# Tables Configuration Reference

The tables configuration is probably the most important section in the configuration file. Tables usually make up the core of the database objects any application wants to work with and this section is by far the more configurable one, by providing multiple options and tweaks to the developer.

## The <table> Tag

The <table> tag tells HotRod which tables you want to use. Each <table> tag adds one table to the code generation and produces a DAO. If the database schema has more tables in it they will be ignored if not listed using a <table> tag. Essentially HotRod uses a white list approach when configuring tables.

The <table> tag attributes are:

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required** |
| name | The name of the database table. | Yes |
| java-name | The name of the DAO Java class, if different from the default name produced by HotRod. | No |
| column-seam | A character String that glues columns together when naming methods that use composite columns in their names. Used when navigating composite FKs and composite unique indexes. | No |

A single table definition can be very simple—it can only declare the name of a table—or can very quite long—by including numerous extra attributes and sub-tags to add extra functionality or tweaks. In its simplest form the <table> tag looks like:

<table name="application\_log" />

This informs HotRod there's an APPLICATION\_LOG table that is to included in the code generation. The DAO Java name—and the properties names for its columns—will be generated by default from the database table structure, and the Java types will also be produced using the default rules HotRod has for this specific database. Basically, there is no custom configuration on it.

The name of the DAO Java class is taken from the table name, and prepended and appended with the prefix and suffix as specified in the header configuration section of the configuration file. By default there's no prefix, and by default the suffix is DAO. Therefore, in the example above the APPLICATION\_LOG table will produce a DAO Java class ApplicationLogDAO.java.

Now, if you want to use a different name for the DAO Java class, you can specify it in the <table> tag using the java-name attribute. Mind this name must be a valid Java class name, starting with an upper case letter. If, for example, we wanted to use a custom name for the DAO the configuration could look like:

<table name="application\_log" java-name="AppLog" />

This configuration will produce the DAO Java class AppLog.java. Notice that when you openly specify the DAO Java name, HotRod does not prepend or append the prefix and suffix, but it takes it exactly as you declared it.

## The <auto-generated-column> tag

Most tables with a primary key produce their values either as an auto-generated column with values provided by the database itself, or using a sequence whose values are also generated by the database. Some databases offer the first solution, other ones offer the second, and some databases offer both options.

The <auto-generated-column> tag tells the code generator the database will provide values for the each row when it's inserted into the table. Because of the different PK generation strategies mentioned before the <auto-generated-column> tag can take different forms.

The <auto-generated-column> tag attributes are:

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required** |
| name | The name of the primary key column. | Yes |
| sequence | The name of the database sequence to use when generating values for the PK while inserting new rows. If not specified, the PK is considered to be an identity PK. | No |
| allows-specified-value | Tells the code generator that if the DAO property for the PK has a non-null value, this value should be used while inserting the row, ignoring the auto-generation mechanism. Valid values are true or false. Defaults to false. | No |

**Identity Primary Key**

If the PK value is generated as an identity PK the table could be configured as:

<table name="client">

<auto-generated-column name="client\_id" />

</table>

This configuration setting tell the code generator the table CLIENT has primary key column CLIENT\_ID whose value is generated by the database on every inserted row. Any value on the clientId DAO property the developer may have set before inserting the row will be ignored, in favor of the value produced by the database. Additionally, the newly inserted PK value will be loaded back into the DAO right after the insertion, so the application can start using it right away.

**Sequence Generated Primary Key**

If the PK value is generated using a database sequence the table could be configured as:

<table name="branch">

<auto-generated-column name="branch\_id" sequence="branch\_seq" />

</table>

These configuration settings tell the code generator the table BRANCH has a primary key column BRANCH\_ID whose value is generated by the database on every inserted row using the sequence BRANCH\_SEQ. Any value on the branchId DAO property the developer may have set before inserting the row will be ignored, in favor of the value produced by the database sequence. Additionally, the newly inserted PK value will be loaded back into the DAO right after the insertion, so the application can start using it right away.

**Optionally Generated Identity Primary Key**

Some databases can allow database tables to honor the PK value set by the application while inserting the row on tables with auto-generated PK values. If you happen to have a database or a table configured to support this feature, it can be configured in HotRod as:

<table name="purchase">

<auto-generated-column name="purchase\_id"

allows-specified-value="true" />

</table>

These configuration settings tell the code generator the table PURCHASE has a primary key column PURCHASE\_ID whose value is generated by the database on every inserted row. If no value is set (or a null value is set) on the purchaseId property of the DAO the database will generate the PK value, and this newly generated value will be loaded back into the DAO right after the insertion. However, if a non-null value is set on the purchaseId property of the DAO this value will be honored while inserting the row so no auto-generated value will be produced by the database.

The example below for H2 database depicts the usage of this feature using plain SQL:

create table test(id bigint auto\_increment, name varchar(255));

insert into test(name) values('hello');

insert into test(id, name) values(123, 'world'); -- Overrides PK!

select \* from test;

The last SQL select shows both rows inserted with the PK values 1 and 123. If the id column hadn't been specified on the second insert its PK value would have been 2 instead of 123.

