MyBatis

# What is MyBatis?

MyBatis is a first class persistence framework with support for custom SQL, stored procedures and advanced mappings. MyBatis eliminates almost all of the JDBC code and manual setting of parameters and retrieval of results. MyBatis can use simple XML or Annotations for configuration and map primitives, Map interfaces and Java POJOs (Plain Old Java Objects) to database records.

# Getting started

## Installation

To use the MyBatis you just need to include the mybatis-x.x.x.jar file in the classpath.

## Building SqlSessionFactory from XML

Every MyBatis application centers around an instance of SqlSessionFactory. A SqlSessionFactory instance can be acquired by using the SqlSessionFactoryBuilder. SqlSessionFactoryBuilder can build a SqlSessionFactory instance from an XML configuration file, of from a custom prepared instance of the Configuration class.

Building a SqlSessionFactory instance from an XML file is very simple. It is recommended that you use a classpath resource for this configuration, but you could use any InputStream instance, including one created from a literal file path or a file:// URL. MyBatis includes a utility class, called Resources, that contains a number of methods that make it simpler to load resources from the classpath and other locations.

String resource = "org/mybatis/example/mybatis-config.xml";

InputStream inputStream = Resources.getResourceAsStream(resource);

SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

The configuration XML file contains settings for the core of the MyBatis system, including a DataSource for acquiring database Connection instances, as well as a TransactionManager for determining how transactions should be scoped and controlled.

<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE configuration

PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>

<environments default="development">

<environment id="development">

<transactionManager type="JDBC"/>

<dataSource type="POOLED">

<property name="driver" value="${driver}"/>

<property name="url" value="${url}"/>

<property name="username" value="${username}"/>

<property name="password" value="${password}"/>

</dataSource>

</environment>

</environments>

<mappers>

<mapper resource="org/mybatis/example/BlogMapper.xml"/>

</mappers>

</configuration>

## Acquiring a SqlSession from SqlSessionFactory

Now that you have a SqlSessionFactory, as the name suggests, you can acquire an instance of SqlSession. The SqlSession contains absolutely every method needed to execute SQL commands against the database. You can execute mapped SQL statements directly against the SqlSession instance. For exmaple:

SqlSession session = sqlSessionFactory.openSession();

try {

Blog blog = session.selectOne("org.mybatis.example.BlogMapper.selectBlog", 101);

} finally {

session.close();

}

While this approach works, and is familiar to users of previous versions of MyBatis, there is now a cleaner approach. Using an interface (e.g. BlogMapper.class) that properly describes the parameter and return value for a given statement, you can now execute cleaner and more type safe code, without error prone string literals and casting.

SqlSession session = sqlSessionFactory.openSession();

try {

BlogMapper mapper = session.getMapper(BlogMapper.class);

Blog blog = mapper.selectBlog(101);

} finally {

session.close();

}

## Exploring Mapped SQL Statements

<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE mapper

PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="org.mybatis.example.BlogMapper">

<select id="selectBlog" parameterType="int" resultType="Blog">

select \* from Blog where id = #{id}

</select>

</mapper>

While this looks like a lot of overhead for this simple example, it is actually very light. You can define as many mapped statements in a single mapper XML file as you like, so you get a lot of mileage out of the XML header and doctype declaration. The rest of the file is pretty self explanatory. It defines a name for the mapped statement “selectBlog”, in the namespace “org.mybatis.example.BlogMapper”, which would allow you to call it by specifying the fully qualified name of “org.mybatis.example.BlogMapper.selectBlog”, as we did above in the following example:

Blog blog = session.selectOne("org.mybatis.example.BlogMapper.selectBlog", 101);

Notice how similar this is to calling a method on a fully qualified Java class, and there's a reason for that. This name can be directly mapped to a Mapper class of the same name as the namespace, with a method that matches the name, parameter, and return type as the mapped select statement. This allows you to very simply call the method against the Mapper interface as you saw above, but here it is again in the following example:

BlogMapper mapper = session.getMapper(BlogMapper.class);

Blog blog = mapper.selectBlog(101);

## Scope and Lifecycle

It's very important to understand the various scopes and lifecycles classes we've discussed so far. Using them incorrectly can cause severe concurrency problems.

