Mapper XML Files

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

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

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

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

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">
```

Select Attributes

Attribute	Description
`id`	A unique identifier in this namespace that can be used to reference this statement.
`parameterType`	The fully qualified class name or alias for the parameter that will be passed into this statement. This attribute is optional because MyBatis can calculate the TypeHandler to use out of the actual parameter passed to the statement. Default is `unset`.
~~`parameterMap`~~	~~This is a deprecated approach to referencing an external `parameterMap`. Use inline parameter mappings and the `parameterType` attribute.~~
`resultType`	The fully qualified class name or alias for the expected type that will be returned from this statement. Note that in the case of collections, this should be the type that the collection contains, not the type of the collection itself. Use `resultType` OR `resultMap`, not both.
`resultMap`	A named reference to an external `resultMap`. Result maps are the most powerful feature of MyBatis, and with a good understanding of them, many difficult mapping cases can be solved. Use `resultMap` OR `resultType`, not both.
`flushCache`	Setting this to true will cause the local and 2nd level caches to be flushed whenever this statement is called. Default: `false` for select statements.
`useCache`	Setting this to true will cause the results of this statement to be cached in 2nd level cache. Default: `true` for select statements.
`timeout`	This sets the number of seconds the driver will wait for the database to return from a request, before throwing an exception. Default is `unset` (driver dependent).

Attribute	Description
`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).
`databaseId`	In case there is a configured databaseIdProvider, MyBatis will load all statements with no `databaseId` attribute or with a `databaseId` that matches the current one. If case the same statement if found with and without the `databaseId` the latter will be discarded.
`resultOrdered`	This is only applicable for nested result select statements: If this is true, it is assumed that nested results are contained or grouped together such that when a new main result row is returned, no references to a previous result row will occur anymore. This allows nested results to be filled much more memory friendly. Default: `false`.
`resultSets`	This is only applicable for multiple result sets. It lists the result sets that will be returned by the statement and gives a name to each one. Names are separated by commas.

insert, update and delete

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

```
<insert
  id="insertAuthor"
  parameterType="domain.blog.Author"
  flushCache="true"
  statementType="PREPARED"
  keyProperty=""
  keyColumn=""
  useGeneratedKeys=""
  timeout="20">

<update
  id="insertAuthor"
  parameterType="domain.blog.Author"
  flushCache="true"
  statementType="PREPARED"
  timeout="20">

<delete
  id="insertAuthor"
  parameterType="domain.blog.Author"
  flushCache="true"
  statementType="PREPARED"
  timeout="20">
```

Insert, Update and Delete Attributes

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

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

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

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

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

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

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

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

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

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

```
<insert id="insertAuthor">
  <selectKey keyProperty="id" resultType="int" order="BEFORE">
    select CAST(RANDOM()*1000000 as INTEGER) a from SYSIBM.SYSDUMMY1
  </selectKey>
  insert into Author
    (id, username, password, email,bio, favourite_section)
  values
    (#{id}, #{username}, #{password}, #{email}, #{bio}, #{favouriteSection,jdbcType=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 behavior to an auto-generated key in your database without complicating your Java code.

The selectKey element is described as follows:

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

selectKey Attributes

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

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" resultType="map">
  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" 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 behavior is a little different. For example:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Result Maps

The `resultMap` element is the most important and powerful element in MyBatis. It's what allows you to do away with 90% of the code that JDBC requires to retrieve data from `ResultSet`s, 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 `ResultMap`s is such that simple statements don't require explicit result mappings at all, and more complex statements require no more than is absolutely necessary to describe the relationships.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Now if only the world were always that simple.

Advanced Result Maps

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

For example, how would we map this statement?

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

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

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

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

resultMap

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

ResultMap Attributes

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

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:

Id and Result Attributes

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

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	BLOG	NVARCHAR
SMALLINT	DOUBLE	LONGVARCHAR	VARBINARY	CLOB	NCHAR
INTEGER	NUMERIC	DATE	LONGVARBINARY	BOOLEAN	NCLOB
BIGINT	DECIMAL	TIME	NULL	CURSOR	ARRAY

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.

