mybatis

v. 3.5.1

User Guide

MyBatis.org **2019-04-07**

Table of Contents

1 Introduction

1.1 Introduction

1.1.1 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.

1.1.2 Help make this documentation better…

If you find this documentation lacking in any way, or missing documentation for a feature, then the best thing to do is learn about it and then write the documentation yourself!

Sources of this manual are available in xdoc format at project's Git Fork the repository, update them and send a pull request.

You're the best author of this documentation, people like you have to read it!

1.1.3 Translations

Users can read about MyBatis in following translations:

- English
- Español
- ###
- ###
- ####

Do you want to read about MyBatis in your own native language? File an issue providing patches with your mother tongue documentation!

2 Getting Started

2.1 Getting started

2.1.1 Installation

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

If you are using Maven just add the following dependency to your pom.xml:

```
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>x.x.x</version>
</dependency>
```

2.1.2 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, or 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. The full details of the XML configuration file can be found later in this document, but here is a simple example:

```
<?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>
```

While there is a lot more to the XML configuration file, the above example points out the most critical parts. Notice the XML header, required to validate the XML document. The body of the environment element contains the environment configuration for transaction management and connection pooling. The mappers element contains a list of mappers – the XML files and/or annotated Java interface classes that contain the SQL code and mapping definitions.

2.1.3 Building SqlSessionFactory without XML

If you prefer to directly build the configuration from Java, rather than XML, or create your own configuration builder, MyBatis provides a complete Configuration class that provides all of the same configuration options as the XML file.

```
DataSource dataSource = BlogDataSourceFactory.getBlogDataSource();
TransactionFactory transactionFactory =
  new JdbcTransactionFactory();
Environment environment =
  new Environment("development", transactionFactory, dataSource);
Configuration configuration = new Configuration(environment);
configuration.addMapper(BlogMapper.class);
SqlSessionFactory sqlSessionFactory =
  new SqlSessionFactoryBuilder().build(configuration);
```

Notice in this case the configuration is adding a mapper class. Mapper classes are Java classes that contain SQL Mapping Annotations that avoid the need for XML. However, due to some limitations of Java Annotations and the complexity of some MyBatis mappings, XML mapping is still required for the most advanced mappings (e.g. Nested Join Mapping). For this reason, MyBatis will automatically look for and load a peer XML file if it exists (in this case, BlogMapper.xml would be loaded based on the classpath and name of BlogMapper.class). More on this later.

2.1.4 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 example:

```
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.

For example:

```
SqlSession session = sqlSessionFactory.openSession();
try {
  BlogMapper mapper = session.getMapper(BlogMapper.class);
  Blog blog = mapper.selectBlog(101);
} finally {
  session.close();
}
```

Now let's explore what exactly is being executed here.

2.1.5 Exploring Mapped SQL Statements

At this point you may be wondering what exactly is being executed by the SqlSession or Mapper class. The topic of Mapped SQL Statements is a big one, and that topic will likely dominate the majority of this documentation. But to give you an idea of what exactly is being run, here are a couple of examples.

In either of the examples above, the statements could have been defined by either XML or Annotations. Let's take a look at XML first. The full set of features provided by MyBatis can be realized by using the XML based mapping language that has made MyBatis popular over the years. If you've used MyBatis before, the concept will be familiar to you, but there have been numerous improvements to the XML mapping documents that will become clear later. Here is an example of an XML based mapped statement that would satisfy the above SqlSession calls.

```
<?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" 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);
```

The second approach has a lot of advantages. First, it doesn't depend on a string literal, so it's much safer. Second, if your IDE has code completion, you can leverage that when navigating your mapped SQL statements.

NOTE **A note about namespaces.**

Namespaces were optional in previous versions of MyBatis, which was confusing and unhelpful. Namespaces are now required and have a purpose beyond simply isolating statements with longer, fully-qualified names.

Namespaces enable the interface bindings as you see here, and even if you don’t think you’ll use them today, you should follow these practices laid out here in case you change your mind. Using the namespace once, and putting it in a proper Java package namespace will clean up your code and improve the usability of MyBatis in the long term.

Name Resolution: To reduce the amount of typing, MyBatis uses the following name resolution rules for all named configuration elements, including statements, result maps, caches, etc.

- Fully qualified names (e.g. “com.mypackage.MyMapper.selectAllThings”) are looked up directly and used if found.
- Short names (e.g. “selectAllThings”) can be used to reference any unambiguous entry. However if there are two or more (e.g. “com.foo.selectAllThings and com.bar.selectAllThings”), then you will receive an error reporting that the short name is ambiguous and therefore must be fully qualified.

There's one more trick to Mapper classes like BlogMapper. Their mapped statements don't need to be mapped with XML at all. Instead they can use Java Annotations. For example, the XML above could be eliminated and replaced with:

```
package org.mybatis.example;
public interface BlogMapper {
  @Select("SELECT * FROM blog WHERE id = #{id}")
  Blog selectBlog(int id);
}
```

The annotations are a lot cleaner for simple statements, however, Java Annotations are both limited and messier for more complicated statements. Therefore, if you have to do anything complicated, you're better off with XML mapped statements.

It will be up to you and your project team to determine which is right for you, and how important it is to you that your mapped statements be defined in a consistent way. That said, you're never locked into a single approach. You can very easily migrate Annotation based Mapped Statements to XML and vice versa.

2.1.6 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.

NOTE **Object lifecycle and Dependency Injection Frameworks**

Dependency Injection frameworks can create thread safe, transactional SqlSessions and mappers and inject them directly into your beans so you can just forget about their lifecycle. You may want to have a look at MyBatis-Spring or MyBatis-Guice sub-projects to know more about using MyBatis with DI frameworks.

2.1.6.1 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.

2.1.6.2 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.

2.1.6.3 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 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();
}
```

Using this pattern consistently throughout your code will ensure that all database resources are properly closed.

2.1.6.4 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();
}
```

3 Configuration XML

3.1 Configuration

The MyBatis configuration contains settings and properties that have a dramatic effect on how MyBatis behaves. The high level structure of the document is as follows:

- configuration
  - properties
  - settings
  - typeAliases
  - typeHandlers
  - objectFactory
  - plugins
  - environments
    - environment
      - transactionManager
      - dataSource
  - databaseIdProvider
  - mappers

3.1.1 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 SqlSessionFactoryBuilder.build() methods. For example:

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

// ... or ...

SqlSessionFactory factory =
  new 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.

Since the MyBatis 3.4.2, your can specify a default value into placeholder as follow:

```
<dataSource type="POOLED">
  <!-- ... -->
  <property name="username" value="${username:ut_user}"/> <!-- If 'username' proper
</dataSource>
```

This feature is disabled by default. If you specify a default value into placeholder, you should be enable this feature by adding a special property as follow:

```
<properties resource="org/mybatis/example/config.properties">
  <!-- ... -->
  <property name="org.apache.ibatis.parsing.PropertyParser.enable-default-value" va
</properties>
```

NOTE Also If you are used already the `":"` as property key(e.g. `db:username`) or you are used already the ternary operator of OGNL expression(e.g. `${tableName != null ? tableName : 'global_constants'}`) on your sql definition, you should be change the character that separate key and default value by adding a special property as follow:

```
<properties resource="org/mybatis/example/config.properties">
  <!-- ... -->
  <property name="org.apache.ibatis.parsing.PropertyParser.default-value-separator"
</properties>
```

```
<dataSource type="POOLED">
  <!-- ... -->
  <property name="username" value="${db:username?:ut_user}"/>
</dataSource>
```

3.1.2 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 enabled, all relations will be lazily loaded. This value can be superseded for an specific relation by using the `fetchType` attribute on it.	true | false	false
aggressiveLazyLoading	When enabled, any method call will load all the lazy properties of the object. Otherwise, each property is loaded on demand (see also `lazyLoadTriggerMeth`	true | false	false (true in ≤3.4.1)
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
autoMappingUnknownColur	Specify the behavior when detects an unknown column (or unknown property type) of automatic mapping target. • `NONE`: Do nothing • `WARNING`: Output warning log (The log level of `'org.apache.iba` must be set to `WARN`) • `FAILING`: Fail mapping (Throw `SqlSessionExcep`	NONE, WARNING, FAILING	NONE `havior'`
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)
defaultFetchSize	Sets the driver a hint as to control fetching size for return results. This parameter value can be override by a query setting.	Any positive integer	Not Set (null)
safeRowBoundsEnabled	Allows using RowBounds on nested statements. If allow, set the false.	true | false	False
safeResultHandlerEnabled	Allows using ResultHandler on nested statements. If allow, set the false.	true | false	True

mapUnderscoreToCamelCa	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=STATEM 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.XMLLangu
defaultEnumTypeHandler	Specifies the `TypeHandler` used by default for Enum. (Since: 3.4.5)	A type alias or fully qualified class name.	org.apache.ibatis.type.EnumTypeHandler
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

returnInstanceForEmptyRo	MyBatis, by default, returns `null` when all the columns of a returned row are NULL. When this setting is enabled, MyBatis returns an empty instance instead. Note that it is also applied to nested results (i.e. collectioin and association). Since: 3.4.2	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	JAVASSIST (MyBatis 3.3 or above)
vfsImpl	Specifies VFS implementations	Fully qualified class names of custom VFS implementation separated by commas.	Not set
useActualParamName	Allow referencing statement parameters by their actual names declared in the method signature. To use this feature, your project must be compiled in Java 8 with `-parameters` option. (Since: 3.4.1)	true | false	true
configurationFactory	Specifies the class that provides an instance of `Configuration`. The returned Configuration instance is used to load lazy properties of deserialized objects. This class must have a method with a signature `static Configuration getConfiguration()`. (Since: 3.2.3)	A type alias or fully qualified class name.	Not set

An example of the settings element fully configured is as follows:

```
<settings>
  <setting name="cacheEnabled" value="true"/>
  <setting name="lazyLoadingEnabled" value="true"/>
  <setting name="multipleResultSetsEnabled" value="true"/>
  <setting name="useColumnLabel" value="true"/>
  <setting name="useGeneratedKeys" value="false"/>
  <setting name="autoMappingBehavior" value="PARTIAL"/>
  <setting name="autoMappingUnknownColumnBehavior" value="WARNING"/>
  <setting name="defaultExecutorType" value="SIMPLE"/>
  <setting name="defaultStatementTimeout" value="25"/>
  <setting name="defaultFetchSize" value="100"/>
  <setting name="safeRowBoundsEnabled" value="false"/>
  <setting name="mapUnderscoreToCamelCase" value="false"/>
  <setting name="localCacheScope" value="SESSION"/>
  <setting name="jdbcTypeForNull" value="OTHER"/>
  <setting name="lazyLoadTriggerMethods"
    value="equals,clone,hashCode,toString"/>
</settings>
```

3.1.3 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>
```

With this configuration, `Blog` can now be used anywhere that `domain.blog.Blog` could be.

You can also specify a package where MyBatis will search for beans. For example:

```
<typeAliases>
  <package name="domain.blog"/>
</typeAliases>
```

Each bean found in `domain.blog` , if no annotation is found, will be registered as an alias using uncapitalized non-qualified class name of the bean. That is `domain.blog.Author` will be registered as `author`. If the `@Alias` annotation is found its value will be used as an alias. See the example below:

```
@Alias("author")
public class Author {
    ...
}
```

There are many built-in type aliases for common Java types. They are all case insensitive, note the special handling of primitives due to the overloaded names.

Alias	Mapped Type
_byte	byte
_long	long
_short	short
_int	int
_integer	int
_double	double
_float	float
_boolean	boolean
string	String
byte	Byte
long	Long
short	Short
int	Integer
integer	Integer
double	Double
float	Float
boolean	Boolean
date	Date
decimal	BigDecimal
bigdecimal	BigDecimal
object	Object
map	Map
hashmap	HashMap
list	List
arraylist	ArrayList
collection	Collection
iterator	Iterator

3.1.4 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.

NOTE Since version 3.4.5, The MyBatis has been supported JSR-310(Date and Time API) by default.

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 SMALLINT
IntegerTypeHandler	java.lang.Integer, int	Any compatible NUMERIC or INTEGER
LongTypeHandler	java.lang.Long, long	Any compatible NUMERIC or BIGINT
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
ClobReaderTypeHandler	java.io.Reader	-
ClobTypeHandler	java.lang.String	CLOB, LONGVARCHAR
NStringTypeHandler	java.lang.String	NVARCHAR, NCHAR
NClobTypeHandler	java.lang.String	NCLOB
BlobInputStreamTypeHandle	java.io.InputStream	-
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).
SqlxmlTypeHandler	java.lang.String	SQLXML
InstantTypeHandler	java.time.Instant	TIMESTAMP
LocalDateTimeTypeHandler	java.time.LocalDateTime	TIMESTAMP
LocalDateTypeHandler	java.time.LocalDate	DATE
LocalTimeTypeHandler	java.time.LocalTime	TIME
OffsetDateTimeTypeHandler	java.time.OffsetDateTime	TIMESTAMP

OffsetTimeTypeHandler	java.time.OffsetTime	TIME
ZonedDateTimeTypeHandler	java.time.ZonedDateTime	TIMESTAMP
YearTypeHandler	java.time.Year	INTEGER
MonthTypeHandler	java.time.Month	INTEGER
YearMonthTypeHandler	java.time.YearMonth	VARCHAR or LONGVARCHAR
JapaneseDateTypeHandler	java.time.chrono.Japanese	DATE

You can override the type handlers or create your own to deal with unsupported or non-standard types. To do so, implement the interface `org.apache.ibatis.type.TypeHandler` or extend the convenience class `org.apache.ibatis.type.BaseTypeHandler` and optionally map it to a JDBC type. For example:

```
// ExampleTypeHandler.java
@MappedJdbcTypes(JdbcType.VARCHAR)
public class ExampleTypeHandler extends BaseTypeHandler<String> {

  @Override
  public void setNonNullParameter(PreparedStatement ps, int i,
    String parameter, JdbcType jdbcType) throws SQLException {
    ps.setString(i, parameter);
  }

  @Override
  public String getNullableResult(ResultSet rs, String columnName)
    throws SQLException {
    return rs.getString(columnName);
  }

  @Override
  public String getNullableResult(ResultSet rs, int columnIndex)
    throws SQLException {
    return rs.getString(columnIndex);
  }

  @Override
  public String getNullableResult(CallableStatement cs, int columnIndex)
    throws SQLException {
    return cs.getString(columnIndex);
  }
}
```

```
<!-- mybatis-config.xml -->
<typeHandlers>
  <typeHandler handler="org.mybatis.example.ExampleTypeHandler"/>
</typeHandlers>
```

Using such a TypeHandler would override the existing type handler for Java String properties and VARCHAR parameters and results. Note that MyBatis does not introspect upon the database metadata to determine the type, so you must specify that it’s a VARCHAR field in the parameter and

result mappings to hook in the correct type handler. This is due to the fact that MyBatis is unaware of the data type until the statement is executed.

MyBatis will know the the Java type that you want to handle with this TypeHandler by introspecting its generic type, but you can override this behavior by two means:

- Adding a `javaType` attribute to the typeHandler element (for example: `javaType="String"`)
- Adding a `@MappedTypes` annotation to your TypeHandler class specifying the list of java types to associate it with. This annotation will be ignored if the `javaType` attribute as also been specified.

Associated JDBC type can be specified by two means:

- Adding a `jdbcType` attribute to the typeHandler element (for example: `jdbcType="VARCHAR"`).
- Adding a `@MappedJdbcTypes` annotation to your TypeHandler class specifying the list of JDBC types to associate it with. This annotation will be ignored if the `jdbcType` attribute as also been specified.

When deciding which TypeHandler to use in a `ResultMap`, the Java type is known (from the result type), but the JDBC type is unknown. MyBatis therefore uses the combination `javaType=[TheJavaType], jdbcType=null` to choose a TypeHandler. This means that using a `@MappedJdbcTypes` annotation *restricts* the scope of a TypeHandler and makes it unavailable for use in `ResultMap`s unless explicity set. To make a TypeHandler available for use in a `ResultMap`, set `includeNullJdbcType=true` on the `@MappedJdbcTypes` annotation. Since Mybatis 3.4.0 however, if a **single** TypeHandler is registered to handle a Java type, it will be used by default in `ResultMap`s using this Java type (i.e. even without `includeNullJdbcType=true`).

