MyBatis 3

User Guide

Warning about Copying Code from this Document

No, this is not a legal warning. It is one to help you keep your sanity. Modern word processors do a great job of making text readable and formatted in an aesthetically pleasing way. However, they also tend to completely ruin code examples by inserting special characters, sometimes that look exactly the same as the one you think you want. "Quotes" and hyphens are a perfect example –the quotes and hyphen you see to the left will not work as quotes in an IDE or text editor, at least not the way you intend.

So read this document, enjoy it and hopefully it is helpful to you. When it comes to code examples, seek out the examples included with the download (including unit tests etc.), or examples from the website or mailing list.

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!

We accept public documentation contributions through our wiki at:

http://opensource.atlassian.com/confluence/oss/display/IBATIS/Contribute+Documentation

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

Contents

What is MyBatis? 5

Getting Started 5

- Building SqlSessionFactory from XML 5
- Building SqlSessionFactory without XML 6
- Acquiring a SqlSession from SqlSessionFactory 6
- Exploring Mapped SQL Statements 7
  - A Note about Namespaces 8
- Scope and Lifecycle 9

Mapper Configuration XML 10

- properties 11
- settings 12
- typeAliases 13
- typeHandlers 14
- objectFactory 15
- plugins 16
- environments 17
  - transactionManager 18
  - dataSource 19
- mappers 21

SQL Map XML Files 21

- select 22
- insert, update, delete 24
- sql 26
- Parameters 26

resultMap 28
Advanced Result Mapping 30
id, result 32
Supported JDBC Types 33
constructor 33
association 34
collection 37
discriminator 40
cache 41
Using a Custom Cache 43
cache-ref 44
Dynamic SQL 44
if 44
choose, when, otherwise 45
trim, where, set 45
foreach 47
Java API 49
Directory Structure 49
SqlSessions 50
SqlSessionFactoryBuilder 50
SqlSessionFactory 52
SqlSession 54
SelectBuilder 60
SqlBuilder 63

What is MyBatis?

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

Getting Started

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

Building SqlSessionFactory from XML

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 Reader 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/Configuration.xml";
Reader reader = Resources.getResourceAsReader(resource);
sqlMapper = new SqlSessionFactoryBuilder().build(reader);
```

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 that contain the SQL code and mapping definitions.

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.

Acquiring a SqlSession from SqlSessionFactory

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

```
SqlSession session = sqlMapper.openSession();
try {
  Blog 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.

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

While this looks like a lot of overhead for this simple example, it is actually very light. You can define as many mapped statements in a single mapper XML file as you like, so you get a lot of milage 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 = (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 sawabove, 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. Third, you don't need to cast the return type, as the BlogMapper interface can have clean, typesafe return types (and a typesafe parameter).

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.

Scope and Lifecycle

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

SqlSessionFactoryBuilder

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

SqlSessionFactory

Once created, the SqlSessionFactory should exist for the duration of your application execution. There should be little or no reason to ever dispose of it or recreate it. It's a best practice to not rebuild the SqlSessionFactory multiple times in an application run. Doing so should be considered a “bad smell”. Therefore the best scope of SqlSessionFactory is application scope. This can be achieved a number of ways. The simplest is to use a Singleton pattern or Static Singleton pattern. However, neither of those is widely accepted as a best practice. Instead, you might prefer to investigate a dependency injection container such as Google Guice or Spring. Such frameworks will allow you to create providers that will manage the singleton lifecycle of SqlSessionFactory for you.

SqlSession

Each thread should have its own instance of SqlSession. Instances of SqlSession are not to be shared and are not thread safe. Therefore the best scope is request or method scope. Never keep references to a SqlSession instance in a static field or even an instance field of a class. Never keep references to a SqlSession in any sort of managed scope, such as HttpSession of of the Servlet framework. If you're using a web framework of any sort, consider the SqlSession to follow a similar scope to that of an HTTP request. In other words, upon recieving 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 (assuming you did not pass in your own connection, which is an indication to MyBatis that you wish to manage your own connection resources).