The following table shows which databases support the optional identity PK value:

|  |  |
| --- | --- |
| **Database** | **Support Details** |
| Oracle Database | Not supported on Oracle 11g or older. Starting on Oracle 12c there is support for identity PKs. If using Oracle 12c, then:   * Allowed on generated by default PKs. * Not allowed on generated always PKs. |
| IBM DB2 | Allowed on generated by default PKs.  Not allowed on generated always PKs. |
| SAP ASE (ex-Sybase) | Not supported by HotRod.  SAP ASE provides this functionality (turned off by default) by turning it on temporarily in the session for a single table at a time using:  set identity\_insert <TABLE> on  Since this strategy is cumbersome and error prone HotRod does not support explicit PK values in SAP ASE. |
| Microsoft SQL Server | Not supported by HotRod.  SQL Server provides this functionality (turned off by default) by turning it on temporarily in the session for a single table at a time using:  set identity\_insert <TABLE> on  Since this strategy is cumbersome and error prone HotRod does not support explicit PK values in Microsoft SQL Server. |
| PostgreSQL | Allowed on all smallserial, serial, and bigserial columns. |
| MySQL | Allowed on all auto\_increment columns. |
| MariaDB | Same as in MySQL: allowed on all auto\_increment columns. |
| HyperSQL (HSQLDB) | Allowed on all identity PKs. |
| H2 | Allowed on all identity PKs. |

## The <version-control-column> tag

Some applications opt to implement the Optimistic Locking strategy to deal with data concurrency issues while updating and/or deleting database rows.

Optimistic Locking is a persistence strategy that considers the probability of a row being updated or deleted by another process while the current thread is working with it is fairly low. This strategy requires the table to have a PK, and works by designating another numeric column of the table as the "version control value" of each row. This way it can find out when a row had been updated by another process or thread.

The <version-control-column> tag attributes are:

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required** |
| name | The name of the table column to be used for row version number purposes. Must be a column of a numeric type, specifically of an integer-like type. | Yes |

HotRod implements this strategy automatically behind the scenes if configured to do so. To implement optimistic locking on a table its configuration should look like:

<table name="agent">

<auto-generated-column name="agent\_id" />

<version-control-column name="row\_version" />

</table>

The configuration above tells the code generator the table AGENT will use optimistic locking while updating and deleting by primary key, by using the column ROW\_VERSION column for version control purposes.

In the example above, if the application reads a row and later tries to update it, the value of the rowVersion DAO property will be checked against the value of the ROW\_VERSION column of the table row. If they match the update operation will actually take place. Otherwise—when the row had been updated or deleted by another process or thread—, the update operation will fail and will throw a StaleDataException.

Also, in the example above, if the application reads a row and later tries to delete it, the value of the rowVersion DAO property will be checked against the value of the ROW\_VERSION column of the table row. If they match the delete operation will actually take place. Otherwise—when the row had been updated or deleted by another process or thread—, the delete operation will fail and will throw a StaleDataException.

**Version control column values**

The values of the version control columns are set according to the following rules. When inserting new rows, any value the application may have set on DAO property of the version control column is ignored, and the initial value is inserted on table column of the new row.

On every row update the value incremented, and may eventually cycle back to the initial values after reaching its maximum value.

The initial, maximum, and minimum values depend on the column data type. The initial value is by default zero. The maximum and minimum values depend on the specifics of the column data type. For example, if the column is of type NUMERIC(5), then the maximum value will be 99999 and the minimum value will be -99999 or zero, depending on if the database accepts negatives number or not for this specific column type.

However, the default initial, maximum, and minimum values can be overridden by the developer by specifying a <column> tag that defines the attributes initial-value, max-value, and min-value respectively. See the <column> tag definition for details.

Not all columns are suitable to be used as version control columns. First, they must be numeric types. Second, they must accept integer numbers only. Therefore, a VARCHAR does not qualify, and neither a DOUBLE, a DATE, or a DECIMAL(10, 2) do. Types such as INTEGER, NUMERIC(10) or BIGINT usually qualify, but this in the end depends on the specifics of each database.

## The <column> Tag

Each <table> tag can include one or more <column> tags to stipulate specific properties for some or all the columns of a table. The table columns with no <column> definition use the default properties provided by the code generator.

For details on the settings of a <column> tag see the Configuration Reference section for Columns.

## The <sequence> Tag

Each <table> tag can include one or more <sequence> tags. These definitions generate a Java method to directly retrieve a value from the specified sequence. There's no relationship between the named sequence and the table tag; the table DAO only serves as a Java class where to place the corresponding Java method to retrieve the value.

For details on the settings of a <sequence> tag see the Configuration Reference section for Sequences.

## The <query> Tag

Each <table> tag can include one or more <query> tags. Each <query> tag contains a SQL statement the developer wants to attach as a method to the DAO. There's no relationship between the SQL statement and the table; the table DAO only serves as a Java class where to place the corresponding Java method that executes the SQL statement. The SQL statement must not necessarily be an SQL update, but can also be an SQL insert, or SQL delete.

For details on how to define an <query> tag see the Configuration Reference section for Updates.