And finally you can let MyBatis search for your TypeHandlers:

```
<!-- mybatis-config.xml -->
<typeHandlers>
  <package name="org.mybatis.example"/>
</typeHandlers>
```

Note that when using the autodiscovery feature JDBC types can only be specified with annotations.

You can create a generic TypeHandler that is able to handle more than one class. For that purpose add a constructor that receives the class as a parameter and MyBatis will pass the actual class when constructing the TypeHandler.

```
//GenericTypeHandler.java
public class GenericTypeHandler<E extends MyObject> extends BaseTypeHandler<E> {

  private Class<E> type;

  public GenericTypeHandler(Class<E> type) {
    if (type == null) throw new IllegalArgumentException("Type argument cannot be n
    this.type = type;
  }
  ...
```

`EnumTypeHandler` and `EnumOrdinalTypeHandler` are generic TypeHandlers. We will learn about them in the following section.

3.1.5 Handling Enums

If you want to map an `Enum`, you'll need to use either `EnumTypeHandler` or `EnumOrdinalTypeHandler`.

For example, let's say that we need to store the rounding mode that should be used with some number if it needs to be rounded. By default, MyBatis uses `EnumTypeHandler` to convert the `Enum` values to their names.

Note `EnumTypeHandler` is special in the sense that unlike other handlers, it does not handle just one specific class, but any class that extends `Enum`

However, we may not want to store names. Our DBA may insist on an integer code instead. That's just as easy: add `EnumOrdinalTypeHandler` to the `typeHandlers` in your config file, and now each `RoundingMode` will be mapped to an integer using its ordinal value.

```
<!-- mybatis-config.xml -->
<typeHandlers>
  <typeHandler handler="org.apache.ibatis.type.EnumOrdinalTypeHandler"
    javaType="java.math.RoundingMode"/>
</typeHandlers>
```

But what if you want to map the same `Enum` to a string in one place and to integer in another?

The auto-mapper will automatically use `EnumOrdinalTypeHandler`, so if we want to go back to using plain old ordinary `EnumTypeHandler`, we have to tell it, by explicitly setting the type handler to use for those SQL statements.

(Mapper files aren't covered until the next section, so if this is your first time reading through the documentation, you may want to skip this for now and come back to it later.)

```
<!DOCTYPE mapper
    PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="org.apache.ibatis.submitted.rounding.Mapper">
    <resultMap type="org.apache.ibatis.submitted.rounding.User" id="usermap">
        <id column="id" property="id"/>
        <result column="name" property="name"/>
        <result column="funkyNumber" property="funkyNumber"/>
        <result column="roundingMode" property="roundingMode"/>
    </resultMap>

    <select id="getUser" resultMap="usermap">
        select * from users
    </select>
    <insert id="insert">
        insert into users (id, name, funkyNumber, roundingMode) values (
            #{id}, #{name}, #{funkyNumber}, #{roundingMode}
        )
    </insert>

    <resultMap type="org.apache.ibatis.submitted.rounding.User" id="usermap2">
        <id column="id" property="id"/>
        <result column="name" property="name"/>
        <result column="funkyNumber" property="funkyNumber"/>
        <result column="roundingMode" property="roundingMode"
         typeHandler="org.apache.ibatis.type.EnumTypeHandler"/>
    </resultMap>
    <select id="getUser2" resultMap="usermap2">
        select * from users2
    </select>
    <insert id="insert2">
        insert into users2 (id, name, funkyNumber, roundingMode) values (
            #{id}, #{name}, #{funkyNumber}, #{roundingMode, typeHandler=org.apache.
        )
    </insert>

</mapper>
```

Note that this forces us to use a `resultMap` instead of a `resultType` in our select statements.

3.1.6 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:

```
// ExampleObjectFactory.java
public class ExampleObjectFactory extends DefaultObjectFactory {
  public Object create(Class type) {
    return super.create(type);
  }
  public Object create(Class type, List<Class> constructorArgTypes, List<Object> co
    return super.create(type, constructorArgTypes, constructorArgs);
  }
  public void setProperties(Properties properties) {
    super.setProperties(properties);
  }
  public <T> boolean isCollection(Class<T> type) {
    return Collection.class.isAssignableFrom(type);
  }}
```

```
<!-- mybatis-config.xml -->
<objectFactory type="org.mybatis.example.ExampleObjectFactory">
  <property name="someProperty" value="100"/>
</objectFactory>
```

The ObjectFactory interface is very simple. It contains two create methods, one to deal with the default constructor, and the other to deal with parameterized constructors. Finally, the setProperties method can be used to configure the ObjectFactory. Properties defined within the body of the objectFactory element will be passed to the setProperties method after initialization of your ObjectFactory instance.

3.1.7 plugins

MyBatis allows you to intercept calls to at certain points within the execution of a mapped statement. By default, MyBatis allows plug-ins to intercept method calls of:

- Executor (update, query, flushStatements, commit, rollback, getTransaction, close, isClosed)
- ParameterHandler (getParameterObject, setParameters)
- ResultSetHandler (handleResultSets, handleOutputParameters)
- StatementHandler (prepare, parameterize, batch, update, query)

The details of these classes methods can be discovered by looking at the full method signature of each, and the source code which is available with each MyBatis release. You should understand the behaviour of the method you’re overriding, assuming you’re doing something more than just monitoring calls. If you attempt to modify or override the behaviour of a given method, you’re likely to break the core of MyBatis. These are low level classes and methods, so use plug-ins with caution.

Using plug-ins is pretty simple given the power they provide. Simply implement the Interceptor interface, being sure to specify the signatures you want to intercept.

```
// ExamplePlugin.java
@Intercepts({@Signature(
  type= Executor.class,
  method = "update",
  args = {MappedStatement.class,Object.class})})
public class ExamplePlugin implements Interceptor {
  public Object intercept(Invocation invocation) throws Throwable {
    return invocation.proceed();
  }
  public Object plugin(Object target) {
    return Plugin.wrap(target, this);
  }
  public void setProperties(Properties properties) {
  }
}
```

```
<!-- mybatis-config.xml -->
<plugins>
  <plugin interceptor="org.mybatis.example.ExamplePlugin">
    <property name="someProperty" value="100"/>
  </plugin>
</plugins>
```

The plug-in above will intercept all calls to the "update" method on the Executor instance, which is an internal object responsible for the low level execution of mapped statements.

NOTE **Overriding the Configuration Class**

In addition to modifying core MyBatis behaviour with plugins, you can also override the Configuration class entirely. Simply extend it and override any methods inside, and pass it into the call to the SqlSessionFactoryBuilder.build(myConfig) method. Again though, this could have a severe impact on the behaviour of MyBatis, so use caution.

3.1.8 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:

- **One SqlSessionFactory instance per database**

To specify which environment to build, you simply pass it to the SqlSessionFactoryBuilder as an optional parameter. The two signatures that accept the environment are:

```
SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader, environmen
SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader, environmen
```

If the environment is omitted, then the default environment is loaded, as follows:

```
SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader);
SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader, properties
```

The environments element defines how the environment is configured.

```
<environments default="development">
  <environment id="development">
    <transactionManager type="JDBC">
      <property name="..." value="..."/>
    </transactionManager>
    <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>
```

Notice the key sections here:

- The default Environment ID (e.g. default="development").
- The Environment ID for each environment defined (e.g. id="development").
- The TransactionManager configuration (e.g. type="JDBC")
- The DataSource configuration (e.g. type="POOLED")

The default environment and the environment IDs are self explanatory. Name them whatever you like, just make sure the default matches one of them.

transactionManager

There are two TransactionManager types (i.e. type="[JDBC|MANAGED]") that are included with MyBatis:

- JDBC – This configuration simply makes use of the JDBC commit and rollback facilities directly. It relies on the connection retrieved from the dataSource to manage the scope of the transaction.
- MANAGED – This configuration simply does almost nothing. It never commits, or rolls back a connection. Instead, it lets the container manage the full lifecycle of the transaction (e.g. a JEE Application Server context). By default it does close the connection. However, some containers don't expect this, and thus if you need to stop it from closing the connection, set the "closeConnection" property to false. For example:

```
<transactionManager type="MANAGED">
  <property name="closeConnection" value="false"/>
</transactionManager>
```

NOTE If you are planning to use MyBatis with Spring there is no need to configure any TransactionManager because the Spring module will set its own one overriding any previously set configuration.

Neither of these TransactionManager types require any properties. However, they are both Type Aliases, so in other words, instead of using them, you could put your own fully qualified class name or Type Alias that refers to your own implementation of the TransactionFactory interface.

```
public interface TransactionFactory {
  void setProperties(Properties props);
  Transaction newTransaction(Connection conn);
  Transaction newTransaction(DataSource dataSource, TransactionIsolationLevel level
}
```

Any properties configured in the XML will be passed to the setProperties() method after instantiation. Your implementation would also need to create a Transaction implementation, which is also a very simple interface:

```
public interface Transaction {
  Connection getConnection() throws SQLException;
  void commit() throws SQLException;
  void rollback() throws SQLException;
  void close() throws SQLException;
  Integer getTimeout() throws SQLException;
}
```

Using these two interfaces, you can completely customize how MyBatis deals with Transactions.

dataSource

The dataSource element configures the source of JDBC Connection objects using the standard JDBC DataSource interface.

- Most MyBatis applications will configure a dataSource as in the example. However, it's not required. Realize though, that to facilitate Lazy Loading, this dataSource is required.

There are three build-in dataSource types (i.e. type="[UNPOOLED|POOLED|JNDI]"):

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)
- `poolMaximumLocalBadConnectionTolerance` – This is a low level setting about tolerance of bad connections got for any thread. If a thread got a bad connection, it may still have another chance to re-attempt to get another connection which is valid. But the retrying times should not more than the sum of `poolMaximumIdleConnections` and `poolMaximumLocalBadConnectionTolerance`. Default: 3 (Since: 3.4.5)
- `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 with `env.`, for example:

- `env.encoding=UTF8`

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

You can plug any 3rd party DataSource by implementing the interface `org.apache.ibatis.datasource.DataSourceFactory`:

```
public interface DataSourceFactory {
  void setProperties(Properties props);
  DataSource getDataSource();
}
```

`org.apache.ibatis.datasource.unpooled.UnpooledDataSourceFactory` can be used as super class to build new datasource adapters. For example this is the code needed to plug C3P0:

```
import org.apache.ibatis.datasource.unpooled.UnpooledDataSourceFactory;
import com.mchange.v2.c3p0.ComboPooledDataSource;

public class C3P0DataSourceFactory extends UnpooledDataSourceFactory {

  public C3P0DataSourceFactory() {
    this.dataSource = new ComboPooledDataSource();
  }
}
```

To set it up, add a property for each setter method you want MyBatis to call. Follows below a sample configuration which connects to a PostgreSQL database:

```
<dataSource type="org.myproject.C3P0DataSourceFactory">
  <property name="driver" value="org.postgresql.Driver"/>
  <property name="url" value="jdbc:postgresql:mydb"/>
  <property name="username" value="postgres"/>
  <property name="password" value="root"/>
</dataSource>
```

3.1.9 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. In case the same statement is 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="DB_VENDOR" />
```

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

```
<databaseIdProvider type="DB_VENDOR">
  <property name="SQL Server" value="sqlserver"/>
  <property name="DB2" value="db2"/>
  <property name="Oracle" value="oracle" />
</databaseIdProvider>
```

When properties are provided, the DB_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;
}
```

3.1.10 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 classpath relative resource references, fully qualified url references (including `file:///` URLs), class names or package names. 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.

4 Mapper XML Files

4.1 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.

The next sections will describe each of these elements in detail, starting with the statements themselves.

4.1.1 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 are 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:

```
// Similar JDBC code, NOT MyBatis…
String selectPerson = "SELECT * FROM PERSON WHERE ID=?";
PreparedStatement ps = conn.prepareStatement(selectPerson);
ps.setInt(1,id);
```

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. There's a lot more to know about parameter and result mapping. Those details warrant their own section, which follows later in this section.

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="10"
  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. This attribute is optional because MyBatis can calculate the TypeHandler to use out of the actual parameter passed to the statement. Default is `unset`.
`parameterMap`	This is a deprecated approach to referencing an external `parameterMap`. Use inline parameter mappings and the `parameterType` 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` OR `resultMap`, 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: `false` for 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` or `CallableStatement` respectively. Default: `PREPARED`.
`resultSetType`	Any one of `FORWARD_ONLY`| `SCROLL_SENSITIVE`| `SCROLL_INSENSITIVE`| `DEFAULT`(same as unset). Default is `unset` (driver dependent).
`databaseId`	In case there is a configured databaseIdProvider, 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.
`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.

Select Attributes

4.1.2 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="updateAuthor"
  parameterType="domain.blog.Author"
  flushCache="true"
  statementType="PREPARED"
  timeout="20">

<delete
  id="deleteAuthor"
  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. This attribute is optional because MyBatis can calculate the TypeHandler to use out of the actual parameter passed to the statement. Default is `unset`.
`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` or `CallableStatement` respectively. Default: `PREPARED`.

`useGeneratedKeys`	(insert and update 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 and update only) Identifies a property into which MyBatis will set the key value returned by `getGeneratedKeys`, or by a `selectKey` child element of the insert statement. Default: `unset`. Can be a comma separated list of property names if multiple generated columns are expected.
`keyColumn`	(insert and update 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. Can be a comma separated list of columns names if multiple generated columns are expected.
`databaseId`	In case there is a configured databaseIdProvider, 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.

Insert, Update and Delete Attributes

The following are some examples of insert, update and delete statements.

```
<insert id="insertAuthor">
  insert into Author (id,username,password,email,bio)
  values (#{id},#{username},#{password},#{email},#{bio})
</insert>

<update id="updateAuthor">
  update Author set
    username = #{username},
    password = #{password},
    email = #{email},
    bio = #{bio}
  where id = #{id}
</update>

<delete id="deleteAuthor">
  delete from Author where id = #{id}
</delete>
```

As mentioned, insert is a little bit more rich in that it has a few extra attributes and sub-elements that allow it to deal with key generation in a number of ways.

First, if your database supports auto-generated key fields (e.g. MySQL and SQL Server), then you can simply set `useGeneratedKeys="true"` and set the `keyProperty` to the target property and you're done. For example, if the `Author` table above had used an auto-generated column type for the id, the statement would be modified as follows:

```
<insert id="insertAuthor" useGeneratedKeys="true"
    keyProperty="id">
  insert into Author (username,password,email,bio)
  values (#{username},#{password},#{email},#{bio})
</insert>
```

If your database also supports multi-row insert, you can pass a list or an array of `Authors` and retrieve the auto-generated keys.

```
<insert id="insertAuthor" useGeneratedKeys="true"
    keyProperty="id">
  insert into Author (username, password, email, bio) values
  <foreach item="item" collection="list" separator=",">
    (#{item.username}, #{item.password}, #{item.email}, #{item.bio})
  </foreach>
</insert>
```

MyBatis has another way to deal with key generation for databases that don't support auto-generated column types, or perhaps don't yet support the JDBC driver support for auto-generated keys.

Here's a simple (silly) example that would generate a random ID (something you'd likely never do, but this demonstrates the flexibility and how MyBatis really doesn't mind):

```
<insert id="insertAuthor">
  <selectKey keyProperty="id" resultType="int" order="BEFORE">
    select CAST(RANDOM()*1000000 as INTEGER) a from SYSIBM.SYSDUMMY1
  </selectKey>
  insert into Author
    (id, username, password, email,bio, favourite_section)
  values
    (#{id}, #{username}, #{password}, #{email}, #{bio}, #{favouriteSection,jdbcType
</insert>
```

In the example above, the selectKey statement would be run first, the `Author` id property would be set, and then the insert statement would be called. This gives you a similar behavior to an auto-generated key in your database without complicating your Java code.

The selectKey element is described as follows:

```
<selectKey
  keyProperty="id"
  resultType="int"
  order="BEFORE"
  statementType="PREPARED">
```

Attribute	Description
`keyProperty`	The target property where the result of the `selectKey` statement should be set. Can be a comma separated list of property names if multiple generated columns are expected.

