# Dynamic SQL

One of the most powerful features of MyBatis has always been its Dynamic SQL capabilities. If you have any experience with JDBC or any similar framework, you understand how painful it is to conditionally concatenate strings of SQL together, making sure not to forget spaces or to omit a comma at the end of a list of columns. Dynamic SQL can be downright painful to deal with.

While working with Dynamic SQL will never be a party, MyBatis certainly improves the situation with a powerful Dynamic SQL language that can be used within any mapped SQL statement.

The Dynamic SQL elements should be familiar to anyone who has used JSTL or any similar XML based text processors. In previous versions of MyBatis, there were a lot of elements to know and understand. MyBatis 3 greatly improves upon this, and now there are less than half of those elements to work with. MyBatis employs powerful OGNL based expressions to eliminate most of the other elements:

- if
- choose (when, otherwise)
- trim (where, set)
- foreach

## if

The most common thing to do in dynamic SQL is conditionally include a part of a where clause. For example:

```
<select id="findActiveBlogWithTitleLike"
     resultType="Blog">
  SELECT * FROM BLOG
  WHERE state = ‘ACTIVE’
  <if test="title != null">
    AND title like #{title}
  </if>
</select>
```

This statement would provide an optional text search type of functionality. If you passed in no title, then all active Blogs would be returned. But if you do pass in a title, it will look for a title like that (for the keen eyed, yes in this case your parameter value would need to include any masking or wildcard characters).

What if we wanted to optionally search by title and author? First, I’d change the name of the statement to make more sense. Then simply add another condition.

```
<select id="findActiveBlogLike"
     resultType="Blog">
  SELECT * FROM BLOG WHERE state = ‘ACTIVE’
  <if test="title != null">
    AND title like #{title}
  </if>
  <if test="author != null and author.name != null">
    AND author_name like #{author.name}
  </if>
</select>
```

## choose, when, otherwise

Sometimes we don’t want all of the conditionals to apply, instead we want to choose only one case among many options. Similar to a switch statement in Java, MyBatis offers a choose element.

Let’s use the example above, but now let’s search only on title if one is provided, then only by author if one is provided. If neither is provided, let’s only return featured blogs (perhaps a strategically list selected by administrators, instead of returning a huge meaningless list of random blogs).

```
<select id="findActiveBlogLike"
     resultType="Blog">
  SELECT * FROM BLOG WHERE state = ‘ACTIVE’
  <choose>
    <when test="title != null">
      AND title like #{title}
    </when>
    <when test="author != null and author.name != null">
      AND author_name like #{author.name}
    </when>
    <otherwise>
      AND featured = 1
    </otherwise>
  </choose>
</select>
```

## trim, where, set

The previous examples have been conveniently dancing around a notorious dynamic SQL challenge. Consider what would happen if we return to our "if" example, but this time we make "ACTIVE = 1" a dynamic condition as well.

```
<select id="findActiveBlogLike"
     resultType="Blog">
  SELECT * FROM BLOG
  WHERE
  <if test="state != null">
    state = #{state}
  </if>
  <if test="title != null">
    AND title like #{title}
  </if>
  <if test="author != null and author.name != null">
    AND author_name like #{author.name}
  </if>
</select>
```

What happens if none of the conditions are met? You would end up with SQL that looked like this:

```
SELECT * FROM BLOG
WHERE
```

This would fail. What if only the second condition was met? You would end up with SQL that looked like this:

```
SELECT * FROM BLOG
WHERE
AND title like ‘someTitle’
```

This would also fail. This problem is not easily solved with conditionals, and if you’ve ever had to write it, then you likely never want to do so again.

MyBatis has a simple answer that will likely work in 90% of the cases. And in cases where it doesn’t, you can customize it so that it does. With one simple change, everything works fine:

```
<select id="findActiveBlogLike"
     resultType="Blog">
  SELECT * FROM BLOG
  <where>
    <if test="state != null">
         state = #{state}
    </if>
    <if test="title != null">
        AND title like #{title}
    </if>
    <if test="author != null and author.name != null">
        AND author_name like #{author.name}
    </if>
  </where>
</select>
```

The *where* element knows to only insert "WHERE" if there is any content returned by the containing tags. Furthermore, if that content begins with "AND" or "OR", it knows to strip it off.

If the *where* element does not behave exactly as you like, you can customize it by defining your own trim element. For example, the trim equivalent to the *where* element is:

```
<trim prefix="WHERE" prefixOverrides="AND |OR ">
  ...
</trim>
```

The *prefixOverrides* attribute takes a pipe delimited list of text to override, where whitespace is relevant. The result is the removal of anything specified in the *prefixOverrides* attribute, and the insertion of anything in the *prefix* attribute.

