Accessing Identity Data via EJB

Claudio A. Ardagna, Ernesto Damiani, Fulvio Frati and Umberto Raimondi
Department of Information Technologies University of Milan (Italy)

1 Introduction

In this paper we explore the critical issue of accessing identity data, e.g. during access control and authentication processes. A worked-out case study, the *Web-based Multiprotocol User Interface* (IMW) will demonstrate how interfacing business objects with an identity database can be straightforwardly carried out by adopting an object-oriented approach. Our technique entirely relies on standard Enterprise JavaBeans^{TM}technology, as a direct interface to business objects, and on the Data Access Object pattern as a way to interact with the underlying database. This simple yet effective technique is a first step toward understanding and using advanced persistence management tools such as Hibernate [1, 7], that will be covered in a future article. In Section 2 we briefly introduce all technologies we rely on and, in Section 3 we explain in detail how our interface is implemented. Finally, in Section 4, we describe alternative implementation approaches.

2 Technology Overview

2.1 Enterprise JavaBeans™

The J2EE (Java 2 Platform, Enterprise Edition) platform specification includes a notable component, the Enterprise JavaBeans architecture (EJB) [6, 4], for the development and deployment of component-based distributed business applications. The EJB architecture specification defines three types of objects, all named beans:

- *Session Beans*, objects that represent and implement the user interaction with the system and contain the business logic. Session Beans can be *stateless*: in this case, a new instance of the object is created at every request and no state information is stored for next calls. Otherwise, they can be *stateful*, i.e. a bean is created in response to the initial client request and then manages all the subsequent requests until the client connection is dropped or the bean is explicitly removed;
- *Entity Beans*, that implement business objects in a persistent storage mechanism. Typically, each entity bean has an underlying table in a relational database, and each instance of the bean corresponds to a row in that table. Entity beans are persistent, allow shared access, and may participate in relationships with other entity beans;
- *Message-driven Beans*, enterprise beans that allow J2EE applications to process messages asynchronously.

Our sample IMW relies on entity beans to provide a reliable database interface to business objects. Database rows are mapped in class variables and *getter* and *setter* methods are provided. Business objects access appropriate entity beans ignoring all about the underlying database. Database-dependent features are entirely hidden by Data Access Objects (DAO), explained in the following section. Our DAO-based approach will permit us to deal with a relational database as if it were an object-oriented database.

2.2 The Data Access Object Pattern

Patterns are a well-known software engineering technique that emerged from the object-oriented community. J2EE patterns, in particular, are a set of generic class templates that can be used to design and implement almost all types of software systems. In this Section we give an overview of the well-known Data Access Object pattern [5] (DAO), used to implement classes that directly communicate with data repositories. Generally speaking, the access interface to a database is highly dependent on the underlying DBMS and on its data model. The DAO pattern is a simple and elegant solution to give to developers a unique and reliable interface to data stored in all types of databases while adhering to the ODBMS paradigm. The DAO pattern abstracts and encapsulates all access to data sources, managing the connections with each data source to store and retrieve data. DAO uses the native interface provided by the data source and supplies a more sophisticated and ad-hoc interface to calling objects.

The advantages of using DAO as interface to data sources, can be summarized as follows:

- *Transparency*: business objects can access data sources without knowing their specific internal details;
- *Easier Migration*: data source migration is simplified by DAO objects, since it implies changes to DAO objects only, without modifications to any business objects;
- *Business Objects Code Simplification*, problems related to data source management and queries are delegated to DAO objects;

In our sample application, Entity Beans interact with an identity database through DAO objects. Database specifications are filtered by DAO objects that supply an uniform and basic interface to entity beans.