`keyColumn`	The column name(s) in the returned result set that match the properties. Can be a comma separated list of column names if multiple generated columns are expected.
`resultType`	The type of the result. MyBatis can usually figure this out, but it doesn't hurt to add it to be sure. MyBatis allows any simple type to be used as the key, including Strings. If you are expecting multiple generated columns, then you can use an Object that contains the expected properties, or a Map.
`order`	This can be set to `BEFORE` or `AFTER`. If set to `BEFORE`, then it will select the key first, set the `keyProperty` and then execute the insert statement. If set to `AFTER`, it runs the insert statement and then the `selectKey` statement – which is common with databases like Oracle that may have embedded sequence calls inside of insert statements.
`statementType`	Same as above, MyBatis supports `STATEMENT`, `PREPARED` and `CALLABLE` statement types that map to `Statement`, `PreparedStatement` and `CallableStatement` respectively.

selectKey Attributes

4.1.3 sql

This element can be used to define a reusable fragment of SQL code that can be included in other statements. It can be statically (during load phase) parametrized. Different property values can vary in include instances. For example:

```
<sql id="userColumns"> ${alias}.id,${alias}.username,${alias}.password </sql>
```

The SQL fragment can then be included in another statement, for example:

```
<select id="selectUsers" resultType="map">
  select
    <include refid="userColumns"><property name="alias" value="t1"/></include>,
    <include refid="userColumns"><property name="alias" value="t2"/></include>
  from some_table t1
    cross join some_table t2
</select>
```

Property value can be also used in include refid attribute or property values inside include clause, for example:

```
<sql id="sometable">
  ${prefix}Table
</sql>

<sql id="someinclude">
  from
    <include refid="${include_target}"/>
</sql>

<select id="select" resultType="map">
  select
    field1, field2, field3
  <include refid="someinclude">
    <property name="prefix" value="Some"/>
    <property name="include_target" value="sometable"/>
  </include>
</select>
```

4.1.4 Parameters

In all of the past statements, you've seen examples of simple parameters. Parameters are very powerful elements in MyBatis. For simple situations, probably 90% of the cases, there's not much to them, for example:

```
<select id="selectUsers" resultType="User">
  select id, username, password
  from users
  where id = #{id}
</select>
```

The example above demonstrates a very simple named parameter mapping. The parameterType is set to `int`, so therefore the parameter could be named anything. Primitive or simple data types such as `Integer` and `String` have no relevant properties, and thus will replace the full value of the parameter entirely. However, if you pass in a complex object, then the behavior is a little different. For example:

```
<insert id="insertUser" parameterType="User">
  insert into users (id, username, password)
  values (#{id}, #{username}, #{password})
</insert>
```

If a parameter object of type User was passed into that statement, the id, username and password property would be looked up and their values passed to a `PreparedStatement` parameter.

That's nice and simple for passing parameters into statements. But there are a lot of other features of parameter maps.

First, like other parts of MyBatis, parameters can specify a more specific data type.

```
#{property,javaType=int,jdbcType=NUMERIC}
```

Like the rest of MyBatis, the javaType can almost always be determined from the parameter object, unless that object is a `HashMap`. Then the `javaType` should be specified to ensure the correct `TypeHandler` is used.

NOTE The JDBC Type is required by JDBC for all nullable columns, if `null` is passed as a value. You can investigate this yourself by reading the JavaDocs for the `PreparedStatement.setNull()` method.

To further customize type handling, you can also specify a specific `TypeHandler` class (or alias), for example:

```
#{age,javaType=int,jdbcType=NUMERIC,typeHandler=MyTypeHandler}
```

So already it seems to be getting verbose, but the truth is that you'll rarely set any of these.

For numeric types there's also a `numericScale` for determining how many decimal places are relevant.

```
#{height,javaType=double,jdbcType=NUMERIC,numericScale=2}
```

Finally, the mode attribute allows you to specify `IN`, `OUT` or `INOUT` parameters. If a parameter is `OUT` or `INOUT`, the actual value of the parameter object property will be changed, just as you would expect if you were calling for an output parameter. If the `mode=OUT` (or `INOUT`) and the `jdbcType=CURSOR` (i.e. Oracle REFCURSOR), you must specify a `resultMap` to map the `ResultSet` to the type of the parameter. Note that the `javaType` attribute is optional here, it will be automatically set to `ResultSet` if left blank with a `CURSOR` as the `jdbcType`.

```
#{department, mode=OUT, jdbcType=CURSOR, javaType=ResultSet, resultMap=departmentRe
```

MyBatis also supports more advanced data types such as structs, but you must tell the statement the type name when registering the out parameter. For example (again, don't break lines like this in practice):

```
#{middleInitial, mode=OUT, jdbcType=STRUCT, jdbcTypeName=MY_TYPE, resultMap=departm
```

Despite all of these powerful options, most of the time you'll simply specify the property name, and MyBatis will figure out the rest. At most, you'll specify the `jdbcType` for nullable columns.

```
#{firstName}
#{middleInitial,jdbcType=VARCHAR}
#{lastName}
```

4.1.4.1 String Substitution

By default, using the `#{}` syntax will cause MyBatis to generate `PreparedStatement` properties and set the values safely against the `PreparedStatement` parameters (e.g. ?). While this is safer, faster and almost always preferred, sometimes you just want to directly inject an unmodified string into the SQL Statement. For example, for ORDER BY, you might use something like this:

```
ORDER BY ${columnName}
```

Here MyBatis won't modify or escape the string.

String Substitution can be very useful when the metadata(i.e. table name or column name) in the sql statement is dynamic, for example, if you want to `select` from a table by any one of its columns, instead of writing code like:

```
@Select("select * from user where id = #{id}")
User findById(@Param("id") long id);

@Select("select * from user where name = #{name}")
User findByName(@Param("name") String name);

@Select("select * from user where email = #{email}")
User findByEmail(@Param("email") String email);

// and more "findByXxx" method
```

you can just write:

```
@Select("select * from user where ${column} = #{value}")
User findByColumn(@Param("column") String column, @Param("value") String value);
```

in which the `${column}` will be substituted directly and the `#{value}` will be "prepared". Thus you can just do the same work by:

```
User userOfId1 = userMapper.findByColumn("id", 1L);
User userOfNameKid = userMapper.findByColumn("name", "kid");
User userOfEmail = userMapper.findByColumn("email", "noone@nowhere.com");
```

This idea can be applied to substitute the table name as well.

NOTE It's not safe to accept input from a user and supply it to a statement unmodified in this way. This leads to potential SQL Injection attacks and therefore you should either disallow user input in these fields, or always perform your own escapes and checks.

4.1.5 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 the `ResultMaps` 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.

You've already seen examples of simple mapped statements that don't have an explicit `resultMap`. For example:

```
<select id="selectUsers" resultType="map">
  select id, username, hashedPassword
  from some_table
  where id = #{id}
</select>
```

Such a statement simply results in all columns being automatically mapped to the keys of a `HashMap`, as specified by the `resultType` attribute. While useful in many cases, a `HashMap` doesn't make a very good domain model. It's more likely that your application will use JavaBeans or POJOs (Plain Old Java Objects) for the domain model. MyBatis supports both. Consider the following JavaBean:

```
package com.someapp.model;
public class User {
  private int id;
  private String username;
  private String hashedPassword;

  public int getId() {
    return id;
  }
  public void setId(int id) {
    this.id = id;
  }
  public String getUsername() {
    return username;
  }
  public void setUsername(String username) {
    this.username = username;
  }
  public String getHashedPassword() {
    return hashedPassword;
  }
  public void setHashedPassword(String hashedPassword) {
    this.hashedPassword = hashedPassword;
  }
}
```

Based on the JavaBeans specification, the above class has 3 properties: id, username, and hashedPassword. These match up exactly with the column names in the select statement.

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

```
<select id="selectUsers" resultType="com.someapp.model.User">
  select id, username, hashedPassword
  from some_table
  where id = #{id}
</select>
```

And remember that TypeAliases are your friends. 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" resultType="User">
  select id, username, hashedPassword
  from some_table
  where id = #{id}
</select>
```

In these cases MyBatis is automatically creating a `ResultMap` behind the scenes to auto-map the columns to the JavaBean properties based on name. If the column names did not match exactly, you could employ select clause aliases (a standard SQL feature) on the column names to make the labels match. For example:

```
<select id="selectUsers" resultType="User">
  select
    user_id             as "id",
    user_name           as "userName",
    hashed_password     as "hashedPassword"
  from some_table
  where id = #{id}
</select>
```

The great thing about `ResultMap`s 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 external `resultMap`, 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" resultMap="userResultMap">
  select user_id, user_name, hashed_password
  from some_table
  where id = #{id}
</select>
```

Now if only the world was always that simple.

4.1.5.1 Advanced Result Maps

MyBatis was created with one idea in mind: Databases aren't always what you want or need them to be. While we'd love every database to be perfect 3rd normal form or BCNF, they aren't. And it would be great if it was possible to have a single database map perfectly to all of the applications that use it, it's not. Result Maps are the answer that MyBatis provides to this problem.

For example, how would we map this statement?

```
<!-- Very Complex Statement -->
<select id="selectBlogDetails" resultMap="detailedBlogResultMap">
  select
       B.id as blog_id,
       B.title as blog_title,
       B.author_id as blog_author_id,
       A.id as author_id,
       A.username as author_username,
       A.password as author_password,
       A.email as author_email,
       A.bio as author_bio,
       A.favourite_section as author_favourite_section,
       P.id as post_id,
       P.blog_id as post_blog_id,
       P.author_id as post_author_id,
       P.created_on as post_created_on,
       P.section as post_section,
       P.subject as post_subject,
       P.draft as draft,
       P.body as post_body,
       C.id as comment_id,
       C.post_id as comment_post_id,
       C.name as comment_name,
       C.comment as comment_text,
       T.id as tag_id,
       T.name as tag_name
  from Blog B
       left outer join Author A on B.author_id = A.id
       left outer join Post P on B.id = P.blog_id
       left outer join Comment C on P.id = C.post_id
       left outer join Post_Tag PT on PT.post_id = P.id
       left outer join Tag T on PT.tag_id = T.id
  where B.id = #{id}
</select>
```

You'd probably want to map it to an intelligent object model consisting of a Blog that was written by an Author, and has many Posts, each of which may have zero or many Comments and Tags. The following is a complete example of a complex ResultMap (assume Author, Blog, Post, Comments and Tags are all type aliases). Have a look at it, but don't worry, we're going to go through each step. While it may look daunting at first, it's actually very simple.

```
<!-- Very Complex Result Map -->
<resultMap id="detailedBlogResultMap" type="Blog">
  <constructor>
    <idArg column="blog_id" javaType="int"/>
  </constructor>
  <result property="title" column="blog_title"/>
  <association property="author" javaType="Author">
    <id property="id" column="author_id"/>
    <result property="username" column="author_username"/>
    <result property="password" column="author_password"/>
    <result property="email" column="author_email"/>
    <result property="bio" column="author_bio"/>
    <result property="favouriteSection" column="author_favourite_section"/>
  </association>
  <collection property="posts" ofType="Post">
    <id property="id" column="post_id"/>
    <result property="subject" column="post_subject"/>
    <association property="author" javaType="Author"/>
    <collection property="comments" ofType="Comment">
      <id property="id" column="comment_id"/>
    </collection>
    <collection property="tags" ofType="Tag" >
      <id property="id" column="tag_id"/>
    </collection>
    <discriminator javaType="int" column="draft">
      <case value="1" resultType="DraftPost"/>
    </discriminator>
  </collection>
</resultMap>
```

The `resultMap` element has a number of sub-elements and a structure worthy of some discussion. The following is a conceptual view of the `resultMap` element.

4.1.5.2 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 `resultMap`s themselves, or can refer to one
- `collection` – a collection of complex types
  - nested result mappings – collections are `resultMap`s 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.

ResultMap Attributes

Best Practice Always build ResultMaps incrementally. Unit tests really help out here. If you try to build a gigantic `resultMap` like the one above all at once, it's likely you'll get it wrong and it will be hard to work with. Start simple, and evolve it a step at a time. And unit test! The downside to using frameworks is that they are sometimes a bit of a black box (open source or not). Your best bet to ensure that you're achieving the behaviour that you intend, is to write unit tests. It also helps to have them when submitting bugs.

The next sections will walk through each of the elements in more detail.

4.1.5.3 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).

Each has a number of attributes:

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 to `resultSet.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.

Id and Result Attributes

4.1.5.4 Supported JDBC Types

For future reference, MyBatis supports the following JDBC Types via the included JdbcType enumeration.

BIT	FLOAT	CHAR	TIMESTAMP	OTHER	UNDEFINED
TINYINT	REAL	VARCHAR	BINARY	BLOB	NVARCHAR
SMALLINT	DOUBLE	LONGVARCHAR	VARBINARY	CLOB	NCHAR
INTEGER	NUMERIC	DATE	LONGVARBINAR	BOOLEAN	NCLOB
BIGINT	DECIMAL	TIME	NULL	CURSOR	ARRAY

4.1.5.5 constructor

While properties will work for most Data Transfer Object (DTO) type classes, and likely most of your domain model, there may be some cases where you want to use immutable classes. Often tables that contain reference or lookup data that rarely or never changes is suited to immutable classes. Constructor injection allows you to set values on a class upon instantiation, without exposing public methods. MyBatis also supports private properties and private JavaBeans properties to achieve this, but some people prefer Constructor injection. The *constructor* element enables this.

Consider the following constructor:

```
public class User {
   //...
   public User(Integer id, String username, int age) {
     //...
  }
//...
}
```

In order to inject the results into the constructor, MyBatis needs to identify the constructor for somehow. In the following example, MyBatis searches a constructor declared with three parameters: `java.lang.Integer`, `java.lang.String` and `int` in this order.

```
<constructor>
   <idArg column="id" javaType="int"/>
   <arg column="username" javaType="String"/>
   <arg column="age" javaType="_int"/>
</constructor>
```

When you are dealing with a constructor with many parameters, maintaining the order of arg elements is error-prone.
Since 3.4.3, by specifying the name of each parameter, you can write arg elements in any order. To reference constructor parameters by their names, you can either add `@Param` annotation to them or compile the project with '-parameters' compiler option and enable `useActualParamName` (this option is enabled by default). The following example is valid for the same constructor even though the order of the second and the third parameters does not match with the declared order.

```
<constructor>
   <idArg column="id" javaType="int" name="id" />
   <arg column="age" javaType="_int" name="age" />
   <arg column="username" javaType="String" name="username" />
</constructor>
```

`javaType` can be omitted if there is a property with the same name and type.

The rest of the attributes and rules are the same as for the regular id and result elements.

Attribute	Description
`column`	The column name from the database, or the aliased column label. This is the same string that would normally be passed to `resultSet.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 an 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.

`select`	The ID of another mapped statement that will load the complex type required by this property mapping. The values retrieved from columns specified in the column attribute will be passed to the target select statement as parameters. See the Association element for more.
`resultMap`	This is the ID of a ResultMap that can map the nested results of this argument into an appropriate object graph. This is an alternative to using a call to another select statement. It allows you to join multiple tables together into a single `ResultSet`. Such a `ResultSet` will contain duplicated, repeating groups of data that needs to be decomposed and mapped properly to a nested object graph. To facilitate this, MyBatis lets you "chain" result maps together, to deal with the nested results. See the Association element below for more.
`name`	The name of the constructor parameter. Specifying name allows you to write arg elements in any order. See the above explanation. Since 3.4.3.

4.1.5.6 association

```
<association property="author" javaType="Author">
  <id property="id" column="author_id"/>
  <result property="username" column="author_username"/>
</association>
```

The association element deals with a "has-one" type relationship. For example, in our example, a Blog has one Author. An association mapping works mostly like any other result. You specify the target property, the `javaType` of the property (which MyBatis can figure out most of the time), the jdbcType if necessary and a typeHandler if you want to override the retrieval of the result values.

Where the association differs is that you need to tell MyBatis how to load the association. MyBatis can do so in two different ways:

- Nested Select: By executing another mapped SQL statement that returns the complex type desired.
- Nested Results: By using nested result mappings to deal with repeating subsets of joined results.

First, let's examine the properties of the element. As you'll see, it differs from a normal result mapping only by the select and resultMap attributes.

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`.

`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 an 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.

