# Selects Configuration Reference

The out of the box DAO database operation include the CRUD, the byExample methods as well as the search and navigation by primary key, unique indexes, and foreign keys.

Even though, a whole lot can be done using these out of the box DAO methods, at some point more specialized SQL select methods are suitable to join multiple tables, group rows, or use database enhanced SQL syntax. The out of the box DAO methods can help only so much to deal with the increasing needs the application user starts to demand.

HotRod provides a way of automating the execution of arbitrary SQL select statements by using <select> tags. A <select> tag produces a DAO whose properties are the columns returned by the SQL statement, fully typed and named, and that includes a single Java method that executes the specified SQL statement.

Unlike the <update> tags, the <select> tags produce fully independent DAO classes and do not need to be included inside another DAO. In other words they are defined at the first level, just by the side of the <table>, <view>, or <dao> tags. The body of the <select> supports a single SQL select.

## The <select> Tag

The <select> tag attributes are:

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required** |
| java-class-name | The name of the DAO Java class generated for this SQL statement. It must be a valid Java class name starting with an upper case letter. The name must be unique and different to all other names produced from a <table>, <view>, <select>, and/or <dao> tag since they live in the same namespace. | Yes |
| complement-start | The start delimiter for a complement SQL section, if different from {\* | No |
| complement-end | The end delimiter for a complement SQL section, if different from \*} | No |

Now, the following <select> tag illustrates a few features of the definition:

<select java-class-name="BigTransaction">

<![CDATA[

select a.\*, t.recorded\_at, t.amount

from account a

join transaction t on (t.account\_id) = (a.id)

where t.amount &gt; 100.00

]]>

</select>

This <select> tag will produce the BigTransaction DAO with a static select() method:

public static List<BigTransaction> select() { ... }

As you see the execution of the select() method return a java.util.List of the DAOs. Also note that, even though the name of the class does not end with DAO this class is still one, since it communicates with the database.

It's important to note that the resulting columns of the SQL select must all have different names. This is necessary to produce the properties of the DAO java class. If you try to use a DAO with duplicate column names HotRod will stop the code generation and will show an error.

## CDATA and XML entities

Please note the SQL statement above includes a greater than comparison in the last lines. Since the > character is a reserved XML character it's escaped as &gt;. If you forget to escape it, this character could be interpreted as a start or end of an XML tag, and you would be entering the realm of dynamic SQL (described later on in this section). That is why it's escaped to ensure it's interpreted as a simple number comparison rather than an XML tag.

There are three reserved characters that need to be escaped. This means:

* any < must be escaped as &lt;
* any > character must be escaped as &gt;
* any & character must be escaped as &amp;

Also, when using dynamic SQL (not this case) the SQL must be enclosed in a CDATA section—as shown in the example. In the case above there's no dynamic SQL, but it's still a good practice to do it, in case you later on decide to add dynamic tags to it.

## Parameters

The <select> tag can also accept parameters to be applied into the SQL statement. For example, the SQL statement below receives 4 parameters:

<select java-class-name="RankedEmployee">

<![CDATA[

select e.\*, r.position from employee e

left join rank r on (r.employee\_id) = (e.id)

{\*

where r.year =

#{year,javaType=java.lang.Integer,jdbcType=NUMERIC}

and e.branch\_id =

#{branchId,javaType=java.lang.Long,jdbcType=NUMERIC}

order by r.position

\*}

]]>

</select>

This tag will produce the DAO RankedEmployee. This name must be different to all other DAOs produced from a <table>, <view>, <select> and/or <dao> tag since they live in the same namespace. This DAO includes the static DAO java method:

public class RandekEmployee {

public static List<RandekEmployee> select(

java.lang.Integer year,

java.lang.Long branchId) { ... }

}

As you can see both parameters defined in the SQL select statement are included in the Java method.

Each parameter definition takes the form:

#{**name**,javaType=**javatype**,jdbcType=**jdbctype**}

We can see the definition has three sections separated by commas:

* the first one includes the java parameter name.
* the second one is the full class name of the java parameter.
* finally, the last one is the JDBC type of the parameter as specified in the java.sql.Types class. As defined by the JDBC specification the JDBC type is needed for cases when the java parameter is null.

If the same parameter needs to be inserted twice or more in the SQL statement, only the first occurence should have the "full definition" as shown above. The second and subsequent occurrences should include the "short definition" that only includes the name of it, as in:

#{**name**}

In a <select> tag parameters can only be present in a complement SQL section, the sections enclosed in {\* and \*}. In the previous example you probably noticed the where and order by sections surrounded by {\* and \*}.

## Complement SQL Section

The {\* and \*} delimiters mark which sections of the SQL are complements vs a foundational parts of it. HotRod requires the developer to differentiate these sections to retrieve the column metadata of the SQL select.

The short explanation is that all parts of the SQL select starting from the where statement are to be enclosed by the {\* and \*} delimiters.