2.3 Oracle 9i Database: Table Types and Stored Procedures

To make our example more concrete, let us assume the underlying database facility is Oracle 9i, one of the most widespread database management system. Oracle relies on PL/SQL that is a procedural extension to standard SQL. Oracle 9i supports seven major types of tables [8]:

1. *Heap Organized Tables*: standard database tables, where data is managed like a heap. When a record is added, the first free space available that can fit the data is used. When a record is removed from the table, the free space will become available for subsequent "update" or "insert" operations;

2. *Index Organized Tables*: tables are stored in an indexed structure, where data is sorted according to the primary key;

3. *Clustered Tables*: data are stored in cluster and retrieved by cluster key. Data from different tables can be stored in the same block; moreover, all data that contains the same cluster key are physically stored together;

4. *Hash Clustered Tables*: like the previous category, data are stored in cluster and retrieved by cluster key, but the cluster key is hashed directly from stored data;

5. *Nested Tables*: these tables are part of the Object Relational extension to Oracle and add to traditional tables the concept of inheritance. For instance, if the table "Employer" has a foreign key that points to "Department", Employer is considered a child of Department;

6. *Temporary Tables*: tables that store temporary data for the duration of a transaction;

7. *Object Tables*: tables built based on a object type.

Another important feature of Oracle databases is the ability to implement stored procedures, i.e. PL/SQL routines that extend DBMS functionalities running inside the DBMS itself. Stored procedures can address common transactions-based systems issues, as for instance:

- *State Management*: external procedures will lose their state at the end of each transaction. Well written stored procedures, since they are running within the database, can implement a mechanism to maintain and establish a persistent state;

- *Tracing mechanism*: stored procedures allow writing routines to debug and test transactions on server side;

- *Parameter setting*: stored procedures can be easily parameterized to adapt their behavior, in response to external requests;

- *Error handling*: stored procedures can report to end user errors occurring during transactions.

3 Database Implementation: IMW Approach

We are now ready to explain our implementation focusing on identity management. IMW application is designed as a three layers structure composed on the bottom by the business components (Session Beans), in the middle by the database proxy layer constituted by both DAO and Entity Bean solutions and finally on the top by the database platform (see Fig. 1).

More in detail, the business components layer is composed by four modules, two core modules and two optional ones:

1. ST (Startup): the core module that manages IMW startup and initialization;

2. UM (User Management): the core module that manages IMW users profiles and login mechanism;

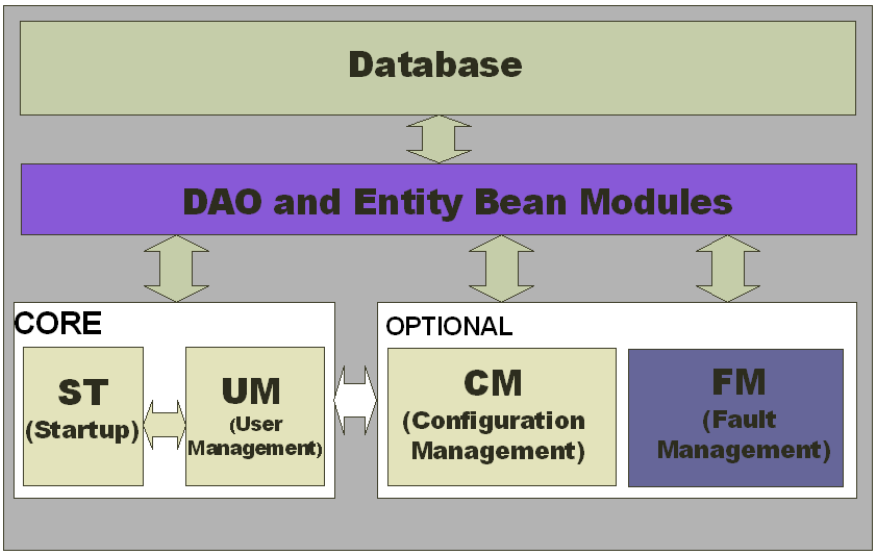

Figure 1: IMW Structure

3. FM (Fault Management): the optional module that manages errors and notifications;

4. CM (Command Management): is an optional module that allows users to check network state and send configuration commands to solve network problems.

Next, we provide a description of the IMW database interface implementation, starting from the description of fine-grained database elements, the identity tables and stored procedures, and going on to the description of meaningful parts of DAO and EJB Entity Bean solutions.

3.1 Database Layer

3.1.1 Identity table

Our implementation stores the IMW user profiles information into a Oracle table *User*. This table has been created using PL/SQL code, whose most interesting part is showed below in Fig. 2.

