + e);
      } finally {
          return cust;
      }
    } // getCustomerData
} // end class
```

Like servlets and JSPs, EJBs run in a container on a server. The container arbitrates all communication between beans and clients. The beans are mapped for the container in XML configuration files. These files can get rather complex. The reference implementation provides a GUI tool to aid in the configuration.

5 Performance

Various researchers have analyzed combinations of architectures, examining them for efficiency of implementation and ease of development. While the results are always

provisional due to the rapid change in the industry, the results do give insight into how these architectures can develop.

5.1 Testing Many Architectures

Cecchet, Marguerite, and Zwaenepoel at Rice University conducted an extensive test of EBJ architectures comparing the scalability and performance of different combinations of Session and Entity beans with a servlets-only implementation as a baseline. In all cases, the presentation logic was handled by servlets.

One of the goals of their study was to assess the differences in performance between Bean Managed Persistence (BMP) and Container Managed Persistence (CMP). The conclusion was that BMP outperformed CMP on the test platforms. However, both were outperformed by the servlets-only implementation.

Another architecture in the Rice experiments used Session beans only and that architecture performed almost as well. They theorize that the reason for this is that the container is not relied upon for much beyond simple communication. A Session façade pattern also performed better than a direct EJB architecture. A Session façade pattern has Session beans communicating with the presentation layer and serving as the interface to the backend entity beans.

The Rice experiments used two open source J2EE containers and a lot of the limitations on performance and scalability appeared to come from how the containers were implemented. One container, JBoss, relied on Java Reflection[3] for analyzing the EJBs. The other, JOnAS, did more at compile time. The latter performed better.

A final experiment used the part of EJB 2.0 standard that allowed for EJBs to define local interfaces. The prior standard of home and remote interfaces required network access even when all of the requested resources were on the same physical machine. The local interfaces cut out this network access with a corresponding increase in performance.

The study also included some statistics regarding the complexity of the various implementations. The servlet-only implementation was most straightforward. The EJB requirement for multiple source files to implement the required interfaces resulted in a lot

[3] Reflection is a technology that allows a Java object to reveal its attributes to the outside without requiring source code access.

of extra complexity. Although each bean class may be simple in itself, the number of classes can grow dramatically.

A major caveat with the study is that the EJB container implementations were just snap-shots in time. Each container continues to evolve. Any results are similarly only snap-shots in time. For example, the JBoss container tested only supported EBJ 1.1. Today the test might go differently.

5.2 Testing Many Servers

Another study (Gorton and Liu, 2003) focused on just two EJB architectures but expanded the number of containers tested.

The Gorton and Liu study focused on the stateless session-bean-only and the stateless session façade patterns. They ran their tests on six containers, some commercial and some open source. Standalone and clustered configurations were used. The Stock-Online system was used to simulate an e-commerce application. The experiments simulated various loads of connections and transaction requests. The analysis included an evaluation of the ease of development as well as the ease of maintenance.

The results showed some of the commercial containers to be very capable. The Borland Enterprise Server outperformed all others with the stateless session-bean-only architecture and was one of the top performers for the session façade pattern.

The study concluded that the session-bean-only pattern was more likely to scale better than the session façade pattern. Their reasoning was that session façade had too much additional overhead for inter-bean communication.

5.3 NCDC Study

Only one study (Baldwin, 2003) included JDO as an option. The problem being addressed by the Baldwin study was the handling of the large amounts of weather data held by the National Climatic Data Center (NCDC). Some of the data dates back to the 19th Century.

The NCDC needs to be able to retrieve sets of data from its Integrated Surface Hourly (ISH) data store. Each ISH data set contains 40 elements (temperature, visibility, *etc*.)

from which additional summaries can be calculated. To be able to store and efficiently retrieve this data, the Baldwin study looked at several options.

The first option was a simple file system. The Java objects are serialized[4] and written to a file. The second used a simple key/value database. The serialized object is stored in a database using the key. The third used a RDBMS and the last an OODBMS.

The RDBMS method could be implemented using just JDBC, or JDO, or as EJBs. The EJBs could then be implemented with BMP or CMP. (Baldwin asserts that CMP has "largely replaced" BMP.)

In the tests conducted for the study, the fastest load of data was from the key/value pair database. The quickest fetch of data was from the simple file system. These approaches have their problems, however, in managing the large data sets. The ISH data is being arranged as a data mart with the materialized views following a star model. The study concluded that for ISH, the data mart, key/value databases and JDO would each have a role in the solution. Baldwin does note, however, that some database vendors are not supporting the current JDO standard.[5]

6 Conclusions and Suggestions

The motivation for preparing this paper was to answer the question, "Does JDO provide a better way of handling databases in Java applications?" Unfortunately, the answer is not definitive. It does appear that JDO is a better way, but only if other aspects of the problem being addressed do not encourage using other technologies.

From the Rice study (Cecchet, *et al*. 2002) it is clear that simple servlets still have a place. These can use either JDBC or JDO. They could use SQLJ if that fits better with the other tools being used for the solution.

While a large, distributed application could benefit from an EJB-based architecture, this does add significantly to the complexity of the solution. However, this added complexity must be balanced with real project needs. The more complex the solution, the easier it is to get wrong.

---

[4] Serialization is a way to export a Java object from a JVM and later to reactivate it in another JVM. Other technologies use the term "marshalling."

[5] As this paper is being written, a new JDO specification was released in final form so database vendors may change their view of JDO.

