



REFERENCE

DOCUMENTATION

[Introduction](#)[Getting Started](#)[Configuration](#) >
[XML](#)[Mapper XML](#) >
[Files](#)[Dynamic SQL](#)[Java API](#) >[SQL Builder Class](#)[Logging](#)

PROJECT

DOCUMENTATION

[Project](#) >[Information](#)[Project Reports](#) >

Logging

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

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

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

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

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

Valid values are SLF4J, LOG4J, LOG4J2, JDK_LOGGING, COMMONS_LOGGING, STDOUT_LOGGING, NO_LOGGING or a full





qualified class name that implements `org.apache.ibatis.logging.Log` and gets an string as a constructor parameter.

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

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

If you choose to call one of these methods, you should do so before calling any other MyBatis method. Also, these methods will only switch to the requested log implementation if that implementation is available on the runtime classpath. For example, if you try to select Log4J logging and Log4J is not available at runtime, then MyBatis will ignore the request to use Log4J and will use it's normal algorithm for discovering logging implementations.

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

- [SLF4J](#) 
- [Apache Commons Logging](#) 
- [Apache Log4j 1.x and 2.x](#) 
- [JDK Logging API](#) 

Logging Configuration

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

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

Step 1: Add the Log4J JAR file

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

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

Step 2: Configure Log4J

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

```
package org.mybatis.example;

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

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

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

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

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

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

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

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

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

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

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

```
<?xml version="1.0" encoding="UTF-8" ?>
```

```
<!DOCTYPE mapper
PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="org.mybatis.example.BlogMapper">
  <select id="selectBlog" resultType="Blog">
    select * from Blog where id = #{id}
  </select>
</mapper>
```

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

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

Or for an specific statement:

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

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

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

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