Starting with Java Persistence Architecture

Introduction

Java Database Connectivity (JDBC), the traditional mechanism for accessing relational data bases from Java programs, is getting pretty close to being "deprecated".

JDBC too low level

The JDBC API is low-level. It is too concerned with the mechanics – getting connections, creating statements, composing strings with SQL directives, arranging for the SQL to be run, and pulling data out of result sets one datum at a time. Such JDBC code breaks the predominant "invoke method on object" style of Java in much the same way as the java.net (sockets) API breaks the object model when invoking operations of an object in another process.

An insulating layer – hide low level detail

Java's Remote Method Invocation API was created years ago to hide the network level and provide a consistent object model for distributed programs. The new Java Persistence Architecture (JPA) APIs and supporting technologies will in future provide a similar insulating layer that will conceal low level details of data persistence mechanisms so allowing an application developer to work in a consistent "object programming" style.

JDBC code is in the JPA library

Of course, the JDBC libraries will remain in use. All that is happening is that a new level of software is being introduced. This JPA level automates the object to relational-table mapping. The JDBC code is still present – it is simply in the supplied JPA implementation library.

JPA implementations

There are several implementations available including those based on Hibernate, Kodo, an OpenJPA implementation for Apache Geronimo, and an implementation for SAP's application server. The "standard" implementation is currently Oracle's

Toplink. The Toplink libraries are included in the Java enterprise development kits downloaded from Sun.

JDBC code complexity

In a traditional JDBC program, it is commonplace to use simple data classes that correspond to the rows of database tables. Instances of these classes have data members that hold data copied to and from corresponding columns in a relational table. Code utilizing such classes is simple, but relatively long winded and clumsy. As example, the code needed to create an object with the data for a record with a given key involves parameterising and running a query, getting a ResultSet row, creating an empty object, and then successively extracting data elements for each column and using these to set members in the new object. Such code is very regular, and consequently can easily be handled automatically if given a mapping from table columns to data members.

Persistence automation – EJB style

Mechanisms for automating persistence were defined for Enterprise Java versions 1 and 2; but those mechanisms were complex and restrictive. The programmer had to create entity classes that fitted into the EJB framework; those classes had to extend framework supplied base-classes or implement specific interfaces. In addition, factory classes and other helper classes had to be coded.

Evolution of JPA

Many developers found the EJB framework overly complex and intrusive and looked for other mechanisms. From around 2000, numerous open-source and commercial projects created Java libraries that allowed simpler persistence models; these projects include JDO, Kodo, Hibernate and others. Any of those libraries could be used for persistence, but they were not standard reference implementations. The JPA represents the next generation; it benefits from all the work on the earlier persistence mechanisms and provides a solution that is part of the Java reference standard.

JPA code simplicity – leave the complexities to the Entity Manaaer

A JPA application makes use of an "entity manager" that serves as its "intelligent" connection to the relational data store. The code needed to instantiate an in-memory object with the data associated with a primary key is simplified down to a request that this entity manger "find" that object – one line of code rather than an entire procedure that runs a query and processes the results in the ResultSet.

Meta-data relate "EntityManager" and database

The "entity manager" must of course be supplied with data identifying the data source and specific details of which table must be used and how the table columns relate to data members of the Java class used to represent the in-memory object. These necessary "meta-data" can be supplied separately from the Java code; an XML file can contain details of the data source and the table mappings.

"Annotations" simplify meta-data

The JPA system provides an additional facility for aiding rapid application development through the use of Java code annotations that allow the programmer to define most of the mapping details with the code. (While the annotation approach facilitates development, the use of an XML file with all the meta-data may be more appropriate for production environments.)

JPA and entity relationships

The JPA goes beyond handling simple object-to-table mappings. It also handles relations among different entities. Consider the classic case of an "order" that is composed of some order data and a collection of many "line-items". With JPA one

can define an Order class one of whose members is a Collection (e.g. java.util.ArrayList) of LineItem objects and provide supplementary meta-data that identify how orders and lineitems are related in the database (typically, lineitems would have the order number as a foreign key). JPA will allow an application to retrieve an Order object; when the application first attempts to access the collection of line items, JPA will arrange to run the JDBC code needed to retrieve these records as well.

Classes define tables, OR tables define classes

JPA implementations typically give you a choice. You can use existing tables in some data base; the JPA implementation will automatically generate the basic definitions of the entity classes that will be used when instantiating in-memory copies of persistent data. (The developer will generally need to add other business methods to these generated classes and provide effective definitions for methods such as toString()). Alternatively, you can start by defining Java classes and the JPA system will create the database tables from your class definitions.

An example – JDBC/JPA applications, servlets, enterprise …

The rest of this note presents an example illustrating JPA. The example involves the creation, reading, updating and deletion of a number of simple records. Versions of the code start with the traditional JDBC code, and then present the JPA alternative. JPA is illustrated for use in standard (Java SE) applications and also for "container managed" applications (EJB style). There are some subtle differences in the way that JPA is used in different contexts.

Example Application

More CRUD – create, read, update and delete; a schools enrolment database

The example problem domain that is used here to illustrate conventional JDBC and newer JPA styles has hypothetical data relating to a small local government region's primary schools, its teachers, and their pupils. The application is to provide mechanisms for creating, reading, updating, and deleting school enrolment data. It takes the form of a simple "menu-select" program that allows a user to do things like add a pupil, assign him/her to a school and then to a particular class ("K", "1", ..., "6") in that school, and get a listing of all pupils in that class.

The data and relationships among data entities

There are three types of data:

- Schools – The data characterizing a school are limited to the school name and its address.
- Teachers – Teachers are identified by an employment number, and naturally have a title, a first-name, initials, and a surname. Most teachers are assigned to specific schools; there are a few "relief" teachers who have no fixed assignment. A teacher may be a school principal, or may be a "class teacher" responsible for one of the classes ("K", "1", ..., "6") in the primary school. A school may have other teachers (e.g. physical education teachers, remedial teachers, English as second language teachers, etc) who are not assigned a particular class.
- Pupils – Pupils have enrolment numbers, initials, and a surname; their gender is also recorded. Pupils newly enrolled in the school district will appear on the roll for a while before they are assigned to a specific school. Soon after joining a school, a pupil will be allocated to one of the classes ("K", "1", ..., "6").

Such data are easy to represent in relational tables – School, Teacher, and Pupil. Each table has a natural primary key (school-name, employment-number, enrolment-number). There are no "many-to-many" relations; but there are several "one-to-many" relations: a school has many staff members, a class in a particular school has many pupils. Such relations could be represented via auxiliary tables but with these simple data they can be handled by a table column in the "many" side that is effectively a foreign key.

The tables:

Oracle tables

The actual table definitions, as given here, are for Oracle:

```
create table school (
     schoolname varchar2(64) primary key,
     address varchar2(128)
);

create table teacher (
     employeenumber number(12) primary key,
     surname varchar2(16) not null,
     firstname varchar2(16) not null,
     initials varchar2(4),
     title varchar2(4)
          constraint title_check
               check (title in ('Dr', 'Mr', 'Mrs', 'Ms', 'Miss')),
     schoolname varchar2(64),
     constraint valid_school
          foreign key(schoolname)
                    references school(schoolname),
     role varchar(10),
          constraint role_type
               check (role in
                   ('PRINCIPAL', 'K', '1', '2', '3', '4', '5', '6' ))
);

create table pupil (
     enrolnumber number(12) primary key,
     surname varchar2(16) not null,
     initials varchar2(4) not null,
     gender varchar2(1)
          constraint gender_check
               check (gender in ('M', 'F')),
     schoolname varchar2(64),
     classlvl varchar2(2),
     constraint enrol_school
          foreign key(schoolname)
               references school(schoolname),
     constraint enrol_class_lvl
          check (classlvl in ('K', '1', '2', '3', '4', '5', '6' ))
);
```

Relationship amongst entities expressed via "foreign key" constraint

Populating the tables – scripts and initialization code supplied

Data for tables:

SQL scripts are used to create these tables and populate them with around 40 teacher records and about 250 pupil records. The initial data does not define any school or

role values for teachers, nor any school or class values for pupils. The application code includes a section where, if necessary, these additional data are generated pseudo-randomly and used to update the initial tables.

Menu-commands:

Program inputs

The Java SE version of the application uses a hierarchy of command menus that allow a user to select processing options. The top-level menu allows the user to select create, read, update, or delete operations. The "create menu" offers options for adding teachers or pupils; the "update menu" allows assignments of teachers or pupils to be changed. The "read" menu has options for retrieving details of individual pupils or teachers and for obtaining collections such as details of all teachers employed in a particular school. It does not use an elaborate GUI interface. Inputs are handled via JOptionPane dialogs while outputs are mostly to the System.out console stream.

Reports:

Program outputs

The following illustrates part of the program's output to the console for read requests for information on a school, its staff, and pupils in one of its classes:

```
Selected Huntley Vale Primary
Huntley Vale Primary
        Landsdown Road, Huntley
        Principal: 2007040003: Mr Thomas K Pollock
1979120023: Ms Joanne R Peach 6
1989050031: Ms Mary P O'Farrel 2
…
1994120067: Ms Deborah T Black K
Selected class K
Class teacher : 1994120067: Ms Deborah T Black
20031012928: VN Bennet (F)
...
20031412826: P Henderson (F)
```

The other versions of the application use a WWW style interface handled directly via servlets; responses take the form of HTML pages with tabular data.

JDBC style

Program outline
JDBC selection queries
Read operations
Simple "bean" classes
Inserting rows

Applications of this nature are basically procedural. The application has a main() that invokes static functions that setup database connections, and then run command loops wherein the user can enter processing commands and view results.

The main class will have a number of static variables and constants including such data as the database driver class, database connection string, and strings for SQL

commands. The example program created PreparedStatements for all queries that might get used repeatedly.

JDBC program outline:

The basic outline of the code is as follows:

“Procedural style” code – static functions and data

```
public class Main {
    // Constants and static variables
    private static final String dbDriverName =
            "oracle.jdbc.driver.OracleDriver";
    private static final String dbURL =
            "jdbc:oracle:thin:@host:1521:orcl";
    // Additional data for db-username/password etc
    …
    // Variables like array of school names, database connection
    private static String[] schoolArray;
    private static Connection dbConnection;

    // Strings for SQL queries
    private static final String getSpecificTeacher =
            "select *  from Teacher  where EMPLOYEENUMBER=?";
    …
    // Prepared statments that are used extensively
    private static PreparedStatement pstmtSpecificTeacher;
    …
```

Connect to database

```
    /**
     * Create database connection and prepare statements
     * that will be used extensively.
     */
     private static void connectToDatabase() {

         try {
             // Ensure driver loaded, then open
             // database connection
            Class.forName(dbDriverName);
            dbConnection = DriverManager.getConnection(
                  dbURL, userName, userPassword);
            // Create prepared statements as well
            pstmtSpecificTeacher =
                  dbConnection.prepareStatement(getSpecificTeacher);
            …
            }
        catch (Exception e) { … }
    }

    …
    …
```

Command loop

```
    /**
       * Menu-select loop getting commands from user
       */

     private static void interact() {
       String[] optionArray = {"Create records", … };
        for (;;) {
            Object selected = JOptionPane.showInputDialog(null,
```

```
                    "Select processing option", "CRUD options",
                   JOptionPane.INFORMATION_MESSAGE, null,
                   optionArray, optionArray[4]);
            if (selected == null) {
                break;
            }
            if (selected == optionArray[0]) {
                createRecords();
            } else …
        }
    }
```

Main()

```
    public static void main(String[] args) {
        try {
            connectToDatabase();
            initializeSchoolsArray();
            …
            interact();
            dbConnection.close();
        } catch (SQLException ex) { … }
    }
}
```

Simple JDBC selection query:

The program options that allow teachers and pupils to be assigned to schools require a JOptionPane dialog that shows the school names; so an array of school names must be created. This leads to the first operations involving the database; a simple query must be run that will retrieve the names of all schools, gathering these into a temporary dynamic collection, then converting these data to an array form. There is nothing complex about such code; it is just a little tedious as it necessitates all the low level details being defined explicitly:

Run a query, retrieve a ResultSet, laboriously extract data!

```
private static void initializeSchoolsArray() {
    try {
       final String getSchoolNames =
            "select schoolname from School";
       PreparedStatement pstmtSchoolNames =
             dbConnection.prepareStatement(getSchoolNames);

       ArrayList<String> schoolNames = new ArrayList<String>();
       ResultSet rs = pstmtSchoolNames.executeQuery();
       while (rs.next()) {
           String aschoolname = rs.getString(1);
           schoolNames.add(aschoolname);
           }
       rs.close();
       schoolArray =  schoolNames.toArray(new String[0]);
       pstmtSchoolNames.close();
     } catch (SQLException ex) { … }
    }
```

Example “Read” operations:

The code for the other “read” operations is similarly simple, straightforward, but rather tediously long. The following illustrate a variety of read operations – some returning individual rows, some collections of rows, and others getting aggregate properties. The JPA versions of the same operations are illustrated in the next section.

