HotRod Manual

Automated Data Persistence for Java

September 2016

Introduction

HotRod is a comprehensive DAO generator for MyBatis and SpringJDBC.

For MyBatis, it generates DAO Java classes, and MyBatis mappers for database tables, views, and select queries.

For SpringJDBC it generates...

It's assummed that you have knowledge of MyBatis or Spring JDBC in order to use HotRod, since HotRod automates the generation of the persistence layer. When running the code, it's MyBatis or Spring JDBC who actually executes all SQL statements.

HotRod works by retrieving the database metadata from a live database through JDBC. This means that you first create the tables and views in a database schema, and then you run HotRod to automaticall produce your DAO Java classes and mappers.

The generated DAOs provide simple methods to access the database. Primary keys make up the structural base for the simple select, update, and delete methods. Unique indexes, foreign keys, and database views provide the underlying structure to generate complex SQL selects. Using DAOs as data examples, methods are also provided to select, update, or delete multiple rows at a time. All these operations are available through simple Java DAO methods.

Complex SQL select statements with joins are also automated by DAOs. When a SQL select is specified in the configuration file a generated DAO offers simple methods with parameters to execute any intricate SQL select, even with database specific exotic features. This greatly boost the development speed since the developer does not need to write parameter or result-set Java classes for each complex query anymore or tedious, error-prone JDBC code.

HotRod works with Oracle, PostgreSQL, Sybase, DB2, MySQL, and HyperSQL. It automatically resolves column types to Java types depending on the column type for the specific database. Nonetheless, these default types can be overridden by the developer using the configuration file.

In terms of transactions, all persistence methods operate by default in auto-commit mode. When transactions are required for multiple operations the are readily available from within the DAOs, in order to initiate transactions and later to commit or to roll them back. Transactions can also be locally controlled, or can participate in external transactions using JTA (this is a feature of MyBatis). Transactions can also be linear or interleaved, the latter only available if supported by the database and JDBC driver.

DAO Generation

HotRod generates the Java classes (DAOs) and the mappers that MyBatis need for the DAO persistence. In order to do that you'll need to configure the generation and then run it.

It's necessary to have a basic understanding on how MyBatis works. A short perusing in the MyBatis web site can help explaining the details on how it works. In short, MyBatis is a lightweight library that executes SQL queries and maps Java POJOs into SQL parameters back and forth.

The HotRod generation reads a database structure and generates all the files MyBatis needs. Every time you make changes to the database structure, you can run the generation again to reflect the late changes of it.

To run the generation you'll need to provide:

1. HotRod library (.jar).
2. The JDBC driver of your database.
3. An Ant target that executes the generation:
```
<taskdef name="empusamb" classname=
    "org.nocrala.tools.persistence.empusamb.ant.EmpusaMbAntTask">
  <classpath>
    <pathelement location="empusa-mybatis-2.3.jar" />
    <pathelement location="${driverclasspath}" />
  </classpath>
</taskdef>
<empusamb driver="${driverclass}"
          url="${url}"
          userid="${username}"
          password="${password}"
          division="${schema}"
          configfile="dao-config.xml"
          display="list" />
```
4. The DAO configuration file.
5. The MyBatis Template file.
6. A Java session factory class.

You can see all the above in action in the Quick Start example included in this release.

Per each table, view, and select specified in the DAO configuration file the generator will create:

- A DAO Java class. Never overwritten.
- A DAO Java primitive class. Overwritten on every DAO generation.
- A mapper file. Overwritten on every DAO generation.

Additionally, it will generate the main MyBatis configuration file together with the mappers under the name `mybatis-configuration.xml`. This file will include all generated mapper files, as well as any custom mapper you may have written manually found in the directory `custom`, parallel to the `primitives` directory.

The DAO Configuration File

The DAO configuration file is the core of the configuration and specifies all the generation options, including the file system path were the generated files will be placed, as well as the database tables, views, and selects that are going to be included.

The Quick Start example includes the following DAO configuration file:

```
<?xml version="1.0"?>
<!DOCTYPE empusa-mybatis SYSTEM "empusa-mybatis.dtd">

<empusa-mybatis>

  <layout>
    <daos gen-base-dir="src/main/java" package="com.company.daos" />
    <mappers gen-base-dir="src/mybatis" relative-dir="mappers" />
    <mybatis-configuration-template file="mybatis-template.xml" />
    <session-factory singleton-full-class-name=
      "com.company.sessionfactory.SessionFactory" />
    <select-generation temp-view-base-name="mybatis_temp_view" />
  </layout>

  <table name="account">
    <auto-generated-column name="id" />
  </table>

  <table name="transaction">
    <auto-generated-column name="id" />
  </table>

  <view name="account_debit" />

  <collection name="sequences_dao">
    <sequence name="employee_seq" />
    <sequence name="account_seq" />
  </collection>

  <select name="big_deposit">
      <![CDATA[
    select a.name , t.*
      from account a, transaction t
      where t.account_id = a.id
        {* and t.amount >=
           #{minAmount,javaType=java.lang.Integer,jdbcType=NUMERIC} *}
      order by a.current_balance
      ]]>
  </select>

</empusa-mybatis>
```

The main tag in the file is the <empusa-mybatis> tag. In it we'll find one <layout> tag that defines the main parameters, and then one or more <table>, <view>, and/or <select> tags.

The <layout> tag

The layout tag is mandatory. It includes one <daos>, one <mappers>, one <mybatis-configuration-template>, one <session-factory>, and one <select-generation> tag.

The <daos> tag indicates where the DAO java classes are going to be placed. Its attributes are:

gen-base-dir: The base dir of the source java classes.
package: The base package of the DAO java classes, relative to the gen-base-dir attribute.

The <mappers> tag indicates where the mappers are going to be placed. Its attributes are:

gen-base-dir: The base dir of the generation. This dir should be placed as part of the classpath when running your application.
relative-dir: The relative dir, relative to the gen-base-dir attribute, where the mapper files will be located.

