The SQL Builder Class

The Problem

One of the nastiest things a Java developer will ever have to do is embed SQL in Java code. Usually this is done because the SQL has to be dynamically generated - otherwise you could externalize it in a file or a stored proc. As you've already seen, MyBatis has a powerful answer for dynamic SQL generation in its XML mapping features. However, sometimes it becomes necessary to build a SQL statement string inside of Java code. In that case, MyBatis has one more feature to help you out, before reducing yourself to the typical mess of plus signs, quotes, newlines, formatting problems and nested conditionals to deal with extra commas or AND conjunctions. Indeed, dynamically generating SQL code in Java can be a real nightmare. For example:

```
String sql = "SELECT P.ID, P.USERNAME, P.PASSWORD, P.FULL_NAME, "
"P.LAST_NAME,P.CREATED_ON, P.UPDATED_ON " +
"FROM PERSON P, ACCOUNT A " +
"INNER JOIN DEPARTMENT D on D.ID = P.DEPARTMENT_ID " +
"INNER JOIN COMPANY C on D.COMPANY_ID = C.ID " +
"WHERE (P.ID = A.ID AND P.FIRST_NAME like ?) " +
"OR (P.LAST_NAME like ?) " +
"GROUP BY P.ID " +
"HAVING (P.LAST_NAME like ?) " +
"OR (P.FIRST_NAME like ?) " +
"ORDER BY P.ID, P.FULL_NAME";
```

The Solution

MyBatis 3 offers a convenient utility class to help with the problem. With the SQL class, you simply create an instance lets you call methods against it to build a SQL statement one step at a time. The example problem above would look like this when rewritten with the SQL class:

```
private String selectPersonSql() {
  return new SQL() {{
    SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FULL_NAME");
    SELECT("P.LAST_NAME, P.CREATED_ON, P.UPDATED_ON");
    FROM("PERSON P");
    FROM("ACCOUNT A");
```

```
    INNER_JOIN("DEPARTMENT D on D.ID = P.DEPARTMENT_ID");
    INNER_JOIN("COMPANY C on D.COMPANY_ID = C.ID");
    WHERE("P.ID = A.ID");
    WHERE("P.FIRST_NAME like ?");
    OR();
    WHERE("P.LAST_NAME like ?");
    GROUP_BY("P.ID");
    HAVING("P.LAST_NAME like ?");
    OR();
    HAVING("P.FIRST_NAME like ?");
    ORDER_BY("P.ID");
    ORDER_BY("P.FULL_NAME");
  }}.toString();
}
```

What is so special about that example? Well, if you look closely, it doesn't have to worry about accidentally duplicating "AND" keywords, or choosing between "WHERE" and "AND" or none at all. The SQL class takes care of understanding where "WHERE" needs to go, where an "AND" should be used and all of the String concatenation.

The SQL Class

it is used with @SelectProvider annotation

Here are some examples:

```
// Anonymous inner class
public String deletePersonSql() {
  return new SQL() {{
    DELETE_FROM("PERSON");
    WHERE("ID = ${id}");
  }}.toString();
}

// Builder / Fluent style
public String insertPersonSql() {
  String sql = new SQL()
    .INSERT_INTO("PERSON")
    .VALUES("ID, FIRST_NAME", "${id}, ${firstName}")
    .VALUES("LAST_NAME", "${lastName}")
    .toString();
  return sql;
}

// With conditionals (note the final parameters, required for the anonymous inner
class to access them)
public String selectPersonLike(final String id, final String firstName, final String la
stName) {
  return new SQL() {{
    SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FIRST_NAME, P.LAST_N
AME");
    FROM("PERSON P");
```

```
    if (id != null) {
      WHERE("P.ID like ${id}");
    }
    if (firstName != null) {
      WHERE("P.FIRST_NAME like ${firstName}");
    }
    if (lastName != null) {
      WHERE("P.LAST_NAME like ${lastName}");
    }
    ORDER_BY("P.LAST_NAME");
  }}.toString();
}

public String deletePersonSql() {
  return new SQL() {{
    DELETE_FROM("PERSON");
    WHERE("ID = ${id}");
  }}.toString();
}

public String insertPersonSql() {
  return new SQL() {{
    INSERT_INTO("PERSON");
    VALUES("ID, FIRST_NAME", "${id}, ${firstName}");
    VALUES("LAST_NAME", "${lastName}");
  }}.toString();
}

public String updatePersonSql() {
  return new SQL() {{
    UPDATE("PERSON");
    SET("FIRST_NAME = ${firstName}");
    WHERE("ID = ${id}");
  }}.toString();
}
```

Method	Description
`SELECT(String)`	Starts or appends to a `SELECT` clause. Can be called more than once, and parameters will be appended to the `SELECT` clause. The parameters are usually a comma separated list of columns and aliases, but can be anything acceptable to the driver.