The first query uses SQL to return counts of records satisfying particular constraints, in this case counts of all teacher records or just those for “relief” teachers:

```
private static final String getTeacherCount =
      "select count(*) from Teacher";
private static final String getReliefTeacherCount =
      "select count(*) from Teacher where schoolname is NULL";
…
private static PreparedStatement pstmtCountTeachers;
private static PreparedStatement pstmtCountReliefTeachers;
// PreparedStatements created after connection to database
…

private static void countTeachers(boolean onlyRelief) {
   try {
      PreparedStatement pstmt = (onlyRelief) ?
             pstmtCountReliefTeachers : pstmtCountTeachers;
      ResultSet rs = pstmt.executeQuery();
      if (rs.next()) {
          int count = rs.getInt(1);
          if (onlyRelief) {
              System.out.println("There are " + count +
                        " relief teachers");
          } else {
              System.out.println("The department employs " + count +
                         " teachers");
          }
      } else {
          System.out.println("Failed to retrieve teacher data");
      }
      rs.close();
   } catch (SQLException ex) { … }
}
```

Programmatically construct “beans” to hold retrieved data

The program does make some use of simple POJO (Plain Old Java Object) classes to hold the data for records of Teachers, Pupils etc.

```
private static Teacher getTeacher(Long id) {
    Teacher t = null;
    // Attempt to retrieve Teacher record given employment
    // number
    try {
      pstmtSpecificTeacher.setLong(1, id);
      ResultSet rs = pstmtSpecificTeacher.executeQuery();
      if (rs.next()) {
            t = new Teacher(
                  rs.getString("EMPLOYEENUMBER"),
                  rs.getString("TITLE"),
                  rs.getString("SURNAME"),
                  rs.getString("FIRSTNAME"),
                  rs.getString("INITIALS"),
                  rs.getString("SCHOOLNAME"),
```

```
                    rs.getString("ROLE"));
            }
            rs.close();
        } catch (SQLException ex) { …  }
        return t;
}
```

Often, simple CRUD programs implemented using JDBC will forgo entity classes and work simply with the raw data associated with SQL results and queries. Though common, such a programming style further reduces the intelligibility of the programs:

```
private static final String getTeachersAtSchool =
     "select * from Teacher where schoolname=?";
private static PreparedStatement pstmtTeachersForSchool;
…

private static void listTeachersBySchool(String schoolName) {
    try {
        pstmtTeachersForSchool.setString(1, schoolName);
        ResultSet rs = pstmtTeachersForSchool.executeQuery();
        int count = 0;
        while (rs.next()) {
           if (count == 0)
              System.out.println("Teachers employed at" +
                   schoolName);

           String empNum = rs.getString(1);
           String title = rs.getString(5);
           String surname = rs.getString(2);
           String firstname = rs.getString(3);
           String inits = rs.getString(4);
           if(inits==null) inits =" ";
           String lvl = rs.getString(7);
           if(lvl==null) lvl = "-";
           System.out.println("\t" +
                       empNum +": " +
                       title + " " +
                       firstname + " " +
                       inits + " " +
                       surname + " " +
                       lvl
                       );
             count++;
          }
          if (count == 0) {
             System.out.println(
                  "There are no teachers currently employed at " +
                  schoolName);
      }
      rs.close();
    } catch (SQLException ex) { … }
}
```

Try to manage without “beans” – the code is even worse!

Use of simple “bean” classes

Classes such as Teacher will just be “beans” – a few private data members corresponding to the columns in the relational table, and appropriate accessor and mutator functions:

Definition of a simple “bean” class

```
public class Teacher {
    private String employeeNumber;
    private String title;
    private String firstName;
    private String initials;
    private String surName;
    private String schoolName;
    private String role;

    public Teacher() { }

    public Teacher(String anEmpNum, String atitle,
         String asurName, String afirstName, String inits,
         String sName, String rolename
         )
    {
        employeeNumber = anEmpNum;
        title = atitle;
        …
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    …

    @Override
    public String toString()
    {
        // Some effective definition of toString
        …
    }

}
```

Entity relationships not clearly expressed in bean classes.

Note how a Teacher object references its associated School. It has a string for the schoolname – a primary key for the School table. If the application code needed to print out details of where a teacher with specified employment number was employed, it would have to first run one query to retrieve the teacher record and then another query to load the School record. These classes are limited; they don’t really capture anything of the relationships that can exist between class instances. With the JPA style, we will manage better.

Example “create” operations:

Other operations, such as those that create new records in the database, are equally simple but tiresome as a consequence of their use of the low level JDBC API. Thus,

the following procedure correctly updates the database but fails to convey the idea of “create a new Teacher object”:

```
private static final String addTeacher =
      "insert into Teacher values(?,?,?,?,?,null,null)";
private static PreparedStatement pstmtAddTeacher;
…

private static void addTeacher() {
    try {
        String[] titlesArray = {"Dr", "Mrs", "Ms", "Miss", "Mr"};
        String empNumber = JOptionPane.showInputDialog(
                  "Employment number");
        Object selected = JOptionPane.showInputDialog(null,
            "Select title", "Person\'s title",
            JOptionPane.INFORMATION_MESSAGE,
            null, titlesArray, titlesArray[0]);
        if (selected == null)
                return;

        String title = (String) selected;
        String firstName = JOptionPane.showInputDialog(
                  "First name");

        String initials = JOptionPane.showInputDialog("Initials");

        String surName = JOptionPane.showInputDialog("Surame");
```

Inserting rows? Hack, hack, hack!

```
        pstmtAddTeacher.setString(1, empNumber);
        pstmtAddTeacher.setString(2, surName );
        pstmtAddTeacher.setString(3, firstName );
        pstmtAddTeacher.setString(4, initials);
        pstmtAddTeacher.setString(5, title );
        pstmtAddTeacher.executeUpdate();
     } catch (SQLException ex) { … }
 }
```

With JPA, the code for all such operations becomes much more intelligible.

JPA style

The application was recoded to work with JPA. JPA uses formally defined “entity classes”. Though still just POJO classes, these entity classes can be designed to express the relations that exist among entities.

Entity relationships expressed in classes:

Entity relationships through "object references"

In this example application, "one-to-many" relationships exist between a school and both its staff and its students. These relationships can be defined by having a School class whose data members include Collection<Teacher> and Collection<Pupil>. Rather than have a Teacher object have a String with the name of the school were that teacher is employed, the data member can be a reference to the appropriate School object. When these relationships are captured in the class structure, the code becomes more readily comprehended.

Entity classes can be defined by the application developer and can be processed to create the tables. But, here, the tables already existed in the database making it easier to have the classes auto-generated. A JPA system can read table meta-data and construct initial outlines for the entity classes. This was the first step taken in the development of the JPA version of the application.

The definition generated (using the Oracle Toplink JPA implementation via NetBeans) for Teacher is as follows:

Teacher object has reference to School object – not school-name String

```
@Entity
@Table(name = "TEACHER")
…
public class Teacher implements Serializable {
    @Id
    @Column(name = "EMPLOYEENUMBER", nullable = false)
    private BigDecimal employeenumber;
    @Column(name = "SURNAME", nullable = false)
    private String surname;
    @Column(name = "FIRSTNAME", nullable = false)
    private String firstname;
    @Column(name = "INITIALS")
    private String initials;
    @Column(name = "TITLE")
    private String title;
    @Column(name = "ROLE")
    private String role;
    @JoinColumn(name = "SCHOOLNAME",
                referencedColumnName = "SCHOOLNAME")
    @ManyToOne
    private School schoolname;

    public Teacher() {
    }

    public Teacher(BigDecimal employeenumber) {
        this.employeenumber = employeenumber;
    }

    public Teacher(BigDecimal employeenumber,
            String surname, String firstname) {
        this.employeenumber = employeenumber;
        this.surname = surname;
        this.firstname = firstname;
    }
```

```
    public BigDecimal getEmployeenumber() {
        return employeenumber;
    }

    public void setEmployeenumber(BigDecimal employeenumber) {
        this.employeenumber = employeenumber;
    }

    public void setSchoolname(School schoolname) {
        this.schoolname = schoolname;
    }

    …
}
```

Annotations are used to provide the meta-data that relate the defined class to its table, identify its primary key field, and identify the column names associated with each data member. The JPA translation system has used the "foreign key" relationship specified in the table meta-data to infer the relationship between a Teacher object and a School object; so here, the schoolname field is of type School rather than being just a String with the school's name.

The definition of class School is in part as follows:

Foreign key relations interpreted – School "owns" a collection of Teachers and a collection of Pupils

```
@Entity
@Table(name = "SCHOOL")
…
public class School implements Serializable {
    @Id
    @Column(name = "SCHOOLNAME", nullable = false)
    private String schoolname;
    @Column(name = "ADDRESS")
    private String address;
    @OneToMany(mappedBy = "schoolname")
    private Collection<Teacher> teacherCollection;
    @OneToMany(mappedBy = "schoolname")
    private Collection<Pupil> pupilCollection;

    public School() {
    }

    public School(String schoolname) {
        this.schoolname = schoolname;
    }

    …

}
```

The JPA system has generated fields, of type Collection, representing the staff and pupils associated with the school. A School object can be instantiated without these collections having to be populated. Obviously it wouldn't be desirable to load all associated Teacher and Pupil records every time a School object is instantiated; instead, such data are fetched on demand.

Use of the Teacher and School entities is illustrated in the code below.

JPA program outline:

The overall program structure is similar to that of the JDBC version. There is an initiation step, then a loop where JOptionPane dialogs are employed to allow the user to select processing options.

```
public class Main {
    private static EntityManagerFactory emf;
    private static EntityManager em;
    private static String[] schoolArray;
    …

    private static void setupDataPersistence()
    {
        emf = Persistence.createEntityManagerFactory(
            "JavaApplication12PU");
        em = emf.createEntityManager();
    }

    private static void closedownDataPersistence()
    {
        em.close();
        emf.close();
    }

    private static void interact() {
        // As shown above
    …
  }
  public static void main(String[] args) {
    setupDataPersistence();
    initalizeSchoolsArray();
    …
    interact();
    closedownDataPersistence();
  }

}
```

As in the JDBC version, the program starts by creating its "connection" to the datasource. The connection is handled through an EntityManager object created using a factory class.

Defining the "connection" meta-data: persistence.xml

The program does not appear to provide the data needed to identify the driver, the database URL, or the user-name and password. These data are contained in a separate "persistence.xml" file; this file should be included in the same directory area as the codebase of the program, it gets read in the library code for the EntityManagerFactory. This mechanism is obviously more convenient than having database details in the code (requiring recompilations for changes), and it is simpler than the other alternative of arranging for a separate JNDI resource lookup system.

The contents of persistence.xml are typically created with the aid of some "wizard". For this example, the file reads as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
     xmlns="http://java.sun.com/xml/ns/persistence"
     …>
  <persistence-unit
     name="JavaApplication12PU"
     transaction-type="RESOURCE_LOCAL">
```

IDE "Wizard" creates XML file using information from dialogs

```
<provider>oracle.toplink.essentials.PersistenceProvider</provider>
    <class>javaapplication12.Pupil</class>
    <class>javaapplication12.School</class>
    <class>javaapplication12.Teacher</class>
    <properties>
      <property name="toplink.jdbc.user" value="user"/>
      <property name="toplink.jdbc.password" value="password"/>
      <property name="toplink.jdbc.url"
          value="jdbc:oracle:thin:@host:1521:orcl"/>
      <property name="toplink.jdbc.driver"
          value="oracle.jdbc.OracleDriver"/>
    </properties>
  </persistence-unit>
</persistence>
```

The "EntityManager" – an "intelligent" database connection

EntityManager – running the requests, synchronizing data

An EntityManager has several roles. It is the EntityManager that will run ad-hoc queries (such as the queries that return counts of teachers etc); it can also fetch objects given primary keys, and can automatically build collections such as the collection of Teacher objects associated with a particular school. It also "manages" these objects once they have been loaded. The EntityManager can observe changes made to objects and can arrange to synchronize persisted versions of the data with the updated in-memory objects.

The exact way in which an EntityManager works does however vary. The factory that creates an entity manager sets control parameters that define its approach to managing instances of entity classes. There are differences between the settings for Java SE programs like the current example, and container managed applications like the EE versions illustrated later.

"extended persistence" context

In a Java SE application, an entity manager will use an "extended persistence context" and "resource-local transactions". In this mode, the entity manager will keep track of entities once they have been loaded into memory and will, *at its convenience*, synchronize any changes to objects with the database. (It is possible to explicitly remove an entity from the persistence context, so allowing it to be changed without the changes being written to the database.) When is it convenient for an entity manager to write changes to the database? That is determined by the implementation; updates of the data tables may be delayed for a long time. Typically though, changes are made at the same time as the next explicit transactional operation.

Transactional updates

In EE container managed applications, it is the container that arranges for transactional operations. In SE applications, the application code must organize transactions explicitly. Read operations (finding an entity and displaying its contents) do not require transactions. But operations that make change changes – creation, update, removal – will only be written to the database as part of a transaction. A developer of SE-style JPA applications must remember to arrange for transactions to complete such operations.

JPA Simple selection query:

The program again requires an array with the school names loaded from the database. The JPA defines a "Java Persistence Query Language" JPQL that allows for select requests that are similar to SQL select requests as used with JDBC; of course, JPQL does differ, in part because now one is retrieving objects not rowsets. (There are advanced options that allow for native SQL queries in cases where these are really necessary or where the developer wishes to take advantage of some database specific SQL feature.)

The JDBC-based code that loaded the schools' names was illustrated earlier. The JPA code is somewhat cleaner:

EntityManager, giveme a the list of the school names