The <mybatis-configuration-template> tag specifies which template file will be used to generate the main MyBatis configuration file. This template will specify the data source properties, as well as other MyBatis tweaks, and JDBC properties. See the MyBatis documentation for details of this file. The template must contain the text `@@mappers@@` that will be replaced will the full list of mappers in the resulting MyBatis configuration file. Its attributes are:

file: the location of the MyBatis main configuration file template.

The <select-generation> tag helps the <select> queries generation. Its attribute is:

temp-view-base-name: a database view prefix name that is not used in the database. Views with this prefix will be created and dropped in the database during the DAO generation to retrieve and inspect the <select> queries metadata.

Note: if you use the <select> tag you must have privileges to create and to drop database views in your (dev) database.

The <table> Tag

Add one <table> tag per each table you want to include in the DAO generation.

Its attribute name specifies the table name and is used to generate the DAO. The name must be unique for all <table>s, <view>s, and <select>s.

The <table> tag can include the <auto-generated-column> tag. Use this tag to specify how an auto-generated column (usually the PK) will be generated when inserting a new row in the table. It can take four forms:

- The <auto-generated-column> tag is not specified: In this case the developer will need to provide values for all the properties of the row that's being inserted, including the

PK. This is useful when the developer wants to have full control on the PK values generation, or for tables that don't have a PK.

- The <auto-generated-column> tag is specified and takes a form similar to:
  ```
    <auto-generated-column name="id" />
  ```
  In this case, the DAO generator (and MyBatis) will consider the value for the column `id` will be auto-generated by the database, probably as an identity column, or auto-generated column. The details depend on the specific RDBMS. The value of the `id` column provided by the program (if any) will be ignored, and after the insert is executed the DAO will be automatically populated with the inserted value.
- The <auto-generated-column> tag is specified and takes a form similar to:
  ```
  <auto-generated-column name="id" sequence="account_seq" />
  ```
  In this case, the DAO generator (and MyBatis) will consider the value of the column comes from a database sequence. In this case the database sequence is named `account_seq`. Again, the value of the `id` column provided by the program (if any) will be ignored, and after the insert is executed the DAO will be populated with the inserted value.
- The <auto-generated-column> tag is specified and takes a form similar to:
  ```
  <auto-generated-column name="id" allows-specified-value="true" />
  ```
  In this case, the DAO generator (and MyBatis) will consider the value of the column comes will be auto-generated by the database, probably as an identity column, or auto-generated column. The details depend on the specific RDBMS. If the attribute allows-specified-value is not specified its value defaults to false. Now, if its value is true, and a non-null value is present in the DAO property, the DAO value is used; if the property value is null, then the value will be auto-generated by the database.

The <table> tag can accept one or more <column> tags. These tags take the form.

```
  <column name="column_1" java-type="java.lang.Integer" />
  <column name="column_2" java-type="java.lang.Long" jdbc-type="NUMERIC" />
```

These tags force the DAO generator to use the specified java and JDBC data types to persist the specified columns. This can be suitable when the developer wants to avoid changing the existing java source code when the default java type for the database is not the desired one.

This can help with conversion types, for example, in cases when the developer wants to treat numbers with non-standard precisions: i.e. to treat a NUMERIC(4) as Integer instead of (the default type) Short.

Any column not specifically configured will assume the default java type during the DAO generation. To find out about the default java and JDBC type see the corresponding Appendix for each RBMS at the end of this document.

The jdbc-type attribute is optional. Do not specify it unless you know what you're doing. Its value is retrieved from the database during the DAO generation and corresponds to one of the constant values defined in the java.sql.Types class. Normally you will not specify/override this value, unless you suspect the JDBC driver is reporting a wrong JDBC type.

The <view> Tag

Add one <view> tag per each database view you want to include in the DAO generation.

Its attribute name specifies the view name, and is used to generate the DAO. The name must be unique for all <table>s, <view>s, and <select>s.

The DAO considers all view as read-only queries. Therefore, the only java methods in the DAO are or selecting data from the view.

As with the <table> tag, the <view> tag can include one or more <column> tags to specify the java type for the desired columns.

The <select> Tag

Add one <select> tag per each complex SQL select and/or join you want to use to retrieve data from the database.

The name attribute specifies the name you are giving to the select query and is used to generate the DAO. The name must be unique for all <table>s, <view>s, and <select>s.

As with the <table> tag, the <view> tag can include one or more <column> tags to specify the java type for the desired columns.

The select-prefix and select-suffix attributes are optional and specify which character string to use to delimit the “parameters sections” of the SQL select. They default to {* and *}.

The body of the <select> tag includes the SQL select you want to be executed. One or more parameters can be added in the form:

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

and will make up the parameter list of the related java method. If no parameters are added the SQL select will be executed just as specified.

A parameter can be used several times in the SQL select you want to execute. When adding it multiple time, the first occurrence is considered the definition of the parameter, while subsequent occurrences are considered references to the parameter. The definition must use the complete form of the parameter as in

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

while the references must use the simple form of it (just including the name) as in

```
#{minAmount}
```

Even if you use a parameter many times in the SQL select, the DAO will include it only once in the method call.

Please note that the sections that contain parameters must be enclosed between {* and *}. This is necessary to tell the DAO generator that these sections must be ignored during the generation while creating a database view for the select. These database views are only created temporarily during the generation to retrieve the SQL select metadata and create the DAO accordingly.

The SQL query without the parameters sections is called the “reduced select” and **MUST BE A VALID SQL**; otherwise, the DAO generation will fail and will point out the details accordingly. Therefore, place the {* and *} delimiters strategically, so to ensure the reduced SQL select is a valid query.

