# Arbitrary SQL Queries Configuration Reference

All but the most basic applications reach a point where arbitrary SQL queries need to be devised to perform database changes in an optimal manner.

On very simple applications most of the persistence can be dealt with using the basic standard CRUD methods but sooner or later each application reaches a point where it needs to use more advanced custom made SQL statements. This is when the <query> tag comes into place by providing the developer with an easy way to execute any arbitrary SQL update, delete, or insert statements directly by calling a DAO class method.

In HotRod one or more <query> tags can be added to any <table>, <view>, or <dao> tag to produce Java methods that execute the SQL statement included in the body of the <query> tag.

## The <query> Tag

The <query> tag attributes are:

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required** |
| java-method-name | The name of the DAO Java method that will execute the SQL statement. It must be a valid Java method name starting with a lower case letter. | Yes |

The body of the <query> supports a single SQL statement. It's recommended its body to be enclosed in a CDATA section. That is, an <query> tag could look like:

<table name="promotion">

...

<query java-method-name="applyAccountPromotion74">

<![CDATA[

update account set balance = balance + 100

where type = 6

]]>

</query>

...

</table>

This <query> tag (included in the PROMOTION's <table> tag) will produce the static method:

public static int applyAccountPromotion74() { ... }

in the PromotionsDAO java class. This method executes the included SQL update statement. It returns an int value that corresponds to the total number of affected database rows.

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

<query java-method-name="applyAccountPromotion75">

<![CDATA[

update account a set balance = balance +

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

where created\_on

between #{from,javaType=java.util.Date,jdbcType=TIMESTAMP}

and #{to,javaType=java.util.Date,jdbcType=TIMESTAMP}

and (select count(\*) from transaction

where account\_id = a.id) &gt;

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

]]>

</query>

This tag will produce the static DAO java method:

public static int applyAccountPromotion75(

java.lang.Integer reward,

java.util.Date from,

java.util.Date to,

java.lang.Integer minTransactions) { ... }

All four parameters defined in the SQL statement are included in the corresponding 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**}

Also, it's important to mention that HotRod doesn't impose or require any relationship to exist between the SQL statement used by the <query> tag and the <table>, <view>, or <dao> tags that enclose it. The DAO Java classes produced by these tags only serve as containers to place the Java method that executes the SQL query.

## Deleting and Inserting

The <query> tag can actually include any valid SQL statement that does not return tabular data. In particular, it can be used to execute SQL delete or SQL insert statements. For example, the definition below defines a more complex SQL delete:

<query java-method-name="wipeClosedAccounts">

<![CDATA[

delete from account a

where state in ('CLOSED', 'REJECTED')

and account\_type =

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

and last\_activity &lt;

#{cutDate,javaType=java.util.Date,jdbcType=TIMESTAMP}

]]>

</query>

This tag will produce the static DAO java method:

public static int wipeClosedAccounts(

java.lang.Integer type,

java.util.Date cutDate) { ... }

This method will execute the specified SQL delete statement applying both parameter values, and will return the number of deleted rows as an int. Using the same strategy a SQL insert can be used to insert one or multiple rows using any valid combination of parameters, embedded SQL select statement, etc.

## Dynamic SQL

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

When using MyBatis, HotRod provides dynamic SQL capabilities.

The following example shows how a SQL update conditionally adds conditions to the SQL statement:

<query java-method-name="applyFreeShipping">

<![CDATA[

update outstanding\_order set free\_shipping = 1

<where>

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

region = #{region}

</if>

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

AND type = #{clientType}

</if>

<if test="#{productType,javaType=java.lang.String,jdbcType=VARCHAR} != null">

AND productType = #{productType}

</if>

</where>

]]>

</query>

The example above applies free shipping to order, depending on any combination of region, clientType, and/or productType. Depending on the non-null values provided at runtime, the SQL update statement will add (or not) each one of the sections enclosed in an <if> tag.

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