```
private static void initalizeSchoolsArray() {
   Query q = em.createQuery("Select s.schoolname from School s");
   List<String> schoolNames = q.getResultList();

   schoolArray = (String[]) schoolNames.toArray(new String[0]);

}
```

Here, the entity manager is used first to create a javax.persistence.Query object that has a JPQL request for one column of string data from the School table. The query is executed and, automatically, returns a collection of String objects.

JPQL can handle more complex requests that require data from several columns. Suppose for example, the School table had a more realistic representation of an address with street location, town, and postcode, and that the request was for school name and postcode. One would need to define a simple bean with String members for schoolname and postcode, e.g. NameCode. The JPQL select query would then have a form something like `Select new NameCode(s.schoolname, s.postcode) from School s`.

JPA "Read" operations:

A request for a computed aggregate property again has to be handled via JPQL. Once again, the code is a bit simpler than the JDBC code:

Simple JPA query to get a value computed by database

```
private static void countTeachers(boolean onlyRelief) {
    String fullCount = "select count(t) from Teacher t";
    String reliefCount = "select count(t) from Teacher t" +
      " where t.schoolname IS NULL";

    Query q = em.createQuery((
```

```
            onlyRelief) ? reliefCount : fullCount);
        Long res = null;
        try {
            res  = (Long) q.getSingleResult();
        }
        catch(NoResultException nre) {
            System.out.println("Didn't find any employee data");
            return;
        }
        if(onlyRelief)
            System.out.println("There are " + res.longValue() +
                " relief teachers");
        else
            System.out.println("The department employs " +
                res.longValue() + " teachers");
    }
```

A JPA request that loads a particular object, identified by a primary key, is much simpler than the JDBC version. One simply asks the entity manager to "find" the required object:

Load object with given primary key

```
private static Teacher getTeacher(BigDecimal enumber) {
    Teacher t = em.find(Teacher.class, enumber);
    return t;
}
```

(Because this is an SE-JPA application, the entity manager object keeps a reference to this Teacher object in its "extended persistence context". The entity manager will note any changes and, given an opportunity, will write any such changes to the data table. Things work rather differently in the context of EE-JPA applications.)

The program option for listing details of a school (its name and address, its staff, pupils in selected classes etc) is simplified by the entity style classes that JPA generated from the tables. Once the School object has been loaded, its staff can be obtained by the application requesting the contents of the data member with the Collection<Teacher>:

Get a school object, and its teachers

```
private static void getSchoolInfo(String name) {
    School s = em.find(School.class, name);
    System.out.println(s);
    Collection<Teacher> staff = s.getTeacherCollection();
    Teacher principal = null;
    for(Teacher t : staff)
        if("PRINCIPAL".equals(t.getRole()))
            { principal = t; break; }

    System.out.print("\tPrincipal: ");

    if (principal == null)
        System.out.println("(to be appointed)");
    else
        System.out.println(principal);

    boolean showStaff = false;
    int val = JOptionPane.showConfirmDialog(null,
            "List staff members");
    if (val == JOptionPane.CANCEL_OPTION)
```

```
            return;

    showStaff = (val == JOptionPane.YES_OPTION);
    if (showStaff) {
         for(Teacher t : staff) {
             System.out.print(t);
             if(t.getRole()==null) System.out.println(" -");
             else
                 System.out.println(" " + t.getRole());
         }
    }
    …
}
```

The JDBC version of this code required a call to a separate procedure that explicitly ran another SQL request that selects all Teacher rows where the schoolname field matched the school's name. The JPA code actually runs essentially the same SQL query; but it is all handled automatically resulting in the much clearer code s.getTeacherCollection().

JPA Create and Update operations

Objects for classes like Teacher are just Plain Old Java Objects. They can be created, changed, and discarded as the application requires. They only get associated with data table rows if they are added to the persistence context managed by an entity manager.

Persisting objects, synchronizing on next transaction

A new object, one for which a row will have to be created in the corresponding table, is added to the persistence context through the use of the "persist" operation defined by the EntityManger. In an SE-JPA application, adding an object with the "persist" operation does not result in a write to the database; the row in the database is only added when an explicit transaction operation is committed. Since a transaction is going to be required, the typical code style is along the following lines:

```
// create new object "aThing"
aThing = new …;
// complete initialization of object
aThing.setField(someparameter);
…
// Use the entity manager em to start a transaction
em.getTransaction().begin();
// Add the new Thing object to the persistence context
em.persist(aThing);
// Commit, and write the new row to the database
em.getTranscation().commit();
```

While bracketing the "persist" operation in a transaction context is typical, it isn't essential. In an SE-JPA application with an extended persistence context, it is possible to have code like `em.persist(thing1); …; em.persist(thing2); …; em.getTransaction().begin(); em.getTransaction().commit();` where several objects are added to the persistence context and then a dummy transaction is run to force the entity manager to actually create the rows.

The following code illustrates the JPA version of the “add teacher” operation that was illustrated earlier using JDBC. The revised code makes the semantics much clearer:

```
private static void addTeacher() {
   String[] titlesArray = {"Dr", "Mrs", "Ms", "Miss", "Mr"};
   BigDecimal empNumber = getNumber("Employment number");
   Object selected = JOptionPane.showInputDialog(
      null, "Select title", "Person's title",
      JOptionPane.INFORMATION_MESSAGE, null,
      titlesArray, titlesArray[0]);
   if (selected == null)
               return;

   String title = (String) selected;
   String firstName = JOptionPane.showInputDialog("First name");
   String initials = JOptionPane.showInputDialog("Initials");
   String surName = JOptionPane.showInputDialog("Surname");

   if((firstName==null) || (surName==null)) {
       System.out.println("Invalid data");
       return;
   }

   Teacher t = new Teacher();
   t.setEmployeenumber(empNumber);
   t.setFirstname(firstName);
   t.setInitials(initials);
   t.setSurname(surName);
   t.setTitle(title);
   em.getTransaction().begin();
   try {
       em.persist(t);
    }
    catch(EntityExistsException eee) {
         System.out.println("Add error, duplicate employeenumber");
    }
    em.getTransaction().commit();
}
```

Create new instance;

persist it to database

The following code fragment illustrates an update operation that either assigns a teacher to a specific school or changes their status to “relief teacher” by removing an assignment.

```
private static void changeSchoolForTeacher(Teacher t,
      boolean toRelief) {
    String schoolName = null;
    if(!toRelief) {
       Object selected = pickSchool();
       if (selected == null)
               return;

       schoolName = (String) selected;
       if(schoolName.equals(t.getEmployment())) {
           System.out.println(
             "Teacher already employed at designated school");
          return;
       }
```

```
    }

   em.getTransaction().begin();
   // change school, and remove any existing role
   t.setRole(null);
   if(schoolName != null)
      t.setSchoolname(getSchool(schoolName));
   else
      t.setSchoolname(null);
   em.getTransaction().commit();
}
```

As was the case with creation operations, a transaction is needed to force the revised data to be written to the database. There is no requirement that the changes to the object be done within this transactional context; when an extended persistence context is being used, the objects can be changed at any time. However, since a transaction will be needed anyway, it is probably clearer if the changes are all made within an explicit transactional context.

A delete operation is handled by "finding" the record to be deleted, "removing" it, and a transaction commit. It is typically coded as follows:

```
private static void removeTeacher(BigDecimal empnum) {
   em.getTransaction().begin();
   Teacher t = em.find(Teacher.class, empnum);
   if(t!=null) {
      System.out.println("Removed " + empnum.toString());
      em.remove(t);
    }
    else
       System.out.println("No record of that employee number");
    em.getTransaction().commit();
}
```

When using an extended persistence context, it isn't essential that the find and remove operations be bracketed by the transaction begin and end operations. The find and remove code can be invoked at any time; but the database update will only occur with a transaction. It is clearer if the illustrated coding style is followed with its transaction begin and commit steps bracketing the find and remove operations.

Costs

The code for the JPA version of the application was easier to write; it would be much more intelligible to maintenance programmers who might later need to modify the code; it is shorter (about 25% fewer lines) than the JDBC code. Of course, nothing comes for free.

The JPA code will be more costly to run. The code libraries must rely on things like Java reflection and invoke indirect mechanisms for operations such as setting fields where JDBC style code would involve simple method invocations. The JPA

libraries will involve many processing steps to perform persist and commit operations whereas the JDBC implementation would make just the one invocation of an SQL executeUpdate step. Though there would consequently be additional processing costs, any time differences would be too small to detect in a simple application such as the example.

There are also differences in the way in which operations are performed on the database, the number of operations, and the volume of data traffic. These differences are more noticeable.

A test was composed involving a sequence of twenty operations including listings of all staff in a school, all relief teachers, all pupils in a given class, updates, insertions of new data etc. The same test sequence was run under standard conditions against fresh copies of the tables in the database.

All data traffic between the application client and the database server was captured using the Wireshark utility and summary reports of traffic were generated. For this example, the JDBC implementation ran the test using 342 total data packets amounting to 40,302 bytes. The JPA version required 416 total packets with a data volume of 86,425 bytes. Although it is a very limited test, it does illustrate that there can be considerable changes in data volume.

Such changes in data traffic are very dependent on the database drivers that are employed. If performance does become an issue, it can be worth monitoring the data traffic and evaluating different JDBC database drivers (it may also be necessary to explore the different encoding formats that a driver may offer for the data traffic, some database systems do not deal effectively with 16-bit Unicode data traffic). Another factor, though not one likely to have been of importance in the limited 20 operation test, is the difference between "prepared statements" and "statements". The entity manager's createQuery functions, used in the JPA version, are akin to the use of java.sql.Statements. The JPA offers a "named query" facility that is essentially the same as the use of java.sql.PreparedStatements. Use of named queries should show some performance gains.

JPA functions illustrated:

EntityManager Factory

The `EntityManagerFactory` is typically only used by developers of SE-JPA applications; its one important method is `createEntityManager`. (In container managed applications, it is primarily the container code that utilizes such factory objects.)

EntityManager

The methods of `EntityManager` that have been illustrated are:

- `createQuery(String)`
  Creates a `javax.persistence.Query` object that will be used to execute a JPQL style select operation.
- `find(Class entityClass, Object primaryKey)`
  Loads an object given its primary key.

- `persist(Object)`
  Adds a new object, representing a new database row, to the persistence context. It is essentially an SQL insert operation.
- remove(Object)
  Essentially an SQL delete row operation.
- getTransaction()
  Returns an EntityTransaction needed to commit changes to the database.
- close()
  Closes the EntityManager.

The EntityManager also has a "merge" operation. The merge operation adds an object to the current persistence context, this object being an in-memory version of something that already exists as an entry in the database. (An extended persistence context already remembers everything ever fetched from the database, so there is never any need to add such objects back into context.)

Query

Instances of `javax.persistence.Query` are used to run JPQL (or, sometimes, native SQL) select and update statements. These are typically needed when the application needs aggregate data such as counts, or needs to join data from various tables. Query's methods include:

- `executeUpdate()`
  Run an update (or delete) JPQL/SQL operation.
- `getResultList()`
  Runs a select operation that is expected to return multiple objects.
- `getSingleResult()`
  Runs a select operation that is expected to return a single result.
- `setParameter(String name, Object value)`
  Parameters in JQPL statements can be named placeholders; actual values are substituted before the query executes. Named parameters are clearer, but one can have positional parameters as with JDBC style `PreparedStatement` objects.

An `EntityTransaction` object has typical transactional methods – `begin`, `commit`, `setRollbackonly`, and `rollback`.

Deployment

It is best to use an IDE such as Netbeans to build a jar file with all the required libraries. Netbeans has options for creating a persistence.xml file, and lets you add the Oracle Toplink implementation of JPA to a project's libraries (go to the project-properties/add-library dialog, Toplink essentials should be listed). The developer will typically need to add a jar file with the JDBC drivers, e.g. for Oracle this would be ojdbc14.jar. (Be careful to use the most up-to-date version of ojdbc14; some relatively recent versions still throw silent exceptions when processing table meta-data. These are not caught and will cause the IDE to mysteriously fail to generate correct entity classes from tables.) NetBeans will create a jar file in its dist directory that contains all the code, the libraries, and support meta-data such as the persistence.xml file.

Servlets with JPA

The use of JPA within web-applications is still somewhat problematic as many of the features have only been defined recently and they are not yet that widely supported. Different servlet containers vary in their support for features such as "resource injection". In principle, it is possible to annotate declarations of EntityManager variables or EntityManagerFactory variables with references to elements defined in a persistence.xml file. Some servlet containers support mechanisms whereby the container reads the XML files, identifies the use of the persistence unit, and can automatically initialize ("inject" data into) appropriately annotated variables. Other containers simply ignore the information generated from the annotations and will silently fail to initialize those variables.

"Injecting" initialization data

"Injection" (container-managed initialization) applies only to data members of a class. In general, a servlet should not have data members (apart from things like final Strings etc). A servlet is like a little procedural server that can be running many concurrent service requests in different threads. Instance data members are potentially accessible by all threads and so can be a potential hazard by permitting interference among threads. It may be necessary to use synchronization locks on such members (potentially reducing throughput by the servlet), or to rely on the data member being an instance of a class whose methods are themselves thread-safe. (You may see servlets that have a JDBC Connection object as a data member. Some JDBC implementations guarantee thread-safe operations by Connections; others don't support concurrent use. Such servlets will work with some databases, but not others.)

Inject a "factory" into the servlet;

Use factory to create EntityManager as needed

EnityManager implementations are not required to be thread safe. Therefore, a servlet should either avoid using data members that are instances of EntityManager or should arrange its own synchronization locks. Of course, methods like doGet or doPost can declare local variables of type EntityManager; each thread will obtain its own instance. With servlets, it is probably best to "inject" an EntityManagerFactory and then create, use, and destroy an EntityManager within individual servlet methods. The persistence contexts associated with each EntityManager will then also have lifetimes that match the use of the method.

The servlet examples illustrated below were developed with NetBeans 6 and AppServer 9. (When these examples were developed, Tomcat 6 did not handle injection of EntityManagerFactory objects; so the examples could not be run in that container.) A detailed point-by-point account of how to set up a basic JPA servlet in the NetBeans-6/AppServer-9 is provided in an attached document.

Injecting an EntityManagerFactory and using an EntityManager

The accompanying document illustrating the basics of creating and using a JPA servlet start with an example servlet whose doGet method will return a page with an ordered list of school names.

Inject factory using PersistenceUnit annotation;

Use factory to create EntityManager