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 requestsd. 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();
}
```

Mapper Configuration XML

The MyBatis XML configuration file 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
    - mappers

properties

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

```
<properties resource="org/mybatis/example/config.properties">
  <property name="username" value="dev_user"/>
  <property name="password" value="F2Fa3!33TYyg"/>
</properties>
```

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

```
<dataSource type="POOLED">
  <property name="driver" value="${driver}"/>
  <property name="url" value="${url}"/>
  <property name="username" value="${username}"/>
  <property name="password" value="${password}"/>
</dataSource>
```

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

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

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

// ... or ...

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

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

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

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

settings

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

Setting	Description	Valid Values	Default
cacheEnabled	Globally enables or disables any caches configured in any mapper under this configuration.	true | false	true
lazyLoadingEnabled	Globally enables or disables lazy loading. When disabled, all associations will be eagerly loaded.	true | false	true
aggressiveLazyLoading	When enabled, an object with lazy loaded properties will be loaded entirely upon a call to any of the lazy properties. Otherwise, each property is loaded on demand.	true | false	true
multipleResultSetsEnabled	Allows or disallows multiple ResultSets to be returned from a single statement (compatible driver required).	true | false	true
useColumnLabel	Uses the column label instead of the column name. Different drivers behave differently in this respect. Refer to the driver documentation, or test out both modes to determine how your driver behaves.	true | false	true
useGeneratedKeys	Allows JDBC support for generated keys. A compatible driver is required. This setting forces generated keys to be used if set to true, as some drivers deny compatibility but still work (e.g. Derby).	true | false	False
autoMappingBehavior	Specifies if and how MyBatis should automatically map columns to fields/properties. PARTIAL will only auto-map simple, non-nested results. FULL will auto-map result mappings of any complexity (nested or otherwise).	NONE, PARTIAL, FULL	PARTIAL
defaultExecutorType	Configures the default executor. SIMPLE executor does nothing special. REUSE executor reuses prepared statements. BATCH executor reuses statements and batches updates.	SIMPLE REUSE BATCH	SIMPLE
defaultStatementTimeout	Sets the timeout that determines how long the driver will wait for a response from the database.	Any positive integer	Not Set (null)

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="enhancementEnabled" value="false"/>
  <setting name="defaultExecutorType" value="SIMPLE"/>
  <setting name="defaultStatementTimeout" value="25000"/>
</settings>
```

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.

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

typeHandlers

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

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

You can override the type handlers or create your own to deal with unsupported or non-standard types. To do so, simply implementing the TypeHandler interface (org.mybatis.type) and map your new TypeHandler class to a Java type, and optionally a JDBC type. For example:

```
// ExampleTypeHandler.java
public class ExampleTypeHandler implements TypeHandler {
  public void setParameter(
```

```
      PreparedStatement ps, int i, Object parameter,JdbcType jdbcType)
            throws SQLException {
    ps.setString(i, (String) parameter);
  }
  public Object getResult(
            ResultSet rs, String columnName)
                  throws SQLException {
    return rs.getString(columnName);
  }
  public Object getResult(
            CallableStatement cs, int columnIndex)
                  throws SQLException {
    return cs.getString(columnIndex);
  }
}

// MapperConfig.xml
<typeHandlers>
      <typeHandler javaType="String" jdbcType="VARCHAR"
            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.

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> constructorArgs) {
    return super.create(type, constructorArgTypes, constructorArgs);
  }
  public void setProperties(Properties properties) {
    super.setProperties(properties);
  }
}

// MapperConfig.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.

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) {
  }
}
```

```
// MapperConfig.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.

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.

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

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

```
SqlSessionFactory factory = sqlSessionFactoryBuilder.build(reader);
SqlSessionFactory factory = 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. Spring or 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>
```

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, boolean autoCommit);
}
```

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();
  void commit() throws SQLException;
  void rollback() throws SQLException;
  void close() 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 – 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 four properties:

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

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

- **driver.encoding=UTF8**

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

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

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

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

JNDI – This implementation of DataSource is intended for use with containers such as Spring 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.

