# MyBatis Generator Overview

HotRod's MyBatis Generator produces all necessary files that MyBatis needs to connect and operate on a database. The developer provides the list of tables, views, SQL selects, and SQL queries he/she wants to use and HotRod takes care of the rest.

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

The main strategy the HotRod's MyBatis Generator uses is to generate one DAO per database table, database view, arbitrary SQL select statement, and custom DAO definition. Each DAO class doubles as a VO since it includes all the table columns as properties with their corresponding getters and setters. Apart from the DAO classes it also produces all the corresponding MyBatis mapper files, and the main MyBatis configuration file, all of them ready to use.

**The DAO Java Classes**

All DAOs are specified in the configuration file for tables, views, SQL select, and custom DAO using the tags <table>, <view>, <select>, and <dao> accordingly. See the Reference section of the online manual for details.

Database tables generate DAO classes that have static methods used to retrieve database rows in the form of single or lists of DAOs. These DAOs also offers regular (non-static) methods the objects (table rows) can call to perform tasks such as inserting a row, updating it, or retrieving FK-related rows from another table, once you have retrieved table rows.

The same strategy also works well for views. Database views also produce DAO Java classes. These ones are, however, more limited since views are treated as a read only database objects. Only static methods are provided to retrieve data, but no update, insert, or delete operations are available. In essence DAO produced from views are much simpler than DAOs produced from tables.

DAOs are also produced from for arbitrary SQL select statements. Since SQL select statements produce table-like readable data they behave in a very similar way compared to views. These DAO are, however, much more limited than the ones generated from tables and views. Apart from the setters and getters they only offer a single method select(). This method executes the specified SQL select statement and returns a list of DAOs.

All these DAO definitions included in the configuration file can also include in their body multiple optional entries to access database sequence values using the tag <sequence> tag and/or to execute arbitrary SQL updates using the tag <update> tag. These entries generate DAO static methods to perform these activities.

Finally, the configuration file can also specify custom DAOs, using the <dao> tag, that do not relate to any database object or SQL select, but that only group <sequence> and <update> tags. The DAO generated Java class does not have any properties setters or getters, but only methods corresponding to the <sequence> and <update> tags.

**The DAO Primitives class vs the DAO class**

There's one more key design strategy on the DAO Java classes. HotRod actually generates two Java classes per each DAO: first, the "DAO Primitives" class is an abstract Java class that includes all database logic HotRod can automate; second, the "DAO" is a concrete class that extends from the DAO Primitives class and is initially empty. The former is never referenced from the application; only the normal DAO is, and all methods are available in it since it extends from the DAO Primitives.

This separation of the DAOs in two Java classes offers one key benefit. Since the DAO Primitives class does not include any custom code, it is safely rewritten and updated by HotRod every time the database changes and the code is regenerated. This way all the DAO database related methods are always up to date to the latest changes.

The normal DAO class, however, is generated once by HotRod and never rewritten. The developer can add custom logic to the Normal DAOs for any purposes he/she deems necessary. HotRod never touches this class so it's always safe to perform a code re-generation when needed. The developer can add any custom logic in it, usually persistence level logic or data manipulation logic.

**The DAO Properties and Methods**

Each DAO generated for a <table> tag includes:

* One class property for each table column.
* All getters and setters for each property.
* A select() method that selects a row by primary key—only if the table has a primary key.
* An insert() method that inserts a row into the table.
* An update() method that updates the row by primary key—only if the table has a primary key.
* A delete() method that deletes a row by primary key—only if the table has a primary key.
* The selectByUI...() methods—for each unique index the table has. These methods retrieve a row using the corresponding unique index. To differentiate these methods their names append the list of columns included on each unique index.
* The static selectByExample(), updateByExample(), and deleteByExample() methods. These methods are always included. They retrieve, update, or delete multiples rows at a time.
* The selectChildren...().by...() and selectParent...().by...() methods. These methods allows you to retrieve related DAO objects using the foreign keys defined for the table. There are as many methods as imported and exported foreign keys the table has. To differentiate their names the selectChildren...() and selectParent...() methods append the name of the related table the foreign key is linked to (including self references). Then the subsequent method by...() appends the list of column names the foreign key uses to reference the related table.
* A static method produced for each embedded <sequence> tag, as in selectSequence...().
* A static method produced for each embedded <update> tag.
* A getTxManager() method for transaction management, an enum with all default sort orders, and a toString() method.

Each DAO generated for a <view> tag includes:

* One class property for each view column.
* All getters and setters for each property.
* The static selectByExample() method.
* A static method produced for each embedded <sequence> tag, as in selectSequence...().
* A static method produced for each embedded <update> tag.
* A getTxManager() method for transaction management, an enum with all default sort orders, and a toString() method.