Also, in XML is a good practice to specify any non-trivial body text of a tag in a CDATA section. This allows you to indiscriminately use the characters < (less than), > (greater than), & (ampersand), ' (single quote) and “ (double quotes) XML characters without the need of escaping them. You can omit the <![CDATA[and]]> XML delimiters but I highly recommend using them to simplify your life.

DAO Java classes structure

Each generated DAO Java class is actually divided into two Java classes: the main DAO (mostly empty) class and the DAO primitives class.

The main DAO class is the class you'll use most of the time. This class offers you the room to add custom Java logic to suit any domain-related behavior. This class is never overwritten by the DAO generator so your valuable custom code is safe.

The DAO primitives class includes all the “goodies” you automatically get from the DAO generator. This class is overwritten every time you execute the DAO generator.

For ALL tables, views, and select, the DAO generator includes:

- All columns of the related table, view, or select as Java properties.
- All the corresponding getters and setters.

For tables it includes:

- The select(), update() and delete() methods: these methods use the PK to operate and, therefore, are available only if the table has a primary key.
- The selectByUICOLUMNS() methods. This methods retrieve a DAO (database row) using a unique index. There are as many of these methods as the number of unique constraint the table has.
- The selectByExample(), updateByExample(), and deleteByExample() methods. These methods are always present, and can retrieve or update multiples rows at a time.
- The selectChildrenTABLE().byCOLUMNS() and selectParentTABLE().byCOLUMNS() methods. These methods allows you to retrieve related DAO objects (database rows) using the foreign keys present in the database. There are as many methods as imported and exported foreign keys the table has.

For views it includes:

- The selectByExample() method.

For select it includes:

- The select() method. This method includes all the parameters defined in the body of the <select> tag, if any, as Java parameters.

A Word on the Development Effort of the Persistence Layer

The persistence layer of a Java application are the Java operations that interact with the database in order to retrieve data, or to produce data changes in the database: in sum all the SQL select, insert, update, and delete statements that the application needs to fully operate. These operations are highly dependent on the database details, and need to be in sync with all its changes to operate correctly. In particular they need to consider all the table details, columns, data types, tables constraints, relationships, etc.

The persistence layer--the set of persistence operations that perform specific database tasks--can be grouped into operations related to specific areas such as tables, views, and/or other general queries. We can organize all these operations as methods of DAOs, where each DAO represents one the groups. Considering this strategy, the effort of developing the entire persistence layer of a Java application could be considered as the sum of the efforts of coding each DAO necessary for the application. Now, the effort of writing a DAO usually boils down to:

❶ Map the related database column types to suitable Java types.

❷ Write the DAO Java class with its properties, setters, and getters.

❸ Write the DAO methods with all the tedious JDBC code that sends and retrieves data to and from the database.

❹ Free JDBC resources after interacting with the database, even under error conditions.

On the other hand, we can group the same persistence operations by their complexity and reusability--something that is usually tied to their level of optimization. This grouping is shown below, ordered from the most basic ones--meant to be highly reusable--to the complex and very tailored ones--meant to be very optimized but probably not very reusable:

Basic CRUD Operations
These are the most basic database operations to interact with tables. They include the typical select by PK, update by PK, delete by PK, and insert using sequences or auto-generated PKs.

Advanced Operations
These vary, but usually include select, update, and delete by example. They may use database views to retrieve data with joins and complex filtering. They may also leverage the database constraints to navigate the data model using unique and foreign key constraints. Advanced operations mostly fall in the SQL standard.

Exotic SQL
These are highly sophisticated and tuned parameterized SQL selects, updates, deletes, and inserts that include one or more of the features below:

- Manage data in a massive way.

- Join multiple tables and retrieve combined join data.
- Heavily use functions.
- Limit returned/processed rows.
- Prepare in-memory or temporary structures for cubes, pivoting or other.
- Add hinting to specify execution plans.
- Use other non-standard SQL extensions specific to the RDBMS.

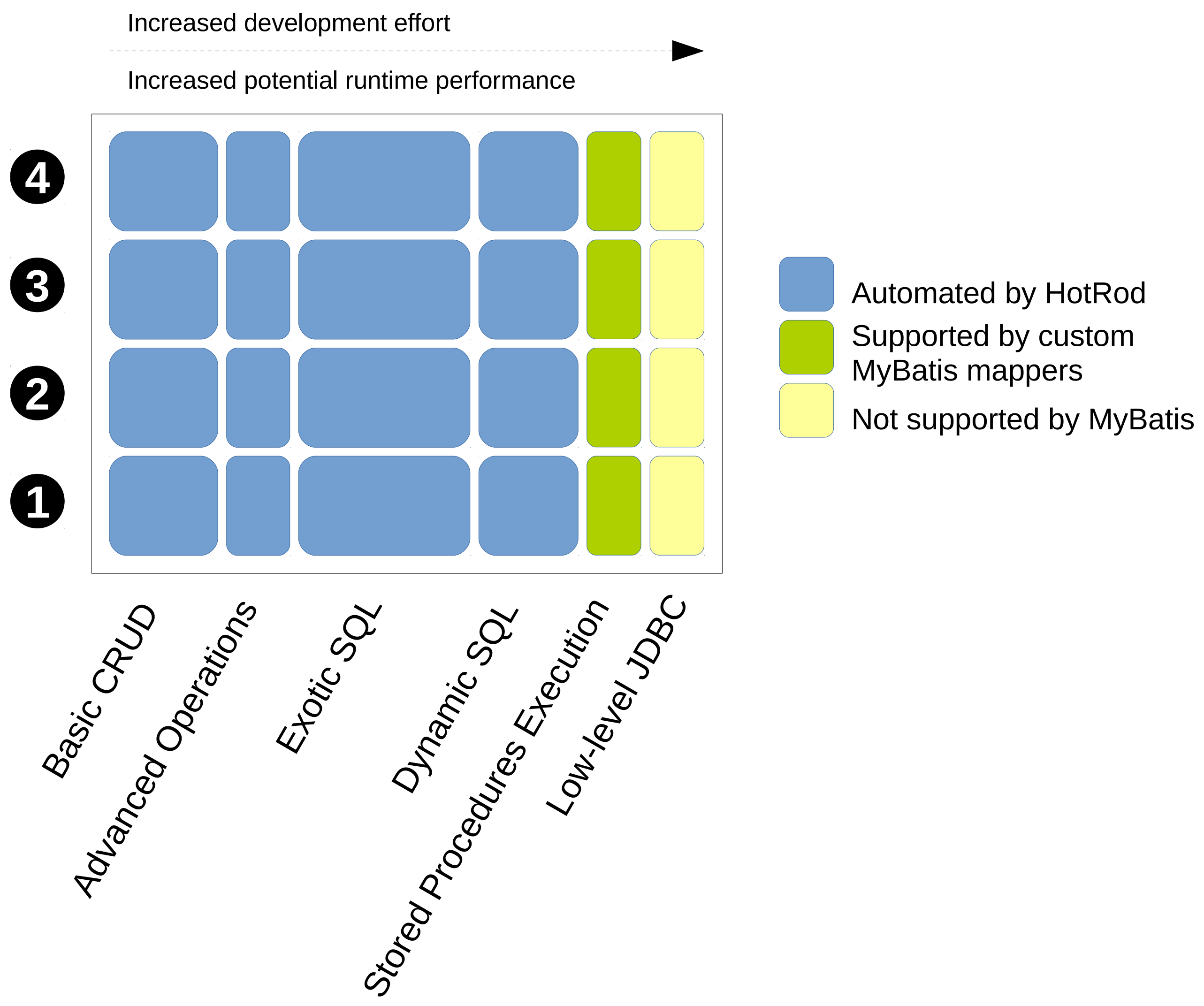
These SQL statements are carefully written to optimize the resources and speed using indexes, table segmentation, in-memory access and other database features. They can also severely reduce the network calls and latency needed to produce the desired data change.