4.1.5.7 Nested Select for Association

Attribute	Description
`column`	The column name from the database, or the aliased column label that holds the value that will be passed to the nested statement as an input parameter. This is the same string that would normally be passed to `resultSet.getString(columnName)`. Note: To deal with composite keys, you can specify multiple column names to pass to the nested select statement by using the syntax `column="{prop1=col1,prop2=col2}"`. This will cause `prop1` and `prop2` to be set against the parameter object for the target nested select statement.
`select`	The ID of another mapped statement that will load the complex type required by this property mapping. The values retrieved from columns specified in the column attribute will be passed to the target select statement as parameters. A detailed example follows this table. Note: To deal with composite keys, you can specify multiple column names to pass to the nested select statement by using the syntax `column="{prop1=col1,prop2=col2}"`. This will cause `prop1` and `prop2` to be set against the parameter object for the target nested select statement.
`fetchType`	Optional. Valid values are `lazy` and `eager`. If present, it supersedes the global configuration parameter `lazyLoadingEnabled` for this mapping.

For example:

```
<resultMap id="blogResult" type="Blog">
  <association property="author" column="author_id" javaType="Author" select="selec
</resultMap>

<select id="selectBlog" resultMap="blogResult">
  SELECT * FROM BLOG WHERE ID = #{id}
</select>

<select id="selectAuthor" resultType="Author">
  SELECT * FROM AUTHOR WHERE ID = #{id}
</select>
```

That's it. We have two select statements: one to load the Blog, the other to load the Author, and the Blog's resultMap describes that the `selectAuthor` statement should be used to load its author property.

All other properties will be loaded automatically assuming their column and property names match.

While this approach is simple, it will not perform well for large data sets or lists. This problem is known as the "N+1 Selects Problem". In a nutshell, the N+1 selects problem is caused like this:

- You execute a single SQL statement to retrieve a list of records (the "+1").
- For each record returned, you execute a select statement to load details for each (the "N").

This problem could result in hundreds or thousands of SQL statements to be executed. This is not always desirable.

The upside is that MyBatis can lazy load such queries, thus you might be spared the cost of these statements all at once. However, if you load such a list and then immediately iterate through it to access the nested data, you will invoke all of the lazy loads, and thus performance could be very bad.

And so, there is another way.

4.1.5.8 Nested Results for Association

Attribute	Description
`resultMap`	This is the ID of a ResultMap that can map the nested results of this association into an appropriate object graph. This is an alternative to using a call to another select statement. It allows you to join multiple tables together into a single ResultSet. Such a ResultSet will contain duplicated, repeating groups of data that needs to be decomposed and mapped properly to a nested object graph. To facilitate this, MyBatis lets you "chain" result maps together, to deal with the nested results. An example will be far easier to follow, and one follows this table.
`columnPrefix`	When joining multiple tables, you would have to use column alias to avoid duplicated column names in the ResultSet. Specifying columnPrefix allows you to map such columns to an external resultMap. Please see the example explained later in this section.

`notNullColumn`	By default a child object is created only if at least one of the columns mapped to the child's properties is non null. With this attribute you can change this behaviour by specifiying which columns must have a value so MyBatis will create a child object only if any of those columns is not null. Multiple column names can be specified using a comma as a separator. Default value: unset.
`autoMapping`	If present, MyBatis will enable or disable automapping when mapping the result to this property. This attribute overrides the global autoMappingBehavior. Note that it has no effect on an external resultMap, so it is pointless to use it with `select` or `resultMap` attribute. Default value: unset.

You've already seen a very complicated example of nested associations above. The following is a far simpler example to demonstrate how this works. Instead of executing a separate statement, we'll join the Blog and Author tables together, like so:

```
<select id="selectBlog" resultMap="blogResult">
  select
    B.id            as blog_id,
    B.title         as blog_title,
    B.author_id     as blog_author_id,
    A.id            as author_id,
    A.username      as author_username,
    A.password      as author_password,
    A.email         as author_email,
    A.bio           as author_bio
  from Blog B left outer join Author A on B.author_id = A.id
  where B.id = #{id}
</select>
```

Notice the join, as well as the care taken to ensure that all results are aliased with a unique and clear name. This makes mapping far easier. Now we can map the results:

```
<resultMap id="blogResult" type="Blog">
  <id property="id" column="blog_id" />
  <result property="title" column="blog_title"/>
  <association property="author" resultMap="authorResult" />
</resultMap>

<resultMap id="authorResult" type="Author">
  <id property="id" column="author_id"/>
  <result property="username" column="author_username"/>
  <result property="password" column="author_password"/>
  <result property="email" column="author_email"/>
  <result property="bio" column="author_bio"/>
</resultMap>
```

In the example above you can see at the Blog's "author" association delegates to the "authorResult" resultMap to load the Author instance.

Very Important: id elements play a very important role in Nested Result mapping. You should always specify one or more properties that can be used to uniquely identify the results. The truth is that MyBatis will still work if you leave it out, but at a severe performance cost. Choose as few properties as possible that can uniquely identify the result. The primary key is an obvious choice (even if composite).

Now, the above example used an external resultMap element to map the association. This makes the Author resultMap reusable. However, if you have no need to reuse it, or if you simply prefer to co-locate your result mappings into a single descriptive resultMap, you can nest the association result mappings. Here's the same example using this approach:

```
<resultMap id="blogResult" type="Blog">
  <id property="id" column="blog_id" />
  <result property="title" column="blog_title"/>
  <association property="author" javaType="Author">
    <id property="id" column="author_id"/>
    <result property="username" column="author_username"/>
    <result property="password" column="author_password"/>
    <result property="email" column="author_email"/>
    <result property="bio" column="author_bio"/>
  </association>
</resultMap>
```

What if the blog has a co-author? The select statement would look like:

```
<select id="selectBlog" resultMap="blogResult">
  select
    B.id            as blog_id,
    B.title         as blog_title,
    A.id            as author_id,
    A.username      as author_username,
    A.password      as author_password,
    A.email         as author_email,
    A.bio           as author_bio,
    CA.id           as co_author_id,
    CA.username     as co_author_username,
    CA.password     as co_author_password,
    CA.email        as co_author_email,
    CA.bio          as co_author_bio
  from Blog B
  left outer join Author A on B.author_id = A.id
  left outer join Author CA on B.co_author_id = CA.id
  where B.id = #{id}
</select>
```

Recall that the resultMap for Author is defined as follows.

```
<resultMap id="authorResult" type="Author">
  <id property="id" column="author_id"/>
  <result property="username" column="author_username"/>
  <result property="password" column="author_password"/>
  <result property="email" column="author_email"/>
  <result property="bio" column="author_bio"/>
</resultMap>
```

Because the column names in the results differ from the columns defined in the resultMap, you need to specify `columnPrefix` to reuse the resultMap for mapping co-author results.

```
<resultMap id="blogResult" type="Blog">
  <id property="id" column="blog_id" />
  <result property="title" column="blog_title"/>
  <association property="author"
    resultMap="authorResult" />
  <association property="coAuthor"
    resultMap="authorResult"
    columnPrefix="co_" />
</resultMap>
```

4.1.5.9 Multiple ResultSets for Association

Attribute	Description
`column`	When using multiple resultset this attribute specifies the columns (separated by commas) that will be correlated with the `foreignColumn` to identify the parent and the child of a relationship.
`foreignColumn`	Identifies the name of the columns that contains the foreign keys which values will be matched against the values of the columns specified in the `column` attibute of the parent type.
`resultSet`	Identifies the name of the result set where this complex type will be loaded from.

Starting from version 3.2.3 MyBatis provides yet another way to solve the N+1 problem.

Some databases allow stored procedures to return more than one resultset or execute more than one statement at once and return a resultset per each one. This can be used to hit the database just once and return related data without using a join.

In the example, the stored procedure executes the following queries and returns two result sets. The first will contain Blogs and the second Authors.

```
SELECT * FROM BLOG WHERE ID = #{id}

SELECT * FROM AUTHOR WHERE ID = #{id}
```

A name must be given to each result set by adding a `resultSets` attribute to the mapped statement with a list of names separated by commas.

```
<select id="selectBlog" resultSets="blogs,authors" resultMap="blogResult" statement
  {call getBlogsAndAuthors(#{id,jdbcType=INTEGER,mode=IN})}
</select>
```

Now we can specify that the data to fill the "author" association comes in the "authors" result set:

```
<resultMap id="blogResult" type="Blog">
  <id property="id" column="id" />
  <result property="title" column="title"/>
  <association property="author" javaType="Author" resultSet="authors" column="auth
    <id property="id" column="id"/>
    <result property="username" column="username"/>
    <result property="password" column="password"/>
    <result property="email" column="email"/>
    <result property="bio" column="bio"/>
  </association>
</resultMap>
```

You've seen above how to deal with a "has one" type association. But what about "has many"? That's the subject of the next section.

4.1.5.10 collection

```
<collection property="posts" ofType="domain.blog.Post">
  <id property="id" column="post_id"/>
  <result property="subject" column="post_subject"/>
  <result property="body" column="post_body"/>
</collection>
```

The collection element works almost identically to the association. In fact, it's so similar, to document the similarities would be redundant. So let's focus on the differences.

To continue with our example above, a Blog only had one Author. But a Blog has many Posts. On the blog class, this would be represented by something like:

```
private List<Post> posts;
```

To map a set of nested results to a List like this, we use the collection element. Just like the association element, we can use a nested select, or nested results from a join.

4.1.5.11 Nested Select for Collection

First, let's look at using a nested select to load the Posts for the Blog.

```
<resultMap id="blogResult" type="Blog">
  <collection property="posts" javaType="ArrayList" column="id" ofType="Post" selec
</resultMap>

<select id="selectBlog" resultMap="blogResult">
  SELECT * FROM BLOG WHERE ID = #{id}
</select>

<select id="selectPostsForBlog" resultType="Post">
  SELECT * FROM POST WHERE BLOG_ID = #{id}
</select>
```

There are a number things you'll notice immediately, but for the most part it looks very similar to the association element we learned about above. First, you'll notice that we're using the collection element. Then you'll notice that there's a new "ofType" attribute. This attribute is necessary to

distinguish between the JavaBean (or field) property type and the type that the collection contains. So you could read the following mapping like this:

```
<collection property="posts" javaType="ArrayList" column="id" ofType="Post" select=
```

Read as: "A collection of posts in an ArrayList of type Post."

The `javaType` attribute is really unnecessary, as MyBatis will figure this out for you in most cases. So you can often shorten this down to simply:

```
<collection property="posts" column="id" ofType="Post" select="selectPostsForBlog"/
```

4.1.5.12 Nested Results for Collection

By this point, you can probably guess how nested results for a collection will work, because it's exactly the same as an association, but with the same addition of the `ofType` attribute applied.

First, let's look at the SQL:

```
<select id="selectBlog" resultMap="blogResult">
  select
  B.id as blog_id,
  B.title as blog_title,
  B.author_id as blog_author_id,
  P.id as post_id,
  P.subject as post_subject,
  P.body as post_body,
  from Blog B
  left outer join Post P on B.id = P.blog_id
  where B.id = #{id}
</select>
```

Again, we've joined the Blog and Post tables, and have taken care to ensure quality result column labels for simple mapping. Now mapping a Blog with its collection of Post mappings is as simple as:

```
<resultMap id="blogResult" type="Blog">
  <id property="id" column="blog_id" />
  <result property="title" column="blog_title"/>
  <collection property="posts" ofType="Post">
    <id property="id" column="post_id"/>
    <result property="subject" column="post_subject"/>
    <result property="body" column="post_body"/>
  </collection>
</resultMap>
```

Again, remember the importance of the id elements here, or read the association section above if you haven't already.

Also, if you prefer the longer form that allows for more reusability of your result maps, you can use the following alternative mapping:

```
<resultMap id="blogResult" type="Blog">
  <id property="id" column="blog_id" />
  <result property="title" column="blog_title"/>
  <collection property="posts" ofType="Post" resultMap="blogPostResult" columnPrefi
</resultMap>

<resultMap id="blogPostResult" type="Post">
  <id property="id" column="id"/>
  <result property="subject" column="subject"/>
  <result property="body" column="body"/>
</resultMap>
```

4.1.5.13 Multiple ResultSets for Collection

As we did for the association, we can call a stored procedure that executes two queries and returns two result sets, one with Blogs and another with Posts:

```
SELECT * FROM BLOG WHERE ID = #{id}

SELECT * FROM POST WHERE BLOG_ID = #{id}
```

A name must be given to each result set by adding a `resultSets` attribute to the mapped statement with a list of names separated by commas.

```
<select id="selectBlog" resultSets="blogs,posts" resultMap="blogResult">
  {call getBlogsAndPosts(#{id,jdbcType=INTEGER,mode=IN})}
</select>
```

We specify that the "posts" collection will be filled out of data contained in the result set named "posts":

```
<resultMap id="blogResult" type="Blog">
  <id property="id" column="id" />
  <result property="title" column="title"/>
  <collection property="posts" ofType="Post" resultSet="posts" column="id" foreignC
    <id property="id" column="id"/>
    <result property="subject" column="subject"/>
    <result property="body" column="body"/>
  </collection>
</resultMap>
```

NOTE There's no limit to the depth, breadth or combinations of the associations and collections that you map. You should keep performance in mind when mapping them. Unit testing and performance testing of your application goes a long way toward discovering the best approach for your application. The nice thing is that MyBatis lets you change your mind later, with very little (if any) impact to your code.

Advanced association and collection mapping is a deep subject. Documentation can only get you so far. With a little practice, it will all become clear very quickly.

4.1.5.14 discriminator

```
<discriminator javaType="int" column="draft">
  <case value="1" resultType="DraftPost"/>
</discriminator>
```

Sometimes a single database query might return result sets of many different (but hopefully somewhat related) data types. The discriminator element was designed to deal with this situation, and others, including class inheritance hierarchies. The discriminator is pretty simple to understand, as it behaves much like a switch statement in Java.

A discriminator definition specifies column and javaType attributes. The column is where MyBatis will look for the value to compare. The javaType is required to ensure the proper kind of equality test is performed (although String would probably work for almost any situation). For example:

```
<resultMap id="vehicleResult" type="Vehicle">
  <id property="id" column="id" />
  <result property="vin" column="vin"/>
  <result property="year" column="year"/>
  <result property="make" column="make"/>
  <result property="model" column="model"/>
  <result property="color" column="color"/>
  <discriminator javaType="int" column="vehicle_type">
    <case value="1" resultMap="carResult"/>
    <case value="2" resultMap="truckResult"/>
    <case value="3" resultMap="vanResult"/>
    <case value="4" resultMap="suvResult"/>
  </discriminator>
</resultMap>
```

In this example, MyBatis would retrieve each record from the result set and compare its vehicle type value. If it matches any of the discriminator cases, then it will use the `resultMap` specified by the case. This is done exclusively, so in other words, the rest of the resultMap is ignored (unless it is extended, which we talk about in a second). If none of the cases match, then MyBatis simply uses the resultMap as defined outside of the discriminator block. So, if the carResult was declared as follows:

```
<resultMap id="carResult" type="Car">
  <result property="doorCount" column="door_count" />
</resultMap>
```

Then ONLY the doorCount property would be loaded. This is done to allow completely independent groups of discriminator cases, even ones that have no relationship to the parent resultMap. In this case we do of course know that there's a relationship between cars and vehicles, as a Car is-a Vehicle. Therefore, we want the rest of the properties loaded too. One simple change to the resultMap and we're set to go.

```
<resultMap id="carResult" type="Car" extends="vehicleResult">
  <result property="doorCount" column="door_count" />
</resultMap>
```

Now all of the properties from both the vehicleResult and carResult will be loaded.

Once again though, some may find this external definition of maps somewhat tedious. Therefore there's an alternative syntax for those that prefer a more concise mapping style. For example:

```
<resultMap id="vehicleResult" type="Vehicle">
  <id property="id" column="id" />
  <result property="vin" column="vin"/>
  <result property="year" column="year"/>
  <result property="make" column="make"/>
  <result property="model" column="model"/>
  <result property="color" column="color"/>
  <discriminator javaType="int" column="vehicle_type">
    <case value="1" resultType="carResult">
      <result property="doorCount" column="door_count" />
    </case>
    <case value="2" resultType="truckResult">
      <result property="boxSize" column="box_size" />
      <result property="extendedCab" column="extended_cab" />
    </case>
    <case value="3" resultType="vanResult">
      <result property="powerSlidingDoor" column="power_sliding_door" />
    </case>
    <case value="4" resultType="suvResult">
      <result property="allWheelDrive" column="all_wheel_drive" />
    </case>
  </discriminator>
</resultMap>
```

NOTE Remember that these are all Result Maps, and if you don't specify any results at all, then MyBatis will automatically match up columns and properties for you. So most of these examples are more verbose than they really need to be. That said, most databases are kind of complex and it's unlikely that we'll be able to depend on that for all cases.

