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.
        /mybatis-config.xml
        /BlogMapper.java
        /BlogMapper.xml
      /model
      /service
      /view
    /properties         <-- Properties included in your XML Configuratio
n go here.
  /test
    /org/myapp/
      /action
      /data
      /model
      /service
      /view
    /properties
  /web
    /WEB-INF
      /web.xml
```

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

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

SqlSessions

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

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

SqlSessionFactoryBuilder

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

```
SqlSessionFactory build(InputStream inputStream)
SqlSessionFactory build(InputStream inputStream, String envir
onment)
SqlSessionFactory build(InputStream inputStream, Properties p
roperties)
SqlSessionFactory build(InputStream inputStream, String env,
Properties props)
SqlSessionFactory build(Configuration config)
```

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

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

If you call a build method that takes the environment parameter, then MyBatis will use the configuration for that environment. Of course, if you specify an invalid environment, you will receive an error. If you call one of the build methods that does not take the environment parameter, then the default environment is uses (which is specified as default="development" in the example above).

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

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

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

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

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

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

config.xml file.

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

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

```
URL getResourceURL(String resource)
URL getResourceURL(ClassLoader loader, String resource)
InputStream getResourceAsStream(String resource)
InputStream getResourceAsStream(ClassLoader loader, String re
source)
Properties getResourceAsProperties(String resource)
Properties getResourceAsProperties(ClassLoader loader, String
 resource)
Reader getResourceAsReader(String resource)
Reader getResourceAsReader(ClassLoader loader, String resourc
e)
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 JdbcTransactionFa
ctory();

Environment environment = new Environment("development", tran
sactionFactory, 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.cla
ss);
configuration.addMapper(BoundBlogMapper.class);
configuration.addMapper(BoundAuthorMapper.class);

SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuild
er();
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,TransactionIsola
tionLevel level)
SqlSession openSession(ExecutorType execType)
SqlSession openSession(ExecutorType execType, boolean autoCom
mit)
SqlSession openSession(ExecutorType execType, Connection conn
ection)
Configuration getConfiguration();
```

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

- A transaction scope will be started (i.e. NOT auto-commit).
- A `Connection` object will be acquired from the DataSource instance configured by the active environment.
- The transaction isolation level will be the default used by the driver or data source.
- No PreparedStatements will be reused, and no updates will be batched.

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

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

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

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

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

SqlSession

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

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

Statement Execution Methods

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

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

The difference between selectOne and selectList is only in that selectOne must return exactly one object or null (none). If any more than one, an exception will be thrown. If you don't' know how many objects are expected, use selectList. If you want to check for the existence of an object, you're better off returning a count (0 or 1). The selectMap is a special case in that it is designed to convert a list of results into a Map based on one of the properties in the resulting objects. Because not all statements require a parameter, these methods are overloaded with versions that do not require the parameter object.

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

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

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

```
<E> List<E> selectList (String statement, Object parameter, R
owBounds rowBounds)
<K,V> Map<K,V> selectMap(String statement, Object parameter,
String mapKey, RowBounds rowbounds)
void select (String statement, Object parameter, ResultHandle
r handler)
void select (String statement, Object parameter, RowBounds ro
wBounds, 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.apache.ibatis.session;
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.

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

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

Transaction Control Methods

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

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

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

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

Local Cache

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

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

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

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

You can clear the local cache at any time calling:

```
void clearCache()
```

Ensuring that SqlSession is Closed

```
void close()
```

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

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

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

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

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

```
Configuration getConfiguration()
```

Using Mappers

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

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

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

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

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

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

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

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

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

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

Mapper Annotations

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

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

The Annotations are as follows:

Annotation	Target	XML equivalent	Description
`@CacheNamespace`	`Class`	`<cache>`	Configures the cache for the given namespace (i.e. class). Attributes: `implementation`, `eviction`, `flushInterval`, `size`, `readWrite`.
`@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 `Arg`s.
`@Arg`	`Method`	• `<arg>` • `<idArg>`	A single constructor argument that is part of a ConstructorArgs collection. Attributes: `id`, `column`, `javaType`, `jdbcType`, `typeHandler`, `select`, `resultMap`. The id attribute is a boolean value that identifies the property to be used for comparisons, similar to the `<idArg>` XML element.
`@TypeDiscriminator`	`Method`	`<discriminator>`	A group of value cases that can be used to determine the result mapping to perform. Attributes: `column`, `javaType`, `jdbcType`, `typeHandler`, `cases`. The cases attribute is an array of `Case`s.