mappers

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

```
// Using classpath relative resources
<mappers>
    <mapper resource="org/mybatis/builder/AuthorMapper.xml"/>
    <mapper resource="org/mybatis/builder/BlogMapper.xml"/>
    <mapper resource="org/mybatis/builder/PostMapper.xml"/>
</mappers>

// Using url fully qualified paths
<mappers>
    <mapper url="file:///var/sqlmaps/AuthorMapper.xml"/>
    <mapper url="file:///var/sqlmaps/BlogMapper.xml"/>
    <mapper url="file:///var/sqlmaps/PostMapper.xml"/>
</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.

SQL Map XML Files

The true power of MyBatis is in the Mapped Statements. This is where the magic happens. For all of their power, the SQL Map 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 SQL Map 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 DELEETE statement.
- **select** – A mapped SELECT statement.

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

select

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

```
<select id="selectPerson" parameterType="int" resultType="hashmap">
  SELECT * FROM PERSON WHERE ID = #{id}
</select>
```

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

Notice the parameter notation:

```
#{id}
```

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

```
// 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="10000"
      fetchSize="256"
      statementType="PREPARED"
      resultSetType="FORWARD_ONLY"
      >
```

Attribute	Description
id	A unique identifier in this namespace that can be used to reference this statement.
parameterType	The fully qualified class name or alias for the parameter that will be passed into this statement.
~~parameterMap~~	~~This is a deprecated approach to referencing an external parameterMap. Use inline parameter mappings and 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 cache 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. Default: true for select statements.
timeout	This sets the maximum time 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 is

	unset (driver dependent).

insert, update, 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=""
      useGeneratedKeys=""
      timeout="20000">

<update
      id="insertAuthor"
      parameterType="domain.blog.Author"
      flushCache="true"
      statementType="PREPARED"
      timeout="20000">

<delete
      id="insertAuthor"
      parameterType="domain.blog.Author"
      flushCache="true"
      statementType="PREPARED"
      timeout="20000">
```

Attribute	Description
id	A unique identifier in this namespace that can be used to reference this statement.
parameterType	The fully qualified class name or alias for the parameter that will be passed into this statement.
~~parameterMap~~	~~This is a deprecated approach to referencing an external parameterMap. Use inline parameter mappings and the parameterType attribute.~~
flushCache	Setting this to true will cause the cache to be flushed whenever this statement is called. Default: false for select statements.
timeout	This sets the maximum time 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 only) This tells MyBatis to use the JDBC getGeneratedKeys method to retrieve keys generated internally by the database (e.g. auto increment fields in RDBMS like MySQL or SQL Server). Default: false
keyProperty	(insert only) Identifies a property into which MyBatis will set the key value returned by getGeneratedKeys, or by a selectKey child element of the insert statement. Default: unset.

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

```
<insert id="insertAuthor" parameterType="domain.blog.Author">
```

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

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

<delete id="deleteAuthor" parameterType="int">
    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" parameterType="domain.blog.Author"
      useGeneratedKeys="true" keyProperty="id">
    insert into Author (username,password,email,bio)
    values (#{username},#{password},#{email},#{bio})
</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" parameterType="domain.blog.Author">
  <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=VARCHAR}
  )
</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 behaviour 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.
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.
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.

sql

This element can be used to define a reusable fragment of SQL code that can be included in other statements. For example:

```
<sql id="userColumns"> id,username,password </sql>
```

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

```
<select id="selectUsers" parameterType="int" resultType="hashmap">
      select <include refid="userColumns"/>
      from some_table
      where id = #{id}
</select>
```

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 too them, for example:

```
<select id="selectUsers" parameterType="int" 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 simply 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 behaviour 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=departmentResultMap}
```

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=departmentResultMap}
```

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}
```

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 a string unmodified 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.

➔ **IMPORTANT:** 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.

resultMap

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" parameterType="int" resultType="hashmap">
      select id, username, hashedPassword
      from some_table
      where id = #{id}
</sql>
```

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

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

```
<!-- In Config XML file -->
<typeAlias type="com.someapp.model.User" alias="User"/>

<!-- In SQL Mapping XML file -->
<select id="selectUsers" parameterType="int"
      resultType="User">
      select id, username, hashedPassword
      from some_table
      where id = #{id}
```

```
</sql>
```

In these cases MyBatis is automatically creating a ResultMap behind the scenes to 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" parameterType="int" resultType="User">
      select
            user_id           as "id",
            user_name         as "userName",
            hashed_password   as "hashedPassword"
      from some_table
      where id = #{id}
</sql>
```