Dynamic SQL
These are SQL statements (select, insert, update, and delete) enhanced by an extra descriptive layer of logic (such as OGNL) that automatically alters their structure based on the received parameters. A very simple example of this is a SQL select that filters by different columns depending on a logic related to the values of the received parameters.

Stored Procedures Execution
These are simple calls to existing stored procedures in the database. The development effort includes the writing of Java classes to send parameters and receive results usually in the form of plain values, scalars, lists, or database cursors.

Low-level JDBC
These SQL statements require low-level JDBC to usually access features such as updateable resultsets, low-level locking, batch SQL processing, etc.

The level of effort to develop and test each one of these groups depends on the popularity of them (how heavily used) on each application and also on the complexity of their tweaks and level of customization. The chart below depicts a common the level of effort for each one of the groups as well as the level of support HotRod provides for each group.


Increased development effort
Increased potential runtime performance
4
3
2
1
Automated by HotRod
Supported by custom MyBatis mappers
Not supported by MyBatis
Basic CRUD
Advanced Operations
Exotic SQL
Dynamic SQL
Stored Procedures Execution
Low-level JDBC

In this chart the rectangles depict the level of development effort if all aspects (1, 2, 3, and 4) are coded by hand. For example, “Advanced Operations” is quite narrow, since even though these are not difficult to write, the may more difficult to test. “Exotic SQL”, on the other hand, are heavily used in any but the most basic applications.

The chart also depicts in blue the DAO operations that HotRod automatically covers, and in green the operations that can be used using custom MyBatis mappers. The area in yellow depicts JDBC usage out of the scope of MyBatis and also out of the scope of HotRod.

From left to right the complexity increases, and so does the potential performance that can be obtained from JDBC.

Appendix A - Default Data Types for Oracle database

The HotRod Oracle adapter automatically maps known column types to Java types. The table below shows which Java types are generated by the DAO Generator for each Oracle column type.

Please note that the Java types for the Oracle columns may vary depending on the specific version and variant of Oracle, the operating system where the database engine is running, and the JDBC driver version.

Default Java Types

If no special rule is defined for a column in the configuration file the DAO generator will use the following rules to decide which Java type to use for each column. In yellow is the DAO property type. In parenthesis the actual object type returned by the Oracle JDBC driver, that sometimes is different.

Oracle Data Type	Default Java Type

`NUMBER(p,s),`
`DECIMAL(p,s),`
`DEC(p,s),`
`NUMERIC(p,s),`
`NUM(p,s)`
	If neither p or s are specified:
	• `java.math.BigDecimal`
	If both are specified and s is different from zero the Java type is:
	• `java.math.BigDecimal`
	if p is specified and s is not specified or specified with a value of zero:
	• if p <= 2: `java.lang.Byte`
	• if 2 < p <= 4: `java.lang.Short`
	• if 4 < p <= 9: `java.lang.Integer`
	• if 8 < p <= 18: `java.lang.Long`
	• if p > 18: `java.math.BigInteger`

`SMALLINT,`
`INTEGER,`
`INT`
	`java.math.BigInteger`
	Note: SMALLINT, INTEGER, and INT are equivalent to NUMBER(38).