### SqlSessionFactoryBuilder

This class can be instantiated, used and thrown away. There is no need to keep it around once you've created your SqlSessionFactory. Therefore the best scope for instances of SqlSessionFactoryBuilder is method scope (i.e. a local method variable). You can reuse the SqlSessionFactoryBuilder to build multiple SqlSessionFactory instances, but it's still best not to keep it around to ensure that all of the XML parsing resources are freed up for more important things.

### SqlSessionFactory

Once created, the SqlSessionFactory should exist for the duration of your application execution. There should be little or no reason to ever dispose of it or recreate it. It's a best practice to not rebuild the SqlSessionFactory multiple times in an application run. Doing so should be considered a “bad smell”. Therefore the best scope of SqlSessionFactory is application scope. This can be achieved a number of ways. The simplest is to use a Singleton pattern or Static Singleton pattern.

### SqlSession

Each thread should have its own instance of SqlSession. Instances of SqlSession are not to be shared and are not thread safe. Therefore the best scope is request or method scope. Never keep references to a SqlSession instance in a static field or even an instance field of a class. Never keep references to a SqlSession in any sort of managed scope, such as HttpSession of of the Servlet framework. If you're using a web framework of any sort, consider the SqlSession to follow a similar scope to that of an HTTP request. In other words, upon receiving an HTTP request, you can open a SqlSession, then upon returning the response, you can close it. Closing the session is very important. You should always ensure that it's closed within a finally block. The following is the standard pattern for ensuring that SqlSessions are closed:

SqlSession session = sqlSessionFactory.openSession();

try {

// do work

} finally {

session.close();

}

### Mapper Instances

Mappers are interfaces that you create to bind to your mapped statements. Instances of the mapper interfaces are acquired from the SqlSession. As such, technically the broadest scope of any mapper instance is the same as the SqlSession from which they were requested. However, the best scope for mapper instances is method scope. That is, they should be requested within the method that they are used, and then be discarded. They do not need to be closed explicitly. While it's not a problem to keep them around throughout a request, similar to the SqlSession, you might find that managing too many resources at this level will quickly get out of hand. Keep it simple, keep Mappers in the method scope. The following example demonstrates this practice.

SqlSession session = sqlSessionFactory.openSession();

try {

BlogMapper mapper = session.getMapper(BlogMapper.class);

// do work

} finally {

session.close();

}

# Configuration

## properties

These are externalizable, substitutable properties that can be configured in a typical Java Properties file instance, or passed in through sub-elements of the properties element. For example:

<properties resource="org/mybatis/example/config.properties">

<property name="username" value="dev\_user"/>

<property name="password" value="F2Fa3!33TYyg"/>

</properties>

The properties can then be used throughout the configuration files to substitute values that need to be dynamically configured. For example:

<dataSource type="POOLED">

<property name="driver" value="${driver}"/>

<property name="url" value="${url}"/>

<property name="username" value="${username}"/>

<property name="password" value="${password}"/>

</dataSource>

The username and password in this example will be replaced by the values set in the properties elements. The driver and url properties would be replaced with values contained from the config.properties file. This provides a lot of options for configuration.

Properties can also be passed into the SqlSessionBuilder.build() methods. For example:

SqlSessionFactory factory = sqlSessionFactoryBuilder.build(reader, props);

// ... or ...

SqlSessionFactory factory = sqlSessionFactoryBuilder.build(reader, environment, props);

If a property exists in more than one of these places, MyBatis loads them in the following order:

* Properties specified in the body of the properties element are read first,
* Properties loaded from the classpath resource or url attributes of the properties element are read second, and override any duplicate properties already specified,
* Properties passed as a method parameter are read last, and override any duplicate properties that may have been loaded from the properties body and the resource/url attributes.

Thus, the highest priority properties are those passed in as a method parameter, followed by resource/url attributes and finally the properties specified in the body of the properties element.