```
public class FormServlet extends HttpServlet {
    @PersistenceUnit(unitName="WebSchoolDemo2PU")
    private EntityManagerFactory emf;

    private String[] getSchoolNames() {
        String[] results = null;
        EntityManager em = null;

        em = emf.createEntityManager();
        Query q = em.createQuery(
            "Select s.schoolname from School s");
        List<String> names = q.getResultList();
        results = names.toArray(new String[0]);
        em.close();
        return results;
    }


    protected void doGet(HttpServletRequest request,
       HttpServletResponse response)
    throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        String[] names = getSchoolNames();
        out.println(…);
        …
        out.println("<h1>Erehwon School Board</h1>");
        out.println("<h2>Schools</h2>");
        out.println("<ul>");

                for(String name : names)
                    out.println("<li>" + name + "</li>");
         out.println("</ul>");
         out.println("</body></html>");

         …
    }

    …

}
```

The attached note provides further details of how this servlet was deployed.

Transactions with JPA in servlets

Alternative transaction libraries

There are two ways in which transactions can be used in servlets, and unfortunately essentially no documentation that would guide one's choice between the mechanisms. It is possible to use EntityManager transactions as illustrated in the Java SE example; however, the default appears to be the use of the JTA transaction API.

If one is using the JTA, then it is necessary to get a reference to the UserTransaction object injected into the servlet. (The methods of UserTransaction are thread specific so it is safe to use a servlet data member to hold the reference to this object.) The use of a UserTransaction is illustrated in the following fragment from a NewTeacherServlet. The doGet method of this servlet displays a form for input of data defining the properties of a new teacher; the doPost method validates these input data, creates a Teacher object, and then within a UserTransaction transaction uses the EntityManager to write the data to the database:

"Inject" factory and transaction context

```
public class NewTeacherServlet extends HttpServlet {

    @PersistenceUnit(unitName="WebSchoolDemo2PU")
    private EntityManagerFactory emf;
    @Resource
    private UserTransaction utx;


    protected void doGet(HttpServletRequest request,
      HttpServletResponse response)
    throws ServletException, IOException {
        // Generate data entry form …
        …
    }

    private void badData(HttpServletResponse response,
       String reason)
            throws IOException {
        // Generate a response page reporting errors such as
        // invalid employee number, missing data, duplicate
        // employee number etc
        …
    }

    protected void doPost(HttpServletRequest request,
       HttpServletResponse response)
    throws ServletException, IOException {
       // Read and validate input data
       String empNumStr = request.getParameter("enumber");
       …

       // Create and initialize Teacher object
       Teacher t = new Teacher();
       t.setFirstname(firstName);
       t.setEmployeenumber(empNum);
       …
```

Create EntityManager within transaction (or explicitly add it to a transaction)

```
       try {
           // Establish transactional context

            utx.begin();
           EntityManager em = emf.createEntityManager();

            em.persist(t);

            utx.commit();
            em.close();
       }
```

```
        catch(EntityExistsException eee) { … }
        catch(Exception e) {
            …
        }
       // Generate success response HTML page
       …
    }

 }
```

Remove operations also require transactions as illustrated in this code fragment from a servlet that handles removal of teacher records:

```
protected void doPost(HttpServletRequest request,
      HttpServletResponse response)
    throws ServletException, IOException {
       String empNumStr = request.getParameter("enumber");

       BigDecimal enrlNum = null;
       try {
           enrlNum = new BigDecimal(empNumStr);
       }
       catch(NumberFormatException nfe) {
           badData(response,"invalid employment number");
           return;
       }
      EntityManager em = null;
       try {
            utx.begin();
           em = emf.createEntityManager();
           Teacher t = em.find(Teacher.class, enrlNum);
           if(t==null) {
               badData(response, "No such teacher");
               try { utx.rollback(); } catch(Exception er) { }
               return;
           }
            em.remove(t);

            utx.commit();
       }
       catch(Exception e) {
           try { utx.rollback(); } catch (Exception er) { }
           badData(response, e.getMessage());
           return;
       }
       finally {
            em.close();

       }


        // Generate report HTML page
        …

    }
```

Reading data with JPA and servlets

Don't require transaction contexts if simply reading data

Transactions are not required for read-only operations. The following fragments illustrate some of these operations on the "schools" data. They also illustrate the differences between immediate and deferred loading of data. A "Pupil" object has a reference to its School (possibly a null reference); when a Pupil is loaded, the School object is also instantiated. A "School" object has a Collection<Pupil> member; but the Pupil objects are only loaded when the collection is accessed. The differences show in aspects like when the EntityManager connection gets closed (it is important to close EntityManagers, otherwise they may act as resource leaks and may hold onto database connections).

Don't leak; close EntityManagers

One of the options in the "PupilServlet" presents details of a particular pupil whose enrolment number has been entered in a HTML form generated in the doGet method of this servlet. This option is handled in the following method invoked from the doPost method:

```
private void doIndividual(HttpServletRequest request,
       HttpServletResponse response)
    throws ServletException, IOException {
   String eNumStr = request.getParameter("enumber");

   if(eNumStr==null) {
        badData(response,
            "no enrolment number given for individual");
        return;
   }
   BigDecimal enrlNum = null;
   try {
       enrlNum = new BigDecimal(eNumStr);
   }
    catch(NumberFormatException nfe) {
       badData(response,"invalid enrolment number");
       return;
    }

    EntityManager em = null;
    em = emf.createEntityManager();
    Pupil p =  em.find(Pupil.class, enrlNum);
    em.close();
    if(p==null) {
          badData(response, " no such Pupil");
           return;
     }
     response.setContentType("text/html;charset=UTF-8");
     PrintWriter out = response.getWriter();
     out.println(…);
     …
     School s = p.getSchoolname();
     if(s==null) {  out.println("currently not enrolled"); }
     …
}
```

In the above function, the EnityManager (connection to database) can be closed a soon as the Pupil has been explicitly loaded; the associated School object has also been loaded and can be accessed later.

The following code fragment is from the servlet that displays information about schools. The form generated by the doGet method of this servlet has a HTML selection for the school name, a checkbox to indicate whether a listing of staff is required, and a multi-selection choice for listing details of class membership. These data are handled in this doPost method. Here the EntityManager object is only closed after its collection members have been accessed.

```
protected void doPost(HttpServletRequest request,
      HttpServletResponse response)
    throws ServletException, IOException {

 String schoolname = request.getParameter("sname");
 // Should get a null value for showstaff if checkbox not selected
 String showstaff = request.getParameter("showstaff");
 String[] classes = request.getParameterValues("classes");
 EntityManager em = null;

 em = emf.createEntityManager();
 School s = em.find(School.class, schoolname);
 // Assuming that all data will be valid input from form!
 // No checks!
 response.setContentType("text/html;charset=UTF-8");
 PrintWriter out = response.getWriter();
 out.println(…);
 …

 out.println("<p>Address: " + s.getAddress() + "</p>");

 Collection<Teacher> staff = s.getTeacherCollection();

 Teacher principal = null;
 for(Teacher t : staff)
   if("PRINCIPAL".equals(t.getRole())) { principal = t; break; }

   out.print("<br /><p>Principal: ");

   if (principal == null) {
     out.println("(to be appointed)</p>");
   }
   else {
     out.println(principal + "</p>");
   }

   …

  if(classes.length>0) {
     out.println("<br /><br />");
     Collection<Pupil> pupils = s.getPupilCollection();
            for(String aclass : classes) {
                …
            }
   }
   out.println("</body></html>");
   out.close();

   em.close();

}
```

Enterprise style!

EJBs? overkill?

!

If you did have to implement the "School Board Administration" application then your best choice would probably be a combination of servlets, JPA-style entity beans, and JSPs to generate response pages using data selected and then forwarded by the control servlets. The use of full Enterprise Java with session beans in an application server would be overkill. EJB solutions are really only appropriate when one has complex transactions involving more than one datastore (database and/or message queue).

Rich-client/EJB session bean in application server

However, if you wanted a "rich client interface", rather than HTML forms and response pages, then it probably would be worth considering a three tier EJB solution rather than the simple two tier Java SE/JPA-database solution illustrated earlier. (Of course, a "rich client" would use something better than the JOptionPane dialogs illustrated in the example – but that isn't material to this set of examples.)

One would define a "session bean" that had business methods supporting all the required queries and updates. This (stateless) session bean would be deployed in an EJB container and would be accessed by instances of the "rich client" application running on other machines.

Session bean to organize and systematize business code

One can even argue for use of the EJB session bean based solution in the context of a web-client implementation. In the servlet example presented in the previous section, the business code is scattered across many different servlets. If the business code is all in a session bean, maintenance and enhancements may well be easier. The web components are then simplified; the servlets take on their proper function of control components, the EJB session bean manipulates the entities and returns selected data that are forwarded for display by JSPs.

EJB session bean

A session bean and its entities

A new Enterprise Application project was created in Netbeans with its ejb component containing a stateless session bean along with entity beans once again generated from the Oracle tables. (Curiously, the code generator created definitions for these entity beans that differed from those generated earlier for the entities used with the SE environment and re-used in the servlets. The new definitions use Long instead of BigDecimal for all data members mapped from Oracle number fields. The reason for this discrepancy was not pursued.)

"Inject" an EntityManager

Of course, a session will use a JPA EntityManager to manipulate its entities. EJB beans are always "thread safe"; so the session bean has the EntityManager injected by the framework. (If there are multiple threads running, they will be using different

instances of the EJB stateless session bean. The framework will "inject" (initialize) each of these beans with its own private EntityManger before it invokes any business method.)

"Remote" and "Local" interfaces

In this example, it was intended that the session bean be used both with a re-written "rich client" and with re-written servlets. An application client supposedly invokes operations of the session bean using RMI invocation requests that are packaged for network transmission using CORA style IIOP messages. Such a client sees the "remote" interface defined by the session bean. Servlet clients differ in that they run in servlet containers that typically will be managed by the same Java virtual machine as runs the EJB container. Such servlet clients will see the "local" interface of the session bean. It is possible for a session bean to expose different functionality through its local and remote interfaces. (One quite common convention is for web style clients – and hence local interfaces – handle "customer" requests, while administrator control is effected through rich client applications using a remote interface.)

Common interfaces; "final" arguments

In this case, the business methods defined for the session bean were all published through both remote and local interfaces. (A development environment may generate warnings relating to such use. The semantics of local and remote invocations can vary with regard to the treatment of data passed as arguments. Remote invocations always follow "pass by value" conventions; local invocations work through pass by reference for arguments derived from java.lang.Object. In this case, all arguments were defined as `final` – so in effect pass by value was used consistently.)

Business interface

The common business interface (Remote and Local) for the session bean was defined as:

```
@Remote
public interface SchoolsSessionRemote {
    String[] getSchoolNames();
    void removeTeacher(final long empNum)
      throws ApplicationException;
    void removePupil(long enrlNum) throws ApplicationException;
    void addPupil(Pupil newPupil) throws ApplicationException;
    void addTeacher(Teacher newTeacher)throws ApplicationException;
    Pupil getPupil(final long enrlNum) throws ApplicationException;
    Teacher getTeacher(final long empNum)
      throws ApplicationException;
    List<Pupil> listPupils(final boolean onlyNonEnrolled);
    long countTeachers(final boolean onlyRelief);
    Collection<Teacher> listTeachersBySchool(
      final String schoolName);
    List<Teacher> listReliefTeachers();
    School getSchool(final String schoolName);
    void changeSchoolForPupil(final long enrolNum,
      final String newschool);
    void changeClassForPupil(final long enrlNum,
      final String classid);
    void changeSchoolForTeacher(final long empNum,
      final String schoolName);
```

