Configuration

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

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

properties

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

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

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

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

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

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

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

// ... or ...

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

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

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

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

settings

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

Setting	Description	Valid Values	Default
cacheEnabled	Globally enables or disables any caches configured in any mapper under this configuration.	true | false	true

lazyLoadingEnabled	Globally enables or disables lazy loading. When enabled, all relations will be lazily loaded. This value can be superseded for an specific relation by using the `fetchType` attribute on it.	true | false	false
aggressiveLazyLoading	When enabled, an object with lazy loaded properties will be loaded entirely upon a call to any of the lazy properties. Otherwise, each property is loaded on demand.	true | false	true
multipleResultSetsEnabled	Allows or disallows multiple ResultSets to be returned from a single statement (compatible driver required).	true | false	true
useColumnLabel	Uses the column label instead of the column name. Different drivers behave differently in this respect. Refer to the driver documentation, or test out both modes to determine how your driver behaves.	true | false	true
useGeneratedKeys	Allows JDBC support for generated keys. A compatible driver is required. This setting forces generated keys to be used if set to true, as some drivers deny compatibility but still work (e.g. Derby).	true | false	False
autoMappingBehavior	Specifies if and how MyBatis should automatically map columns to fields/properties. NONE disables auto-mapping. PARTIAL will only auto-map results with no nested result mappings defined inside. FULL will auto-map result mappings of any complexity (containing nested or otherwise).	NONE, PARTIAL, FULL	PARTIAL
defaultExecutorType	Configures the default executor. SIMPLE executor does nothing special. REUSE executor reuses prepared statements. BATCH executor reuses statements and batches updates.	SIMPLE REUSE BATCH	SIMPLE
defaultStatementTimeout	Sets the number of seconds the driver will wait for a response from the database.	Any positive integer	Not Set (null)
safeRowBoundsEnabled	Allows using RowBounds on nested statements.	true | false	False
mapUnderscoreToCamelCase	Enables automatic mapping from classic database column names A_COLUMN to camel case classic Java property names aColumn.	true | false	False
localCacheScope	MyBatis uses local cache to prevent circular references and speed up repeated nested queries. By default (SESSION) all queries executed during a session are cached. If	SESSION | STATEMENT	SESSION

	localCacheScope=STATEMENT local session will be used just for statement execution, no data will be shared between two different calls to the same SqlSession.		
jdbcTypeForNull	Specifies the JDBC type for null values when no specific JDBC type was provided for the parameter. Some drivers require specifying the column JDBC type but others work with generic values like NULL, VARCHAR or OTHER.	JdbcType enumeration. Most common are: NULL, VARCHAR and OTHER	OTHER
lazyLoadTriggerMethods	Specifies which Object's methods trigger a lazy load	A method name list separated by commas	equals,clone,hashCode,toString
defaultScriptingLanguage	Specifies the language used by default for dynamic SQL generation.	A type alias or fully qualified class name.	org.apache.ibatis.scripting.xmltags.XMLDynamicLanguageDriver
callSettersOnNulls	Specifies if setters or map's put method will be called when a retrieved value is null. It is useful when you rely on Map.keySet() or null value initialization. Note primitives such as (int,boolean,etc.) will not be set to null.	true | false	false
logPrefix	Specifies the prefix string that MyBatis will add to the logger names.	Any String	Not set
logImpl	Specifies which logging implementation MyBatis should use. If this setting is not present logging implementation will be autodiscovered.	SLF4J | LOG4J | LOG4J2 | JDK_LOGGING | COMMONS_LOGGING | STDOUT_LOGGING | NO_LOGGING	Not set
proxyFactory	Specifies the proxy tool that MyBatis will use for creating lazy loading capable objects.	CGLIB | JAVASSIST	CGLIB

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

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

typeAliases

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

```
<typeAliases>
  <typeAlias alias="Author" type="domain.blog.Author"/>
  <typeAlias alias="Blog" type="domain.blog.Blog"/>
  <typeAlias alias="Comment" type="domain.blog.Comment"/>
  <typeAlias alias="Post" type="domain.blog.Post"/>
  <typeAlias alias="Section" type="domain.blog.Section"/>
  <typeAlias alias="Tag" type="domain.blog.Tag"/>
</typeAliases>
```

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

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

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

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

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

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

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

list	List
arraylist	ArrayList
collection	Collection
iterator	Iterator

typeHandlers

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

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

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

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

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

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

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

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

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

Using such a TypeHandler would override the existing type handler for Java String properties and VARCHAR parameters and results. Note that MyBatis does not introspect upon the database metadata to determine the type, so you must specify that it's a VARCHAR field in the parameter and result mappings to hook in the correct type handler. This is due to the fact that MyBatis is unaware of the data type until the statement is executed.

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

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

Associated JDBC type can be specified by two means:

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

And finally you can let MyBatis search for your TypeHandlers:

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

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

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

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

  private Class<E> type;

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

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

Handling Enums

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

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

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

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

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

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

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

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

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

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

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

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

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

</mapper>
```

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

objectFactory

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

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

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

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

plugins

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

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

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

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

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

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

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

NOTE **Overriding the Configuration Class**

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

environments

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

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

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

- **One SqlSessionFactory instance per database**

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

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

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

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

The environments element defines how the environment is configured.

```
<environments default="development">
  <environment id="development">
    <transactionManager type="JDBC">
      <property name="..." value="..."/>
    </transactionManager>
    <dataSource type="POOLED">
      <property name="driver" value="${driver}"/>
      <property name="url" value="${url}"/>
      <property name="username" value="${username}"/>
      <property name="password" value="${password}"/>
    </dataSource>
  </environment>
</environments>
```

Notice the key sections here:

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

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

transactionManager

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

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

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

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

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

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

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

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

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

dataSource

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

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

There are three build-in dataSource types (i.e. type="????"):

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

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

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

- `driver.encoding=UTF8`

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

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

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

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

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

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

Similar to the other DataSource configurations, it's possible to send properties directly to the InitialContext by prefixing those properties with `env.`, for example:

- `env.encoding=UTF8`

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

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

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

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

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

public class C3P0DataSourceFactory extends UnpooledDataSourceFactory {

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

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

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

databaseIdProvider

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

```
<databaseIdProvider type="DB_VENDOR" />
```

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

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

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

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

```
public interface DatabaseIdProvider {
  void setProperties(Properties p);
  String getDatabaseId(DataSource dataSource) throws SQLException;
}
```

mappers

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

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

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

```
<!-- Using mapper interface classes -->
<mappers>
  <mapper class="org.mybatis.builder.AuthorMapper"/>
  <mapper class="org.mybatis.builder.BlogMapper"/>
  <mapper class="org.mybatis.builder.PostMapper"/>
</mappers>
```

```
<!-- Register all interfaces in a package as mappers -->
<mappers>
  <package name="org.mybatis.builder"/>
</mappers>
```

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