Day - 3 HF JDBC Model

AbstractDAO, Transactions, general HF RDBMS rules to follow

JDBC - HF way

- If you think about SQL queries - they are TEMPLATE oriented.
- Typical INSERT syntax:

```
INSERT INTO <table_name>
(<column_name>)
VALUES
(<value>);
```

- Typical UPDATE syntax:

```
UPDATE <table_name>
SET <column_name> = <value>
```

AbstractDAO

- JDBC Template
- This class encapsulates the typical JDBC interaction you will have with your application.
- If you can identify what is the table name, what is the primary key, how you generate the primary key, what are the column, where do the column values come from, how to map the resultset and create a POJO - you should be able to deal with RDBMS without redoing the CRUD statements over and over again for each new TABLE created.
- AbstractDAO help accomplish the same. The concrete class is responsible to fill the template with the needful

AbstractDAO - contd..

- In the concrete class you identify the tablename, primary key, provide the mapping rules from column to POJO(SELECT stmt), POJO instance vars to columns (INSERT, UPDATE).

SampleDAO

```
 */
public class DemoDAO extends AbstractDAO {
    @Override
    protected String getTableName() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    protected String getPrimaryKey() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    protected Object loadColumns(Map<String, Object> _columns) {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    protected Map<String, Object> buildColumns(Object _obj, boolean _isInsert) {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    protected boolean isReadOnlyTable() {
        // TODO Auto-generated method stub
        return false;
    }
}
```

BuildColumn - why?

```
/**
 * Convenience method to create a Map to be used for building the
 * INSERT/UPDATE statement column/column values
 *
 * @param _obj
 *            Object to insert/update in the database
 * @param _isInsert
 *            Boolean indicator for whether its an insert or an update
 * @return Map with key = Table Column Name and Value = Column Value to
 *         Insert
 */
protected abstract Map<String, Object> buildColumns(Object _obj, boolean _isInsert);
```

This method is used in building the INSERT statetment for the table. The MAP returned :

KEY = COLUMN NAME
VALUE = COLUMN VALUE to save.

BuildColumn - contd..

- _isInsert - indicates whether you want your primary key generated or not.
- Save by default makes it true, In cases where you need to generate the primary key ahead of time to tie the child objects - while saving set this indicator to false for not generating.
- Update makes it false - since primary key already exists.
-

```
@Override
protected Map<String, Object> buildColumns(Object _obj, boolean _isInsert) {
    AllowedCcareIPAddress ipAddress = (AllowedCcareIPAddress) _obj;
    Map<String, Object> values = new HashMap<String, Object>();
    if (_isInsert) {
        ipAddress.setIpAddressId(getSequence());
    }
    values.put(COLUMN_NAME.IP_ADDRESS_ID.name(), ipAddress.getIpAddressId());
    values.put(COLUMN_NAME.IP_ADDRESS.name(), ipAddress.getIpAddress());
    return values;
}
```

BuildColumn - contd..

```
/**
 * Convenience method to create the insert statement based on the Map
 * received.
 *
 * @param _columns
 *            Map with key = Table Column Name , value = Table Column Value
 *            to Insert
 * @param _conn
 *            Connection
 * @throws SQLException
 * @return PreparedStatement
 */
public PreparedStatement buildInsert(Map<String, Object> _columns, Connection _conn) throws SQLException {
    StringBuffer sb = new StringBuffer();
    sb.append(" INSERT INTO ").append(getTableName());
    sb.append(" ( ");
    Object[] keys = _columns.keySet().toArray();
    Object[] values = new Object[keys.length];
    for (int i = 0; i < keys.length; i++) {
        String key = (String) keys[i];
        sb.append(key).append(" ");
        if (i < keys.length - 1) {
            sb.append(" , ");
        }
        values[i] = _columns.get(key);
    }
    sb.append(" ) VALUES ( ");
    for (int j = 0; j < values.length; j++) {
         if ("sysdate".equals(values[j])) {
            sb.append("sysdate");
        } else if ("empty_blob()".equals(values[j])) {
            sb.append("empty_blob()");
        } else if (values[j] instanceof String && ((String) values[j]).startsWith("XMLTYPE")) {
            sb.append((String) values[j]);
```

LoadColumns - why?

```
/**
 * Convenience method to extract the column values and set the fields on the
 * Object returned from the result set. Used by the find SQLs
 *
 * @param _columns
 *            Map of table column name and column values
 * @return Newly created object mapped for the resultset Map
 */
protected abstract Object loadColumns(Map<String, Object> _columns);
```

LoadColumns does the reverse - used in SELECT statements to map the Column value to the object.

```
@Override
protected Object loadColumns(Map<String, Object> _columns) {
    AllowedCcareIPAddress data = new AllowedCcareIPAddress();
    data.setIpAddressId(getLong((BigDecimal) _columns.get(COLUMN_NAME.IP_ADDRESS_ID.name())));
    data.setIpAddress((String) _columns.get(COLUMN_NAME.IP_ADDRESS.name()));
    return data;
}
```

LoadColumns - contd..

```
public Object find(long _id) throws InfrastructureException {
    String sql = "SELECT * FROM " + getTableName() + " WHERE " + getPrimaryKey() + " = ? ";
    PreparedStatement pst = null;
    Connection conn = null;
    ResultSet rs = null;
    Object obj = null;
    try {
        conn = getConnection();
        pst = conn.prepareStatement(sql);
        pst.setLong(1, _id);
        traceParameterizedQuery(sql, _id);
        rs = pst.executeQuery();
        ResultSetMetaData rsmd = rs.getMetaData();
        Map<String, Object> columns = getColumnMap(rsmd, rs);
        if (columns != null) {
            obj = loadColumns(columns);
        }
        columns = null;
    } catch (SQLException e) {
        throw new InfrastructureException(e);
    } finally {
        closeAll(conn, pst, rs);
    }
    return obj;
}
```

Find is a template method - where the concrete class defines given a SELECT result set how do I map the columns.