```
    String changeClassForTeacher(final long empNum,
      final String role);
}
```

Some of these methods return instances of School, Teacher, and Pupil to the client so that details may be displayed. Now a Teacher object contains a reference to its School, while a School object contains a collection of Teachers and a collection of Pupils. Clearly, one wouldn't wish to retrieve across the network the complete school enrolment if simply desiring details of a Teacher. There are a few subtleties here with regard to the data that are actually returned to a client.

The update methods, like changeClassForTeacher, take simply the identifier of the Teacher and details of the data that are to change. The code in the session bean deals with all the mechanics of the change; this may involve updating more than one record in the database (e.g. a teacher being assigned to a school class may be replacing another teacher whose record must change to no class assignment).

A class ApplicationException was defined for the project. It is simply an exception subclass used to return error messages such as warnings of invalid employment numbers.

Session bean and sample methods

The bean definition is:

"Inject" the EntityManager

No need to provide a transaction – that has been done by the EJB container

```
@Stateless
public class SchoolsSessionBean
    implements SchoolsSessionRemote, SchoolsSessionLocal {
    @PersistenceContext(unitName="EESchools3-ejbPU")
    private EntityManager em;

    public String[] getSchoolNames() {
        Query q = em.createQuery(
            "Select s.schoolname from School s");
        List<String> schoolNames = q.getResultList();
        String[] schoolArray = (String[])
             schoolNames.toArray(new String[0]);
        return schoolArray;
    }

    public void removeTeacher(final long empNum)
      throws ApplicationException {
        Teacher t = em.find(Teacher.class, empNum);
        if(t==null) {
           throw new ApplicationException(
            "No teacher with employment number " +
            empNum);
        }
        em.remove(t);
    }

    …

    public void addPupil(Pupil newPupil)
      throws ApplicationException {
        try {
```

```
            em.persist(newPupil);
        }
        catch(EntityExistsException eee) {
            throw new ApplicationException(
      "There is already a pupil with that enrollment number");
        }
    }

    …

    public Teacher getTeacher(final long empNum)
      throws ApplicationException {
        Teacher t = em.find(Teacher.class, empNum);
        if(t==null) throw new ApplicationException(
            "No such teacher");
        return t;
    }

    …

    public String changeClassForTeacher(final long empNum,
      final String role) {
        Teacher t = em.find(Teacher.class, empNum);
        // There may already be a teacher in this role at this
        // school. Clear any entry
        School s = t.getSchoolname();
        Teacher existing = getTeacherByRole(s, role);
        String report = null;
        if (existing == null) {
            report = t.getEmployeenumber() +
                  " will fill vacant position";
        } else {
            report = "Replacing current incumbent " +
                  existing.toString();
        }
        if (existing != null)
              existing.setRole(null);
        t.setRole(role);
        return report;
    }

}
```

An EntityManager can be injected using the PersistenceContext annotation. This refers to a persistence unit (XML file) that has been defined as a part of the EJB component of the enterprise application.

The example getTeacher method will return a Teacher object that has a member that is an instance of class School. This School object will be returned as part of the Teacher. It will have its name and address (String) fields set, but its two collection members (the list of all teachers employed at that school, and the list of all pupils) will both be invalid. (I had hoped that they would be null; seems that isn't so. They contain something that causes your application to die if you touch it. Yuk.) This is more or less what one would want – one gets the data needed to display employment details for an individual teacher.

If the client requests a School object for display then, in this particular implementation of the application, it wants all information about the school –

including Teachers and Pupils. (If one only wanted the address, then another method should have been added to the session bean that would return just the address string.) The code for getSchool must force the collections to be instantiated; the code used is as follows:

Force loading of collection members when needed

```
    public School getSchool(final String schoolName) {
        School s = em.find(School.class, schoolName);
        // Have loaded in school, but not the teacher list
        // or pupil lists
        // "Touch" these before trying to serialize and return
        // the School object
        int countEmployees = s.getTeacherCollection().size();
        int countPupils = s.getPupilCollection().size();
        // Hope no optimising compiler removes that work

        return s;
    }
```

(One could change the entity bean definition so that the collections weren't "lazy loaded". But one wouldn't want that! Every time you touched a school object you would load all the pupils.)

Using the EJB session bean in a client application

Three-tier: client, EJB application server, database

In this three-tier model, the client application no longer directly accesses the database so it does not require any reference to an EntityManagerFactory or EntityManager. Instead, it has to be initialized with a reference to the stateless session bean that it will employ. It will get a reference to an object that implements the "Remote" interface; the client is going to be sending requests across the network to the machine where the EJB application server is running.

Client runs in "application client container"

The client looks as if it is a free standing Java SE program; but it is not. It has to be run inside an application client container. This container has the framework code that loads appropriate configuration files that contain the details of the application server used, and which then "injects" these data into the application. If security controls were to be placed on some or all of the methods of the session bean, this application client framework code would also take responsibility for getting a user to enter name and password details at application launch time.

Deployment details hidden in configuration files!

In a typical learning environment, client and server will run on the same machine and will most likely be run using control features in the Netbeans or other IDE. However, provided one copies all the application-client container jar files etc, it is possible to deploy a client onto another machine and then to run the application client system using an ant script. Unfortunately, details of where the server runs are fairly well hidden. These details are in one of the configuration files used by the client application container – you just have to dig until you find them and then edit them in a text editor to set hostname and port number etc.

```
public class Main {

    @EJB
    private static SchoolsSessionRemote schoolsSessionBean;
    private static final String classIDS[] = {
```

Inject (framework initializes) reference to session bean

```
            "K", "1", "2", "3", "4", "5", "6"
        };
        private static String[] schoolArray;

        private static void initalizeSchoolsArray() {
            schoolArray = schoolsSessionBean.getSchoolNames();
            for (String s : schoolArray) {
                System.out.println(s);
            }
        }

    …

    private static void listReliefTeachers() {
            List<Teacher> reliefs =
                schoolsSessionBean.listReliefTeachers();
            if (reliefs.size() == 0) {
                System.out.println("There are no relief teachers");
            } else {
                for (Teacher t : reliefs) {
                    System.out.println(t);
                }
            }
        }

        private static void countTeachers(boolean onlyRelief) {

            long res = schoolsSessionBean.countTeachers(onlyRelief);
            if (onlyRelief) {
                System.out.println(
                      "There are " + res + " relief teachers");
            } else {
                System.out.println(
                      "The department employs " + res + " teachers");
            }
        }

    …
        public static void main(String[] args) {
            initalizeSchoolsArray();
            interact();
        }
}
```

Request that bean returns data

Use of the session bean from servlets

Multi-tier: browser, web-app, EJB-app, database

The servlets can be rewritten to utilize the session bean instead of using an EntityManager as an intelligent connection to the database. The resulting architecture then has a browser client, a web application layer with the servlets (and the JSPs that would in practice be used for final output of response pages), an EJB application layer in the application server, and the database.

Local or Remote interfaces?

It is possible to configure the overall application so that the web-app layer and the EJB-app layer run on different machines. (Such a deployment might be appropriate in a business environment because it permits additional firewalling of the core business applications.) If these layers are on separate machines communicating across the network, then the servlets will use the “remote” interfaces for the session

bean just like the application client illustrated in the previous subsection. The simpler deployment keeps the web-layer and EJB layer distinct but runs them in the same JVM. In this case, the servlets will use the local interface for the session bean. (If the remote interface were used, the system would go to all the work of preparing IIOP requests for transmission on TCP/IP networks and then just unpack these request messages to extract the data. Local invocations eliminate that network encoding overhead.)

Each servlet will need to be re-written. It will now get a reference to a session bean injected (rather than an EntityManagerFactory), and requests will be delegated to the session bean.

Inject reference to session bean

```
public class SchoolsServlet extends HttpServlet {
    @EJB
    private SchoolsSessionLocal schoolsSessionBean;

    protected void doGet(HttpServletRequest request,
       HttpServletResponse response)
    throws ServletException, IOException {
```

Use bean

```
        String[] schools = schoolsSessionBean.getSchoolNames();
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        out.println(
            "<html><head><title>Information on schools</title>");

            Code to produce HTML data entry form including a HTML
            select with school names from schools[] as options

        out.println("</body></html>");
        out.close();
    }

    protected void doPost(HttpServletRequest request,
       HttpServletResponse response)
    throws ServletException, IOException {

       String schoolname = request.getParameter("sname");
       String showstaff = request.getParameter("showstaff");
       String[] classes = request.getParameterValues("classes");
```

Use bean; collect School along with its Teachers and Pupils

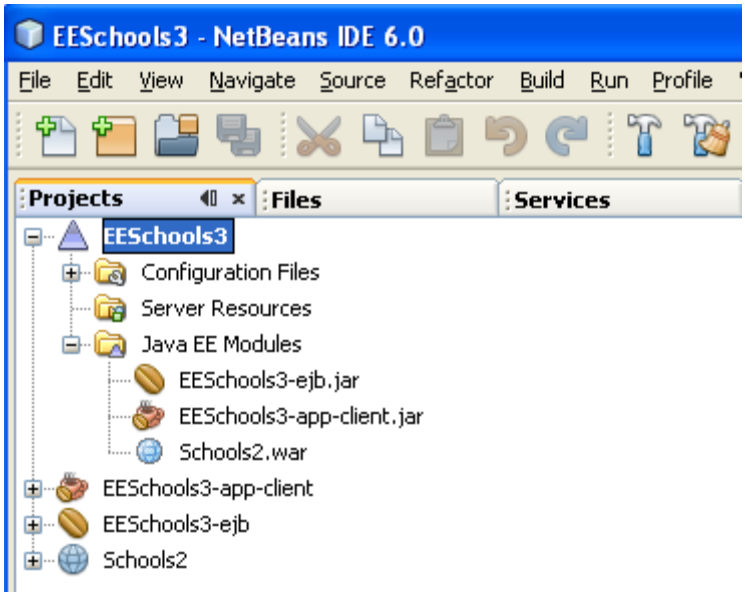
```
       School s = schoolsSessionBean.getSchool(schoolname);
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        …
        out.println("<h1>" + schoolname + "</h1>");

        out.println("<p>Address: " + s.getAddress() + "</p>");

        Collection<Teacher> staff = s.getTeacherCollection();

        Teacher principal = null;
        for(Teacher t : staff)
            if("PRINCIPAL".equals(t.getRole()))
                  { principal = t; break; }

        out.print("<br /><p>Principal: ");

        if (principal == null) {
```

```
            out.println("(to be appointed)</p>");
        }
        else {
            out.println(principal + "</p>");
        }