4.1.6 Auto-mapping

As you have already seen in the previous sections, in simple cases MyBatis can auto-map the results for you and in others you will need to build a result map. But as you will see in this section you can also mix both strategies. Let's have a deeper look at how auto-mapping works.

When auto-mapping results MyBatis will get the column name and look for a property with the same name ignoring case. That means that if a column named *ID* and property named *id* are found, MyBatis will set the *id* property with the *ID* column value.

Usually database columns are named using uppercase letters and underscores between words and java properties often follow the camelcase naming covention. To enable the auto-mapping between them set the setting `mapUnderscoreToCamelCase` to true.

Auto-mapping works even when there is an specific result map. When this happens, for each result map, all columns that are present in the ResultSet that have not a manual mapping will be auto-mapped, then manual mappings will be processed. In the following sample *id* and *userName* columns will be auto-mapped and *hashed_password* column will be mapped.

```
<select id="selectUsers" resultMap="userResultMap">
  select
    user_id             as "id",
    user_name           as "userName",
    hashed_password
  from some_table
  where id = #{id}
</select>
```

```
<resultMap id="userResultMap" type="User">
  <result property="password" column="hashed_password"/>
</resultMap>
```

There are three auto-mapping levels:

- `NONE` - disables auto-mapping. Only manually mapped properties will be set.
- `PARTIAL` - will auto-map results except those that have nested result mappings defined inside (joins).
- `FULL` - auto-maps everything.

The default value is `PARTIAL`, and it is so for a reason. When `FULL` is used auto-mapping will be performed when processing join results and joins retrieve data of several different entities in the same row hence this may result in undesired mappings. To understand the risk have a look at the following sample:

```
<select id="selectBlog" resultMap="blogResult">
  select
    B.id,
    B.title,
    A.username,
  from Blog B left outer join Author A on B.author_id = A.id
  where B.id = #{id}
</select>
```

```
<resultMap id="blogResult" type="Blog">
  <association property="author" resultMap="authorResult"/>
</resultMap>

<resultMap id="authorResult" type="Author">
  <result property="username" column="author_username"/>
</resultMap>
```

With this result map both *Blog* and *Author* will be auto-mapped. But note that *Author* has an *id* property and there is a column named *id* in the ResultSet so Author's id will be filled with Blog's id, and that is not what you were expecting. So use the `FULL` option with caution.

Regardless of the auto-mapping level configured you can enable or disable the automapping for an specific ResultMap by adding the attribute `autoMapping` to it:

```
<resultMap id="userResultMap" type="User" autoMapping="false">
  <result property="password" column="hashed_password"/>
</resultMap>
```

4.1.7 cache

MyBatis includes a powerful transactional query caching feature which is very configurable and customizable. A lot of changes have been made in the MyBatis 3 cache implementation to make it both more powerful and far easier to configure.

By default, just local session caching is enabled that is used solely to cache data for the duration of a session. To enable a global second level of caching you simply need to add one line to your SQL Mapping file:

```
<cache/>
```

Literally that's it. The effect of this one simple statement is as follows:

- All results from select statements in the mapped statement file will be cached.
- All insert, update and delete statements in the mapped statement file will flush the cache.
- The cache will use a Least Recently Used (LRU) algorithm for eviction.
- The cache will not flush on any sort of time based schedule (i.e. no Flush Interval).
- The cache will store 1024 references to lists or objects (whatever the query method returns).
- The cache will be treated as a read/write cache, meaning objects retrieved are not shared and can be safely modified by the caller, without interfering with other potential modifications by other callers or threads.

NOTE The cache will only apply to statements declared in the mapping file where the cache tag is located. If you are using the Java API in conjunction with the XML mapping files, then statements declared in the companion interface will not be cached by default. You will need to refer to the cache region using the @CacheNamespaceRef annotation.

All of these properties are modifiable through the attributes of the cache element. For example:

```
<cache
  eviction="FIFO"
  flushInterval="60000"
  size="512"
  readOnly="true"/>
```

This more advanced configuration creates a FIFO cache that flushes once every 60 seconds, stores up to 512 references to result objects or lists, and objects returned are considered read-only, thus modifying them could cause conflicts between callers in different threads.

The available eviction policies available are:

- `LRU` – Least Recently Used: Removes objects that haven't been used for the longst period of time.
- `FIFO` – First In First Out: Removes objects in the order that they entered the cache.
- `SOFT` – Soft Reference: Removes objects based on the garbage collector state and the rules of Soft References.
- `WEAK` – Weak Reference: More aggressively removes objects based on the garbage collector state and rules of Weak References.

The default is LRU.

The flushInterval can be set to any positive integer and should represent a reasonable amount of time specified in milliseconds. The default is not set, thus no flush interval is used and the cache is only flushed by calls to statements.

The size can be set to any positive integer, keep in mind the size of the objects your caching and the available memory resources of your environment. The default is 1024.

The readOnly attribute can be set to true or false. A read-only cache will return the same instance of the cached object to all callers. Thus such objects should not be modified. This offers a significant performance advantage though. A read-write cache will return a copy (via serialization) of the cached object. This is slower, but safer, and thus the default is false.

NOTE Second level cache is transactional. That means that it is updated when a SqlSession finishes with commit or when it finishes with rollback but no inserts/deletes/updates with flushCache=true where executed.

4.1.7.1 Using a Custom Cache

In addition to customizing the cache in these ways, you can also completely override the cache behavior by implementing your own cache, or creating an adapter to other 3rd party caching solutions.

```
<cache type="com.domain.something.MyCustomCache"/>
```

This example demonstrates how to use a custom cache implementation. The class specified in the type attribute must implement the org.apache.ibatis.cache.Cache interface and provide a constructor that gets an String id as an argument. This interface is one of the more complex in the MyBatis framework, but simple given what it does.

```
public interface Cache {
  String getId();
  int getSize();
  void putObject(Object key, Object value);
  Object getObject(Object key);
  boolean hasKey(Object key);
  Object removeObject(Object key);
  void clear();
}
```

To configure your cache, simply add public JavaBeans properties to your Cache implementation, and pass properties via the cache Element, for example, the following would call a method called `setCacheFile(String file)` on your Cache implementation:

```
<cache type="com.domain.something.MyCustomCache">
  <property name="cacheFile" value="/tmp/my-custom-cache.tmp"/>
</cache>
```

You can use JavaBeans properties of all simple types, MyBatis will do the conversion. And you can specify a placeholder(e.g. `${cache.file}`) to replace value defined at configuration properties.

Since 3.4.2, the MyBatis has been supported to call an initialization method after it's set all properties. If you want to use this feature, please implements the `org.apache.ibatis.builder.InitializingObject` interface on your custom cache class.

```
public interface InitializingObject {
  void initialize() throws Exception;
}
```

NOTE Settings of cache (like eviction strategy, read write..etc.) in section above are not applied when using Custom Cache.

It's important to remember that a cache configuration and the cache instance are bound to the namespace of the SQL Map file. Thus, all statements in the same namespace as the cache are bound by it. Statements can modify how they interact with the cache, or exclude themselves completely by

using two simple attributes on a statement-by-statement basis. By default, statements are configured like this:

```
<select ... flushCache="false" useCache="true"/>
<insert ... flushCache="true"/>
<update ... flushCache="true"/>
<delete ... flushCache="true"/>
```

Since that's the default, you obviously should never explicitly configure a statement that way. Instead, only set the flushCache and useCache attributes if you want to change the default behavior. For example, in some cases you may want to exclude the results of a particular select statement from the cache, or you might want a select statement to flush the cache. Similarly, you may have some update statements that don't need to flush the cache upon execution.

4.1.7.2 cache-ref

Recall from the previous section that only the cache for this particular namespace will be used or flushed for statements within the same namespace. There may come a time when you want to share the same cache configuration and instance between namespaces. In such cases you can reference another cache by using the cache-ref element.

```
<cache-ref namespace="com.someone.application.data.SomeMapper"/>
```

5 Dynamic SQL

5.1 Dynamic SQL

One of the most powerful features of MyBatis has always been its Dynamic SQL capabilities. If you have any experience with JDBC or any similar framework, you understand how painful it is to conditionally concatenate strings of SQL together, making sure not to forget spaces or to omit a comma at the end of a list of columns. Dynamic SQL can be downright painful to deal with.

While working with Dynamic SQL will never be a party, MyBatis certainly improves the situation with a powerful Dynamic SQL language that can be used within any mapped SQL statement.

The Dynamic SQL elements should be familiar to anyone who has used JSTL or any similar XML based text processors. In previous versions of MyBatis, there were a lot of elements to know and understand. MyBatis 3 greatly improves upon this, and now there are less than half of those elements to work with. MyBatis employs powerful OGNL based expressions to eliminate most of the other elements:

- if
- choose (when, otherwise)
- trim (where, set)
- foreach

5.1.1 if

The most common thing to do in dynamic SQL is conditionally include a part of a where clause. For example:

```
<select id="findActiveBlogWithTitleLike"
     resultType="Blog">
  SELECT * FROM BLOG
  WHERE state = ‘ACTIVE’
  <if test="title != null">
    AND title like #{title}
  </if>
</select>
```

This statement would provide an optional text search type of functionality. If you passed in no title, then all active Blogs would be returned. But if you do pass in a title, it will look for a title like that (for the keen eyed, yes in this case your parameter value would need to include any masking or wildcard characters).

What if we wanted to optionally search by title and author? First, I’d change the name of the statement to make more sense. Then simply add another condition.

```
<select id="findActiveBlogLike"
     resultType="Blog">
  SELECT * FROM BLOG WHERE state = ‘ACTIVE’
  <if test="title != null">
    AND title like #{title}
  </if>
  <if test="author != null and author.name != null">
    AND author_name like #{author.name}
  </if>
</select>
```

5.1.2 choose, when, otherwise

Sometimes we don’t want all of the conditionals to apply, instead we want to choose only one case among many options. Similar to a switch statement in Java, MyBatis offers a choose element.

Let’s use the example above, but now let’s search only on title if one is provided, then only by author if one is provided. If neither is provided, let’s only return featured blogs (perhaps a strategically list selected by administrators, instead of returning a huge meaningless list of random blogs).

```
<select id="findActiveBlogLike"
     resultType="Blog">
  SELECT * FROM BLOG WHERE state = ‘ACTIVE’
  <choose>
    <when test="title != null">
      AND title like #{title}
    </when>
    <when test="author != null and author.name != null">
      AND author_name like #{author.name}
    </when>
    <otherwise>
      AND featured = 1
    </otherwise>
  </choose>
</select>
```

5.1.3 trim, where, set

The previous examples have been conveniently dancing around a notorious dynamic SQL challenge. Consider what would happen if we return to our "if" example, but this time we make "ACTIVE = 1" a dynamic condition as well.

```
<select id="findActiveBlogLike"
     resultType="Blog">
  SELECT * FROM BLOG
  WHERE
  <if test="state != null">
    state = #{state}
  </if>
  <if test="title != null">
    AND title like #{title}
  </if>
  <if test="author != null and author.name != null">
    AND author_name like #{author.name}
  </if>
</select>
```

What happens if none of the conditions are met? You would end up with SQL that looked like this:

```
SELECT * FROM BLOG
WHERE
```

This would fail. What if only the second condition was met? You would end up with SQL that looked like this:

```
SELECT * FROM BLOG
WHERE
AND title like ‘someTitle’
```

This would also fail. This problem is not easily solved with conditionals, and if you’ve ever had to write it, then you likely never want to do so again.

MyBatis has a simple answer that will likely work in 90% of the cases. And in cases where it doesn’t, you can customize it so that it does. With one simple change, everything works fine:

```
<select id="findActiveBlogLike"
     resultType="Blog">
  SELECT * FROM BLOG
  <where>
    <if test="state != null">
         state = #{state}
    </if>
    <if test="title != null">
        AND title like #{title}
    </if>
    <if test="author != null and author.name != null">
        AND author_name like #{author.name}
    </if>
  </where>
</select>
```

The *where* element knows to only insert "WHERE" if there is any content returned by the containing tags. Furthermore, if that content begins with "AND" or "OR", it knows to strip it off.

If the *where* element does not behave exactly as you like, you can customize it by defining your own trim element. For example, the trim equivalent to the *where* element is:

```
<trim prefix="WHERE" prefixOverrides="AND |OR ">
  ...
</trim>
```

The *prefixOverrides* attribute takes a pipe delimited list of text to override, where whitespace is relevant. The result is the removal of anything specified in the *prefixOverrides* attribute, and the insertion of anything in the *prefix* attribute.

There is a similar solution for dynamic update statements called *set*. The *set* element can be used to dynamically include columns to update, and leave out others. For example:

```
<update id="updateAuthorIfNecessary">
  update Author
    <set>
      <if test="username != null">username=#{username},</if>
      <if test="password != null">password=#{password},</if>
      <if test="email != null">email=#{email},</if>
      <if test="bio != null">bio=#{bio}</if>
    </set>
  where id=#{id}
</update>
```

Here, the *set* element will dynamically prepend the SET keyword, and also eliminate any extraneous commas that might trail the value assignments after the conditions are applied.

If you're curious about what the equivalent *trim* element would look like, here it is:

```
<trim prefix="SET" suffixOverrides=",">
  ...
</trim>
```

Notice that in this case we're overriding a suffix, while we're still appending a prefix.

5.1.4 foreach

Another common necessity for dynamic SQL is the need to iterate over a collection, often to build an IN condition. For example:

```
<select id="selectPostIn" resultType="domain.blog.Post">
  SELECT *
  FROM POST P
  WHERE ID in
  <foreach item="item" index="index" collection="list"
      open="(" separator="," close=")">
        #{item}
  </foreach>
</select>
```

The *foreach* element is very powerful, and allows you to specify a collection, declare item and index variables that can be used inside the body of the element. It also allows you to specify opening and closing strings, and add a separator to place in between iterations. The element is smart in that it won't accidentally append extra separators.

NOTE You can pass any Iterable object (for example List, Set, etc.), as well as any Map or Array object to foreach as collection parameter. When using an Iterable or Array, index will be the number

of current iteration and value item will be the element retrieved in this iteration. When using a Map (or Collection of Map.Entry objects), index will be the key object and item will be the value object.

This wraps up the discussion regarding the XML configuration file and XML mapping files. The next section will discuss the Java API in detail, so that you can get the most out of the mappings that you've created.

5.1.5 bind

The `bind` element lets you create a variable out of an OGNL expression and bind it to the context. For example:

```
<select id="selectBlogsLike" resultType="Blog">
  <bind name="pattern" value="'%' + _parameter.getTitle() + '%'" />
  SELECT * FROM BLOG
  WHERE title LIKE #{pattern}
</select>
```

5.1.6 Multi-db vendor support

If a databaseIdProvider was configured a "_databaseId" variable is available for dynamic code, so you can build different statements depending on database vendor. Have a look at the following example:

```
<insert id="insert">
  <selectKey keyProperty="id" resultType="int" order="BEFORE">
    <if test="_databaseId == 'oracle'">
      select seq_users.nextval from dual
    </if>
    <if test="_databaseId == 'db2'">
      select nextval for seq_users from sysibm.sysdummy1"
    </if>
  </selectKey>
  insert into users values (#{id}, #{name})
</insert>
```

5.1.7 Pluggable Scripting Languages For Dynamic SQL

Starting from version 3.2 MyBatis supports pluggable scripting languages, so you can plug a language driver and use that language to write your dynamic SQL queries.