The longer explanation: In order to automatically find out the full list of the columns the resulting tabular data the SQL select statement produces, HotRod uses the strategy of creating a temporary database view with it, and then retrieves the database view metadata. Of course, the temporary view must be created using a fully valid SQL create view statement and there is where the “foundation” and “complement” sections play a key role. The foundation sections of the SQL are used when creating the view, while complement SQL sections are ignored while creating it.

Therefore, the foundation SQL section—that is all the SQL code outside the {\* and \*} delimiters—must be a valid SQL select statement that can be used to create a view. That is, it cannot have an order by or other SQL sections your database deems unfit for a view.

The previous example would generate the following view:

create view hotrod\_temp\_view\_001 as

select e.\*, r.position from employee e

left join rank r on (r.employee\_id) = (e.id)

Note all the complement SQL sections included by {\* and \*} are excluded, so this SQL statement can actually be executed. Once the view is created the column metadata is easily retrieved, and the DAO Java class can be fully produced.

Also, note the temporary view name is based on the configuration parameter temp-view-base-name on the header section of HotRod's configuration file.

Please note the temporary view is created, used, and removed automatically by HotRod behind the scenes. Once the code generation is complete the view is automatically dropped.

To decide which sections to mark as a complement SQL section consider the following examples:

select ... from ... join ... {\* where ... \*}

select ... from ... join ... {\* order by ... \*}

select ... from ... join ... {\* group by ... \*}

select ... from ... join ... {\* group by ... having ... \*}

select ... from ... join ... {\* union ... \*}

select ... from ... join ... {\* intersect ... \*}

select ... from ... join ... {\* except ... \*}

select ... from ... join ... {\* where ... order by ...

group by ... having ... union ... intersect ... except ... \*}

That is, the SQL sections select, from and join sections are included, but everything else should be excluded.

Now, if the SQL statement happens to include the delimiters {\* and/or \*} you can specify custom delimiters using the complement-start and/or complement-end attributes of the <select> tag, as in:

<select java-class-name="RankedEmployee"

complement-start="//\*" complement-end="\*//">

<![CDATA[

select count(\*), avg(sales\_price) as price

from property p

//\*

where type like '{\*' or classification = '\*}'

\*//

]]>

</select>

This way, the SQL sections will be correctly interpreted.

## Columns

The resulting columns of the SQL select are automatically discovered by HotRod. Their Java names and Java types are based on the resulting name and database type of the column in the result set. This is because a column in the result set may not only correspond to a table or view column, but could also be a runtime expression with a name and valid column type.

The result set column names must be all different to each other since they are used to produce the default DAO Java properties names.

The result set columns are used to produce the default DAO Java properties types, using the same rules used when generating properties for the <table> and <view> tags.

In most cases the names and types would be suitable for the application. However, if it happens the developer considers a different name or type is better suited for the application requirements he/she can override the default values using a <column> tag. This can be quite useful for cryptic column names, specially on legacy databases.

A <column> tag can be specified for one or more columns of the result set. If no <column> tag is specified for a column, the default name and type produced by HotRod are used.

<column> tags need to be added outside the CDATA section. If added inside they don't have any effect while generating the DAO, and will most likely produce a runtime error.

The example below shows a column tag superseding the default column name and type.

<select java-class-name="AvailableBook">

<column name="bkcurpc" java-name="price"

java-type="java.lang.Double" />

<![CDATA[

select \* from book

{\*

where available = 1

\*}

]]>

</select>

Please note:

* The <column> tag is placed outside the CDATA section.
* The result set returns a column for the price with the cryptic name bkcurpc but the DAO will have a property (including getters and setters) with the name price.
* The result set returns a DECIMAL column type with two decimals (not shown in the example) that HotRod would treat by default as a java.math.BigDecimal. The DAO, however, will use the type java.lang.Double overriding the default type.
* Finally, the java-name and java-type attributes do not need to be specified simultaneously, and can be used separately as needed. For more details on how to use the <column> tag see the Configuration Reference section for Columns.

## Dynamic SQL

So far, we have seen the <select> tag to apply parameter values to SQL statements and execute them. However, we haven't seen the SQL statement changing its internal structure.

When using MyBatis, HotRod provides dynamic SQL capabilities, that allow the SQL statement to change at runtime depending on the specified parameter values.

The following example shows a SQL select that adds fragments to the SQL statement to filter and order rows conditionally:

<select java-class-name="applyFreeShipping">

<![CDATA[

select \* from catalog

<where>

<if test="#{parentCatalog,javaType=java.lang.Long,jdbcType=NUMERIC} != null">

parent\_catalog = #{parentCatalog}

</if>

</where>

<if test="#{sort,javaType=java.lang.Boolean}">

order by catalog\_name

</if>

]]>

</update>

The example above retrieves rows from the CATALOG table conditionally filtered by the PARENT\_CATALOG column and conditionally sorted by CATALOG\_NAME. Depending on the values provided at runtime for the parentCatalog and sort parameters, the SQL select statement will add each one of the sections enclosed in an <if> tag.

When using MyBatis, the <select> tag (as well as the <update> tag) support dynamic SQL. For more details on how to use it, see the Configuration Reference section for Dynamic SQL.