## settings

These are extremely important tweaks that modify the way that MyBatis behaves at runtime. The following table describes the settings, their meanings and their default values.

| **Setting** | **Description** | **Valid Values** | **Default** |
| --- | --- | --- | --- |
| cacheEnabled | Globally enables or disables any caches configured in any mapper under this configuration. | true | false | true |
| lazyLoadingEnabled | Globally enables or disables lazy loading. When disabled, all associations will be eagerly loaded. | true | false | true |
| aggressiveLazyLoading | When enabled, an object with lazy loaded properties will be loaded entirely upon a call to any of the lazy properties. Otherwise, each property is loaded on demand. | true | false | true |
| multipleResultSetsEnabled | Allows or disallows multiple ResultSets to be returned from a single statement (compatible driver required). | true | false | true |
| useColumnLabel | Uses the column label instead of the column name. Different drivers behave differently in this respect. Refer to the driver documentation, or test out both modes to determine how your driver behaves. | true | false | true |
| useGeneratedKeys | Allows JDBC support for generated keys. A compatible driver is required. This setting forces generated keys to be used if set to true, as some drivers deny compatibility but still work (e.g. Derby). | true | false | False |
| autoMappingBehavior | Specifies if and how MyBatis should automatically map columns to fields/properties. NONE disables auto-mapping. PARTIAL will only auto-map results with no nested result mappings defined inside. FULL will auto-map result mappings of any complexity (containing nested or otherwise). | NONE, PARTIAL, FULL | PARTIAL |
| defaultExecutorType | Configures the default executor. SIMPLE executor does nothing special. REUSE executor reuses prepared statements. BATCH executor reuses statements and batches updates. | SIMPLE REUSE BATCH | SIMPLE |
| defaultStatementTimeout | Sets the number of seconds the driver will wait for a response from the database. | Any positive integer | Not Set (null) |
| safeRowBoundsEnabled | Allows using RowBounds on nested statements. | true | false | False |
| mapUnderscoreToCamelCase | Enables automatic mapping from classic database column names A\_COLUMN to camel case classic Java property names aColumn. | true | false | False |
| localCacheScope | MyBatis uses local cache to prevent circular references and speed up repeated nested queries. By default (SESSION) all queries executed during a session are cached. If localCacheScope=STATEMENT local session will be used just for statement execution, no data will be shared between two different calls to the same SqlSession. | SESSION | STATEMENT | SESSION |
| jdbcTypeForNull | Specifies the JDBC type for null values when no specific JDBC type was provided for the parameter. Some drivers require specifying the column JDBC type but others work with generic values like NULL, VARCHAR or OTHER. | JdbcType enumeration. Most common are: NULL, VARCHAR and OTHER | OTHER |
| lazyLoadTriggerMethods | Specifies which Object's methods trigger a lazy load | A method name list separated by commas | equals,clone,hashCode,toString |
| defaultScriptingLanguage | Specifies the language used by default for dynamic SQL generation. | A type alias or fully qualified class name. | org.apache.ibatis.scripting.xmltags.XMLDynamicLanguageDriver |
| callSettersOnNulls | Specifies if setters or map's put method will be called when a retrieved value is null. It is useful when you rely on Map.keySet() or null value initialization. Note primitives such as (int,boolean,etc.) will not be set to null. | true | false | false |
| logPrefix | Specifies the prefix string that MyBatis will add to the logger names. | Any String | Not set |
| logImpl | Specifies which logging implementation MyBatis should use. If this setting is not present logging implementation will be autodiscovered. | SLF4J | LOG4J | LOG4J2 | JDK\_LOGGING | COMMONS\_LOGGING | STDOUT\_LOGGING | NO\_LOGGING | Not set |
| proxyFactory | Specifies the proxy tool that MyBatis will use for creating lazy loading capable objects. | CGLIB | JAVASSIST | CGLIB |

## typeAliases

A type alias is simply a shorter name for a Java type. It's only relevant to the XML configuration and simply exists to reduce redundant typing of fully qualified classnames. For example:

<typeAliases>

<typeAlias alias="Author" type="domain.blog.Author"/>

<typeAlias alias="Blog" type="domain.blog.Blog"/>

<typeAlias alias="Comment" type="domain.blog.Comment"/>

<typeAlias alias="Post" type="domain.blog.Post"/>

<typeAlias alias="Section" type="domain.blog.Section"/>

<typeAlias alias="Tag" type="domain.blog.Tag"/>

</typeAliases>

## typeHandlers

Whenever MyBatis sets a parameter on a PreparedStatement or retrieves a value from a ResultSet, a TypeHandler is used to retrieve the value in a means appropriate to the Java type. The following table describes the default TypeHandlers.

| **Type Handler** | **Java Types** | **JDBC Types** |
| --- | --- | --- |
| BooleanTypeHandler | java.lang.Boolean,boolean | Any compatible BOOLEAN |
| ByteTypeHandler | java.lang.Byte, byte | Any compatible NUMERIC or BYTE |
| ShortTypeHandler | java.lang.Short, short | Any compatible NUMERIC or SHORT INTEGER |
| IntegerTypeHandler | java.lang.Integer, int | Any compatible NUMERIC or INTEGER |
| LongTypeHandler | java.lang.Long, long | Any compatible NUMERIC or LONG INTEGER |
| FloatTypeHandler | java.lang.Float, float | Any compatible NUMERIC or FLOAT |
| DoubleTypeHandler | java.lang.Double, double | Any compatible NUMERIC or DOUBLE |
| BigDecimalTypeHandler | java.math.BigDecimal | Any compatible NUMERIC or DECIMAL |
| StringTypeHandler | java.lang.String | CHAR, VARCHAR |
| ClobTypeHandler | java.lang.String | CLOB, LONGVARCHAR |
| NStringTypeHandler | java.lang.String | NVARCHAR, NCHAR |
| NClobTypeHandler | java.lang.String | NCLOB |
| ByteArrayTypeHandler | byte[] | Any compatible byte stream type |
| BlobTypeHandler | byte[] | BLOB, LONGVARBINARY |
| DateTypeHandler | java.util.Date | TIMESTAMP |
| DateOnlyTypeHandler | java.util.Date | DATE |
| TimeOnlyTypeHandler | java.util.Date | TIME |
| SqlTimestampTypeHandler | java.sql.Timestamp | TIMESTAMP |
| SqlDateTypeHandler | java.sql.Date | DATE |
| SqlTimeTypeHandler | java.sql.Time | TIME |
| ObjectTypeHandler | Any | OTHER, or unspecified |
| EnumTypeHandler | Enumeration Type | VARCHAR any string compatible type, as the code is stored (not index). |
| EnumOrdinalTypeHandler | Enumeration Type | Any compatible NUMERIC or DOUBLE, as the position is stored (not the code itself). |

## objectFactory

Each time MyBatis creates a new instance of a result object, it uses an ObjectFactory instance to do so. The default ObjectFactory does little more than instantiate the target class with a default constructor, or a parameterized constructor if parameter mappings exist. If you want to override the default behaviour of the ObjectFactory, you can create your own. For example:

## environments

MyBatis can be configured with multiple environments. This helps you to apply your SQL Maps to multiple databases for any number of reasons. For example, you might have a different configuration for your Development, Test and Production environments. Or, you may have multiple production databases that share the same schema, and you’d like to use the same SQL maps for both. There are many use cases.

**One important thing to remember though: While you can configure multiple environments, you can only choose ONE per SqlSessionFactory instance.**

So if you want to connect to two databases, you need to create two instances of SqlSessionFactory, one for each. For three databases, you’d need three instances, and so on. It’s really easy to remember:

**UNPOOLED** – This implementation of DataSource simply opens and closes a connection each time it is requested. While it’s a bit slower, this is a good choice for simple applications that do not require the performance of immediately available connections. Different databases are also different in this performance area, so for some it may be less important to pool and this configuration will be ideal. The UNPOOLED DataSource is configured with only five properties:

* driver – This is the fully qualified Java class of the JDBC driver (NOT of the DataSource class if your driver includes one).
* url – This is the JDBC URL for your database instance.
* username – The database username to log in with.
* password - The database password to log in with.
* defaultTransactionIsolationLevel – The default transaction isolation level for connections.