You can plug a language by implementing the following interface:

```
public interface LanguageDriver {
  ParameterHandler createParameterHandler(MappedStatement mappedStatement, Object p
  SqlSource createSqlSource(Configuration configuration, XNode script, Class<?> par
  SqlSource createSqlSource(Configuration configuration, String script, Class<?> pa
}
```

Once you have your custom language driver you can set it to be the default by configuring it in the mybatis-config.xml file:

```
<typeAliases>
  <typeAlias type="org.sample.MyLanguageDriver" alias="myLanguage"/>
</typeAliases>
<settings>
  <setting name="defaultScriptingLanguage" value="myLanguage"/>
</settings>
```

Instead of changing the default, you can specify the language for an specific statement by adding the `lang` attribute as follows:

```
<select id="selectBlog" lang="myLanguage">
  SELECT * FROM BLOG
</select>
```

Or, in the case you are using mappers, using the `@Lang` annotation:

```
public interface Mapper {
  @Lang(MyLanguageDriver.class)
  @Select("SELECT * FROM BLOG")
  List<Blog> selectBlog();
}
```

NOTE You can use Apache Velocity as your dynamic language. Have a look at the MyBatis-Velocity project for the details.

All the xml tags you have seen in the previous sections are provided by the default MyBatis language that is provided by the driver `org.apache.ibatis.scripting.xmltags.XmlLanguageDriver` which is aliased as `xml`.

6 Java API

6.1 Java API

Now that you know how to configure MyBatis and create mappings, you're ready for the good stuff. The MyBatis Java API is where you get to reap the rewards of your efforts. As you'll see, compared to JDBC, MyBatis greatly simplifies your code and keeps it clean, easy to understand and maintain. MyBatis 3 has introduced a number of significant improvements to make working with SQL Maps even better.

6.1.1 Directory Structure

Before we dive in to the Java API itself, it's important to understand the best practices surrounding directory structures. MyBatis is very flexible, and you can do almost anything with your files. But as with any framework, there's a preferred way.

Let's look at a typical application directory structure:

```
/my_application
  /bin
  /devlib

/lib
<-- MyBatis *.jar files go here.
  /src
    /org/myapp/
      /action

/data
<-- MyBatis artifacts go here, including, Mapper Classes, XML Configuration, XML Mappin
        /mybatis-config.xml
        /BlogMapper.java
        /BlogMapper.xml
      /model
      /service
      /view

/properties
<-- Properties included in your XML Configuration go here.
  /test
    /org/myapp/
      /action
      /data
      /model
      /service
      /view
    /properties
  /web
    /WEB-INF
      /web.xml
```

Remember, these are preferences, not requirements, but others will thank you for using a common directory structure.

The rest of the examples in this section will assume you're following this directory structure.

6.1.2 SqlSessions

The primary Java interface for working with MyBatis is the SqlSession. Through this interface you can execute commands, get mappers and manage transactions. We'll talk more about SqlSession itself shortly, but first we have to learn how to acquire an instance of SqlSession. SqlSessions are created by a SqlSessionFactory instance. The SqlSessionFactory contains methods for creating instances of SqlSessions all different ways. The SqlSessionFactory itself is created by the SqlSessionFactoryBuilder that can create the SqlSessonFactory from XML, Annotations or hand coded Java configuration.

NOTE When using MyBatis with a dependency injection framework like Spring or Guice, SqlSessions are created and injected by the DI framework so you don't need to use the SqlSessionFactoryBuilder or SqlSessionFactory and can go directly to the SqlSession section. Please refer to the MyBatis-Spring or MyBatis-Guice manuals for further info.

6.1.2.1 SqlSessionFactoryBuilder

The SqlSessionFactoryBuilder has five build() methods, each which allows you to build a SqlSession from a different source.

```
SqlSessionFactory build(InputStream inputStream)
SqlSessionFactory build(InputStream inputStream, String environment)
SqlSessionFactory build(InputStream inputStream, Properties properties)
SqlSessionFactory build(InputStream inputStream, String env, Properties props)
SqlSessionFactory build(Configuration config)
```

The first four methods are the most common, as they take an InputStream instance that refers to an XML document, or more specifically, the mybatis-config.xml file discussed above. The optional parameters are environment and properties. Environment determines which environment to load, including the datasource and transaction manager. For example:

```
<environments default="development">
  <environment id="development">
    <transactionManager type="JDBC">
        ...
    <dataSource type="POOLED">
        ...
  </environment>
  <environment id="production">
    <transactionManager type="MANAGED">
        ...
    <dataSource type="JNDI">
        ...
  </environment>
</environments>
```

If you call a build method that takes the environment parameter, then MyBatis will use the configuration for that environment. Of course, if you specify an invalid environment, you will receive

an error. If you call one of the build methods that does not take the environment parameter, then the default environment is uses (which is specified as default="development" in the example above).

If you call a method that takes a properties instance, then MyBatis will load those properties and make them available to your configuration. Those properties can be used in place of most values in the configuration using the syntax: ${propName}

Recall that properties can also be referenced from the mybatis-config.xml file, or specified directly within it. Therefore it's important to understand the priority. We mentioned it earlier in this document, but here it is again for easy reference:

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.

So to summarize, the first four methods are largely the same, but with overrides to allow you to optionally specify the environment and/or properties. Here is an example of building a SqlSessionFactory from an mybatis-config.xml file.

```
String
resource = "org/mybatis/builder/mybatis-config.xml";
InputStream
inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactoryBuilder
builder = new SqlSessionFactoryBuilder();
SqlSessionFactory
factory = builder.build(inputStream);
```

Notice that we're making use of the Resources utility class, which lives in the org.apache.ibatis.io package. The Resources class, as its name implies, helps you load resources from the classpath, filesystem or even a web URL. A quick look at the class source code or inspection through your IDE will reveal its fairly obvious set of useful methods. Here's a quick list:

```
URL getResourceURL(String resource)
URL getResourceURL(ClassLoader loader, String resource)
InputStream getResourceAsStream(String resource)
InputStream getResourceAsStream(ClassLoader loader, String resource)
Properties getResourceAsProperties(String resource)
Properties getResourceAsProperties(ClassLoader loader, String resource)
Reader getResourceAsReader(String resource)
Reader getResourceAsReader(ClassLoader loader, String resource)
File getResourceAsFile(String resource)
File getResourceAsFile(ClassLoader loader, String resource)
InputStream getUrlAsStream(String urlString)
Reader getUrlAsReader(String urlString)
Properties getUrlAsProperties(String urlString)
Class classForName(String className)
```

The final build method takes an instance of Configuration. The Configuration class contains everything you could possibly need to know about a SqlSessionFactory instance. The Configuration class is useful for introspecting on the configuration, including finding and manipulating SQL maps (not recommended once the application is accepting requests). The configuration class has every configuration switch that you've learned about already, only exposed as a Java API. Here's a simple example of how to manually a Configuration instance and pass it to the build() method to create a SqlSessionFactory.

```
DataSource dataSource = BaseDataTest.createBlogDataSource();
TransactionFactory transactionFactory = new JdbcTransactionFactory();

Environment environment = new Environment("development", transactionFactory, dataSo

Configuration configuration = new Configuration(environment);
configuration.setLazyLoadingEnabled(true);
configuration.setEnhancementEnabled(true);
configuration.getTypeAliasRegistry().registerAlias(Blog.class);
configuration.getTypeAliasRegistry().registerAlias(Post.class);
configuration.getTypeAliasRegistry().registerAlias(Author.class);
configuration.addMapper(BoundBlogMapper.class);
configuration.addMapper(BoundAuthorMapper.class);

SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
SqlSessionFactory factory = builder.build(configuration);
```

Now you have a SqlSessionFactory that can be used to create SqlSession instances.

6.1.2.2 SqlSessionFactory

SqlSessionFactory has six methods that are used to create SqlSession instances. In general, the decisions you'll be making when selecting one of these methods are:

- **Transaction**: Do you want to use a transaction scope for the session, or use auto-commit (usually means no transaction with most databases and/or JDBC drivers)?
- **Connection**: Do you want MyBatis to acquire a Connection from the configured DataSource for you, or do you want to provide your own?
- **Execution**: Do you want MyBatis to reuse PreparedStatements and/or batch updates (including inserts and deletes)?

The set of overloaded openSession() method signatures allow you to choose any combination of these options that makes sense.

```
SqlSession openSession()
SqlSession openSession(boolean autoCommit)
SqlSession openSession(Connection connection)
SqlSession openSession(TransactionIsolationLevel level)
SqlSession openSession(ExecutorType execType,TransactionIsolationLevel level)
SqlSession openSession(ExecutorType execType)
SqlSession openSession(ExecutorType execType, boolean autoCommit)
SqlSession openSession(ExecutorType execType, Connection connection)
Configuration getConfiguration();
```

The default openSession() method that takes no parameters will create a SqlSession with the following characteristics:

- A transaction scope will be started (i.e. NOT auto-commit).

- A `Connection` object will be acquired from the DataSource instance configured by the active environment.
- The transaction isolation level will be the default used by the driver or data source.
- No PreparedStatements will be reused, and no updates will be batched.

Most of the methods are pretty self explanatory. To enable auto-commit, pass a value of `true` to the optional `autoCommit` parameter. To provide your own connection, pass an instance of `Connection` to the `connection` parameter. Note that there's no override to set both the `Connection` and `autoCommit`, because MyBatis will use whatever setting the provided connection object is currently using. MyBatis uses a Java enumeration wrapper for transaction isolation levels, called `TransactionIsolationLevel`, but otherwise they work as expected and have the 5 levels supported by JDBC (`NONE`, `READ_UNCOMMITTED`, `READ_COMMITTED`, `REPEATABLE_READ`, `SERIALIZABLE`).

The one parameter that might be new to you is `ExecutorType`. This enumeration defines 3 values:

- `ExecutorType.SIMPLE`: This type of executor does nothing special. It creates a new PreparedStatement for each execution of a statement.
- `ExecutorType.REUSE`: This type of executor will reuse PreparedStatements.
- `ExecutorType.BATCH`: This executor will batch all update statements and demarcate them as necessary if SELECTs are executed between them, to ensure an easy-to-understand behavior.

NOTE There's one more method on the SqlSessionFactory that we didn't mention, and that is *getConfiguration()*. This method will return an instance of Configuration that you can use to introspect upon the MyBatis configuration at runtime.

NOTE If you've used a previous version of MyBatis, you'll recall that sessions, transactions and batches were all something separate. This is no longer the case. All three are neatly contained within the scope of a session. You need not deal with transactions or batches separately to get the full benefit of them.

6.1.2.3 SqlSession

As mentioned above, the SqlSession instance is the most powerful class in MyBatis. It is where you'll find all of the methods to execute statements, commit or rollback transactions and acquire mapper instances.

There are over twenty methods on the SqlSession class, so let's break them up into more digestible groupings.

6.Statement Execution Methods

These methods are used to execute SELECT, INSERT, UPDATE and DELETE statements that are defined in your SQL Mapping XML files. They are pretty self explanatory, each takes the ID of the statement and the Parameter Object, which can be a primitive (auto-boxed or wrapper), a JavaBean, a POJO or a Map.

```
<T> T selectOne(String statement, Object parameter)
<E> List<E> selectList(String statement, Object parameter)
<T> Cursor<T> selectCursor(String statement, Object parameter)
<K,V> Map<K,V> selectMap(String statement, Object parameter, String mapKey)
int insert(String statement, Object parameter)
int update(String statement, Object parameter)
int delete(String statement, Object parameter)
```

The difference between selectOne and selectList is only in that selectOne must return exactly one object or null (none). If any more than one, an exception will be thrown. If you don't' know how many objects are expected, use selectList. If you want to check for the existence of an object, you're better

off returning a count (0 or 1). The selectMap is a special case in that it is designed to convert a list of results into a Map based on one of the properties in the resulting objects. Because not all statements require a parameter, these methods are overloaded with versions that do not require the parameter object.

A Cursor offers the same results as a List, except it fetches data lazily using an Iterator.

```
try (Cursor<MyEntity> entities = session.selectCursor(statement, param)) {
   for (MyEntity entity:entities) {
      // process one entity
   }
}
```

The value returned by the insert, update and delete methods indicate the number of rows affected by the statement.

```
<T> T selectOne(String statement)
<E> List<E> selectList(String statement)
<T> Cursor<T> selectCursor(String statement)
<K,V> Map<K,V> selectMap(String statement, String mapKey)
int insert(String statement)
int update(String statement)
int delete(String statement)
```

Finally, there are three advanced versions of the select methods that allow you to restrict the range of rows to return, or provide custom result handling logic, usually for very large data sets.

```
<E> List<E> selectList (String statement, Object parameter, RowBounds rowBounds)
<T> Cursor<T> selectCursor(String statement, Object parameter, RowBounds rowBounds)
<K,V> Map<K,V> selectMap(String statement, Object parameter, String mapKey, RowBoun
void select (String statement, Object parameter, ResultHandler<T> handler)
void select (String statement, Object parameter, RowBounds rowBounds, ResultHandler
```

The RowBounds parameter causes MyBatis to skip the number of records specified, as well as limit the number of results returned to some number. The RowBounds class has a constructor to take both the offset and limit, and is otherwise immutable.

```
int offset = 100;
int limit = 25;
RowBounds rowBounds = new RowBounds(offset, limit);
```

Different drivers are able to achieve different levels of efficiency in this regard. For the best performance, use result set types of SCROLL_SENSITIVE or SCROLL_INSENSITIVE (in other words: not FORWARD_ONLY).

The ResultHandler parameter allows you to handle each row however you like. You can add it to a List, create a Map, Set, or throw each result away and instead keep only rolled up totals of calculations. You can do pretty much anything with the ResultHandler, and it's what MyBatis uses internally itself to build result set lists.

Since 3.4.6, ResultHandler passed to a CALLABLE statement is used on every REFCURSOR output parameter of the stored procedure if there is any.

The interface is very simple.

```
package org.apache.ibatis.session;
public interface ResultHandler<T> {
  void handleResult(ResultContext<? extends T> context);
}
```

The ResultContext parameter gives you access to the result object itself, a count of the number of result objects created, and a Boolean stop() method that you can use to stop MyBatis from loading any more results.

Using a ResultHandler has two limitations that you should be aware of:

- Data got from a method called with a ResultHandler will not be cached.
- When using advanced resultmaps MyBatis will probably require several rows to build an object. If a ResultHandler is used you may be given an object whose associations or collections are not yet filled.

6.Batch update statement Flush Method

There is method for flushing(executing) batch update statements that stored in a JDBC driver class at any timing. This method can be used when you use the `ExecutorType.BATCH` as `ExecutorType`.

```
List<BatchResult> flushStatements()
```

6.Transaction Control Methods

There are four methods for controlling the scope of a transaction. Of course, these have no effect if you've chosen to use auto-commit or if you're using an external transaction manager. However, if you're using the JDBC transaction manager, managed by the Connection instance, then the four methods that will come in handy are:

```
void commit()
void commit(boolean force)
void rollback()
void rollback(boolean force)
```

By default MyBatis does not actually commit unless it detects that the database has been changed by a call to insert, update or delete. If you've somehow made changes without calling these methods, then you can pass true into the commit and rollback methods to guarantee that it will be committed (note, you still can't force a session in auto-commit mode, or one that is using an external transaction manager). Most of the time you won't have to call rollback(), as MyBatis will do that for you if you don't call commit. However, if you need more fine grained control over a session where multiple commits and rollbacks are possible, you have the rollback option there to make that possible.

NOTE MyBatis-Spring and MyBatis-Guice provide declarative transaction handling. So if you are using MyBatis with Spring or Guice please refer to their specific manuals.

6.Local Cache

MyBatis uses two caches: a local cache and a second level cache.

Each time a new session is created MyBatis creates a local cache and attaches it to the session. Any query executed within the session will be stored in the local cache so further executions of the same query with the same input parameters will not hit the database. The local cache is cleared upon update, commit, rollback and close.

By default local cache data is used for the whole session duration. This cache is needed to resolve circular references and to speed up repeated nested queries, so it can never be completely disabled but you can configure the local cache to be used just for the duration of an statement execution by setting localCacheScope=STATEMENT.

Note that when the localCacheScope is set to SESSION, MyBatis returns references to the same objects which are stored in the local cache. Any modification of returned object (lists etc.) influences the local cache contents and subsequently the values which are returned from the cache in the lifetime of the session. Therefore, as best practice, do not to modify the objects returned by MyBatis.