The great thing about ResultMaps is that you've already learned a lot about them, but you haven't even seen one yet! These simple cases don't require any more than you've seen here. Just for example sake, let's see what this last example would look like as an 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="username"/>
    <result property="password" column="password"/>
</resultMap>
```

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

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

Now if only the world were always that simple.

Advanced Result Mapping

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" parameterType="int" 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" column="blog_author_id" 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" column="post_author_id" javaType="Author"/>
    <collection property="comments" column="post_id" ofType=" Comment">
      <id property="id" column="comment_id"/>
    </collection>
    <collection property="tags" column="post_id" 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.

resultMap

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

➔ **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.

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 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.

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	LONGVARBINARY	BOOLEAN	NCLOB
BIGINT	DECIMAL	TIME	NULL	CURSOR	

constructor

```
<constructor>
       <idArg column="id" javaType="int"/>
       <arg column="username" javaType="String"/>
</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(int id, String username) {
        //…
```

```
    }
    //…
}
```

In order to inject the results into the constructor, MyBatis needs to identify the constructor by the type of its parameters. Java has no way to introspect (or reflect) on parameter names. So when creating a constructor element, ensure that the arguments are in order, and that the data types are specified.

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

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.

association

```
<association property="author" column="blog_author_id" 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 column to retrieve the value from, 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".
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). **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.**
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.

Nested Select for Association

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.**

For example:

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

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

<select id="selectAuthor" parameterType="int" 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.

Nested Results for Association

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.*

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" parameterType="int" 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="blog_id" column="id" />
  <result property="title" column="blog_title"/>
  <association property="author" column="blog_author_id" javaType="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="blog_id" column="id" />
  <result property="title" column="blog_title"/>
  <association property="author" column="blog_author_id" 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>
```

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.

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.

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="blog_id"
      ofType="Post" select="selectPostsForBlog"/>
</resultMap>

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

<select id="selectPostsForBlog" parameterType="int" resultType="Author">
  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="blog_id"
      ofType="Post" select="selectPostsForBlog"/>
```

➔ 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="blog_id" ofType="Post"
      select="selectPostsForBlog"/>
```

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" parameterType="int" 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"/>
</resultMap>

<resultMap id="blogPostResult" type="Post">
    <id property="id" column="post_id"/>
    <result property="subject" column="post_subject"/>
    <result property="body" column="post_body"/>
</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.

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>
```

➔ **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.

cache

MyBatis has includes a powerful 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, there is no caching enabled, except for local session caching, which improves performance and is required to resolve circular dependencies. To enable a 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.

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.*

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.mybatis.cache.Cache interface. 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();
  ReadWriteLock getReadWriteLock();
}
```

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.

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.

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"/>
```

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

if

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

```
<select id="findActiveBlogWithTitleLike"
      parameterType="Blog" 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"
      parameterType="Blog" 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 title like #{author.name}
  </if>
</select>
```

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"
      parameterType="Blog" 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 title like #{author.name}
    </when>
    <otherwise>
      AND featured = 1
    </otherwise>
  </choose>
</select>
```

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"
      parameterType="Blog" 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 title 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"
      parameterType="Blog" 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 title 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 overrides attribute takes a pipe delimited list of text to override, where whitespace is relevant. The result is the removal of anything specified in the *overrides* attribute, and the insertion of anything in the *with* 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"
          parameterType="domain.blog.Author">
    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.

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 a List instance or an Array to MyBatis as a parameter object. When you do, MyBatis will automatically wrap it in a Map, and key it by name. List instances will be keyed to the name "list" and array instances will be keyed to the name "array".

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.

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.

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 Mapping Files.
                        /SqlMapConfig.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                       Remember, these are preferences, not requirements, but others will thank you for using a common directory structure.
            /properties
      /web
            /WEB-INF
                  /web.xml
```

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

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.

SqlSessionFactoryBuilder

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

```
SqlSessionFactory build(Reader reader)
SqlSessionFactory build(Reader reader, String environment)
SqlSessionFactory build(Reader reader, Properties properties)
SqlSessionFactory build(Reader reader, String env, Properties props)
SqlSessionFactory build(Configuration config)
```

The first four methods are the most common, as they take a *Reader* instance that refers to an XML document, or more specifically, the SqlMapConfig.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="EXTERNAL">
        …
      <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 SqlMapConfig.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 SqlMapConfig.xml file.

```
String resource = "org/mybatis/builder/MapperConfig.xml";
Reader reader = Resources.getResourceAsReader(resource);
SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
SqlSessionFactory factory = builder.build(reader);
```

Notice that we're making use of the *Resources* utility class, which lives in the org.mybatis.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, dataSource);

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.