Optionally, you can pass properties to the database driver as well. To do this, prefix the properties with driver., for example:

* driver.encoding=UTF8

This will pass the property encoding, with the value UTF8, to your database driver via the DriverManager.getConnection(url, driverProperties) method.

**POOLED** – This implementation of DataSource pools JDBC Connection objects to avoid the initial connection and authentication time required to create a new Connection instance. This is a popular approach for concurrent web applications to achieve the fastest response.

In addition to the (UNPOOLED) properties above, there are many more properties that can be used to configure the POOLED datasource:

* poolMaximumActiveConnections – This is the number of active (i.e. in use) connections that can exist at any given time. Default: 10
* poolMaximumIdleConnections – The number of idle connections that can exist at any given time.
* poolMaximumCheckoutTime – This is the amount of time that a Connection can be "checked out" of the pool before it will be forcefully returned. Default: 20000ms (i.e. 20 seconds)
* poolTimeToWait – This is a low level setting that gives the pool a chance to print a log status and re-attempt the acquisition of a connection in the case that it’s taking unusually long (to avoid failing silently forever if the pool is misconfigured). Default: 20000ms (i.e. 20 seconds)
* poolPingQuery – The Ping Query is sent to the database to validate that a connection is in good working order and is ready to accept requests. The default is "NO PING QUERY SET", which will cause most database drivers to fail with a decent error message.
* poolPingEnabled – This enables or disables the ping query. If enabled, you must also set the poolPingQuery property with a valid SQL statement (preferably a very fast one). Default: false.
* poolPingConnectionsNotUsedFor – This configures how often the poolPingQuery will be used. This can be set to match the typical timeout for a database connection, to avoid unnecessary pings. Default: 0 (i.e. all connections are pinged every time – but only if poolPingEnabled is true of course).

**JNDI** – This implementation of DataSource is intended for use with containers such as EJB or Application Servers that may configure the DataSource centrally or externally and place a reference to it in a JNDI context. This DataSource configuration only requires two properties:

* initial\_context – This property is used for the Context lookup from the InitialContext (i.e. initialContext.lookup(initial\_context)). This property is optional, and if omitted, then the data\_source property will be looked up against the InitialContext directly.
* data\_source – This is the context path where the reference to the instance of the DataSource can be found. It will be looked up against the context returned by the initial\_context lookup, or against the InitialContext directly if no initial\_context is supplied.

Similar to the other DataSource configurations, it’s possible to send properties directly to the InitialContext by prefixing those properties withenv., for example:

* env.encoding=UTF8

This would send the property encoding with the value of UTF8 to the constructor of the InitialContext upon instantiation.

## databaseIdProvider

MyBatis is able to execute different statements depending on your database vendor. The multi-db vendor support is based on the mapped statements databaseId attribute. MyBatis will load all statements with no databaseId attribute or with a databaseId that matches the current one. If case the same statement if found with and without the databaseId the latter will be discarded. To enable the multi vendor support add a databaseIdProvider to mybatis-config.xml file as follows:

<databaseIdProvider type="VENDOR" />

The VENDOR implementation databaseIdProvider sets as a databaseId the String returned by DatabaseMetaData#getDatabaseProductName(). As usually that string is too long and also, different versions of the same product return different values, so you may want to translate it to a shorter one by adding properties like follows:

<databaseIdProvider type="VENDOR">

<property name="SQL Server" value="sqlserver"/>

<property name="DB2" value="db2"/>

<property name="Oracle" value="oracle" />

</databaseIdProvider>

When properties are provided, the VENDOR databaseIdProvider will search the property value corresponding to the first key found in the returned database product name or "null" if there is not a matching property. In this case, if getDatabaseProductName() returns "Oracle (DataDirect)" the databaseId will be set to "oracle".