`FLOAT(p)`	if p is not specified (i.e. a 126-bit float):
	• `java.math.BigDecimal`
	if p is specified:
	• if p <= 23: `java.lang.Float`
	• if 24 <= p <= 52: `java.lang.Double`
	• if p >=53: `java.math.BigDecimal`

`REAL`	`java.math.BigDecimal` **Note**: REAL is equivalent to FLOAT(63).
`DOUBLE PRECISION`	`java.math.BigDecimal` **Note**: DOUBLE PRECISION is equivalent to FLOAT(126).
`BINARY_FLOAT`	`java.lang.Float`
`BINARY_DOUBLE`	`java.lang.Double`

`CHAR(n),` `VARCHAR(n),` `VARCHAR2(n),` `NCHAR(n),` `NVARCHAR2(n)`	`java.lang.String`
`CLOB,` `NCLOB`	`java.lang.String` *
`LONG`	No default java type **
`RAW(n),` `LONG RAW`	`byte[]`
`BLOB`	`byte[]` *
`BFILE`	No default java type **
`DATE`	`java.util.Date`
`TIMESTAMP`	`java.sql.Timestamp`
`TIMESTAMP WITH TIME ZONE`	`java.sql.Timestamp`
`TIMESTAMP WITH LOCAL TIME ZONE`	`java.sql.Timestamp`
`ROWID`	`java.lang.String`
`UROWID`	`java.lang.Object` (oracle.sql.ROWID)
`XMLTYPE`	No default java type **
`URITYPE`	`java.lang.Object` (oracle.sql.STRUCT)
`INTERVAL YEAR TO MONTH`	`java.lang.Object` (oracle.sql.INTERVALYM)
`INTERVAL DAY TO SECOND`	`java.lang.Object` (oracle.sql.INTERVALDS)
`VARRAY(n)`	`java.lang.Object` (oracle.sql.ARRAY)
`STRUCT`	`java.lang.Object` (oracle.sql.STRUCT)
`REF`	`java.lang.Object`

* LOB types are by default read all at once into memory as byte arrays. They can also be read/written using streaming instead of loading them all at once. To do this you'll need to write a custom MyBatis TypeHandler.

** It may be possible to read/write from/to these columns using a MyBatis custom TypeHandler. This is, however, not an out-of-the-box solution.

Appendix B - Default Data Types for PostgreSQL database

The HotRod PostgreSQL adapter automatically maps known column types to Java types. The table below shows which Java types are generated by the DAO Generator for each PostgreSQL column type.

Please note that the Java types for the PostgreSQL columns may vary depending on the specific version and variant of PostgreSQL, the operating system where the database engine is running, and the JDBC driver version.

Default Java Types

If no special rule is defined for a column in the configuration file the DAO generator will use the following rules to decide which Java type to use for each column.

PostgreSQL Data Type	Default Java Type

`SMALLINT,`
`INT2,`
`SMALLSERIAL`
	`java.lang.Short`

`INTEGER,`
`INT,`
`INT4,`
`SERIAL`
	`java.lang.Integer`

`BIGINT,`
`INT8,`
`BIGSERIAL`
	`java.lang.Long`

`DECIMAL(p,s),`
`NUMERIC(p,s)`
	If neither p or s are specified:
	• `java.math.BigDecimal`
	If s is specified and different from zero the Java type is:
	• `java.math.BigDecimal`
	if s is not specified or specified with a value of zero:
	• if p <= 2: `java.lang.Byte`
	• if 2 < p <= 4: `java.lang.Short`
	• if 4 < p <= 9: `java.lang.Integer`
	• if 8 < p <= 18: `java.lang.Long`
	• if p > 18: `java.math.BigInteger`

`REAL`	`java.lang.Float`
`DOUBLE PRECISION`	`java.lang.Double`
`MONEY`	`java.math.BigDecimal`

`CHAR(n),`
`CHARACTER(n),`
`VARCHAR(n),`
`CHARACTER VARYING(n),`
	`java.lang.String`

`TEXT`	`java.lang.String`

`BYTEA`	`byte[]`
`DATE`	`java.sql.Date`
`TIMESTAMP(n)`, `TIMESTAMP(n) WITHOUT TIME ZONE`, `TIMESTAMPTZ(n)`, `TIMESTAMP(n) WITH TIME ZONE`	`java.sql.Timestamp`
`TIME(n)`, `TIME(n) WITHOUT TIME ZONE`, `TIMETZ(n)`, `TIME(n) WITH TIME ZONE`	`java.sql.Timestamp` *
`BOOLEAN`, `BOOL`	`java.lang.Boolean`
`INTERVAL <fields> (n)`	No default HotRod data type
`XML`	No default HotRod data type
`POINT`, `LINE`, `LSEG`, `BOX`, `PATH`, `POLYGON`, `CIRCLE`	No default HotRod data type
`CIDR`, `INET`, `MACADDR`	No default HotRod data type
`BIT(n)`, `BIT VARYING(n)`	No default HotRod data type
`UUID`	`java.lang.Object` **
`JSON`, `JSONB`	No default HotRod data type

`(arrays, such as)`
`INTEGER[]`,
`CHAR[][]`,
`INTEGER ARRAY`
	No default HotRod data type

`INT4RANGE`, `INT8RANGE`, `NUMRANGE`, `TSRANGE`, `TSTZRANGE`, `DATERANGE`	No default HotRod data type
`(enum data types)`	No default HotRod data type
`(composite types)`	No default HotRod data type

`OID,` `REGPROC,` `REGPROCEDURE,` `REGOPER,` `REGOPERATOR,` `REGCLASS,` `REGTYPE,` `REGROLE,` `REGNAMESPACE,` `REGCONFIG,` `REGDICTIONARY`	No default HotRod data type

* In the special case of a precision of zero, a `java.sql.Time` type would be enough to store any time of the day without fractional seconds. However, since the majority of cases will have a different precision this type defaults to `java.sql.Timestamp` in all cases; this type can handle up to 9 decimal places.
** Even though, the `java.util.UUID` type is able to save a value into the database, apparently it cannot read from the database into a java program. Therefore, the `java.lang.Object` type is safer, but you'll need to cast it after retrieving a value.

Appendix C - Default Data Types for DB2 database

The HotRod DB2 adapter automatically maps known column types to Java types. The table below shows which Java types are generated by the DAO Generator for each DB2 column type.

Please note that the Java types for the DB2 columns may vary depending on the specific version and variant of DB2, the operating system where the database engine is running, and the JDBC driver version.