SqlSessionFactory

SqlSessionFactory has six methods that are used to create SqlSessionInstances. 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 has 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 sesson. You need not deal with transactions or batches separately to get the full benefit of them.

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.

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

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.

```
Object selectOne(String statement, Object parameter)
List selectList(String statement, Object parameter)
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. If any more than one, or none (null) is returned, 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 oject, you're better off returning a count (0 or 1). Because not all statements require a parameter, these methods are overloaded with versions that do not require the parameter object.

```
Object selectOne(String statement)
List selectList(String statement)
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.

```
List selectList
      (String statement, Object parameter, RowBounds rowBounds)
void select
      (String statement, Object parameter, ResultHandler handler)
void select
      (String statement, Object parameter, RowBounds rowBounds,
            ResultHandler handler)
```

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.

The interface is very simple.

```
package org.mybatis.executor.result;
public interface ResultHandler {
  void handleResult(ResultContext 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.

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.

Clearing the Session Level Cache

```
void clearCache()
```

The SqlSession instance has a local cache that is cleared upon update, commit, rollback and close. To close it explicitly (perhaps with the intention to do more work), you can call clearCache().

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 pseudocod for "doing some work"
    session.insert(…);
    session.update(…);
    session.delete(…);
    session.commit();
} finally {
    session.close();
}
```

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

```
Configuration getConfiguration()
```

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();
  // insert("insertAuthor", author)
  void insertAuthor(Author author);
  // updateAuthor("updateAuhor", author)
  void updateAuthor(Author author);
  // delete("deleteAuthor",5)
  void 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. All of the usual types are supported, including: Primitives, Maps, POJOs and JavaBeans.

➔ 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.

➔ 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 their position in the parameter list by default, for example: #{1}, #{2} 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.

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** and **readWrite**.
@CacheNamespaceRef	Class	<cacheRef>	References the cache of another namespace to use. Attributes: **value**, which should be the string value of a namespace (i.e. a fully

			qualified class name).
@ConstructorArgs	Method	<constructor>	Collects a group of results to be passed to a result object constructor. Attributes: **value**, which is an array of Args.
@Arg	Method	<arg> <idArg>	A single constructor argument that is part of a ConstructorArgs collection. Attributes: **id**, **column**, **javaType**, **jdbcType**, **typeHandler**. 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 Cases.
@Case	Method	<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**, which is an array of Result annotations.
@Result	Method	<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	Method	<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. ***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	Method	<collection>	A mapping to a collection property of a complex types. 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. ***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.*
@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=false, resultSetType=FORWARD_ONLY, statementType=PREPARED, fetchSize=-1, timeout=-1, useGeneratedKeys=false, keyProperty="id".** 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.
@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 **@UpdateProvider** **@DeleteProvider** **@SelectProvider**	Method	<insert> <update> <delete> <select> Allows for creation of dynamic SQL.	These alternative SQL annotations allow you to specify a class name and a method that will return the SQL to run at execution time. Upon executing the mapped statement, MyBatis will instantiate the class, and execute the method, as specified by the provider. The method can optionally accept the parameter object as its sole parameter, but must only specify that parameter, or no parameters.

			Attributes: **type, method**. The type attribute is the fully qualified name of a class. The method is the name of the method on that class. ***Note***: Following this section is a discussion about the SelectBuilder 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 ordinal position (not including any RowBounds parameters). For example #{1}, #{2} etc. is the default. With @Param("person"), the parameter would be named #{person}.

SelectBuilder

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.