You can build your own DatabaseIdProvider by implementing the interface org.apache.ibatis.mapping.DatabaseIdProvider and registering it in mybatis-config.xml:

public interface DatabaseIdProvider {

void setProperties(Properties p);

String getDatabaseId(DataSource dataSource) throws SQLException;

}

## Mappers

Now that the behavior of MyBatis is configured with the above configuration elements, we’re ready to define our mapped SQL statements. But first, we need to tell MyBatis where to find them. Java doesn’t really provide any good means of auto-discovery in this regard, so the best way to do it is to simply tell MyBatis where to find the mapping files. You can use class path relative resource references, or literal, fully qualified url references (including file:/// URLs). For example:

<!-- Using classpath relative resources -->

<mappers>

<mapper resource="org/mybatis/builder/AuthorMapper.xml"/>

<mapper resource="org/mybatis/builder/BlogMapper.xml"/>

<mapper resource="org/mybatis/builder/PostMapper.xml"/>

</mappers>

<!-- Using url fully qualified paths -->

<mappers>

<mapper url="file:///var/mappers/AuthorMapper.xml"/>

<mapper url="file:///var/mappers/BlogMapper.xml"/>

<mapper url="file:///var/mappers/PostMapper.xml"/>

</mappers>

<!-- Using mapper interface classes -->

<mappers>

<mapper class="org.mybatis.builder.AuthorMapper"/>

<mapper class="org.mybatis.builder.BlogMapper"/>

<mapper class="org.mybatis.builder.PostMapper"/>

</mappers>

<!-- Register all interfaces in a package as mappers -->

<mappers>

<package name="org.mybatis.builder"/>

</mappers>

These statement simply tell MyBatis where to go from here. The rest of the details are in each of the SQL Mapping files, and that’s exactly what the next section will discuss.

# Mapper XML Files

The true power of MyBatis is in the Mapped Statements. This is where the magic happens. For all of their power, the Mapper XML files are relatively simple. Certainly if you were to compare them to the equivalent JDBC code, you would immediately see a savings of 95% of the code. MyBatis was built to focus on the SQL, and does its best to stay out of your way.

The Mapper XML files have only a few first class elements (in the order that they should be defined):

* cache – Configuration of the cache for a given namespace.
* cache-ref – Reference to a cache configuration from another namespace.
* resultMap – The most complicated and powerful element that describes how to load your objects from the database result sets.
* parameterMap – Deprecated! Old-school way to map parameters. Inline parameters are preferred and this element may be removed in the future. Not documented here.
* sql – A reusable chunk of SQL that can be referenced by other statements.
* insert – A mapped INSERT statement.
* update – A mapped UPDATE statement.
* delete – A mapped DELETE statement.
* select – A mapped SELECT statement.

## select

The select statement is one of the most popular elements that you'll use in MyBatis. Putting data in a database isn't terribly valuable until you get it back out, so most applications query far more than they modify the data. For every insert, update or delete, there is probably many selects. This is one of the founding principles of MyBatis, and is the reason so much focus and effort was placed on querying and result mapping. The select element is quite simple for simple cases. For example:

<select id="selectPerson" parameterType="int" resultType="hashmap">

SELECT \* FROM PERSON WHERE ID = #{id}

</select>

This statement is called selectPerson, takes a parameter of type int (or Integer), and returns a HashMap keyed by column names mapped to row values.

Notice the parameter notation:

#{id}

This tells MyBatis to create a PreparedStatement parameter. With JDBC, such a parameter would be identified by a "?" in SQL passed to a new PreparedStatement, something like this:

Of course, there's a lot more code required by JDBC alone to extract the results and map them to an instance of an object, which is what MyBatis saves you from having to do.

The select element has more attributes that allow you to configure the details of how each statement should behave.

<select

id="selectPerson"

parameterType="int"

parameterMap="deprecated"

resultType="hashmap"

resultMap="personResultMap"

flushCache="false"

useCache="true"

timeout="10000"

fetchSize="256"

statementType="PREPARED"