LoadColumns - contd..

- ResultSetMetaData = comes to rescue to help identify the template pattern and concrete class just fill the template.
- ResultSetMetaData - you can get the column names and using that from the ResultSet - create a Map.
- KEY = COLUMN NAME
- VALUE = COLUMN VALUE.
-

LoadColumn - contd..

```java
/**
 * Convenience method to create a Map to be used to populate a bean.
 *
 * @param _rsmd
 *            ResultSetMetaData
 * @param _rs
 *            ResultSet
 * @return Map with Key = Table Column Name , Value = Table Column Value
 * @throws SQLException
 */
protected Map<String, Object> getColumnMap(ResultSetMetaData _rsmd, ResultSet _rs) throws SQLException {
    Map<String, Object> columns = null;
    if (_rs.next()) {
        columns = new HashMap<String, Object>();
        int columnCount = _rsmd.getColumnCount();
        for (int i = 1; i <= columnCount; i++) {
            int type = _rsmd.getColumnType(i);
            if (type == Types.DATE || type == Types.TIMESTAMP) {
                columns.put(_rsmd.getColumnName(i), _rs.getTimestamp(i));
            } else {
                columns.put(_rsmd.getColumnName(i), _rs.getObject(i));
            }
        }
    }
    return columns;
```

Using the RSMD - we create the MAP which then the Concrete class uses in the LoadColumns to help fill a POJO instance.

What did we gain using - AbstractDAO?

- You do not need to worry about the things that you would do in a "from the scratch JDBC code"..
- You need to identify what SQL is needed.
- So any Table needed - following things are necessary:
  - Define the POJO.
  - Define the DAO extending AbstractDAO
  - Define the sequence generating the primary key if applicable
  - Define the buildcolumn mapping
  - Define the loadcolumn mapping.
  - Start putting SQLs without worrying about mapping the resultset, iterating the resultset and creating a list of objects.

Sample:

```
/**
 * Method to check is the ipAddress is allowed or not.
 *
 * @param _ipAddress
 * @return
 */
public boolean isAccessAllowed(String _ipAddress) {
    String sql = "select count(*) from " + getTableName() + " where ip_address =? ";
    int cnt = super.getCount(sql, Arrays.asList(new Object[] { _ipAddress }));
    return cnt >= 1 ? true : false;
}

public List<AllowedCcareIPAddress> getListOfAllowedIPAddress() {
    String sql = "select * from " + getTableName();
    return (List<AllowedCcareIPAddress>) super.search(sql);
}

protected Long getSequence() {
    return getSequence("CCARE_IP_ADDRESS_SEQ");
}
```

Ready for SQL - Best Practices:

- You will write SQL queries - always prepared statement.
- Never hard code sql like:

```
public List<String> searchLikeUserName(String _name) {
    String sql = "SELECT USERNAME FROM " + getTableName()
            + " WHERE USERNAME LIKE '" + _name + "%'";
```

- Always write preparedstatements with parameterized SQL. i.e., put ? for parameters.
- Why - DB caches queries if hard coded queries - you can imagine how many to cache depends on all the possible combination of the parameters!!
- So to impact performance to avoid hard parsing of the SQL each time its executed - use parameterized SQL.

Best Practices - contd..

- Override getSequence() - originally when we had one table - one sequence we did not realize this should have been a Abstract Method.. so just be disciplined to remember about this.
- Create ColumnEnum - to avoid spelling mistakes.
- You write your column names in BuildColumn, LoadColumn, ColumnNames.
- Always write one JUNIT test case to make sure you did not mis-spell any columns.

Transactions - if you need to..

```
protected Object executeTransaction() {
    Connection conn = null;
    try {
        setConnection(null);
        conn = getConnection();
        setConnection(conn);
        conn.setAutoCommit(false);
        Object messageObj = transactionSteps.execute(conn);
        conn.commit();
        conn.setAutoCommit(true);
        return messageObj;
    } catch (Exception e) {
        try {
            wasTransactionSuccessful = false;
            e.printStackTrace();
            conn.rollback();
            conn.setAutoCommit(true);
            String err = e.getMessage();
            if (err != null)
                errMsgTransaction = err.replace("java.sql.SQLException:", "");
            return null;
        } catch (SQLException e1) {
            e1.printStackTrace();
            throw new InfrastructureException(e);
        }
    } finally {
        try {
            conn.setAutoCommit(true);
        } catch (SQLException e) {
            e.printStackTrace();
            String name = "";
            try {
                name = transactionSteps.getClass().getName();
```

Transactions - contd..

The transaction steps interface makes the template for the transaction resulting in the previous template method.

The concrete class will fill in the and use it.

See references for the TransactionSteps Interface and usages of those classes to know more.

Few Additional Things:

- What we just saw is a sample of "Close to modification, open to extension"
- We rarely change existing methods of AbstractDAO.
- We keep adding new concrete classes representing new tables.
- Learn more about BatchSaving - since sometime you need to save in batch instead of save in a for loop one item at a time.
- Clob/Blob have special handling - see sample DAOs.
- Beware of connection leaks - if you do not specifically call getConnection() in your method - you are safe, else always call closeAll() in finally.