Default Java Types

If no special rule is defined for a column in the configuration file the DAO generator will use the following rules to decide which Java type to use for each column.

DB2 Data Type	Default Java Type
`SMALLINT`	`java.lang.Short`

`INTEGER,`
`INT`
	`java.lang.Integer`

`BIGINT`	`java.lang.Long`

`DECIMAL(p,s),`
`DEC(p,s),`
`NUMERIC(p,s),`
`NUM(p,s)`
	If s is specified and different from zero the Java type is:
	• `java.math.BigDecimal`
	if s is not specified or it's zero:
	• if p <= 2: `java.lang.Byte`
	• if 2 < p <= 4: `java.lang.Short`
	• if 4 < p <= 9: `java.lang.Integer`
	• if 8 < p <= 18: `java.lang.Long`
	• if p > 18: `java.math.BigInteger`

`DECFLOAT`	`java.math.BigDecimal`
`REAL`	`java.lang.Float`

`FLOAT,`
`DOUBLE`
	`java.lang.Double`

`CHAR(n),`
`CHARACTER(n),`
`VARCHAR(n),`
`CHARACTER VARYING(n),`
`CLOB(n),`
`GRAPHIC(n),`
`NCHAR(n),`
`VARGRAPHIC(n),`
`NVARCHAR(n),`
`DBCLOB(n),`
`NCLOB(n),`
`LONG VARCHAR,`
`LONG VARGRAPHIC`
	`java.lang.String`

VARCHAR FOR BIT DATA, LONG VARCHAR FOR BIT DATA, CHAR(n) FOR BIT DATA, BLOB(n)	`byte[]`
DATE	`java.sql.Date`
TIME	`java.sql.Time`
TIMESTAMP	`java.sql.Timestamp`
XML	`java.lang.String`
BOOLEAN (pseudo type)	`java.lang.String`
BINARY	No default HotRod data type
VARBINARY	No default HotRod data type
ROW	No default HotRod data type
ARRAY	No default HotRod data type

Appendix D - Default Data Types for SQL Server database

The HotRod SQL Server adapter automatically maps known column types to Java types. Please note that the Java types for the SQL Server columns may vary depending on the specific version and variant of SQL Server, the operating system where the database engine is running (Linux support announced), and the JDBC driver version.

Default Java Types

If no special rule is defined for a column in the configuration file HotRod will use the following rules to decide which Java type to use for each column.

SQL Server Data Type	Default Java Type
`BIT`	`java.lang.Byte`
`TINYINT`	`java.lang.Byte`
`SMALLINT`	`java.lang.Short`
`INT`	`java.lang.Integer`
`BIGINT`	`java.lang.Long`
`DECIMAL(p,s)`, `DEC(p,s)`, `NUMERIC(p,s)`	If neither p or s are specified, i.e. DECIMAL(18,0): • `java.lang.Long` If s is specified and different from zero the Java type is: • `java.math.BigDecimal` if s is not specified or it's zero: • if p <= 2: `java.lang.Byte` • if 2 < p <= 4: `java.lang.Short` • if 4 < p <= 9: `java.lang.Integer` • if 8 < p <= 18: `java.lang.Long` • if p > 18: `java.math.BigInteger`
`MONEY`, `SMALLMONEY`	`java.math.BigDecimal`
`FLOAT(n)`	If n is not specified, i.e. a FLOAT(53): • `java.lang.Double` if n is specified: • if n <= 24: `java.lang.Float` • if n >= 25: `java.lang.Double`
`REAL`	`java.lang.Float` **Note**: REAL is equivalent to FLOAT(24),

`CHAR(n),` `CHARACTER(n),` `VARCHAR(n|MAX),` `CHARVARYING(n|MAX),` `CHARACTERVARYING(n|MAX),` `NCHAR(n),` `NATIONAL CHAR(n),` `NATIONAL CHARACTER(n),` `NVARCHAR(n|MAX),` `NATIONAL CHAR VARYING(n|MAX),` `NATIONAL CHARACTER VARYING(n|MAX),` `TEXT,` `NTEXT`	`java.lang.String`
`DATE`	`java.sql.Date`
`DATETIME,` `SMALLDATETIME`	`java.util.Date`
`DATETIME2(n),` `DATETIMEOFFSET(n)`	`java.sql.Timestamp`
`TIME(n)`	If n is not specified, i.e. TIME(7): • `java.sql.Timestamp` If n is specified: • If n <=3: `java.sql.Time` • If n >=4: `java.sql.Timestamp`
`BINARY(n),` `VARBINARY(n|MAX),` `IMAGE`	`byte[]`
`HIERARCHYID`	`byte[]`
`ROWVERSION`	`java.lang.Object` Cannot insert, nor update by PK. Selects and deletes work normally. Rows can be "updated by example" when excluding this column.
`UNIQUEIDENTIFIER`	`java.lang.String`
`SQL_VARIANT`	This type is not supported by the JDBC driver 4.0 provided by SQL Server. A workaround, at least to read it, is to cast this column to a different supported type (maybe using a view or a select) as in th expression: `CAST(`**`<column>`**` AS <`**`type>`**`)`
`XML`	`java.lang.String` *
`GEOGRAPHY`	`byte[]` **
`GEOMETRY`	`byte[]` **
`(pseudo column)` `<col> as <expression>`	Type depends on expression type. Cannot insert, nor update by PK. Selects and deletes work normally. Rows can be "updated by example" when excluding this column.

* Must be a well-formed XML String. Depending on the column definition it may also need to be a valid XML String.

** These data types represent well-formed binary data as specified by the “[MS-SSCLRT]: Microsoft SQL Server CLR Types Serialization Formats” document at https://msdn.microsoft.com/en-us/library/ee320529.aspx.

Appendix E - Default Data Types for MySQL database

The HotRod MySQL adapter automatically maps known column types to Java types. The table below shows which Java types are generated by the DAO Generator for each MySQL columStringn type.