You can clear the local cache at any time calling:

```
void clearCache()
```

6.Ensuring that SqlSession is Closed

```
void close()
```

The most important thing you must ensure is that you close any sessions that you open. The best way to ensure this is to use the following unit of work pattern:

```
SqlSession session = sqlSessionFactory.openSession();
try {
    // following 3 lines pseudocode for "doing some work"
    session.insert(...);
    session.update(...);
    session.delete(...);
    session.commit();
} finally {
    session.close();
}
```

Or, If you are using jdk 1.7+ and MyBatis 3.2+, you can use the try-with-resources statement:

```

try (SqlSession session = sqlSessionFactory.openSession()) {
    // following 3 lines pseudocode for "doing some work"
    session.insert(...);
    session.update(...);
    session.delete(...);
    session.commit();
}
```

NOTE Just like SqlSessionFactory, you can get the instance of Configuration that the SqlSession is using by calling the getConfiguration() method.

```
Configuration getConfiguration()
```

6.Using Mappers

```
<T> T getMapper(Class<T> type)
```

While the various insert, update, delete and select methods above are powerful, they are also very verbose, not type safe and not as helpful to your IDE or unit tests as they could be. We've already seen an example of using Mappers in the Getting Started section above.

Therefore, a more common way to execute mapped statements is to use Mapper classes. A mapper class is simply an interface with method definitions that match up against the SqlSession methods. The following example class demonstrates some method signatures and how they map to the SqlSession.

```
public interface AuthorMapper {
  // (Author) selectOne("selectAuthor", 5);
  Author selectAuthor(int id);
  // (List<Author>) selectList("selectAuthors")
  List<Author> selectAuthors();
  // (Map<Integer,Author>) selectMap("selectAuthors", "id")
  @MapKey("id")
  Map<Integer, Author> selectAuthors();
  // insert("insertAuthor", author)
  int insertAuthor(Author author);
  // updateAuthor("updateAuthor", author)
  int updateAuthor(Author author);
  // delete("deleteAuthor", 5)
  int deleteAuthor(int id);
}
```

In a nutshell, each Mapper method signature should match that of the SqlSession method that it's associated to, but without the String parameter ID. Instead, the method name must match the mapped statement ID.

In addition, the return type must match that of the expected result type for single results or an array or collection for multiple results or Cursor. All of the usual types are supported, including: Primitives, Maps, POJOs and JavaBeans.

NOTE Mapper interfaces do not need to implement any interface or extend any class. As long as the method signature can be used to uniquely identify a corresponding mapped statement.

NOTE Mapper interfaces can extend other interfaces. Be sure that you have the statements in the appropriate namespace when using XML binding to Mapper interfaces. Also, the only limitation is that you cannot have the same method signature in two interfaces in a hierarchy (a bad idea anyway).

You can pass multiple parameters to a mapper method. If you do, they will be named by the literal "param" followed by their position in the parameter list by default, for example: #{param1}, #{param2} etc. If you wish to change the name of the parameters (multiple only), then you can use the @Param("paramName") annotation on the parameter.

You can also pass a RowBounds instance to the method to limit query results.

6.Mapper Annotations

Since the very beginning, MyBatis has been an XML driven framework. The configuration is XML based, and the Mapped Statements are defined in XML. With MyBatis 3, there are new options available. MyBatis 3 builds on top of a comprehensive and powerful Java based Configuration API. This Configuration API is the foundation for the XML based MyBatis configuration, as well as the new Annotation based configuration. Annotations offer a simple way to implement simple mapped statements without introducing a lot of overhead.

NOTE Java Annotations are unfortunately limited in their expressiveness and flexibility. Despite a lot of time spent in investigation, design and trials, the most powerful MyBatis mappings simply cannot be built with Annotations – without getting ridiculous that is. C# Attributes (for example) do not suffer from these limitations, and thus MyBatis.NET will enjoy a much richer alternative to XML. That said, the Java Annotation based configuration is not without its benefits.

The Annotations are as follows:

Annotation	Target	XML equivalent	Description

`@CacheNamespace`	`Class`	`<cache>`	Configures the cache for the given namespace (i.e. class). Attributes: `implementation`, `eviction`, `flushInterval`, `size`, `readWrite`, `blocking`, `properties`.
`@Property`	N/A	`<property>`	Specifies the property value or placeholder(can replace by configuration properties that defined at the `mybatis-config.xml`). Attributes: `name`, `value`. (Available on MyBatis 3.4.2+)
`@CacheNamespaceRef`	`Class`	`<cacheRef>`	References the cache of another namespace to use. Note that caches declared in an XML mapper file are considered a separate namespace, even if they share the same FQCN. Attributes: `value` and `name`. If you use this annotation, you should be specified either `value` or `name` attribute. For the `value` attribute specify a java type indicating the namespace(the namespace name become a FQCN of specified java type), and for the `name` attribute(this attribute is available since 3.4.2) specify a name indicating the namespace.
`@ConstructorArgs`	`Method`	`<constructor>`	Collects a group of results to be passed to a result object constructor. Attributes: `value`, which is an array of `Arg`s.

`@Arg`	N/A	• `<arg>` • `<idArg>`	A single constructor argument that is part of a ConstructorArgs collection. Attributes: `id`, `column`, `javaType`, `jdbcType`, `typeHandler`, `select`, `resultMap`. The id attribute is a boolean value that identifies the property to be used for comparisons, similar to the `<idArg>` XML element.
`@TypeDiscriminator`	`Method`	`<discriminator>`	A group of value cases that can be used to determine the result mapping to perform. Attributes: `column`, `javaType`, `jdbcType`, `typeHandler`, `cases`. The cases attribute is an array of `Case`s.
`@Case`	N/A	`<case>`	A single case of a value and its corresponding mappings. Attributes: `value`, `type`, `results`. The results attribute is an array of Results, thus this `Case` Annotation is similar to an actual `ResultMap`, specified by the `Results` annotation below.
`@Results`	`Method`	`<resultMap>`	A list of Result mappings that contain details of how a particular result column is mapped to a property or field. Attributes: `value`, `id`. The value attribute is an array of `Result` annotations. The id attribute is the name of the result mapping.

`@Result`	N/A	• `<result>` • `<id>`	A single result mapping between a column and a property or field. Attributes: `id`, `column`, `property`, `javaType`, `jdbcType`, `typeHandler`, `one`, `many`. The id attribute is a boolean value that indicates that the property should be used for comparisons (similar to `<id>` in the XML mappings). The one attribute is for single associations, similar to `<association>`, and the many attribute is for collections, similar to `<collection>`. They are named as they are to avoid class naming conflicts.
`@One`	N/A	`<association>`	A mapping to a single property value of a complex type. Attributes: `select`, which is the fully qualified name of a mapped statement (i.e. mapper method) that can load an instance of the appropriate type, `fetchType`, which supersedes the global configuration parameter `lazyLoadingEnabled` for this mapping. NOTE You will notice that join mapping is not supported via the Annotations API. This is due to the limitation in Java Annotations that does not allow for circular references.

`@Many`	N/A	`<collection>`	A mapping to a collection property of a complex type. Attributes: `select`, which is the fully qualified name of a mapped statement (i.e. mapper method) that can load a collection of instances of the appropriate types, `fetchType`, which supersedes the global configuration parameter `lazyLoadingEnabled` for this mapping. NOTE You will notice that join mapping is not supported via the Annotations API. This is due to the limitation in Java Annotations that does not allow for circular references.
`@MapKey`	`Method`		This is used on methods which return type is a Map. It is used to convert a List of result objects as a Map based on a property of those objects. Attributes: `value`, which is a property used as the key of the map.

`@Options`	`Method`	Attributes of mapped statements.	This annotation provides access to the wide range of switches and configuration options that are normally present on the mapped statement as attributes. Rather than complicate each statement annotation, the `Options` annotation provides a consistent and clear way to access these. Attributes: `useCache=true`, `flushCache=FlushCachePolicy.DE` `resultSetType=DEFAULT`, `statementType=PREPARED`, `fetchSize=-1`, `timeout=-1`, `useGeneratedKeys=false`, `keyProperty=""`, `keyColumn=""`, `resultSets=""`. It's important to understand that with Java Annotations, there is no way to specify `null` as a value. Therefore, once you engage the `Options` annotation, your statement is subject to all of the default values. Pay attention to what the default values are to avoid unexpected behavior. Note that `keyColumn` is only required in certain databases (like Oracle and PostgreSQL). See the discussion about `keyColumn` and `keyProperty` above in the discussion of the insert statement for more information about allowable values in these attributes.

• `@Insert` • `@Update` • `@Delete` • `@Select`	`Method`	• `<insert>` • `<update>` • `<delete>` • `<select>`	Each of these annotations represents the actual SQL that is to be executed. They each take an array of strings (or a single string will do). If an array of strings is passed, they are concatenated with a single space between each to separate them. This helps avoid the "missing space" problem when building SQL in Java code. However, you're also welcome to concatenate together a single string if you like. Attributes: `value`, which is the array of Strings to form the single SQL statement.

• `@InsertProvider` Method • `@UpdateProvider` • `@DeleteProvider` • `@SelectProvider`	• `<insert>` • `<update>` • `<delete>` • `<select>`	Allows for creation of dynamic SQL. These alternative SQL annotations allow you to specify a class and a method name that will return the SQL to run at execution time (Since 3.4.6, you can specify the `CharSequence` instead of `String` as a method return type). Upon executing the mapped statement, MyBatis will instantiate the class, and execute the method, as specified by the provider. You can pass objects that passed to arguments of a mapper method, "Mapper interface type", "Mapper method" and "Database ID" via the `ProviderContext`(available since MyBatis 3.4.5 or later) as method argument. (In MyBatis 3.4 or later, it's allow multiple parameters) Attributes: `type`, `method`. The `type` attribute is a class. The `method` is the name of the method on that class (Since 3.5.1, you can omit `method` attribute, the MyBatis will resolve a target method via the `ProviderMethodResolver` interface. If not resolve by it, the MyBatis use the reserved fallback method that named `provideSql`). NOTE Following this section is a discussion about the class, which can help build dynamic SQL in a cleaner, easier to read way.

`@Param`	`Parameter`	N/A	If your mapper method takes multiple parameters, this annotation can be applied to a mapper method parameter to give each of them a name. Otherwise, multiple parameters will be named by their position prefixed with "param" (not including any `RowBounds` parameters). For example `#{param1}`, `#{param2}` etc. is the default. With `@Param("person")`, the parameter would be named `#{person}`.

`@SelectKey`	`Method`	`<selectKey>`	This annotation duplicates the `<selectKey>` functionality for methods annotated with `@Insert`, `@InsertProvider`, `@Update`, or `@UpdateProvider`. It is ignored for other methods. If you specify a `@SelectKey` annotation, then MyBatis will ignore any generated key properties set via the `@Options` annotation, or configuration properties. Attributes: `statement` an array of strings which is the SQL statement to execute, `keyProperty` which is the property of the parameter object that will be updated with the new value, `before` which must be either `true` or `false` to denote if the SQL statement should be executed before or after the insert, `resultType` which is the Java type of the `keyProperty`, and `statementType` is a type of the statement that is any one of `STATEMENT`, `PREPARED` or `CALLABLE` that is mapped to `Statement`, `PreparedStatement` and `CallableStatement` respectively. The default is `PREPARED`.

`@ResultMap`	`Method`	N/A	This annotation is used to provide the id of a `<resultMap>` element in an XML mapper to a `@Select` or `@SelectProvider` annotation. This allows annotated selects to reuse resultmaps that are defined in XML. This annotation will override any `@Results` or `@ConstructorArgs` annotation if both are specified on an annotated select.
`@ResultType`	`Method`	N/A	This annotation is used when using a result handler. In that case, the return type is void so MyBatis must have a way to determine the type of object to construct for each row. If there is an XML result map, use the @ResultMap annotation. If the result type is specified in XML on the `<select>` element, then no other annotation is necessary. In other cases, use this annotation. For example, if a @Select annotated method will use a result handler, the return type must be void and this annotation (or @ResultMap) is required. This annotation is ignored unless the method return type is void.
`@Flush`	`Method`	N/A	If this annotation is used, it can be called the `SqlSession#flushStatements()` via method defined at a Mapper interface.(MyBatis 3.3 or above)

6.Mapper Annotation Examples

This example shows using the @SelectKey annotation to retrieve a value from a sequence before an insert:

```
@Insert("insert into table3 (id, name) values(#{nameId}, #{name})")
@SelectKey(statement="call next value for TestSequence", keyProperty="nameId", befo
true, resultType=
int.class)

int insertTable3(Name name);
```

This example shows using the @SelectKey annotation to retrieve an identity value after an insert:

```
@Insert("insert into table2 (name) values(#{name})")
@SelectKey(statement="call identity()", keyProperty="nameId", before=
false, resultType=
int.class)

int insertTable2(Name name);
```

This example shows using the `@Flush` annotation to call the `SqlSession#flushStatements()`:

```
@Flush
List<BatchResult> flush();
```

These examples show how to name a ResultMap by specifying id attribute of @Results annotation.

```
@Results(id = "userResult", value = {
  @Result(property = "id", column = "uid", id =
true),
  @Result(property = "firstName", column = "first_name"),
  @Result(property = "lastName", column = "last_name")
})
@Select("select * from users where id = #{id}")
User getUserById(Integer id);

@Results(id = "companyResults")
@ConstructorArgs({
  @Arg(property = "id", column = "cid", id =
true),
  @Arg(property = "name", column = "name")
})
@Select("select * from company where id = #{id}")
Company getCompanyById(Integer id);
```

This example shows solo parameter using the SelectProvider annotation:

```
@SelectProvider(type = UserSqlBuilder.class, method = "buildGetUsersByName")
List<User> getUsersByName(String name);

class UserSqlBuilder {
  public static String buildGetUsersByName(final String name) {
    return new SQL(){{
      SELECT("*");
      FROM("users");
      if (name != null) {
        WHERE("name like #{value} || '%'");
      }
      ORDER_BY("id");
    }}.toString();
  }
}
```

This example shows multiple parameters using the Sql Provider annotation:

```
@SelectProvider(type = UserSqlBuilder.class, method = "buildGetUsersByName")
List<User> getUsersByName(
    @Param("name") String name, @Param("orderByColumn") String orderByColumn);

class UserSqlBuilder {

  // If not use @Param, you should be define same arguments with mapper method
  public static String buildGetUsersByName(
      final String name, final String orderByColumn) {
    return new SQL(){{
      SELECT("*");
      FROM("users");
      WHERE("name like #{name} || '%'");
      ORDER_BY(orderByColumn);
    }}.toString();
  }

  // If use @Param, you can define only arguments to be used
  public static String buildGetUsersByName(@Param("orderByColumn") final String ord
    return new SQL(){{
      SELECT("*");
      FROM("users");
      WHERE("name like #{name} || '%'");
      ORDER_BY(orderByColumn);
    }}.toString();
  }
}
```

This example shows usage the default implementation of `ProviderMethodResolver`(available since MyBatis 3.5.1 or later):

```
@SelectProvider(type = UserSqlProvider.class)
List<User> getUsersByName(String name);

// Implements the ProviderMethodResolver on your provider class
class UserSqlProvider implements ProviderMethodResolver {
  // In default implementation, it will resolve a method that method name is matche
  public static String getUsersByName(final String name) {
    return new SQL(){{
      SELECT("*");
      FROM("users");
      if (name != null) {
        WHERE("name like #{value} || '%'");
      }
      ORDER_BY("id");
    }}.toString();
  }
}
```

7 Statement Builders

7.1 The SQL Builder Class

7.1.1 The Problem

One of the nastiest things a Java developer will ever have to do is embed SQL in Java code. Usually this is done because the SQL has to be dynamically generated - otherwise you could externalize it in a file or a stored proc. As you've already seen, MyBatis has a powerful answer for dynamic SQL generation in its XML mapping features. However, sometimes it becomes necessary to build a SQL statement string inside of Java code. In that case, MyBatis has one more feature to help you out, before reducing yourself to the typical mess of plus signs, quotes, newlines, formatting problems and nested conditionals to deal with extra commas or AND conjunctions. Indeed, dynamically generating SQL code in Java can be a real nightmare. For example:

```
String sql = "SELECT P.ID, P.USERNAME, P.PASSWORD, P.FULL_NAME, "
"P.LAST_NAME,P.CREATED_ON, P.UPDATED_ON " +
"FROM PERSON P, ACCOUNT A " +
"INNER JOIN DEPARTMENT D on D.ID = P.DEPARTMENT_ID " +
"INNER JOIN COMPANY C on D.COMPANY_ID = C.ID " +
"WHERE (P.ID = A.ID AND P.FIRST_NAME like ?) " +
"OR (P.LAST_NAME like ?) " +
"GROUP BY P.ID " +
"HAVING (P.LAST_NAME like ?) " +
"OR (P.FIRST_NAME like ?) " +
"ORDER BY P.ID, P.FULL_NAME";
```

7.1.2 The Solution

MyBatis 3 offers a convenient utility class to help with the problem. With the SQL class, you simply create an instance that lets you call methods against it to build a SQL statement one step at a time. The example problem above would look like this when rewritten with the SQL class:

```
private String selectPersonSql() {
  return new SQL() {{
    SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FULL_NAME");
    SELECT("P.LAST_NAME, P.CREATED_ON, P.UPDATED_ON");
    FROM("PERSON P");
    FROM("ACCOUNT A");
    INNER_JOIN("DEPARTMENT D on D.ID = P.DEPARTMENT_ID");
    INNER_JOIN("COMPANY C on D.COMPANY_ID = C.ID");
    WHERE("P.ID = A.ID");
    WHERE("P.FIRST_NAME like ?");
    OR();
    WHERE("P.LAST_NAME like ?");
    GROUP_BY("P.ID");
    HAVING("P.LAST_NAME like ?");
    OR();
    HAVING("P.FIRST_NAME like ?");
    ORDER_BY("P.ID");
    ORDER_BY("P.FULL_NAME");
  }}.toString();
}
```

What is so special about that example? Well, if you look closely, it doesn't have to worry about accidentally duplicating "AND" keywords, or choosing between "WHERE" and "AND" or none at all. The SQL class takes care of understanding where "WHERE" needs to go, where an "AND" should be used and all of the String concatenation.