As shown in Fig 2, the User table is composed by the following eleven fields:

- **USERID,GROUPID**, contains the (unique) identification of the user and of the group which belongs to. These fields are defined as 10-digits numbers and only USERID cannot be null;

- **USERNAME, PSW**, contain username and password for the user. These fields are not null and defined as characters strings;

- **NAME, LASTNAME, ROLE, EMAIL, MOBILE**, string fields describing user personal information;

- **EXPDATE**, indicates password expiration date;

```
CREATE TABLE USERS
(
  USERID    NUMBER(10)          NOT NULL,
  GROUPID   NUMBER(10),
  USERNAME  VARCHAR2(50 BYTE)   NOT NULL,
  PSW       VARCHAR2(50 BYTE)   NOT NULL,
  NAME      VARCHAR2(50 BYTE)   NOT NULL,
  LASTNAME  VARCHAR2(50 BYTE)   NOT NULL,
  ROLE      VARCHAR2(50 BYTE),
  EMAIL     VARCHAR2(50 BYTE),
  MOBILE    VARCHAR2(50 BYTE),
  PASSLIFE  NUMBER(10),
  EXPDATE   NUMBER(10),
  BLOCKED   NUMBER(10)
)
```

Figure 2: PL/SQL code for User table creation

- **BLOCKED**, indicates if the user has been blocked and cannot access the system.

USERID is also defined as table primary key. GROUPID, instead, is defined as *foreign key*, that represents a relation with a similar field contained in another table of the database. To declare a foreign key, we use the following PL/SQL statement:

```
ALTER TABLE USERS ADD (
  CONSTRAINT USERS_FK FOREIGN KEY (GROUPID)
  REFERENCES GROUPS (GROUPID));
```

In the above code, we declare the foreign key *USERS_FK* in the table *USERS*, mapped on field *GROUPID* which, in turn, references the field *GROUPID* of table *GROUPS*. In Fig. 3 we show an example of the table filled with real values. Note that the field *PWD*, that stores the password value, is encrypted with the standard MD5 algorithm.

USERID	GROUPID	USERNAME	PSW	NAME	LASTNAME	ROLE	EMAIL	MOBILE	PASSLIFE	EXPDATE	BLOCKED
1	1	mrossi	6FA4B225...	mario	rossi	Administrator		354321534	30	1117633029124	0
2	2	cverdi	6FA4B225...	carlo	verdi	User		332447788	30	1115041098531	0
3	1	abianchi	6FA4B225...	antonio	bianchi	Administrator		3201772628	30	1114776757578	0

Figure 3: Table USERS with real values.

3.1.2 Stored Procedures

We now show the simple stored procedures that manage three basic functionalities: the creation of a new user, list of all users and verification of user login.

Creation of a new user To create a new user, IMW relies on a parametric stored procedure that accepts all user profile data and saves them in a database row (see Fig 4).

```
CREATE_USER (username in varchar2,
  password in varchar2,
  name in varchar2,
  lastname in varchar2,
  group in number,
  email in varchar2,
  mobile in varchar2,
  expdate in number,
  role in varchar2,
  passlife in number)
AS
BEGIN

  INSERT INTO USERS ("USERNAME", "PSW", "NAME", "LASTNAME",
                     "GROUPID", "EMAIL", "MOBILE", "EXPDATE",
                     "ROLE", "PASSLIFE","BLOCKED")
  VALUES (username, password, name, lastname, group, email,
          mobile, expdate, role, passlife, 0);

END;
```

Figure 4: User creation stored procedure.

List of all users Fig. 5 shows another example of a stored procedure that list all the IMW users.

```
LIST_USERS (curso out riferimenti.curref )
AS
BEGIN
  OPEN curso FOR
  SELECT userid,username,name,lastname,blocked
  FROM USERS ORDER BY lastname;
END;
```

Figure 5: User listing stored procedure.

To extract the list of all users contained in the table and make the data available to IMW, our simple stored procedure uses a *cursor*, defined as a variable that runs through the tuples of some relation. This relation can be a stored table, or it can be the answer to some query. The stored procedure selects some fields of all tables rows, alphabetically ordered by lastname, and inserts them into a cursor for next use.