The research also shows that in some ways, it does not matter what architecture is used by the application, but the architecture of the container itself can have significant impact on system performance.

Confounding the issue further, all such performance and scalability tests are limited in lifespan. Most, perhaps all, of the implementations studied are not today as they were then.

What is needed is a set of reference tests that can be applied to new systems as they develop and evolve. That would not be an easy task, but it could be one of great value to the software industry.

7 References

Almaer, Dion. (2002). Using Java Data Objects. *ONJava.com* 2/6/2002. URL: http://www.onjava.com/lpt/a/1372

Java Data Objects (JDO) is a recent addition to the suite of APIs available for accessing databases from a Java-based environment . This article uses an address book as a simple example of how the JDO technology is used to build Java objects from relational databases.

Baldwin, Richard T. (2003). Views, Objects, and Persistence for Accessing a High Volume Global Data Set. *Digest of Papers IEEE Symposium on Mass Storage Systems (MSS'03)* p 77-81

This paper describes lessons learned by the National Climate Data Center (NCDC) as it decides how to deploy extremely large data sets for use by scientists and researchers world-wide. Prototype systems were developed using direct file access from Java programs, a key/value database bundled with Unix systems, and Java Data Objects (JDO) and Enterprise JavaBeans (EJB) interfaces to RDBMS and ODBMS.

Brown, Jeff. (2002). An Introduction To Java Data Objects. *Object Computing, Inc. - Java News Brief* June 2002. URL: http://www.ociweb.com/jnb/jnbJun2002.html

This web article provides a general overview of the Java Data Objects (JDO) API and how it relates to the prior Java Database Connectivity (JDBC) API. The XML descriptions of the RDBMS-to-Java object mappings are also discussed.

Cecchet, Emmanuel ; Marguerite, Julie and Zwaenepoel, Willy. (2002). Performance and Scalability of EJB Applications. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA, 2002, p 246-261

The paper describes a study of the performance of an e-commerce application utilizing different combinations of Enterprise JavaBeans. The conclusion of the study is that stateless session beans with bean managed persistence out perform alternate combinations of entity beans with container managed persistence. A major factor in their results was the internal architecture of the tested containers. The container that relied upon Java Reflection did not perform as well as the container that did pre-compiling of the bean interface implementations. The study compared JBoss 2.4.4 and JOnAS 2.4.4. The servlet container was Tomcat 3.2.4.

Eisenberg, Andrew and Melton, Jim. (1998). SQLJ Part O, now known as SQL/OLB (Object-Language Bindings). *SIGMOD Record*, Vol. 27, No. 4, December 1998. p 94-100

SQLJ/OLB is a standard for embedding SQL code directly into Java source code. This paper describes the standard and illustrates how it merges with the Java Database Connectivity (JDBC) API.

Gorton, Ian and Liu, Anna. (2003). Evaluating the Performance of EJB Components. *IEEE Internet Computing*, v 7, n 3, May/June, 2003, p 18-23

A report comparing the performance of two common Java 2 Enterprise Edition's (J2EE) Enterprise JavaBean (EJB) application architectures. J2EE allows for management of object persistence either by the server container or the EJBs themselves. The architectures were tested on Borland Enterprise Server 5.02, Interstage Application Server 4.0, SilverStream Application Server 3.7.4, WebLogic Server 6.1, WebSphere Application Server 4.0, and JBoss 2.4.3. Their conclusion was that the session-bean-only pattern was more likely to scale better than the session façade pattern.

Hunter, Jason, William Crawford. (2001). *Java Servlet Programming 2nd Edition*. O'Reilly & Associates.

The book covers Java Servlet standards 2.2 and a draft of the 2.3 standard.

Jordan, David and Russell, Craig. (2003). JDO or CMP? *ONJava.com* 05/21/2003. URL: http://www.onjava.com/lpt/a/3763

This web article provides a bullet comparison of the capabilities of object persistence using Java Data Objects (JDO) and Container Managed Persistence (CMP) of the Java 2 Enterprise Edition standard. It is excerpted from the book *Java Data Objects* published by O'Reilly & Company by the same authors.

Monson-Haefel, Richard. (2001). *Enterprise JavaBeans 3rd Edition*. O'Reilly & Associates.

The book covers the fundamentals of J2EE development using the current EJB 2.0 standard as well as the previous EJB 1.1 standard.

Reese, George. (2000). *Database Programming with JDBC and Java 2nd Edition*. O'Reilly & Associates.

This is a *how-to* book on JDBC and its Java interface. The book provides examples of using various configurations of a JDBC environment and a complete listing of all of the interfaces and classes in the standard. It covers the JDBC 2.0 version of the standard.

Salo, T. and Hill, J. (2000). Building Enterprise Web Applications with Java. *JOOP - Journal of Object-Oriented Programming*, v 13, n 2, May, 2000, p 28-29+47

Five architectures for web applications utilizing different combinations of Java-based technologies are examined. Systems using components comprised of servlets, Java Server Pages (JSP), Java Beans and Enterprise JavaBeans, and combinations are compared.

Tost, A. and Johnson, V.M. (2000). Using JavaBeans Components as Accessors to Enterprise JavaBeans Components. *IBM Systems Journal*, v 39, n 2, 2000, p 293-300

The paper describes how using JavaBean components in a Enterprise JavaBean allows for a clean, three-tier architecture separating client-side and server-side implementations.