7.1.3 The SQL Class

Here are some examples:

```
// Anonymous inner class
public String deletePersonSql() {
  return new SQL() {{
    DELETE_FROM("PERSON");
    WHERE("ID = #{id}");
  }}.toString();
}

// Builder / Fluent style
public String insertPersonSql() {
  String sql = new SQL()
    .INSERT_INTO("PERSON")
    .VALUES("ID, FIRST_NAME", "#{id}, #{firstName}")
    .VALUES("LAST_NAME", "#{lastName}")
    .toString();
  return sql;
}

// With conditionals (note the final parameters, required for the anonymous inner c
public String selectPersonLike(final String id, final String firstName, final Strin
  return new SQL() {{
    SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FIRST_NAME, P.LAST_NAME");
    FROM("PERSON P");
    if (id != null) {
      WHERE("P.ID like #{id}");
    }
    if (firstName != null) {
      WHERE("P.FIRST_NAME like #{firstName}");
    }
    if (lastName != null) {
      WHERE("P.LAST_NAME like #{lastName}");
    }
    ORDER_BY("P.LAST_NAME");
  }}.toString();
}

public String deletePersonSql() {
  return new SQL() {{
    DELETE_FROM("PERSON");
    WHERE("ID = #{id}");
  }}.toString();
}

public String insertPersonSql() {
  return new SQL() {{
    INSERT_INTO("PERSON");
    VALUES("ID, FIRST_NAME", "#{id}, #{firstName}");
    VALUES("LAST_NAME", "#{lastName}");
  }}.toString();
}

public String updatePersonSql() {
  return new SQL() {{
    UPDATE("PERSON");
    SET("FIRST_NAME = #{firstName}");
    WHERE("ID = #{id}");
  }}.toString();
}
```

Method	Description
• SELECT(String) • SELECT(String...)	Starts or appends to a SELECT clause. Can be called more than once, and parameters will be appended to the SELECT clause. The parameters are usually a comma separated list of columns and aliases, but can be anything acceptable to the driver.
• SELECT_DISTINCT(String) • SELECT_DISTINCT(String...)	Starts or appends to a SELECT clause, also adds the DISTINCT keyword to the generated query. Can be called more than once, and parameters will be appended to the SELECT clause. The parameters are usually a comma separated list of columns and aliases, but can be anything acceptable to the driver.
• FROM(String) • FROM(String...)	Starts or appends to a FROM clause. Can be called more than once, and parameters will be appended to the FROM clause. Parameters are usually a table name and an alias, or anything acceptable to the driver.
• JOIN(String) • JOIN(String...) • INNER_JOIN(String) • INNER_JOIN(String...) • LEFT_OUTER_JOIN(String) • LEFT_OUTER_JOIN(String...) • RIGHT_OUTER_JOIN(String) • RIGHT_OUTER_JOIN(String...)	Adds a new JOIN clause of the appropriate type, depending on the method called. The parameter can include a standard join consisting of the columns and the conditions to join on.
• WHERE(String) • WHERE(String...)	Appends a new WHERE clause condition, concatenated by AND. Can be called multiple times, which causes it to concatenate the new conditions each time with AND. Use OR() to split with an OR.
OR()	Splits the current WHERE clause conditions with an OR. Can be called more than once, but calling more than once in a row will generate erratic SQL.
AND()	Splits the current WHERE clause conditions with an AND. Can be called more than once, but calling more than once in a row will generate erratic SQL. Because WHERE and HAVING both automatically concatenate with AND, this is a very uncommon method to use and is only really included for completeness.
• GROUP_BY(String) • GROUP_BY(String...)	Appends a new GROUP BY clause elements, concatenated by a comma. Can be called multiple times, which causes it to concatenate the new conditions each time with a comma.
• HAVING(String) • HAVING(String...)	Appends a new HAVING clause condition, concatenated by AND. Can be called multiple times, which causes it to concatenate the new conditions each time with AND. Use OR() to split with an OR.
• ORDER_BY(String) • ORDER_BY(String...)	Appends a new ORDER BY clause elements, concatenated by a comma. Can be called multiple times, which causes it to concatenate the new conditions each time with a comma.

`DELETE_FROM(String)`	Starts a delete statement and specifies the table to delete from. Generally this should be followed by a WHERE statement!
`INSERT_INTO(String)`	Starts an insert statement and specifies the table to insert into. This should be followed by one or more VALUES() or INTO_COLUMNS() and INTO_VALUES() calls.
• `SET(String)` • `SET(String...)`	Appends to the "set" list for an update statement.
`UPDATE(String)`	Starts an update statement and specifies the table to update. This should be followed by one or more SET() calls, and usually a WHERE() call.
`VALUES(String, String)`	Appends to an insert statement. The first parameter is the column(s) to insert, the second parameter is the value(s).
`INTO_COLUMNS(String...)`	Appends columns phrase to an insert statement. This should be call INTO_VALUES() with together.
`INTO_VALUES(String...)`	Appends values phrase to an insert statement. This should be call INTO_COLUMNS() with together.

Since version 3.4.2, you can use variable-length arguments as follows:

```
public String selectPersonSql() {
  return new SQL()
    .SELECT("P.ID", "A.USERNAME", "A.PASSWORD", "P.FULL_NAME", "D.DEPARTMENT_NAME",
    .FROM("PERSON P", "ACCOUNT A")
    .INNER_JOIN("DEPARTMENT D on D.ID = P.DEPARTMENT_ID", "COMPANY C on D.COMPANY_I
    .WHERE("P.ID = A.ID", "P.FULL_NAME like #{name}")
    .ORDER_BY("P.ID", "P.FULL_NAME")
    .toString();
}

public String insertPersonSql() {
  return new SQL()
    .INSERT_INTO("PERSON")
    .INTO_COLUMNS("ID", "FULL_NAME")
    .INTO_VALUES("#{id}", "#{fullName}")
    .toString();
}

public String updatePersonSql() {
  return new SQL()
    .UPDATE("PERSON")
    .SET("FULL_NAME = #{fullName}", "DATE_OF_BIRTH = #{dateOfBirth}")
    .WHERE("ID = #{id}")
    .toString();
}
```

7.1.4 SqlBuilder and SelectBuilder (DEPRECATED)

Before version 3.2 we took a bit of a different approach, by utilizing a ThreadLocal variable to mask some of the language limitations that make Java DSLs a bit cumbersome. However, this approach is now deprecated, as modern frameworks have warmed people to the idea of using builder-type patterns and anonymous inner classes for such things. Therefore the SelectBuilder and SqlBuilder classes have been deprecated.

The following methods apply to only the deprecated SqlBuilder and SelectBuilder classes.

Method	Description
`BEGIN()` / `RESET()`	These methods clear the ThreadLocal state of the SelectBuilder class, and prepare it for a new statement to be built. `BEGIN()` reads best when starting a new statement. `RESET()` reads best when clearing a statement in the middle of execution for some reason (perhaps if the logic demands a completely different statement under some conditions).
`SQL()`	This returns the generated `SQL()` and resets the `SelectBuilder` state (as if `BEGIN()` or `RESET()` were called). Thus, this method can only be called ONCE!

The SelectBuilder and SqlBuilder classes are not magical, but it's important to know how they work. SelectBuilder and SqlBuilder use a combination of Static Imports and a ThreadLocal variable to enable a clean syntax that can be easily interlaced with conditionals. To use them, you statically import the methods from the classes like this (one or the other, not both):

```
import static org.apache.ibatis.jdbc.SelectBuilder.*;
```

```
import static org.apache.ibatis.jdbc.SqlBuilder.*;
```

This allows us to create methods like these:

```
/* DEPRECATED */
public String selectBlogsSql() {
  BEGIN(); // Clears ThreadLocal variable
  SELECT("*");
  FROM("BLOG");
  return SQL();
}
```

```
/* DEPRECATED */
private String selectPersonSql() {
  BEGIN(); // Clears ThreadLocal variable
  SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FULL_NAME");
  SELECT("P.LAST_NAME, P.CREATED_ON, P.UPDATED_ON");
  FROM("PERSON P");
  FROM("ACCOUNT A");
  INNER_JOIN("DEPARTMENT D on D.ID = P.DEPARTMENT_ID");
  INNER_JOIN("COMPANY C on D.COMPANY_ID = C.ID");
  WHERE("P.ID = A.ID");
  WHERE("P.FIRST_NAME like ?");
  OR();
  WHERE("P.LAST_NAME like ?");
  GROUP_BY("P.ID");
  HAVING("P.LAST_NAME like ?");
  OR();
  HAVING("P.FIRST_NAME like ?");
  ORDER_BY("P.ID");
  ORDER_BY("P.FULL_NAME");
  return SQL();
}
```

8 Logging

8.1 Logging

MyBatis provides logging information through the use of an internal log factory. The internal log factory will delegate logging information to one of the following log implementations:

- SLF4J
- Apache Commons Logging
- Log4j 2
- Log4j
- JDK logging

The logging solution chosen is based on runtime introspection by the internal MyBatis log factory. The MyBatis log factory will use the first logging implementation it finds (implementations are searched in the above order). If MyBatis finds none of the above implementations, then logging will be disabled.

Many environments ship Commons Logging as a part of the application server classpath (good examples include Tomcat and WebSphere). It is important to know that in such environments, MyBatis will use Commons Logging as the logging implementation. In an environment like WebSphere this will mean that your Log4J configuration will be ignored because WebSphere supplies its own proprietary implementation of Commons Logging. This can be very frustrating because it will appear that MyBatis is ignoring your Log4J configuration (in fact, MyBatis is ignoring your Log4J configuration because MyBatis will use Commons Logging in such environments). If your application is running in an environment where Commons Logging is included in the classpath but you would rather use one of the other logging implementations you can select a different logging implementation by adding a setting in mybatis-config.xml file as follows:

```
<configuration>
  <settings>
    ...
    <setting name="logImpl" value="LOG4J"/>
    ...
  </settings>
</configuration>
```

Valid values are SLF4J, LOG4J, LOG4J2, JDK_LOGGING, COMMONS_LOGGING, STDOUT_LOGGING, NO_LOGGING or a full qualified class name that implements `org.apache.ibatis.logging.Log` and gets an string as a constructor parameter.

You can also select the implementation by calling one of the following methods:

```
org.apache.ibatis.logging.LogFactory.useSlf4jLogging();
org.apache.ibatis.logging.LogFactory.useLog4JLogging();
org.apache.ibatis.logging.LogFactory.useLog4J2Logging();
org.apache.ibatis.logging.LogFactory.useJdkLogging();
org.apache.ibatis.logging.LogFactory.useCommonsLogging();
org.apache.ibatis.logging.LogFactory.useStdOutLogging();
```

If you choose to call one of these methods, you should do so before calling any other MyBatis method. Also, these methods will only switch to the requested log implementation if that implementation is available on the runtime classpath. For example, if you try to select Log4J logging

and Log4J is not available at runtime, then MyBatis will ignore the request to use Log4J and will use it's normal algorithm for discovering logging implementations.

The specifics of SLF4J, Apache Commons Logging, Apache Log4J and the JDK Logging API are beyond the scope of this document. However the example configuration below should get you started. If you would like to know more about these frameworks, you can get more information from the following locations:

- SLF4J
- Apache Commons Logging
- Apache Log4j 1.x and 2.x
- JDK Logging API

8.1.1 Logging Configuration

To see MyBatis logging statements you may enable logging on a package, a mapper fully qualified class name, a namespace o a fully qualified statement name.

Again, how you do this is dependent on the logging implementation in use. We'll show how to do it with Log4J. Configuring the logging services is simply a matter of including one or more extra configuration files (e.g. log4j.properties) and sometimes a new JAR file (e.g. log4j.jar). The following example configuration will configure full logging services using Log4J as a provider. There are 2 steps.

8.1.1.1 Step 1: Add the Log4J JAR file

Because we are using Log4J, we will need to ensure its JAR file is available to our application. To use Log4J, you need to add the JAR file to your application classpath. You can download Log4J from the URL above.

For web or enterprise applications you can add the `log4j.jar` to your `WEB-INF/lib` directory, or for a standalone application you can simply add it to the JVM `-classpath` startup parameter.

8.1.1.2 Step 2: Configure Log4J

Configuring Log4J is simple. Suppose you want to enable the log for this mapper:

```
package org.mybatis.example;
public interface BlogMapper {
  @Select("SELECT * FROM blog WHERE id = #{id}")
  Blog selectBlog(int id);
}
```

Create a file called `log4j.properties` as shown below and place it in your classpath:

```
# Global logging configuration
log4j.rootLogger=ERROR, stdout
# MyBatis logging configuration...
log4j.logger.org.mybatis.example.BlogMapper=TRACE
# Console output...
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```

The above file will cause log4J to report detailed logging for `org.mybatis.example.BlogMapper` and just errors for the rest of the classes of your application.

If you want to tune the logging at a finer level you can turn logging on for specific statements instead of the whole mapper file. The following line will enable logging just for the `selectBlog` statement:

```
log4j.logger.org.mybatis.example.BlogMapper.selectBlog=TRACE
```

By the contrary you may want want to enable logging for a group of mappers. In that case you should add as a logger the root package where your mappers reside:

```
log4j.logger.org.mybatis.example=TRACE
```

There are queries that can return huge result sets. In that cases you may want to see the SQL statement but not the results. For that purpose SQL statements are logged at the DEBUG level (FINE in JDK logging) and results at the TRACE level (FINER in JDK logging), so in case you want to see the statement but not the result, set the level to DEBUG.

```
log4j.logger.org.mybatis.example=DEBUG
```

But what about if you are not using mapper interfaces but mapper XML files like this one?

```
<?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" resultType="Blog">
    select * from Blog where id = #{id}
  </select>
</mapper>
```

In that case you can enable logging for the whole XML file by adding a logger for the namespace as shown below:

```
log4j.logger.org.mybatis.example.BlogMapper=TRACE
```

Or for an specific statement:

```
log4j.logger.org.mybatis.example.BlogMapper.selectBlog=TRACE
```

Yes, as you may have noticed, there is no difference in configuring logging for mapper interfaces or for XML mapper files.

NOTE If you are using SLF4J or Log4j 2 MyBatis will call it using the marker MYBATIS.

The remaining configuration in the `log4j.properties` file is used to configure the appenders, which is beyond the scope of this document. However, you can find more information at the Log4J website (URL above). Or, you could simply experiment with it to see what effects the different configuration options have.