Login Check The stored procedure *CHECK_LOGIN* verifies if a pair username/password is presents in the user tables and is presented in Fig. 6:

The *CHECK_LOGIN* stored procedure accepts a username and password pair as input and returns a USERID if the authentication succeeds. The stored procedure tries to match the USERID of the row with the selected username and password. If no rows are selected, then an exception is

```
CHECK_LOGIN (user IN VARCHAR2,
             passwd IN VARCHAR2,
             idUser OUT NUMBER)
IS

BEGIN

  SELECT USERID INTO idUser FROM USERS
  WHERE USERNAME=user AND PSW=passwd;

  EXCEPTION
     WHEN NO_DATA_FOUND THEN
       idUser:=-1;

END;
```

Figure 6: User listing stored procedure.

raised and a "-1" value is returned as *idUser*, otherwise the stored procedure returns as *idUser* the value of the selected USERID.

3.2 Database Proxy Layer

IMW Business components do not see the underlying database; rather, they access information through Entity Beans and DAO components. In this section we show explained examples of Java code snippets.

3.2.1 Data Access Object Examples

The Java class *DAO_UM* manages all connections between IMW and Oracle stored procedures. In Fig. 7 it is presented the Java code used to call the *Create new user* stored procedure (Fig. 4).

This function takes in input ten parameters that represent all user information that have to be stored in *USERS* database. After connection establishment (row 11), a new *CallableStatement* object is created (row 12). This object represents a call to a stored procedure and it is configured following these steps:

1. define the stored procedure to call and the number of parameters to set (row 12). In this case, we call the stored procedure *CREATE_USER* with 11 parameters (indicated as "?");

2. set the value of each one input parameters (rows 14-23). According to the parameter type, we use the respective `setXXXX` method specifying the ordinal number and the value of the parameter;

3. register the output parameter (row 25), indicating position in the parameters list and its type;

The statement and, consequently, the stored procedure are executed (row 26) and the USERID is collected by the appropriate `getXXXX` method.

If all succeeds, *userid* is returned; otherwise, a *UM_ExceptionDAO* is raised.

```
1 public long createUser(String email, long groupid,
2       String password,String mobile, String role,
3       String username,long expdate, String name,
4       String lastname,long passlife) throws UM_ExceptionDAO {
5
6    Connection conn = null;
7    CallableStatement pstmt = null;
8    long userid = -1;
9
10   try {
11     conn = getConnection();
12     pstmt = conn.prepareCall(
13         "{call CREATE_USER(?,?,?,?,?,?,?,?,?,?,?)}");
14     pstmt.setString(1, username);
15     pstmt.setString(2, password);
16     pstmt.setString(3, name);
17     pstmt.setString(4, lastname);
18     pstmt.setLong(5, groupid);
19     pstmt.setString(6, email);
20     pstmt.setString(7, mobile);
21     pstmt.setLong(8, expdate);
22     pstmt.setString(9, role);
23     pstmt.setLong(10, passlife);
24
25     pstmt.registerOutParameter(11, Types.INTEGER);
26     pstmt.execute();
27     userid = pstmt.getLong(11);
28   }
29   catch (Exception e) {
30     throw new UM_ExceptionDAO(msg, e);
31   }
32   finally {
33     closeConnect(conn);
34   }
35   return userid;
36 }
```

Figure 7: Java code for *CREATE_USER* stored procedure calling.

Similar to the previous function, Fig. 8 shows the *allUser* functions, used to call the *LIST_USER* stored procedure (see Fig. 5).

Note that, in row 11 the output parameter is registered as an Oracle cursor, and, once executed, it should be extracted (row 13) a ResultSet with all elements listed by the cursor, containing the users stored in the database. This ResultSet is then formatted as a XML fragment and returned (rows 14-16).

Finally, to communicate with *CHECK_LOGIN* stored procedure, showed in Fig. 6, the function in Fig. 9 is used.

This function accepts a username and a password pair as input, and returns the USERID, if the authentication succeeds, or "-1" if the pair username - password does not exist.

```
1  public String allUsers() throws UM_ExceptionDAO, IMW_ExceptionCreateMessage {
2    Connection conn = null;
3    CallableStatement pstmt = null;
4    String xmlData = null;
5    ResultSet rset;
6
7    try {
8      conn = getConnection();
9      pstmt = conn.prepareCall("{call LIST_USERS(?)}");
10     pstmt.registerOutParameter(1, oracle.jdbc.driver.OracleTypes.CURSOR);
11     pstmt.execute();
12     rset = (ResultSet) pstmt.getObject(1);
13     MessageCreator mc = new MessageCreator();
14     mc.addTable(rset);
15     xmlData = mc.getXmlMessage();
16     pstmt.close();
17   }
18   catch (SQLException ex) {
19     throw new UM_ExceptionDAO(msg, ex);
20   }
21   finally {
22     closeConnect(conn);
23   }
24   return xmlData;
25 }
```

Figure 8: Java code for *LIST_USERS* stored procedure calling.