Each DAO generated for a <select> tag includes:

* One class property for each resulting column from the SQL select statement.
* All getters and setters for each property.
* The static select() method, with all specified parameters.
* A getTxManager() method for transaction management and a toString() method.

Each DAO generated for a <dao> tag includes:

* A static method produced for each embedded <sequence> tag, as in selectSequence...().
* A static method produced for each embedded <update> tag.
* A getTxManager() method for transaction management.

**The Mappers**

For each DAO Java class a complete MyBatis mapper file is also produced. This file contains all the SQL statements necessary for each one of the DAO Java methods to work. This mapper file include the full list of SQL sentences needed to execute SQL inserts, updates, deletes, and selects on the specific table, view, or select. It's fully produced and updated by HotRod and it's not expected the developer will change it.

**The MyBatis Configuration File**

Finally, HotRod's MyBatis Generator produces a complete main MyBatis configuration file with the full list of MyBatis mapper files necessary for all the DAOs to run. This MyBatis Configuration file is based on a template the developer provides where HotRod inserts the list of mappers. The generated file is, therefore, ready to use in a JEE web or standalone Java application.

## HotRod Generation Task and Configuration Files

The code generation is performed by the HotRod jar library executed through an Ant task. This task uses a developer produced configuration file (the HotRod configuration file) that provides configuration details to the generator.

In addition to this, the developer provides the MyBatis Configuration file template. This file includes the full MyBatis configuration, except for the list of mappers that will be populated by HotRod.

Finally, the developer also provides a SessionFactory singleton Java class that is used by all DAOs to retrieve MyBatis SQLSession objects. Its name and location are specified by the developer in HotRod's configuration file.

As as example, the Hello World application includes all the files mentioned above in the following locations:

* The HotRod library lib/hotrod/hotrod-3.0.0.jar
* hotrod Ant Task included in the build.xml file
* HotRod Configuration File hotrod.xml
* MyBatis Configuration file template mybatis-template.xml
* The SessionFactory java/sessionfactory/DatabaseSessionFactory.java

## Generated Files Locations

As described above, HotRod generates the following files:

* DAO Java classes
* DAO Primitives Java classes
* Mappers
* MyBatis Configuration File

The location of these files is configured through the HotRod configuration file. The Hello World example's hotrod.xml file shows the following configuration details:

<generators>

<mybatis>

<daos gen-base-dir=*"auto-generated/java"* dao-package=*"daos"* />

<mappers gen-base-dir=*"auto-generated/mappers"* relative-dir=*"persistence"* />

<mybatis-configuration-template file=*"mybatis-template.xml"* />

<session-factory singleton-full-class-name=*"sessionfactory.DatabaseSessionFactory"* />

...

</mybatis>

</generators>

According to these values the DAO classes will be generated in auto-generated/java/daos. This path is composed by the base dir auto-generated/java and the Java package daos. For example, a DAO for the database table VEHICLE will have the package daos and will be generated as:

auto-generated/java/daos/VehicleDAO.java

Similarly, the DAO Primitives classes will be generated in the package primitives inside the DAOs package. In this example the DAO Primitives Java class will have the package daos/primitives and will be generated as:

auto-generated/java/daos/primitives/VehicleDAOPrimitives.java

The DAO Primitives will reference the Session Factory class as sessionfactory.DatabaseSessionFactory.

The mapper files will be generated in auto-generated/mappers/persistence. Using the same example, the mapper file for the database table VEHICLE will be generated as:

auto-generated/mappers/persistence/primitives/primitives-vehicle-dao.xml

A reference to it will be included in the MyBatis Configuration file using its relative name only persistence/primitives/primitives-vehicle-dao.xml.

Finally, the MyBatis configuration file template will be read from mybatis-template.xml and the resulting fully populated file will be generated in the same directory as the mapper files. That is as:

auto-generated/mappers/persistence/primitives/mybatis-configuration.xml

All this may sound quite confusing, but ends up being very simple, since HotRod takes care of all the details and references between the files.

The generated classes and supporting files are fully functional and ready to use by your application.

## Your Application

Once all the files are generated your Java application needs can simply use the DAOs to retrieve data from the database and to produce changes in it. For example, to insert a row in the VEHICLE table you could write something like:

VehicleDAO skoda = **new** VehicleDAO();

skoda.setBrand("Skoda");

skoda.setModel("Octavia");

skoda.setUsed(**false**);

skoda.setCurrentMileage(7);

skoda.setPurchasedOn(**new** Date(System.*currentTimeMillis*()));

skoda.insert();

The key line is the last one that executes the insert(), that was automatically generated by HotRod. Behind the scenes it performs quite a few tasks for you: first, it obtains a SQLSession object from the SessionFactory singleton you specified, uses all the DAO properties as column values, executes a SQL insert with them, retrieves the newly identity PK value back to the DAO property, and finally frees all the JDBC resources.

See the Hello World example for more details.