        if(showstaff !=null) {
            …
        }

        if(classes.length>0) {
            out.println("<br /><br />");
            Collection<Pupil> pupils = s.getPupilCollection();
            for(String aclass : classes) {
                out.println("<h2>Enrollment in class " +
                  aclass + "</h2>" );
                out.println("<ul>");
                for(Pupil p : pupils)
                    if(aclass.equals(p.getClasslvl()))
                        out.println("<li>" + p + "</li>");
                out.println("</ul>");
            }
        }
        out.println("</body></html>");
        out.close();
    }
}
```

Of course, in a real implementation, the servlet would actually forward the School object (along with its collection of Teachers and Pupils) to a JSP that would handle display generation.

Enterprise application

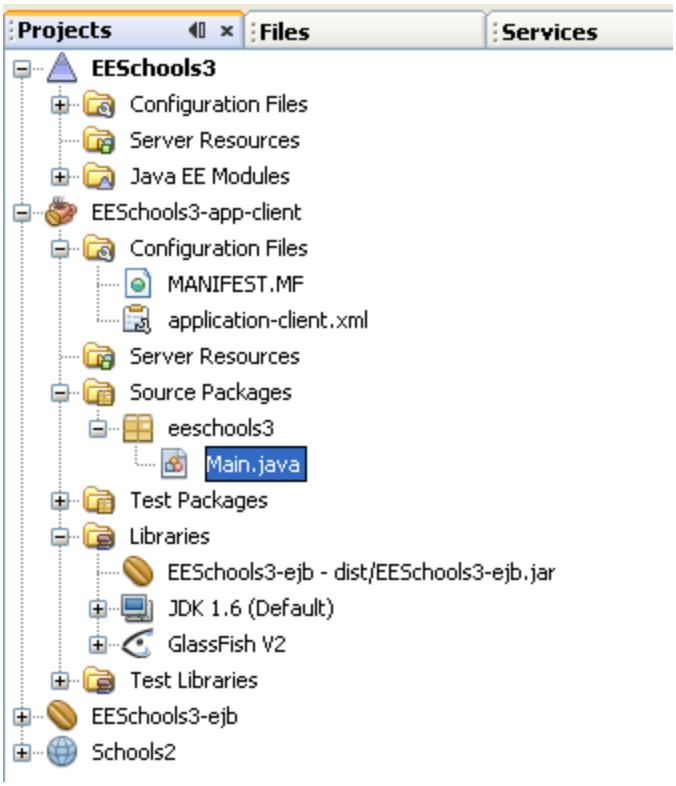
The creation of an enterprise application with its ejb-jar file, its web-app war file, and its application client jar files is a tedious and painful task. Fortunately, this task largely automated by IDEs such as Netbeans. The structure of the application does however remain fairly complex.



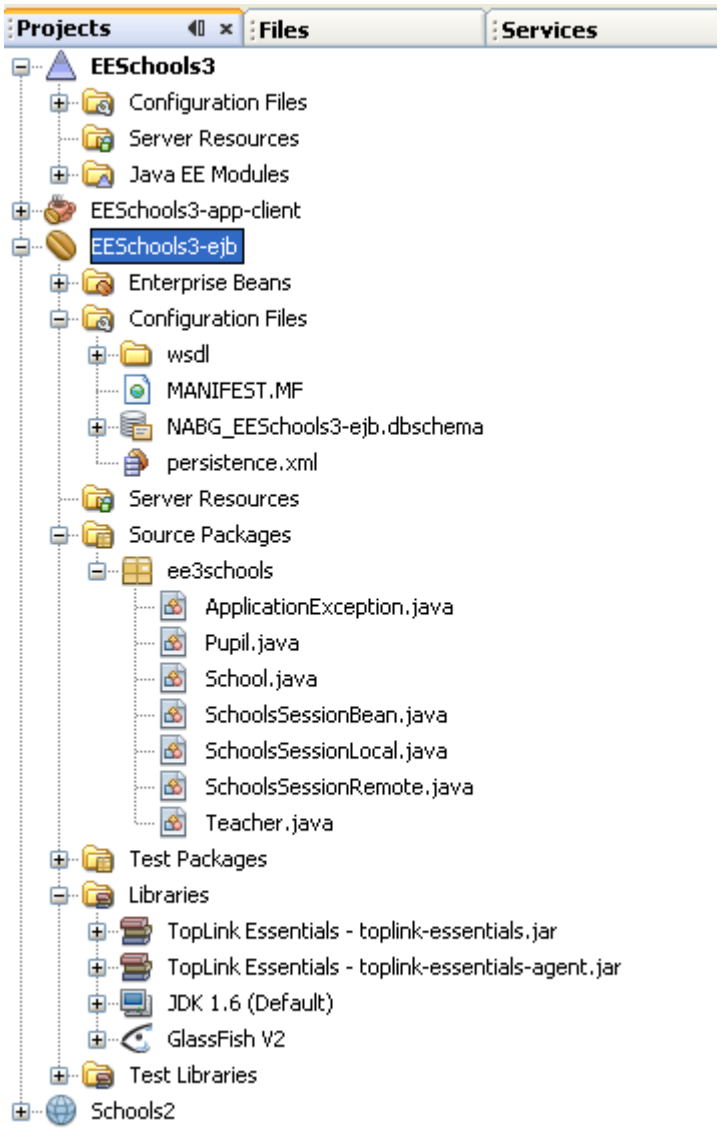
For this example, the Netbeans IDE was used to create firstly a new “Enterprise Project” with just the EJB framework and an application client. The web client components were added once the session bean has been defined and tested via the application client.

As shown in the screen shot above, Netbeans uses four “projects”. There is the overall EESchools3 enterprise project; this is really for deployment consisting of little more than the jar (and war) files generated from the associated projects.

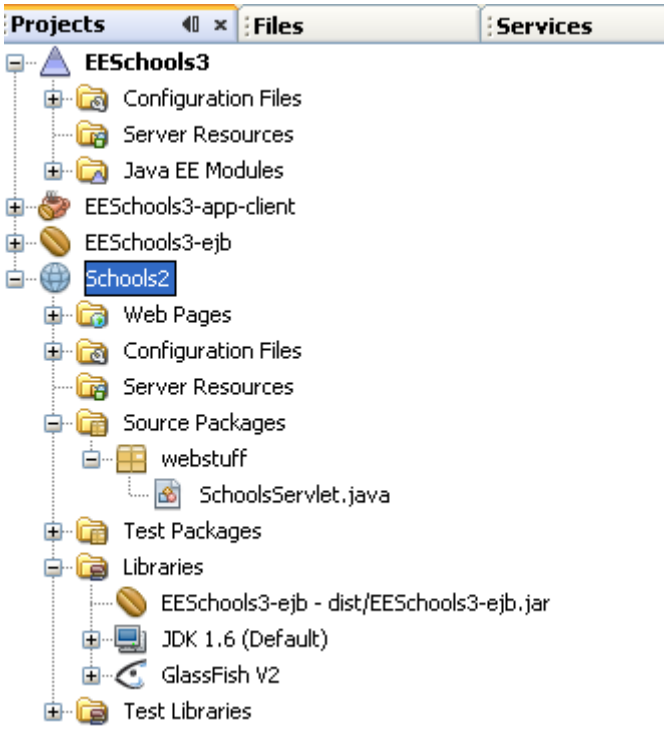
The EESchools3-app-client project contains the Main defined for the client; it needs the GlassFish libraries (code needed to submit requests to the server etc) and the libraries with the definitions of the remote interface of the session bean (and also the application defined exception class).



The ejb project (shown below) contains the code for the stateless session bean and the entity classes. As mentioned above, the entity classes were auto-generated from the table definitions in the Oracle database (copies of the database schema meta-data that were used get added to the configuration component of the project). Once they had been generated, the entity classes underwent some minor editing to add things like useful toString methods and the odd extra data access method. A persistence.xml file, containing details of the database etc, has to be created. This project requires the Toplink libraries along with the Glassfish libraries. (The database driver jar file does not need to be added to the project. When the persistence.xml file is created, the dialogs require details of a JDBC resource defined in the application server. The Netbeans system processes these dialogs and will add the database driver to the application server's own classpath while it is setting up JDBC connection pools and resources.)



The web project contains the definition of the servlets (also JSPs etc). It requires the Glassfish library and the jar file with the definition of the local interface to the session bean.



Concurrent transactions

Funds transfer application

This section uses a different example – banking. The problem that is to be explored here is that of how to handle banking transactions correctly. These transactions involve transfers of funds between accounts in a bank; no funds are added, no funds are removed. This simple model allows for an easy check on correct operation – the total funds in the bank should be invariant.

In practice, realising this simple and obvious consistency constraint proves somewhat problematic.

Database "isolation" levels

The example will be introduced first in the context of JDBC applications that utilize database features that support correct operation through the setting of "isolation levels". Unfortunately, different database vendors approach the problem in different ways and it is not practical to write a single version of the code that will operate with all databases. This obviously makes it very hard to implement a fully automated object-relational mapping system based on isolation levels.

Versioning of database records

Consequently, JPA uses a different approach. It accepts that there could be interfering transactions, but uses a simple versioning mechanism to reveal interference. If two transactions interfere, one succeeds while the other recognizes that it failed but can be retried. This versioning approach is illustrated first with a simulation using standard JDBC and then with Java-SE and EJB code based on JPA.

Finally, there is a brief example illustrating distributed transactions involving multiple databases – again implemented using JPA but now with "XA-aware" database connections. This example works quite well (whereas its realization by

setting isolation levels and without the versioning trick is both difficult to set up and runs extremely slowly.)

Concurrent transactions and database isolation levels

Funds transfer application

The banking application works with a very simple "accounts" table in the database – an account record requires only an identifier and a balance (both integers). Clients will submit transaction requests for funds transfers between accounts. A request should be refused, and an application defined "overdrawn" exception should be thrown, if a requested transfer would result in an account being overdrawn. Otherwise, the requested transfer should be completed by updating the balance fields in the two accounts involved. The tables define one thousand accounts each initially having a balance of $1000. Clients select two distinct accounts for any given transfer; selections are based on a random number generator; a transfer involves an integral number of dollars in the range 1..333.

There can be many transactions operating concurrently; these transaction requests being submitted over different database connections either by different client processes or by different threads within a single client process. (Although less realistic as a model of actual operations, a test may be easier to deploy with a multi-threaded client instead of many concurrent client processes. Each thread of a multi-threaded client uses a separate connection.) Each client runs code of the following form:

Transaction code

```
attemptTransfer(fromAccountID, toAccountID, amount)
{
begin transaction; // implicit with JDBC
fromBalance = load balance in "fromAccount" from database;
toBalance = load balance in "toAccount" from database;
if fromBalance < amount {
     "rollback transaction"
     throw "overdrawn exception";
     }
newFromBalance = fromBalance - amount;
newToBalance = toBalance + amount;
update "fromAccount" set balance = newFromBalance;
update "toAccount" set balance = newToBalance;
commit transaction;
}
```

Of course, it is possible that two concurrent transactions might involve a common account. Thus one might have the following situation:

```
Initial balance account 100 = $1000
Initial balance account 200 = $1000
Initial balance account 300 = $1000
Total funds in affected accounts = $3000
```

```
Process 1                                Process 2
attemptTransfer(100, 200, $500)          attemptTransfer(200,300, $500)
    fromBalance = 1000                       fromBalance = 1000
    toBalance = 1000                         toBalance = 1000
    not overdrawn                            not overdrawn
    newfromBalance = 500                     newfromBalance = 500
    newToBalance = 1500                      newToBalance = 1500
    update “from account”t#100               update “from account”#200
    update “to account”#200                  update “to account”#300

Expected outcome:
Final balance account 100 = $500
Final balance account 200 = $1000
Final balance account 300 = $1500
Total funds in affected accounts = $3000
```

Interfering transactions - consistency violation

```
Probable outcome:
Final balance account 100 = $500
Final balance account 200 = $500
Final balance account 300 = $1500
Total funds in affected accounts = $2500
```

Aren’t transactions “ACID” – atomic, consistent, isolated, and durable?

ACID?

Unfortunately, if one doesn’t go to a lot of work, database transactions are not “ACID”. The implementers of most database systems live by the credo “*live fast, die young, and leave a good looking corpse*.” The default settings for databases allow for rapid processing but can yield incorrect results.

The ACID properties of transactions were first characterized by Haërder and Reuter in 1983. Durability is easy – assuming you have your database on a reliable RAID disk configuration. Consistency should follow automatically if you get atomicity and isolation correct – the data in the table should move from one consistent state to another. Consistency is of course application defined; very simply defined in this banking application as it requires merely invariance in the total funds in the bank. Atomicity requires that all operations be completed; in this example, there are two rows that must be changed by a transfer operation – either both are changed or neither is changed. Database systems can achieve this by making alterations to “shadow” records and then, on commit, using these to overwrite the real records. Problems seem mainly to arise in relation to isolation.

Isolation levels: read uncommitted, read committed, repeatable read, serializable

Four “isolation levels” were defined for database systems – “read uncommitted”, “read committed”, “repeatable read”, and “serializable”. These isolation levels were defined in terms of undesired phenomena that they could prevent.

“Read uncommitted” is useless. It would in effect allow a transaction to see data in the shadow records created for another transaction – data that were in inconsistent and incomplete states. (It is possible that “read uncommitted” might be very fast – but if you don’t require correct results speed is generally easy.)

“Read committed” is the default isolation level for almost all database systems; transactions cannot access data in the shadow records – only data that have been

committed by those transactions completed earlier. "Read committed" prevents the phenomenon of a "dirty read".

"Repeatable read" requires a guarantee that if a transaction were to read some data record at its start, and then re-read that record later in the context of the same transaction, then it would encounter the same values i.e. no other transaction is allowed to change those data between the initial read and the final commit. "Repeatable read" prevents the phenomenon of non-repeatable reads.

The highest isolation level, "serializable", requires that a transaction not be affected by any additions or deletions of rows effected concurrently. It is defined as preventing the phenomenon of a phantom read.

"Undesired phenomena" and real queries

The definition of isolation levels in terms of undesired phenomena is somewhat obscure. In practice, if you are updating records then you will require something approximating "repeatable read' – you need a guarantee that no other transaction will change the records that you are processing during the time of your transaction. Essentially, you have to acquire update locks on the rows that you want to change. You would really only need the more demanding "serializable" level if the correct operation of your transaction depended on some count or an aggregate property of the data; a count or aggregate will change if records are added or deleted. For serializable to work, you would need to lock at least a range of records in the table if not the entire table.

Example needs something equivalent to repeatable read

The transaction code shown above doesn't appear to re-read any data – the records are read once, decisions are made, and new values are written to the respective records. But that code does require that there be no changes to the records – for if there have been changes then the calculated value for the new balance is wrong. Consequently, that code requires that the transaction be at a level equivalent to repeatable read.

This simple case could have exploited database abilities and avoided problems!

This very simple example can be recoded to do the work in the database and so avoid all these concurrency issues. If the update was encoded to use SQL like:

```
update "fromAccount" set balance = balance - amount;
update "toAccount" set balance = balance + amount;
```

then there would be no problems with consistency. However, that encoding could result in overdrawn accounts! The check for an overdrawn account is made before the account is updated – but it is possible that another concurrent transaction is also making a withdrawal. Even that problem can be fixed by placing data constraints in the table definition (balance>=0); such a constraint would result in a SQLException if the transaction would have left an account overdrawn. However, such simple fixes are not generally available and for the most part one must work by setting the isolation level correctly.

Setting the correct isolation levels is where the problems start. In principle, the task is easy – the work is done when the JDBC calls are made to establish a database connection (isolation levels are defined for a connection):

Set isolation level for connection