3.2.2 Entity Beans Examples

Entity beans implement business objects in a persistent storage mechanism. By definition, every Entity Bean has to implement the *EntityBean* interface. The *EntityBean* interface declares a number of methods, such as *ejbCreate*, which must be implemented in the Entity Bean class. In this Section, we present a methods selection of *UsersBean* class, the Entity Bean that manages user information.

When the client invokes a create method, the EJB container invokes the corresponding *ejbCreate* method that usually performs the task of inserting the entity state into the database, initializing the instance value and returning the primary key. In Fig. 10 our *ejbCreate* method is presented.

This function accepts in input all user information and returns its primary key, once the user is stored in database. The Entity uses the previously explained *DAO_UM* class to communicate with the database, hence, in line 6, a new instance of this class is created.

The *createUser* method is then called (line 7) to store in database the user profile. After that, the method returns the newly created primary key that is saved in *userid* variable. from line 9 to 18, the state of the bean is initialized using `setXXXX` methods. Note that, for every instance variable (i.e. every variable that represents a field in database), it is necessary to provide the corresponding *getter* and *setter* methods. Finally, the userid, in the form of *UsersPK* class that represents a user primary key, is returned.

The next method is the *ejbRemove* function (see Fig. 11), that clients invoke to delete the

```
1  public long checkLogin(String username, String password)
2         throws UM_ExceptionDAO {
3
4    Connection conn = null;
5    CallableStatement pstmt = null;
6    long idUser = -1;
7    try {
8      conn = getConnection();
9      pstmt = conn.prepareCall("{call CHECK_LOGIN(?, ?, ?)}");
10     pstmt.setString(1, username);
11     pstmt.setString(2, password);
12     pstmt.registerOutParameter(3, Types.INTEGER);
13     pstmt.execute();
14     idUser = pstmt.getLong(3);
15     pstmt.close();
16   }
17   catch (Exception e) {
18     throw new UM_ExceptionDAO(msg, e);
19   }
20   finally {
21     closeConnect(conn);
22   }
23   return idUser;
24 }
```

Figure 9: Java code for *CHECK_LOGIN* stored procedure calling.

```
1  public UsersPK ejbCreate(String email, BigDecimal groupid,
2                 String password,String mobile, String role,
3                 String username,BigDecimal expdate, String name,
4                 String lastname,BigDecimal passlife) throws CreateException {
5
6    DAO_UM intDaoUm = new DAO_UM();
7    userid = intDaoUm.createUser(email, groupid, password, mobile, role,
8                                 username,expdate, name, lastname, passlife);
9    setEmail(email);
10   setGroupid(groupid);
11   setPassword(password);
12   setMobile(mobile);
13   setRole(role);
14   setUsername(username);
15   setExpdate(expdate);
16   setName(name);
17   setLastname(lastname);
18   setPasslife(passlife);
19
20   return new UsersPK(userid);
22  }
```

Figure 10: Java code for *ejbCreate* function.

entity's state from the database. According to EJB specification, an entity bean can also be removed directly by a database deletion. For example, if an PL/SQL script deletes a row that contains an entity bean state, also the corresponding bean is removed.

```
1  public void ejbRemove() throws RemoveException {
2    DAO_UM intDaoUm = null;
3
4    intDaoUm = new DAO_UM();
5    intDaoUm.removeUser(username);
6  }
```

Figure 11: Java code for *ejbRemove* function.

This method simply instantiates a new *DAO_UM* class (line 4) and call its *removeUser* method to delete the bean instance from the database (line 5).