resultSetType="FORWARD\_ONLY">

| **Attribute** | **Description** |
| --- | --- |
| id | A unique identifier in this namespace that can be used to reference this statement. |
| parameterType | The fully qualified class name or alias for the parameter that will be passed into this statement. |
| parameterMap | This is a deprecated approach to referencing an external parameterMap. Use inline parameter mappings and theparameterType attribute. |
| resultType | The fully qualified class name or alias for the expected type that will be returned from this statement. Note that in the case of collections, this should be the type that the collection contains, not the type of the collection itself. Use resultType ORresultMap, not both. |
| resultMap | A named reference to an external resultMap. Result maps are the most powerful feature of MyBatis, and with a good understanding of them, many difficult mapping cases can be solved. Use resultMap OR resultType, not both. |
| flushCache | Setting this to true will cause the local and 2nd level caches to be flushed whenever this statement is called. Default: falsefor select statements. |
| useCache | Setting this to true will cause the results of this statement to be cached in 2nd level cache. Default: true for select statements. |
| timeout | This sets the number of seconds the driver will wait for the database to return from a request, before throwing an exception. Default is unset (driver dependent). |
| fetchSize | This is a driver hint that will attempt to cause the driver to return results in batches of rows numbering in size equal to this setting. Default is unset (driver dependent). |
| statementType | Any one of STATEMENT, PREPARED or CALLABLE. This causes MyBatis to use Statement, PreparedStatement orCallableStatement respectively. Default: PREPARED. |
| resultSetType | Any one of FORWARD\_ONLY|SCROLL\_SENSITIVE|SCROLL\_INSENSITIVE. Default is unset (driver dependent). |
| databaseId | In case there is a configured databaseIdProvider, MyBatis will load all statements with no databaseId attribute or with adatabaseId that matches the current one. If case the same statement if found with and without the databaseId the latter will be discarded. |
| resultOrdered | This is only applicable for nested result select statements: If this is true, it is assumed that nested results are contained or grouped together such that when a new main result row is returned, no references to a previous result row will occur anymore. This allows nested results to be filled much more memory friendly. Default: false. |
| resultSets | This is only applicable for multiple result sets. It lists the result sets that will be returned by the statement and gives a name to each one. Names are separated by commas. |

## insert, update and delete

The data modification statements insert, update and delete are very similar in their implementation:

<insert

id="insertAuthor"

parameterType="domain.blog.Author"

flushCache="true"

statementType="PREPARED"

keyProperty=""

keyColumn=""

useGeneratedKeys=""

timeout="20">

<update

id="insertAuthor"

parameterType="domain.blog.Author"

flushCache="true"

statementType="PREPARED"

timeout="20">

<delete

id="insertAuthor"

parameterType="domain.blog.Author"

flushCache="true"

statementType="PREPARED"

timeout="20">

| **Attribute** | **Description** |
| --- | --- |
| id | A unique identifier in this namespace that can be used to reference this statement. |
| parameterType | The fully qualified class name or alias for the parameter that will be passed into this statement. |
| parameterMap | This is a deprecated approach to referencing an external parameterMap. Use inline parameter mappings and the parameterType attribute. |
| flushCache | Setting this to true will cause the 2nd level and local caches to be flushed whenever this statement is called. Default:true for insert, update and delete statements. |
| timeout | This sets the maximum number of seconds the driver will wait for the database to return from a request, before throwing an exception. Default is unset (driver dependent). |
| statementType | Any one of STATEMENT, PREPARED or CALLABLE. This causes MyBatis to use Statement, PreparedStatement orCallableStatement respectively. Default: PREPARED. |
| useGeneratedKeys | (insert only) This tells MyBatis to use the JDBC getGeneratedKeys method to retrieve keys generated internally by the database (e.g. auto increment fields in RDBMS like MySQL or SQL Server). Default: false |
| keyProperty | (insert only) Identifies a property into which MyBatis will set the key value returned by getGeneratedKeys, or by aselectKey child element of the insert statement. Default: unset. |
| keyColumn | (insert only) Sets the name of the column in the table with a generated key. This is only required in certain databases (like PostgreSQL) when the key column is not the first column in the table. |
| databaseId | In case there is a configured databaseIdProvider, MyBatis will load all statements with no databaseId attribute or with adatabaseId that matches the current one. If case the same statement if found with and without the databaseId the latter will be discarded. |

## Result Maps

The resultMap element is the most important and powerful element in MyBatis. It's what allows you to do away with 90% of the code that JDBC requires to retrieve data from ResultSets, and in some cases allows you to do things that JDBC does not even support. In fact, to write the equivalent code for something like a join mapping for a complex statement could probably span thousands of lines of code. The design of theResultMaps is such that simple statements don't require explicit result mappings at all, and more complex statements require no more than is absolutely necessary to describe the relationships.