MyBatis 3 introduces a somewhat different concept to deal with the problem. We could have just created an instance of a class that lets you call methods against it to build a SQL statement one step at a time. But then our SQL ends up looking more like Java and less like SQL. Instead, we're trying something a little different. The end result is about as close to a Domain Specific Language that Java will ever achieve in its current form…

The Secrets of SelectBuilder

The SelectBuilder class is not magical, nor does it do any of us any good if you don't know how it works. So right off the bat, let's look at what it does. SelectBuilder uses a combination of Static Imports and a ThreadLocal variable to enable a clean syntax that can be easily interlaced with conditionals and takes care of all of the SQL formatting for you. It allows us to create methods like this:

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

That's a pretty simple example that you might just choose to build statically. So here's a more complicated example:

```
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();
  }
```

Building the above SQL would be a bit of a trial in String concatenation. For example:

```
    "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";
```

If you prefer that syntax, then you're still welcome to use it. It is quite error prone though. Notice the careful addition of a space at the end of each line. Now even if you do prefer that syntax, the next example is inarguably far simpler than Java String concatenation:

```
private String selectPersonLike(Person p){
  BEGIN(); // Clears ThreadLocal variable
  SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FIRST_NAME, P.LAST_NAME");
  FROM("PERSON P");
  if (p.id != null) {
    WHERE("P.ID like #{id}");
  }
  if (p.firstName != null) {
    WHERE("P.FIRST_NAME like #{firstName}");
  }
  if (p.lastName != null) {
```

```
      WHERE("P.LAST_NAME like #{lastName}");
    }
    ORDER_BY("P.LAST_NAME");
    return SQL();
  }
```

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 neither! The statement above will generate a query by example for all PERSON records, ones with ID like the parameter, or the firstName like the parameter, or the lastName like the parameter –or any combination of the three. The SelectBuilder takes care of understanding where "WHEN" needs to go, where an "AND" should be used and all of the String concatenation. Best of all, it does it almost regardless of which order you call these methods in (there's only one exception with the OR() method).

The two methods that may catch your eye are: BEGIN() and SQL(). In a nutshell, every SelectBuilder method should start with a call to BEGIN() and end with a call to SQL(). Of course you can extract methods in the middle to break up your logic, but the scope of the SQL generation should always begin with BEGIN() and end with SQL(). The BEGIN() method clears the ThreadLocal variable, to make sure you don't accidentally carry any state forward, and the SQL() method assembles your SQL statement based on the calls you made since the last call to BEGIN(). Note that BEGIN() has a synonym called RESET(), which does exactly the same thing but reads better in certain contexts.

To use the SelectBuilder as in the examples above, you simply need to import it statically as follows:

```
import static org.mybatis.jdbc.SelectBuilder.*;
```

Once this is imported, the class you're working within will have all of the SelectBuilder methods available to it. The complete set of methods is as follows:

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).
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)	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)	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.

Method	Description
JOIN(String) INNER_JOIN(String) LEFT_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)	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)	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)	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)	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.
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!

SqlBuilder

Similarly to SelectBuilder, MyBatis also includes a generalized SqlBuilder. It includes all the methods for SelectBuilder, as well as methods for building inserts, updates, and deletes. This class can be useful when building SQL strings in a DeleteProvider, InsertProvider, or UpdateProvider (as well as a SelectProvider).

To use the SqlBuilder as in the examples above, you simply need to import it statically as follows:

```
import static org.mybatis.jdbc.SqlBuilder.*;
```

SqlBuilder contains all methods from SelectBuilder, as well as these additional methods:

Method	Description
DELETE_FROM(String)	Starts a delete statement and specifies the table to delete from. Generally this should be followed by a WHERE statement!

Method	Description
INSERT_INTO(String)	Starts an insert statement and specifies the table to insert into. This should be followed by one or more VALUES() calls.
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).

Here are some examples:

```
public String deletePersonSql() {
    BEGIN(); // Clears ThreadLocal variable
    DELETE_FROM("PERSON");
    WHERE("ID = ${id}");
    return SQL();
}

public String insertPersonSql() {
    BEGIN(); // Clears ThreadLocal variable
    INSERT_INTO("PERSON");
    VALUES("ID, FIRST_NAME", "${id}, ${firstName}");
    VALUES("LAST_NAME", "${lastName}");
    return SQL();
}

public String updatePersonSql() {
    BEGIN(); // Clears ThreadLocal variable
    UPDATE("PERSON");
    SET("FIRST_NAME = ${firstName}");
    WHERE("ID = ${id}");
    return SQL();
}
```