```
private static Connection getConnection() {
```

```
     Connection dbConnection = null;
     try {
          Class.forName(dbDriverName);
          dbConnection = DriverManager.getConnection(
               dbURL, dbUsername, dbPassword);
          dbConnection.setAutoCommit(false);
          dbConnection.setTransactionIsolation(
               Connection.APPROPRIATETRANSACTIONLEVEL!);
     }
     catch (Exception e) { … }
     return dbConnection;
}
```

The appropriate transaction level will depend on the database.

Database systems claim to support different isolation levels; further, each tends to have its own vendor specific slant on the meanings of these levels. Oracle supports only “read committed” and “serializable” isolation levels. DB2 supports “read committed”, “repeatable read”, and “serializable”. So, at least in principle, one should set up the test application to use “TRANSACTION_SERIALIZABLE” connections to an Oracle database and “TRANSACTION_REPEATABLE_READ” connections for DB2.

Test application

The basic test application has the form:

```
public class Bank implements Runnable {
  private Connection dbConnection;
  private PreparedStatement pGet;
  private PreparedStatement pUpdate;
  …
  public void main(String[] args) {
     setupTable(); // Re-initialize 1000 accounts
     int nThreads = 5;
     Thread[] threads = new Thread[nThreads];
     for (int i = 0; i < nThreads; i++) {
          Bank aBankConnection = new Bank();
          Thread t = new Thread(aBankConnection);
          threads[i] = t;
     }
     long startTime = System.currentTimeMillis();
     for (int i = 0; i < nThreads; i++) {
          threads[i].start();
     }
     for (int i = 0; i < nThreads; i++) {
          try {
               threads[i].join();
          } catch (InterruptedException ie) { }
     }
     long endTime = System.currentTimeMillis();
     performanceReport();
    }
  }
}
```

This version is multi-threaded; multiple instances of the process can be run to increase the level of concurrent demand on the database. However, as noted later, clients accessing an EJB session bean that is performing the transfers cannot obtain

distinct connections and so those clients are single threaded and scripts must be used to launch multiple concurrent client processes.

Each instance of the Bank class establishes its own connection with the correct isolation level and then creates its own instances of PreparedStatements. The basic form for these is:

```
pGet = dbConnection.prepareStatement(
     "select balance from vaccounts where ident=?");
pUpdate = dbConnection.prepareStatement(
       "update vaccounts set balance=? where ident=?");
```

The run() method defined for the bank class has a loop that is iterated 2000 times. Each iteration picks at random two distinct accounts and an amount to be transferred between these accounts. The actual transfer is effected using code based on the outline given above. Counts are kept of the number of attempted transfers that result in overdrafts and the numbers that are aborted for various reasons.

10,000 requests – how many work?

A test run of the program submits 10,000 transactional requests. Because of scheduling and cpu constraints etc, the effective degree of concurrency is always less than the number of threads or client processes. Typically, there may be two sometimes three transactions actually being processed concurrently in the database engine. Only a small fraction of these would be expected to interfere. There will be a few cases where two pairs of identifiers, selected from the same set of 1000, will have an element in common – for these there will be interference. In principle, there is the possibility of genuine deadlock situations though these will be very rare. One transaction might transfer funds from account #100 to account #200, while another is trying to transfer funds between the same two accounts but in the opposite direction. In a full locking scheme, each of those transactions might acquire just one of the two records and be forced to wait in deadlock for the other. (Such problems can be reduced by accessing and locking records in order of primary key.)

Oracle TRANSACTION_ SERIALIZABLE. That is NOT the right way!

Running at “TRANSACTION_SERIALIZABLE” level with Oracle is a disaster. The majority of transactions are aborted. In a typical run, ~7800 of the ten thousand transactions are aborted as “non-serializable”, ~20 deadlocks occur, and there are ~20 overdraft occurrences among the 2000+ completed transactions. (Most of the successful transactions are completed by either the first thread or the last thread – these tend to be running in isolation.) The constraint of invariant total funds is violated! The problem is that SERIALIZABLE is simply inappropriate in this situation.

DB2 REPEATABLE_ READ? Workable but not happy

DB2’s “REPEATABLE_READ” does work – though it isn’t too happy. The report from an example run is:

```
Final total $1000000
Number of requests 10000
Number of overdraft 'rollbacks' 482
Number of aborted requests 0
Deadlock count 35
Time: 51818 ms
```

The invariance condition for total funds is satisfied, but there is an unexpectedly high number of "deadlocks" and these have to be "timed-out" which resulted in the rather long runtime of 58 seconds. DB2's approach is based on locking. When the transaction code "selects" a record, a read lock is placed on that record. Other transactions can also read the same record. When the transaction reaches the SQL update, DB2 must upgrade the read lock to a write lock. No other transaction using the affected record would start; if there were any read-only transactions using the record they could complete. This acquisition of write locks has to be done for both records. Basically, deadlocks are being reported for every case where there are records in common between two transactions.

The example doesn't run well – because neither database system expects such updates to be handled in this straightforward way. Each employs a non-standard SQL "tweak" to specify the appropriate handling.

SELECT FOR UPDATE

With both Oracle and DB2 one can add an extra "FOR UPDATE" qualifier to a SQL "SELECT" statement. When the database engine handles a "FOR UPDATE" request it places an "intention write lock" on the relevant record(s). The database will automatically prevent any other "FOR UPDATE" transaction from starting until this intention write lock is cleared. If there were other concurrent transactions that required only read access to READ_COMMITTED data they could run. When the SQL update statements are reached, the intention write locks must be upgraded to write locks. At this point other readers can complete. When the transaction has fully locked its rows it can perform the updates. Unfortunately, although both Oracle and DB2 have this feature, they implement it differently.

Oracle – READ_ COMMITTED, select for update

Oracle requires that the transaction isolation level be set at READ_COMMITTED with the SQL select statements specifying FOR_UPDATE.

```
pGet = dbConnection.prepareStatement(
     "select balance from vaccounts where ident=? for update");
```

This works! The report from a typical run is:

```
Final total $1000000
Number of requests 10000
Number of overdraft 'rollbacks' 511
Number of non-serializable requests 0
Time: 96827 ms
```

The invariance of total funds is satisfied, there are no deadlocks. It is a nicely ACID system.

DB2 – REPEATABLE_ READ, select for update

DB2 doesn't understand "FOR_UPDATE" in associated with READ_ COMMITTED, but it does appreciate the significance of this qualifier with REPEATABLE_READ. It places intention locks on the records with the select operations, so stopping other potential update transactions. The transaction completes, its locks are released, and waiting transactions can resume. Without all those "deadlocks" that DB2 got previously when trying to update locks, it runs faster. A report for a typical run is:

```
Final total $1000000
```

```
Number of requests 10000
Number of overdraft 'rollbacks' 587
Number of aborted requests 0
Time: 14321 ms
```

Database specific tweaks? There must be a better way

Each database requires its own specific combination of isolation level and tweaked SQL statements. Generally, such configurations are impossible with auto-generated persistence systems. While EJB engines allow one to set the isolation level for connections from a given database's connection pool, they don't provide any mechanism for tweaking the SQL. One could in this case create a workable though non-optimal DB2 system, but one wouldn't be able to configure an Oracle connection pool correctly for auto-generated persistence. Clearly, there has got to be a better way that will work effectively with auto-generated persistence schemes.

(The times for the Oracle and DB2 systems are not directly comparable. The Oracle database, 10g version 10.2, ran on an elderly SunFire 480, while the DB2 developer edition v9.1 ran on a Windows PC. Neither database was configured for optimal transaction performance. However, these times are useful when comparing the standard JDBC "isolation level" approaches with the times for alternative mechanisms, such as the use of JPA, when used with the same database.)

"Manual" versioning of database records

Rather than relying on database locks, one can redefine the database table and recode the application so that it is the application that is responsible for resolving concurrency issues. The database and its idiosyncrasies are ignored – it is just left to run in its favoured READ_COMMITTED isolation level.

Version field in record

The data table will require an additional "version" field under application control:

```
create table bankaccount (
     ident integer primary key,
     balance integer,
     version integer );
```

The select statements are modified to read both current balance and current version number:

```
pGet = dbConnection.prepareStatement(
     "select balance, version from vaccounts where ident=?");
```

Version and balance are read from the ResultSet returned by running such a query:

```
pGet.setInt(1, from);
rs = pGet.executeQuery();
if (rs.next()) {
     balanceFrom = rs.getInt(1);
     versionFrom = rs.getInt(2);
}
else { … }
```

The checks for overdrawn accounts and calculations of new balances are done as before.

Using the version number in the update

The version number is utilized when a record is updated. The record is being changed, so its version number should be changed along with any other data fields (in this example, the "balance" field). Of course, one only wants to update the record if it hasn't been messed with by some other transaction; if it has been changed by another transaction, one wants this transaction to fail. Consequently, the update code identifies the desired record by combination of its ident (primary key) field and its version number.

```
pUpdate = dbConnection.prepareStatement(
     "update vaccounts set balance=?, version=version+1" +
            " where ident=? and version=?";);
```

The actual code that updates the two affected accounts is along the following lines:

```
//Update 'from' account
pUpdate.setInt(1, newBalanceFrom);
pUpdate.setInt(2, from);
pUpdate.setInt(3, versionFrom);
int updatedFrom = pUpdate.executeUpdate();

//Update 'to' account
pUpdate.setInt(1, newBalanceTo);
pUpdate.setInt(2, to);
pUpdate.setInt(3, versionTo);
int updatedTo = pUpdate.executeUpdate();

// Did they get updated?
if ((updatedFrom != 1) || (updatedTo != 1)) {
     // No! Try again later
     interfereRollbackCount++;
     try {
          dbConnection.rollback();
     } catch (SQLException sqle) {… }
     return;
}
dbConnection.commit();
```

Updates succeed only if the records are unchanged since start of transaction

If the record hasn't been changed, it will be in the data table with its ident and version fields the same as when it was read. This record will be found and updated, and the executeUpdate operation will return a count of 1 for number of records updated. If some other transaction has changed the record, the database will simply not find a record with the specified combination of ident and version number and so the count of records updated will be zero.

The application can check that both rows were updated. If they weren't the transaction is formally rolled back. If the updates were successful, the transaction is committed and the data are changed in the database.

The same code should work for any database. In these experiments, a typical test run against Oracle resulted in the report:

Works fine with Oracle

```
Final total $1000000
Number of requests 10000
Number of overdraft 'rollbacks' 502
```

```
Number of non-isolated changes rolledback 37
Number of non-serializable requests 0
Time: 81411 ms
```

Works fine with DB2

While for DB2, typical results are:

```
Final total $1000000
Number of requests 10000
Number of overdraft 'rollbacks' 544
Number of non-isolated changes rolledback 45
Number of aborted requests 0
Time: 11028 ms
```

It's faster!

This encoding of the application runs faster than that using isolation levels and "for update" SQL hacks. For both databases, it is about 20% faster. The gain in speed results from the lack of any locking being done in the database.

Data traffic

The data traffic is another useful measure of performance. The actual data volume in these small tests is insignificant but the number of packets and total data transfer are both relevant to considering how these applications might scale up. For Oracle, there isn't much difference between the isolation-locking and version styles of coding; in both cases, ten thousand transactions require ~100,000 packets totalling ~13Mbyte. DB2 favours the version style; for 10,000 transactions, the isolation-locking style code generated ~180,000 packets totalling 26Mbyte while the version style code used only 98,000 packets at 19Mbyte.

Failed requests

The version style code does result in about 0.4% of the requested transactions being failed for interference. (In a more realistic application, the frequency of interfering transactions would likely be a lot smaller; after all a real bank should have a million accounts not one thousand so reducing the chance that any two concurrent transactions would actually have a common account.) In any case, these transactions can simply be retried immediately.

This "version number" trick seems to be a very good trick!

Java-SE JPA version of banking application

JPA relies on the versioning approach. Database records should have a version field; this is identified by the @VERSION annotation in the generated JPA entity class definition.

JPA entity class with Version column

```
@Entity
@Table(name = "VACCOUNTS")
public class Vaccounts implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Column(name = "IDENT", nullable = false)
    private Integer ident;
    @Column(name = "BALANCE")
    private int balance;
    @Version
    @Column(name = "VERSION")
    private int version;
```

OptimisticLock Exception

The JPA library (toplink) incorporates all the code needed to work with the version fields. If a conflict is detected when a transaction gets committed, the JPA code throws an "OptimisticLockException". This gets returned, inside a RollbackException, from the transaction.commit() operation.

Naturally, use of JPA simplifies the application coding. In the Java-SE application (multi-threaded application using JPA to directly manipulate the database) an EntityManagerFactory is created in main() and is used in the Bank() constructor to create a separate EntityManager that will be used by a specific thread. The coding for the attemptTransfer operation is now along the following lines:

```
try {
     em.getTransaction().begin();
     Vaccounts fromAccount = em.find(Vaccounts.class, from);
     Vaccounts toAccount = em.find(Vaccounts.class, to);

     int balanceFrom = fromAccount.getBalance();
     int balanceTo = toAccount.getBalance();
     if (balanceFrom < amount) {
          overdraftCount++;
          … // logging code for counts etc
          em.getTransaction().rollback();
          return;
     }
     balanceFrom -= amount;
     balanceTo += amount;

     fromAccount.setBalance(balanceFrom);
     toAccount.setBalance(balanceTo);
     em.getTransaction().commit();