`@Case`	`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, `fetchType`, which supersedes the global configuration parameter `lazyLoadingEnabled` for this mapping. **NOTE** You will

			notice that join mapping is not supported via the Annotations API. This is due to the limitation in Java Annotations that does not allow for circular references.
`@Many`	`Method`	`<collection>`	A mapping to a collection property of a complex type. Attributes: `select`, which is the fully qualified name of a mapped statement (i.e. mapper method) that can load a collection of instances of the appropriate types, `fetchType`, which supersedes the global configuration parameter `lazyLoadingEnabled` for this mapping. **NOTE** You will notice that join mapping is not supported via the Annotations API. This is due to the limitation in Java Annotations that does not allow for circular references.
`@MapKey`	`Method`		This is used on methods which return type is a Map. It is used to convert a List of result objects as a Map based on a property of those objects. Attributes: `value`, which is a property used as the key of the map.
`@Options`	`Method`	Attributes of mapped statements.	This annotation provides access to the wide range of switches and configuration options that are normally present on the mapped statement as attributes. Rather than complicate each statement annotation, the `Options` annotation provides a consistent and clear way to access these. Attributes: `useCache=true`, `flushCache=false`, `resultSetType=FORWARD_ONLY`,

			`statementType=PREPARED`, `fetchSize=-1`, `timeout=-1`, `useGeneratedKeys=false`, `keyProperty="id"`, `keyColumn=""`. It's important to understand that with Java Annotations, there is no way to specify `null` as a value. Therefore, once you engage the `Options` annotation, your statement is subject to all of the default values. Pay attention to what the default values are to avoid unexpected behavior. Note that `keyColumn` is only required in certain databases (like Oracle and PostgreSQL). See the discussion about `keyColumn` and `keyProperty` above in the discussion of the insert statement for more information about allowable values in these attributes.
• `@Insert` • `@Update` • `@Delete` • `@Select`	`Method`	• `<insert>` • `<update>` • `<delete>` • `<select>`	Each of these annotations represents the actual SQL that is to be executed. They each take an array of strings (or a single string will do). If an array of strings is passed, they are concatenated with a single space between each to separate them. This helps avoid the "missing space" problem when building SQL in Java code. However, you're also welcome to concatenate together a single string if you like. Attributes: `value`, which is the array of Strings to form the single SQL statement.
• `@InsertProvider` • `@UpdateProvider` • `@DeleteProvider`	`Method`	• `<insert>` • `<update>` • `<delete>`	Allows for creation of dynamic SQL. These alternative SQL annotations allow you to specify a class

• `@SelectProvider`		• `<select>`	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 position prefixed with "param" (not including any `RowBounds` parameters). For example `#{param1}`, `#{param2}` etc. is the default. With `@Param("person")`, the parameter would be named `#{person}`.
`@SelectKey`	`Method`	`<selectKey>`	This annotation duplicates the `<selectKey>` functionality for methods annotated with `@Insert`, `@InsertProvider`, `@Update`, or `@UpdateProvider`. It is ignored for other methods. If you specify a `@SelectKey`

			annotation, then MyBatis will ignore any generated key properties set via the `@Options` annotation, or configuration properties. Attributes: `statement` an array of strings which is the SQL statement to execute, `keyProperty` which is the property of the parameter object that will be updated with the new value, `before` which must be either `true` or `false` to denote if the SQL statement should be executed before or after the insert, `resultType` which is the Java type of the `keyProperty`, and `statementType` is a type of the statement that is any one of `STATEMENT`, `PREPARED` or `CALLABLE` that is mapped to `Statement`, `PreparedStatement` and `CallableStatement` respectively. The default is `PREPARED`.
`@ResultMap`	`Method`	N/A	This annotation is used to provide the id of a `<resultMap>` element in an XML mapper to a `@Select` or `@SelectProvider` annotation. This allows annotated selects to reuse resultmaps that are defined in XML. This annotation will override any `@Results` or `@ConstructorArgs` annotation if both are specified on an annotated select.
`@ResultType`	`Method`	N/A	This annotation is used when using a result handler. In that case, the return type is void so MyBatis must have a way to determine the type of object to construct for each

	row. If there is an XML result map, use the @ResultMap annotation. If the result type is specified in XML on the `<select>` element, then no other annotation is necessary. In other cases, use this annotation. For example, if a @Select annotated method will use a result handler, the return type must be void and this annotation (or @ResultMap) is required. This annotation is ignored unless the method return type is void.

Mapper Annotation Examples

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

```
@Insert("insert into table3 (id, name) values(#{nameId}, #{na
me})")
@SelectKey(statement="call next value for TestSequence", keyP
roperty="nameId", before=true, resultType=int.class)
int insertTable3(Name name);
```

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

```
@Insert("insert into table2 (name) values(#{name})")
@SelectKey(statement="call identity()", keyProperty="nameId",
 before=false, resultType=int.class)
int insertTable2(Name name);
```