Please note that the Java types for the MySQL columns may vary depending on the specific version and variant of MySQL, the operating system where the database engine is running, and the JDBC driver version.

Default Java Types

If no special rule is defined for a column in the configuration file the DAO generator will use the following rules to decide which Java type to use for each column.

MySQL Data Type	Default Java Type

`TINYINT,`
`INT1`
	`java.lang.Byte`

`TINYINT UNSIGNED,`
`INT1 UNSIGNED`
	`java.lang.Short`

`SMALLINT,`
`INT2`
	`java.lang.Short`

`SMALLINT UNSIGNED,`
`INT2 UNSIGNED`
	`java.lang.Integer`

`MEDIUMINT,`
`INT3`
	`java.lang.Integer`

`MEDIUMINT UNSIGNED,`
`INT3 UNSIGNED`
	`java.lang.Integer`

`INTEGER,`
`INT4`
	`java.lang.Integer`

`INTEGER UNSIGNED,`
`INT4 UNSIGNED`
	`java.lang.Long`

`BIGINT,`
`INT8`
	`java.lang.Long`

`BIGINT UNSIGNED,`
`INT8 UNSIGNED`
	`java.math.BigInteger`

`DECIMAL(p,s),`
`NUMERIC(p,s)`
	If neither p or s are specified:
	• `java.lang.Long`
	If s is specified and different from zero the Java type is:
	• `java.math.BigDecimal`
	if s is not specified or specified with a value of zero:
	• if p <= 2: `java.lang.Byte`
	• if 2 < p <= 4: `java.lang.Short`
	• if 4 < p <= 9: `java.lang.Integer`
	• if 9 < p <= 18: `java.lang.Long`
	• if p > 18: `java.math.BigInteger`

`FLOAT(n),`
`FLOAT4(n),`
`FLOAT(n) UNSIGNED,`
`FLOAT4(n) UNSIGNED`
	If n is not specified:
	• `java.lang.Float`
	If n is specified:
	• if n <= 24: `java.lang.Float`
	• if n >= 25: `java.lang.Double`

`DOUBLE,`
`DOUBLE PRECISION,`
`REAL,`
`FLOAT8,`
`DOUBLE UNSIGNED,`
`DOUBLE PRECISION UNSIGNED,`
`REAL UNSIGNED,`
`FLOAT8 UNSIGNED`
	`java.lang.Double`

`CHAR(n),`
`VARCHAR(n),`
`TINYTEXT,`
`TEXT,`
`MEDIUMTEXT,`
`LONGTEXT`
	`java.lang.String`

`DATE,`
`YEAR`
	`java.sql.Date`

`TIME`	`java.sql.Time`

`DATETIME,`
`TIMESTAMP`
	`java.sql.Timestamp`

`TINYBLOB,`
`BLOB,`
`MEDIUMBLOB,`
`LONGBLOB`
	`byte[]`

`ENUM`	`java.lang.String`
`SET`	`java.lang.String`

Appendix F - Default Data Types for MariaDB database

Please refer to **Appendix E - Default Data Types for MySQL database.**

MariaDB is fully binary compatible with MySQL server. HotRod, as well as any other database client application, does not notice any difference when running against MariaDB compared to MySQL, except for the database identification strings that reveal it's MariaDB. Apart from that the database behaves exactly the same way.

Appendix G - Default Data Types for HyperSQL database

HotRod's HyperSQL adapter automatically maps known column types to Java types. The table below shows which Java types are generated by the DAO Generator for each PostgreSQL column type.

To keep compatibility with Java 6 the default types were gathered from HyperSQL 2.2.9. Please note that the Java types for the HyperSQL columns may vary depending on the specific version and variant of HyperSQL, the operating system where the database engine is running, and the JDBC driver version.

Default Java Types

If no special rule is defined for a column in the configuration file the DAO generator will use the following rules to decide which Java type to use for each column.

HyperSQL data type	Default Java Type
`TINYINT`	`java.lang.Byte`
`SMALLINT`	`java.lang.Short`
`INTEGER`	`java.lang.Integer`
`BIGINT`	`java.lang.Long`

`DECIMAL(p,s)`
`NUMERIC(p,s)`
	If neither p or s are specified:
	• `java.math.BigInteger`
	If s is specified and different from zero the Java type is:
	• `java.math.BigDecimal`
	if s is not specified or specified with a value of zero:
	• if p <= 2: `java.lang.Byte`
	• if 2 < p <= 4: `java.lang.Short`
	• if 4 < p <= 9: `java.lang.Integer`
	• if 8 < p <= 18: `java.lang.Long`
	• if p > 18: `java.math.BigInteger`

`REAL,`
`FLOAT,`
`DOUBLE`
	`java.lang.Double`

`CHAR(n),`
`CHARACTER(n),`
`VARCHAR(n),`
`CHARACTER VARYING(n),`
`CHAR VARYING(n),`
`LONGVARCHAR(n),`
`CLOB(n)`
	`java.lang.String`

`DATE`	`java.sql.Date`

`TIME(n),`
`TIME(n) WITH TIME ZONE`
	`java.sql.Time`

`TIMESTAMP(n)`, `TIMESTAMP(n) WITH TIME ZONE`	`java.sql.Timestamp`
`BINARY(n)`, `VARBINARY(n)`, `BLOB(n)`	`byte[]`
`BOOLEAN`	`java.lang.Boolean`
`OTHER`	`java.lang.Object`
`<type> ARRAY`	`java.lang.Object`
`INTERVAL <qualifier>`	`java.lang.Object`
`BIT(n)`, `BIT VARYING(n)`	`java.lang.Object`

Appendix H - Default Data Types for H2 database

HotRod’s H2 adapter automatically maps known column types to Java types. The table below shows which Java types are generated by the DAO Generator for each PostgreSQL column type.