Attribute	Description
`jdbcType`	The JDBC Type from the list of supported types that follows this table. The JDBC type is only required for nullable columns upon insert, update or delete. This is a JDBC requirement, not an MyBatis one. So even if you were coding JDBC directly, you'd need to specify this type – but only for nullable values.
`typeHandler`	We discussed default type handlers previously in this documentation. Using this property you can override the default type handler on a mapping-by-mapping basis. The value is either a fully qualified class name of a `TypeHandler` implementation, or a type alias.
`select`	The ID of another mapped statement that will load the complex type required by this property mapping. The values retrieved from columns specified in the column attribute will be passed to the target select statement as parameters. See the Association element for more.
`resultMap`	This is the ID of a ResultMap that can map the nested results of this argument into an appropriate object graph. This is an alternative to using a call to another select statement. It allows you to join multiple tables together into a single `ResultSet`. Such a `ResultSet` will contain duplicated, repeating groups of data that needs to be decomposed and mapped properly to a nested object graph. To facilitate this, MyBatis lets you "chain" result maps together, to deal with the nested results. See the Association element below for more.

association

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

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

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

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

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

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

Nested Select for Association

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

For example:

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

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

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

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

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

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

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

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

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

And so, there is another way.

Nested Results for Association

Attribute	Description
`resultMap`	This is the ID of a ResultMap that can map the nested results of this association into an appropriate object graph. This is an alternative to using a call to another select statement. It allows you to join multiple tables together into a single ResultSet. Such a ResultSet will contain duplicated, repeating groups of data that needs to be decomposed and mapped properly to a nested object graph. To facilitate this, MyBatis lets you "chain" result maps together, to deal with the nested results. An example will be far easier to follow, and one follows this table.
`columnPrefix`	When joining multiple tables, you would have to use column alias to avoid duplicated column names in the ResultSet. Specifying columnPrefix allows you to map such columns to an external resultMap. Please see the example explained later in this section.
`notNullColumn`	By default a child object is created only if at least one of the columns mapped to the child's properties is non null. With this attribute you can change this behaviour by specifiying which columns must have a value so MyBatis will create a child object only if any of those columns is not null. Multiple column names can be specified using a comma as a separator. Default value: unset.
`autoMapping`	If present, MyBatis will enable or disable automapping when mapping the result to this property. This attribute overrides the global autoMappingBehavior. Note that it has no effect on an external resultMap, so it is pointless to use it with `select` or `resultMap` attribute. Default value: unset.

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

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

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

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

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

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

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

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

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

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

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

Recall that the resultMap for Author is defined as follows.

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

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

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

Multiple ResultSets for Association

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

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

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

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

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

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

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

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

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

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

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

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="id" ofType="Post" select="selectPostsFo
rBlog"/>
</resultMap>

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

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

There are a number things you'll notice immediately, but for the most part it looks very similar to the association element we learned about above. First, you'll notice that we're using the collection element. Then you'll notice that there's a new "ofType" attribute. This attribute is necessary to distinguish between the JavaBean (or field) property type and the type that the collection contains. So you could read the following mapping like this:

```
<collection property="posts" javaType="ArrayList" column="id" ofType="Post" select="selectPostsForB
log"/>
```

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

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

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

Nested Results for Collection

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

First, let's look at the SQL:

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

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

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

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

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

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

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

Multiple ResultSets for Collection

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

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

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

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

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

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

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

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

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

discriminator

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

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

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

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

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

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

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

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

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

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

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

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

Auto-mapping

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

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

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

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

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

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

There are three auto-mapping levels:

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

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

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

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

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

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

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

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

cache

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

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

```
<cache/>
```

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

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

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

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

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

The available eviction policies available are:

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

The default is LRU.

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

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

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

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

Using a Custom Cache

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

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

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

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

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

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

You can use JavaBeans properties of all simple types, MyBatis will do the conversion.

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

It's important to remember that a cache configuration and the cache instance are bound to the namespace of the SQL Map file. Thus, all statements in the same namespace as the cache are bound by it. Statements can modify how they interact with the cache, or exclude themselves completely by using two simple attributes on a statement-by-statement basis. By default, statements are configured like this:

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

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

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"/>
```