     … // logging code for counts etc
} catch (RollbackException e) {
     Throwable t = e.getCause();
     if (t instanceof OptimisticLockException) {
          interfereRollbackCount++;
          … // logging code for counts etc
     } else {
          … // logging code for counts etc
     }
}
```

The code is of course database independent.

Working for a single multi-threaded application

The code works. For Oracle, a typical test run yields the following performance data:

```
Final total $1000000
Number of requests 10000
Number of overdraft 'rollbacks' 498
Number of non-isolated changes rolledback 111
Number of non-serializable requests 0
Time: 83889 ms
```

While for DB2 the results are like:

```
Final total $1000000
```

```
Number of requests 10000
Number of overdraft 'rollbacks' 533
Number of non-isolated changes rolledback 138
Number of aborted requests 0
Time: 15519 ms
```

(These data were obtained with the logging level for the toplink libraries changed to SEVERE. The default INFO logging level results in excessive log data that slow the application.) The JPA isn't quite as efficient as the hand coded JDBC "manual version" system; in particular, its approach leads to rather more conflicts.

It's broken! ??

However, it all seems to fall apart if one tries running two multi-threaded processes simultaneously (two processes, ten threads total, 20,000 requests). Thus, for Oracle the following are typical results (DB2's results show the same effect):

```
Process-1
     Final total $1000000
     Number of requests 9999
     Number of overdraft 'rollbacks' 342
     Number of non-isolated changes rolledback 5311
     Number of non-serializable requests 3
     Time: 81068 ms
Concurrent process-2
     Final total $1000000
     Number of requests 10000
     Number of overdraft 'rollbacks' 312
     Number of non-isolated changes rolledback 6524
     Number of non-serializable requests 0
     Time: 61083 ms
```

The majority of the transactions appear to interfere.

Effects of caching

This is an impact of "caching" in the JPA architecture. By default, JPA maintains a cache of the entities that it reads from the database. These caches are in effect associated with the EntityManagerFactories. All EntityManager objects created from the same factory share a single cache.

Within the single multi-threaded application the cache works well. The threads utilized the cache to reduce work. The cache will contain instance of entities (complete with version numbers) as they were when last written to the database. If the cache contains a desired entity, it is not reloaded from the database because it is assumed that it couldn't possibly have changed. Unfortunately, if there are other processes also accessing the database, this assumption is wrong. So in the two process example, many commit attempts fail because the record in the database has been changed by the other process.

Disable caching?

If you are confident that only one process should be using a particular table, then the caching helps. However, if there are multiple processes, it is necessary to disable caching (this involves setting an extra property field in the persistence.xml file).

Of course, disabling caching will have a performance impact. (The cache is there after all to improve performance). However, in this example, the impact in terms of overall run-time is not significant.

Oracle – no cache

The results for a typical test run with Oracle (single process, five threads, ten thousand requests total, cache disabled) were:

```
Final total $1000000
Number of requests 10000
Number of overdraft 'rollbacks' 558
Number of non-isolated changes rolledback 95
Number of non-serializable requests 0
Time: 83381 ms
```

With two processes running concurrently, the results were:

```
Process-1
     Final total $1000000
     Number of requests 9997
     Number of overdraft 'rollbacks' 853
     Number of non-isolated changes rolledback 137
     Number of non-serializable requests 0
     Time: 98852 ms
Concurrent Process-2
     Final total $1000000
     Number of requests 10000
     Number of overdraft 'rollbacks' 822
     Number of non-isolated changes rolledback 127
     Number of non-serializable requests 0
     Time: 99541 ms
```

The additional transactions result in more overdraft exceptions, and the overall times do increase because of data traffic competition etc.

DB2 – no cache

The results for DB2 without caching are:

```
Process-1
     Final total $1000000
     Number of requests 10000
     Number of overdraft 'rollbacks' 470
     Number of non-isolated changes rolledback 122
     Number of aborted requests 0
     Time: 24536 ms
Concurrent Process-2
     Final total $1000000
     Number of requests 10000
     Number of overdraft 'rollbacks' 554
     Number of non-isolated changes rolledback 109
     Number of aborted requests 0
     Time: 25235 ms
```

The impact of the loss of caching is more apparent when looking at the data transfers. For Oracle, the cached JPA version handles the standard ten thousand requests with ~84,000 packets totalling 13.7Mbyte; with the cache disabled and entities being reloaded each time, the traffic involves ~133,000 packets and 23Mbyte. The figures for DB2 are: cached ~101,000 packets at 25Mbyte, un-cached ~211,000 packets and 54Mbyte. Even if the apparent run-time differences are not that marked, these changes in data traffic would have a real impact in a more realistic scaled up version of the application.

The auto-generated JPA implementation is somewhat less efficient than the hand coded versioning JDBC code. Run-times are longer and data traffic is higher (but then the coding is much easier).

EJB version of banking application

Client-server banking application

Of course, little stand-alone applications that directly access the database either by JDBC or JPA are an unrealistic model for something like the banking application. This is inherently some client-server system, and one that is quite well suited to an EJB implementation. Testing is easier if "professional" clients are used rather than web-clients – it is easier to have scripts that start little client applications than have scripts submitting faked HTTP requests to a web-server, servlet, EJB system. So, for this section, EJB client applications utilize a stateless EJB session bean that runs the JPA requests.

The service is defined by the remote interface:

EJB remote interface

```
@Remote
public interface BankSessionRemote {
     int reinitialize();
     int getTotalFunds();
     int attemptTransfer(int from, int to, int amount);
}
```

The reinitialize method sets all accounts back to the default state with balance 1000 and version 1; it returns the number of accounts. The getTotalFunds method returns the total funds still in the bank; it can be called at start and end of tests to verify that the consistency constraint is maintained. The attemptTransfer method attempts a transfer; the int result encodes whether there was a successful transfer or and overdrawn condition.

The implementation of attemptTransfer is much simplified:

EJB/JPA transfer code

```
@Stateless
public class BankSessionBean implements BankSessionRemote {

    @PersistenceContext(unitName="DB2BankEJB-ejbPU")
    //@PersistenceContext(unitName="ORABankEJB-ejbPU")
    private EntityManager em;

    public int attemptTransfer(int from, int to, int amount) {
        // Result - 0 = OK, -1 = overdraft,

        int result = 0;
        Vaccounts fromAccount = em.find(Vaccounts.class, from);
        Vaccounts toAccount = em.find(Vaccounts.class, to);
        long balanceFrom = fromAccount.getBalance();
        long balanceTo = toAccount.getBalance();
        if (balanceFrom < amount) {
            return -1;
        }
        balanceFrom -= amount;
        balanceTo += amount;

        fromAccount.setBalance(balanceFrom);
```

```
        toAccount.setBalance(balanceTo);

        return result;
    }

    …
}
```

The persistence unit defined for the application contains definitions of connections for both DB2 and Oracle.

The EJB configuration left caching enabled. (Multiple instances of the EJB session bean will have separate EntityManagers but these will all share the same cache as they are created from a single persistence unit). Logging was again changed to record only SEVERE errors to avoid delays from excessive logging at the default settings.

Now obviously a client application can only have one injected connection to the bean and this one connection should not be used by multiple threads. So the client code can be simplified for single threaded operation:

EJB client code

```
public class Main {
    @EJB
    private static BankSessionRemote bankSessionBean;
    private static Random rgen;
    private static int numAccounts;
    …
    private static void run() {
        …
        for (int i = 0; i < 2000; i++) {
            …
            try {
                int result =
                    bankSessionBean.attemptTransfer(
                        from, to, amount);
                …
            }
            catch (Exception re) {
                …
                interfereRollbackCount++;
            }
            …
        }
        …
    }

    public static void report() { … }

    public static void main(String[] args) {
        numAccounts = bankSessionBean.reinitialize();
        rgen = new Random();
        …
        run();
        report();
    }
}
```

An ant script was used to try to create five concurrent client processes, each of which would submit 2000 requests. In the threaded version, concurrent operations are imperfect – the first and last threads running part of the time in isolation. This effect is a bit more marked with the multiple client process configuration.

Typical results with five client processes and a session bean configured to use Oracle are:

Results with EJB configured to use Oracle

```
Process 1
     Final total $1000000
     Number of requests 2000
     Number of overdraft 'rollbacks' 106
     Number of non-isolated changes rolledback 10
     Time: 87233 ms
Concurrent Process 2
     Final total $1000000
     Number of requests 2000
     Number of overdraft 'rollbacks' 109
     Number of non-isolated changes rolledback 12
     Time: 87175 ms
     …
Concurrent Process 5
     Final total $1000000
     Number of requests 2000
     Number of overdraft 'rollbacks' 104
     Number of non-isolated changes rolledback 12
     Time: 87124 ms
```

Typical DB2 results are:

Results with EJB configured to use DB2

```
Process 1
     Final total $1000000
     Number of requests 2000
     Number of overdraft 'rollbacks' 90
     Number of non-isolated changes rolledback 10
     Time: 24407 ms
…
Concurrent Process 5
     Final total $1000000
     Number of requests 2000
     Number of overdraft 'rollbacks' 106
     Number of non-isolated changes rolledback 14
     Time: 24194 ms
```

The data traffic for a single client submitting 10,000 requests was in the case of Oracle somewhat lower than that of the Java-SE/JPA application (~70,000 packets at 9Mbyte). The data traffic with DB2 was essentially the same as in its Java-SE/JPA implementation.

Concurrent transactions are easy with EJB-3/JPA!

Distributed transactions with XA-aware databases

A version of the application was recoded to simulate a distributed database system with transfers between one bank's accounts as held in a DB2 database, and a second

bank's accounts held in the Oracle database. The consistency check becomes a requirement that the total of funds in the two databases be invariant.

The client code was modified to pick at random the direction of transfer (to/from DB2) as well as randomly selecting the account numbers and amount of funds to be transferred. The session bean code was modified to use two EntityManagers – initialized from the two defined persistence units.

(The NetBeans/Glassfish system creates invalid parameter arguments for a DB2-XA connection; the connection will fail. A DB2-XA connection should have user-name, password, *drivertype*, and rather than a URL the database should be defined by individual parameters specifying host, port number etc. The correct parameters can be set manually via the Glassfish administrative console.)

The distributed version of the application appeared to run reasonably well, though the testing was limited. There was no attempt to temporarily disable a database and thereby force the exploration of the correct operation of the two-phase commit protocol and associated transaction recovery mechanisms.

XA distributed transaction; Oracle & DB2

```
Process 1
     Number of requests 2000
     Number of overdraft 'rollbacks' 36
     Number of non-isolated changes rolledback 9
     Time: 83159 ms
     …
Process 5
     Final total $2000000
     Number of requests 2000
     Number of overdraft 'rollbacks' 39
     Number of non-isolated changes rolledback 6
     Time: 83442 ms
```

Interested readers might like to try creating a correct and effective implementation using EJB-2/CMP technologies with XA connections to databases set to have appropriate isolation levels.

Resources

Local resources

- OldStyleSchools.zip: This contains input files for sqlplus (or similar) to create the tables and partly populate them. It also contains the JDBC version of the code as a Netbeans project (Netbeans 6).
  When run, the program will first ask if the tables are to be completed. If this option is selected, pupils and teachers are pseudo-randomly allocated to the schools. This needs to be done any time the tables have been reinitialized.
  The JDBC application simply uses JOptionPane dialogs to get commands from the user; output goes to System.out (so will typically appear in the "Output" pane of the Netbeans window).

- SchoolsSimpleJPA.zip: This contains the same Java SE application rewritten to use JPA.
- A short tutorial on building a servlet example using Netbeans; this example has a servlet that uses JPA to grab the school names from the Schools datatable on Oracle.
- E5 Another shorter exercise on starting JPA
- WebSchoolDemo2.zip: This contains the version implemented with servlets.
- EESchools3.zip and Schools2.zip: These contain the EE version. The EESchools3 file contains the EJB part and the application client. The Schools2 file contains the web version (only one of the servlets has been rewritten to use the EJB session bean; it is left as an exercise for the reader to convert the other servlets that are defined in the WebSchoolDemo2 project).

Web resources

- Netbeans+AppServer (This download includes Oracle's Toplink Essentials library)
- Sun's Java EE tutorial Chapters 24…27 cover JPA.
- Using Java Persistence in a Web Application This is a NetBeans tutorial with a servlet that accesses a simple customer contact list datatable (it uses the "Derby" database that forms a part of the AppServer suite).
- IBM developerworks
  There are several examples at the developerworks site including:
  1. Get to know Java EE 5
  2. Design enterprise applications with the EJB 3.0 Java Persistence API
- SAP
  SAP's articles include
  1. Getting Started with Java Persistence API and SAP JPA1.0
  2. Understanding the Entity Manager
- Oracle
  Oracle's articles include
  1. Taking JPA for a Test Drive
  2. JPA Annotation Reference All the annotations (@ID, @PersistenceUnit, etc) and their possible attributes are explained.
  3. Simplifying EJB Development with EJB3
  4. Web application with JSF and JPA – get even more automation, use JSF as well!
  5. How to use Toplink in Java SE
  6. Developing Applications Using Toplink JPA
  7. How to configure primary key generation
- Pro EJB 3 (An eBook from Apress)