Such a JavaBean could be mapped to a ResultSet just as easily as the HashMap.

<select id="selectUsers" parameterType="int" resultType="com.someapp.model.User">

select id, username, hashedPassword

from some\_table

where id = #{id}

</select>

And remember that TypeAliases are your friend. Use them so that you don't have to keep typing the fully qualified path of your class out. For example:

<!-- In Config XML file -->

<typeAlias type="com.someapp.model.User" alias="User"/>

<!-- In SQL Mapping XML file -->

<select id="selectUsers" parameterType="int" resultType="User">

select id, username, hashedPassword

from some\_table

where id = #{id}

</select>

The great thing about ResultMaps is that you've already learned a lot about them, but you haven't even seen one yet! These simple cases don't require any more than you've seen here. Just for example sake, let's see what this last example would look like as an externalresultMap, as that is another way to solve column name mismatches.

<resultMap id="userResultMap" type="User">

<id property="id" column="user\_id" />

<result property="username" column="user\_name"/>

<result property="password" column="hashed\_password"/>

</resultMap>

And the statement that references it uses the resultMap attribute to do so (notice we removed the resultType attribute). For example:

<select id="selectUsers" parameterType="int" resultMap="userResultMap">

select user\_id, user\_name, hashed\_password

from some\_table

where id = #{id}

</select>

## resultMap

* constructor - used for injecting results into the constructor of a class upon instantiation
  + idArg - ID argument; flagging results as ID will help improve overall performance
  + arg - a normal result injected into the constructor
* id – an ID result; flagging results as ID will help improve overall performance
* result – a normal result injected into a field or JavaBean property
* association – a complex type association; many results will roll up into this type
  + nested result mappings – associations are resultMaps themselves, or can refer to one
* collection – a collection of complex types
  + nested result mappings – collections are resultMaps themselves, or can refer to one
* discriminator – uses a result value to determine which resultMap to use
  + case – a case is a result map based on some value
    - nested result mappings – a case is also a result map itself, and thus can contain many of these same elements, or it can refer to an external resultMap.

| **Attribute** | **Description** |
| --- | --- |
| id | A unique identifier in this namespace that can be used to reference this result map. |
| type | A fully qualified Java class name, or a type alias (see the table above for the list of built-in type aliases). |
| autoMapping | If present, MyBatis will enable or disable the automapping for this ResultMap. This attribute overrides the global autoMappingBehavior. Default: unset. |

**id & result**

<id property="id" column="post\_id"/>

<result property="subject" column="post\_subject"/>

These are the most basic of result mappings. Both *id*, and *result* map a single column value to a single property or field of a simple data type (String, int, double, Date, etc.).

The only difference between the two is that *id* will flag the result as an identifier property to be used when comparing object instances. This helps to improve general performance, but especially performance of caching and nested result mapping (i.e. join mapping).

| **Attribute** | **Description** |
| --- | --- |
| property | The field or property to map the column result to. If a matching JavaBeans property exists for the given name, then that will be used. Otherwise, MyBatis will look for a field of the given name. In both cases you can use complex property navigation using the usual dot notation. For example, you can map to something simple like: username, or to something more complicated like:address.street.number. |
| column | The column name from the database, or the aliased column label. This is the same string that would normally be passed toresultSet.getString(columnName). |
| javaType | A fully qualified Java class name, or a type alias (see the table above for the list of built-in type aliases). MyBatis can usually figure out the type if you're mapping to a JavaBean. However, if you are mapping to a HashMap, then you should specify the javaType explicitly to ensure the desired behaviour. |
| jdbcType | The JDBC Type from the list of supported types that follows this table. The JDBC type is only required for nullable columns upon insert, update or delete. This is a JDBC requirement, not a MyBatis one. So even if you were coding JDBC directly, you'd need to specify this type – but only for nullable values. |
| typeHandler | We discussed default type handlers previously in this documentation. Using this property you can override the default type handler on a mapping-by-mapping basis. The value is either a fully qualified class name of a TypeHandler implementation, or a type alias. |

## Auto-mapping