There is a similar solution for dynamic update statements called *set*. The *set* element can be used to dynamically include columns to update, and leave out others. For example:

```
<update id="updateAuthorIfNecessary">
  update Author
    <set>
      <if test="username != null">username=#{username},</if>
      <if test="password != null">password=#{password},</if>
      <if test="email != null">email=#{email},</if>
      <if test="bio != null">bio=#{bio}</if>
    </set>
  where id=#{id}
</update>
```

Here, the *set* element will dynamically prepend the SET keyword, and also eliminate any extraneous commas that might trail the value assignments after the conditions are applied.

If you’re curious about what the equivalent *trim* element would look like, here it is:

```
<trim prefix="SET" suffixOverrides=",">
  ...
</trim>
```

Notice that in this case we’re overriding a suffix, while we’re still appending a prefix.

## foreach

Another common necessity for dynamic SQL is the need to iterate over a collection, often to build an IN condition. For example:

```
<select id="selectPostIn" resultType="domain.blog.Post">
  SELECT *
  FROM POST P
  WHERE ID in
  <foreach item="item" index="index" collection="list"
      open="(" separator="," close=")">
        #{item}
  </foreach>
</select>
```

The *foreach* element is very powerful, and allows you to specify a collection, declare item and index variables that can be used inside the body of the element. It also allows you to specify opening and closing strings, and add a separator to place in between iterations. The element is smart in that it won’t accidentally append extra separators.

**NOTE** You can pass a List instance or an Array to MyBatis as a parameter object. When you do, MyBatis will automatically wrap it in a Map, and key it by name. List instances will be keyed to the name "list" and array instances will be keyed to the name "array".

This wraps up the discussion regarding the XML configuration file and XML mapping files. The next section will discuss the Java API in detail, so that you can get the most out of the mappings that you’ve created.

## bind

The `bind` element lets you create a variable out of an OGNL expression and bind it to the context. For example:

```
<select id="selectBlogsLike" resultType="Blog">
  <bind name="pattern" value="'%' + _parameter.getTitle() + '
%'" />
  SELECT * FROM BLOG
  WHERE title LIKE #{pattern}
</select>
```

## Multi-db vendor support

If a databaseIdProvider was configured a "_databaseId" variable is available for dynamic code, so you can build different statements depending on database vendor. Have a look at the following example:

```
<insert id="insert">
  <selectKey keyProperty="id" resultType="int" order="BEFORE"
>
    <if test="_databaseId == 'oracle'">
      select seq_users.nextval from dual
    </if>
    <if test="_databaseId == 'db2'">
      select nextval for seq_users from sysibm.sysdummy1"
    </if>
  </selectKey>
  insert into users values (#{id}, #{name})
</insert>
```

## Pluggable Scripting Languages For Dynamic SQL

Starting from version 3.2 MyBatis supports pluggable scripting languages, so you can plug a language driver and use that language to write your dynamic SQL queries.

You can plug a language by implementing the following interface:

```
public interface LanguageDriver {
  ParameterHandler createParameterHandler(MappedStatement map
pedStatement, Object parameterObject, BoundSql boundSql);
  SqlSource createSqlSource(Configuration configuration, XNod
e script, Class<?> parameterType);
  SqlSource createSqlSource(Configuration configuration, Stri
ng script, Class<?> parameterType);
}
```

Once you have your custom language driver you can set it to be the default by configuring it in the mybatis-config.xml file:

```
<typeAliases>
  <typeAlias type="org.sample.MyLanguageDriver" alias="myLang
uage"/>
</typeAliases>
<settings>
  <setting name="defaultScriptingLanguage" value="myLanguage"
/>
</settings>
```

Instead of changing the default, you can specify the language for an specific statement by adding the `lang` attribute as follows:

```
<select id="selectBlog" lang="myLanguage">
  SELECT * FROM BLOG
</select>
```

Or, in the case you are using mappers, using the `@Lang` annotation:

```
public interface Mapper {
  @Lang(MyLanguageDriver.class)
  @Select("SELECT * FROM BLOG")
  List<Blog> selectBlog();
}
```

**NOTE** You can use Apache Velocity as your dynamic language. Have a look at the MyBatis-Velocity project for the details.

All the xml tags you have seen in the previous sections are provided by the default MyBatis language that is provided by the driver `org.apache.ibatis.scripting.xmltags.XmlLanguageDriver` which is aliased as `xml`.