Another important method prescribed by entity Bean specification, is *ejbLoad*, that synchronizes the instance variables of an entity bean with the corresponding values stored in the database. IMW *ejbLoad* method is shown in Fig 12.

```
1  public void ejbLoad() {
2    DAO_UM intDaoUm = new DAO_UM();
3
4    UsersPK id = (UsersPK) entityContext.getPrimaryKey();
5    UserDatasDb udd = intDaoUm.loadUser(id.userid);
6
7    groupid = udd.groupid;
8    username = udd.username;
9    password = udd.psw;
10   name = udd.name;
11   lastname = udd.lastname;
12   email = udd.email;
13   mobile = udd.mobile;
14   expdate = udd.expdate;
15   role = udd.role;
16   passlife = udd.passlife;
17   blocked = udd.blocked;
18 }
```

Figure 12: Java code for *ejbLoad* function.

First of all, in line 4 the user primary key is retrieved from the context and is used to load the corresponding database insertion (line 5). The bean state is then updated with the current value. This method is automatically invoked by the container after committing the transaction together with the *ejbStore* method that updates the database entry.

Finally, we present a finder method, *ejbFindByPrimaryKey* (see Fig. 13), that allows clients to initialize and load entity beans by means of the primary key. For every finder method available to a client, the entity bean class must implement a corresponding method that begins with the prefix `ejbFind`.

```
public UsersPK ejbFindByPrimaryKey(UsersPK pk) throws FinderException {
  DAO_UM intDaoUm = new DAO_UM;

  UserDatasDb udd = intDaoUm.loadUser(pk.userid);
  userid = udd.userid;
  groupid = udd.groupid;
  username = udd.username;
  password = udd.psw;
  name = udd.name;
  lastname = udd.lastname;
  email = udd.email;
  mobile = udd.mobile;
  expdate = udd.expdate;
  role = udd.role;
  passlife = udd.passlife;
  blocked = udd.blocked;

  return new UsersPK(userid);
}
```

Figure 13: Java code for *ejbFindByPrimaryKey* function.

4 Conclusion and Future Work

In this paper, we have shown a simple yet effective technique for interfacing an identity database with EJB technology. In particular, we used a object oriented approach to access an underlying relational databases. Our solution relies on standard technologies as J2EE specifications, EJB architecture and the DAO pattern.

The advantages of this solution are mainly the usual ones of object-oriented programming. In particular, our solution provides: *i) modularity,* with a clear separation between business components and persistence modules (proxy layer and real database implementation), *ii) scalability,* the system could be extended with low effort and finally, *iii) independence from the underlying database,* allowing administrators to entirely change the database layer without any change to business logic.

In a future article we shall explain how our application can be improved by the adoption of advanced persistence management in object oriented databases, such as the Open Source solution named Hibernate [1, 7], freely downloadable from the Net. In particular, Hibernate provides object/relational persistence and query service for Java, encouraging developers to implement persistent classes following all common Java idioms, including association, inheritance, polymorphism, composition, and the Java collections framework. Hibernate supports the EJB 3.0/JSR-220 persistence standardization effort. It also provides a framework for storing objects in relational tables and an API to load those objects back in memory, either directly by using the primary keys or by using a sophisticated query language called Hibernate Query Language (HQL).

References

[1] Abrahamian, A., Cannon-Brookes, M., Lightbody, P., Walnes, J.:Java^{TM}Open Source Programming With XDoclet, JUnit, WebWork, Hibernate, *Wiley Publishing, Inc.*, 2004.

[2] Ardagna, C.A., Damiani, E., De Capitani di Vimercati, S., Frati, F., Samarati, P.: Single Sign-On for Open Source Application Servers: a Comparative Approach, Submitted to *The 21st ACM Symposium on Applied Computing Track on Computer Security (5th edition), 23-27 April 2006, Dijon, France.*

[3] Ardagna C.A., Damiani E., Frati F., Montel M., Using Open Source Middleware for Securing e-Gov Applications, *The First International Conference on Open Source Systems (OSS 2005)*, Genova, Italy.

[4] Armstrong, E., et al.: The J2EE 1.4 tutorial, `http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html`, June 2005.

[5] Core J2EE Patterns - Data Access Object, `http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html`, July 2002.

[6] DeMichiel, L.G. (Specification Lead): Enterprise JavaBeans Specification, Version 2.1 *Sun Microsystem*, November 2003.

[7] Hibernate, `http://www.hibernate.org/`

[8] Kyte, T.: Expert One-on-One Oracle, *Wrox Press Ltd.*, 2001.