Method	Description
`SELECT_DISTINCT(String)`	Starts or appends to a `SELECT` clause, also adds the `DISTINCT` keyword to the generated query. Can be called more than once, and parameters will be appended to the `SELECT` clause. The parameters are usually a comma separated list of columns and aliases, but can be anything acceptable to the driver.
`FROM(String)`	Starts or appends to a `FROM` clause. Can be called more than once, and parameters will be appended to the `FROM` clause. Parameters are usually a table name and an alias, or anything acceptable to the driver.
• `JOIN(String)` • `INNER_JOIN(String)` • `LEFT_OUTER_JOIN(String)` • `RIGHT_OUTER_JOIN(String)`	Adds a new `JOIN` clause of the appropriate type, depending on the method called. The parameter can include a standard join consisting of the columns and the conditions to join on.
`WHERE(String)`	Appends a new `WHERE` clause condition, concatenated by `AND`. Can be called multiple times, which causes it to concatenate the new conditions each time with `AND`. Use `OR()` to split with an `OR`.
`OR()`	Splits the current `WHERE` clause conditions with an `OR`. Can be called more than once, but calling more than once in a row will generate erratic `SQL`.
`AND()`	Splits the current `WHERE` clause conditions with an `AND`. Can be called more than once, but calling more than once in a row will generate erratic `SQL`. Because `WHERE` and `HAVING` both automatically concatenate with `AND`, this is a very uncommon method to use and is only really included for completeness.
`GROUP_BY(String)`	Appends a new `GROUP BY` clause elements, concatenated by a comma. Can be called multiple times, which causes it to concatenate the new conditions each time with a comma.

Method	Description
`HAVING(String)`	Appends a new `HAVING` clause condition, concatenated by AND. Can be called multiple times, which causes it to concatenate the new conditions each time with `AND`. Use `OR()` to split with an `OR`.
`ORDER_BY(String)`	Appends a new `ORDER BY` clause elements, concatenated by a comma. Can be called multiple times, which causes it to concatenate the new conditions each time with a comma.
`DELETE_FROM(String)`	Starts a delete statement and specifies the table to delete from. Generally this should be followed by a WHERE statement!
`INSERT_INTO(String)`	Starts an insert statement and specifies the table to insert into. This should be followed by one or more VALUES() calls.
`SET(String)`	Appends to the "set" list for an update statement.
`UPDATE(String)`	Starts an update statement and specifies the table to update. This should be followed by one or more SET() calls, and usually a WHERE() call.
`VALUES(String, String)`	Appends to an insert statement. The first parameter is the column(s) to insert, the second parameter is the value(s).

SqlBuilder and SelectBuilder (DEPRECATED)

Before version 3.2 we took a bit of a different approach, by utilizing a ThreadLocal variable to mask some of the language limitations that make Java DSLs a bit cumbersome. However, this approach is now deprecated, as modern frameworks have warmed people to the idea of using builder-type patterns and anonymous inner classes for such things. Therefore the SelectBuilder and SqlBuilder classes have been deprecated.

The following methods apply to only the deprecated SqlBuilder and SelectBuilder classes.

Method	Description
`BEGIN()` / `RESET()`	These methods clear the ThreadLocal state of the SelectBuilder class, and prepare it for a new statement to be built. `BEGIN()` reads best when starting a new statement. `RESET()` reads best when clearing a statement in the middle of execution for some reason (perhaps if the logic demands a completely different statement under some conditions).
`SQL()`	This returns the generated `SQL()` and resets the `SelectBuilder` state (as if `BEGIN()` or `RESET()` were called). Thus, this method can only be called ONCE!

The SelectBuilder and SqlBuilder classes are not magical, but it's important to know how they work. SelectBuilder and SqlBuilder use a combination of Static Imports and a ThreadLocal variable to enable a clean syntax that can be easily interlaced with conditionals. To use them, you statically import the methods from the classes like this (one or the other, not both):

```
import static org.apache.ibatis.jdbc.SelectBuilder.*;
```

```
import static org.apache.ibatis.jdbc.SqlBuilder.*;
```

This allows us to create methods like these:

```
/* DEPRECATED */
public String selectBlogsSql() {
  BEGIN(); // Clears ThreadLocal variable
  SELECT("*");
  FROM("BLOG");
  return SQL();
}
```

```
/* DEPRECATED */
private String selectPersonSql() {
  BEGIN(); // Clears ThreadLocal variable
  SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FULL_NAME");
  SELECT("P.LAST_NAME, P.CREATED_ON, P.UPDATED_ON");
  FROM("PERSON P");
  FROM("ACCOUNT A");
  INNER_JOIN("DEPARTMENT D on D.ID = P.DEPARTMENT_ID");
  INNER_JOIN("COMPANY C on D.COMPANY_ID = C.ID");
  WHERE("P.ID = A.ID");
  WHERE("P.FIRST_NAME like ?");
  OR();
  WHERE("P.LAST_NAME like ?");
  GROUP_BY("P.ID");
  HAVING("P.LAST_NAME like ?");
```

```
  OR();
  HAVING("P.FIRST_NAME like ?");
  ORDER_BY("P.ID");
  ORDER_BY("P.FULL_NAME");
  return SQL();
}
```