Please note that the Java types for the H2 columns may vary depending on the specific version and variant of H2, the operating system where the database engine is running, and the JDBC driver version.

Default Java Types

If no special rule is defined for a column in the configuration file the DAO generator will use the following rules to decide which Java type to use for each column.

H2 Data Type	Default Java Type
`TINYINT`	`java.lang.Byte`

`SMALLINT,`
`INT2,`
`YEAR`
	`java.lang.Short`

`INTEGER,`
`INT,`
`MEDIUMINT,`
`INT4,`
`SIGNED,`
`IDENTITY`
	`java.lang.Integer`

`BIGINT,`
`INT8`
	`java.lang.Long`

`DECIMAL(p,s),`
`DEC(p,s),`
`NUMERIC(p,s),`
`NUMBER(p,s)`
	If neither p or s are specified:
	• `java.math.BigInteger`
	If s is specified and different from zero the Java type is:
	• `java.math.BigDecimal`
	if s is not specified or specified with a value of zero:
	• if p <= 2: `java.lang.Byte`
	• if 2 < p <= 4: `java.lang.Short`
	• if 4 < p <= 9: `java.lang.Integer`
	• if 8 < p <= 18: `java.lang.Long`
	• if p > 18: `java.math.BigInteger`

`REAL,`
`FLOAT4`
	`java.lang.Float`

`DOUBLE,`
`DOUBLE PRECISION,`
`FLOAT,`
`FLOAT8`
	`java.lang.Double`

CHAR(n), CHARACTER(n), NCHAR(n)	`java.lang.String`
VARCHAR(n), LONGVARCHAR(n), VARCHAR2(n), NVARCHAR(n), NVARCHAR2(n), VARCHAR_CASEINSENSITIVE(n), VARCHAR_INGNORECASE(n)	`java.lang.String`
CLOB(n), NCLOB(n), TINYTEXT(n), TEXT(n), MEDIUMTEXT(n), LONGTEXT(n), NTEXT(n)	`java.lang.String`
DATE	`java.sql.Date`
TIME	`java.sql.Time`
TIMESTAMP, DATETIME, SMALLDATETIME	`java.sql.Timestamp`
BINARY(n), VARBINARY(n), LONGVARBINARY(n), RAW(n), BYTEA(n), BLOB(n), TINYBLOB(n), MEDIUMBLOB(n), LONGBLOB(n), IMAGE(n), OID(n)	`byte[]`
BOOLEAN, BIT, BOOL	`java.lang.Boolean`
UUID	`byte[]` *
ARRAY	No default HotRod data type
GEOMETRY	No default HotRod data type
OTHER	`byte[]` **

* Even when H2's documentation states that UUID can be mapped to java.util.UUID this seems to work only when writing a value into H2. When reading the JDBC driver seems to produce a null value in all cases. A byte[] type, on the other hand, works consistently.

** H2's documentation states that a java.lang.Object type can be used, but it does not work well in MyBatis. A byte[] type, on the other hand, works consistently.

Appendix I - Default Data Types for SAP Adaptive Server Enterprise (ex Sybase) database

The HotRod SAP ASE adapter automatically maps known column types to Java types. Please note that the Java types for the SAP ASE columns may vary depending on the specific version and variant of the RDBMS engine, the operating system where the database engine is running, and the JDBC driver version. These default type have been tested with the SAP ASE JDBC driver; using the open source jTDS JDBC driver may show different results.

Default Java Types

If no special rule is defined for a column in the configuration file HotRod will use the following rules to decide which Java type to use for each column.

SAP ASE Data Type	Default Java Type
`BIT`	`java.lang.Byte`
`TINYINT`	`java.lang.Byte`
`UNSIGNED TINYINT`	`java.lang.Byte` *
`SMALLINT`	`java.lang.Short`
`UNSIGNED SMALLINT`	`java.lang.Integer`

`INT,`
`INTEGER`
	`java.lang.Integer`

`UNSIGNED INT,`
`UNSIGNED INTEGER`
	`java.lang.Long`

`BIGINT`	`java.lang.Long`
`UNSIGNED BIGINT`	`java.math.BigInteger`

`DECIMAL(p,s),`
`NUMERIC(p,s)`
	If neither p or s are specified, i.e. DECIMAL(18,0):
	• `java.lang.Long`
	If s is specified and different from zero the Java type is:
	• `java.math.BigDecimal`
	if s is not specified or it's zero:
	• if p <= 2: `java.lang.Byte`
	• if 2 < p <= 4: `java.lang.Short`
	• if 4 < p <= 9: `java.lang.Integer`
	• if 8 < p <= 18: `java.lang.Long`
	• if p > 18: `java.math.BigInteger`

`MONEY,`
`SMALLMONEY`
	`java.math.BigDecimal`

`FLOAT(n),`
`REAL,`
`DOUBLE PRECISION`
	`java.lang.Double` **

CHAR(n), VARCHAR(n), UNICHAR(n), UNIVARCHAR(n), NCHAR(n), NVARCHAR(n), TEXT, UNITEXT, SYSNAME, LONGSYSNAME	`java.lang.String`
DATE	`java.sql.Date`
DATETIME, SMALLDATETIME, BIGDATETIME	`java.sql.Timestamp`
TIME, BIGTIME	`java.sql.Timestamp`
BINARY(n), VARBINARY(n|MAX), IMAGE	`byte[]`

* The SAP ASE JDBC driver does not provide information to differentiate TINYINT from UNSIGNED TINYINT. If you happen to have an UNSIGNED TINYINT column you may want to use the custom type java.lang.Short for it, instead of the default type java.lang.Byte. Or, try to avoid using the UNSIGNED TINYINT type altogether, if possible.
** The SAP ASE JDBC driver does not provide information to differentiate float of different precisions. FLOAT with precision 15 or less could be treated as a java Float. However, since there's no way to find out the precision of the FLOAT the default type is, regardless of their precision, Double.