Getting started

Installation

To use MyBatis you just need to include the `mybatis-x.x.x.jar` file in the classpath.

If you are using Maven just add the following dependency to your pom.xml:

```
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>x.x.x</version>
</dependency>
```

Building SqlSessionFactory from XML

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

Building a SqlSessionFactory instance from an XML file is very simple. It is recommended that you use a classpath resource for this configuration, but you could use any InputStream instance, including one created from a literal file path or a file:// URL. MyBatis includes a utility class, called Resources, that contains a number of methods that make it simpler to load resources from the classpath and other locations.

```
String resource = "org/mybatis/example/mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resou
rce);
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBu
ilder().build(inputStream);
```

The configuration XML file contains settings for the core of the MyBatis system, including a DataSource for acquiring database Connection instances, as well as a TransactionManager for determining how transactions should be scoped and controlled. The full details of the XML configuration file can be found later in this document, but here is a simple example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
  <environments default="development">
    <environment id="development">
      <transactionManager type="JDBC"/>
      <dataSource type="POOLED">
        <property name="driver" value="${driver}"/>
        <property name="url" value="${url}"/>
        <property name="username" value="${username}"/>
        <property name="password" value="${password}"/>
      </dataSource>
    </environment>
  </environments>
  <mappers>
    <mapper resource="org/mybatis/example/BlogMapper.xml"/>
  </mappers>
</configuration>
```

While there is a lot more to the XML configuration file, the above example points out the most critical parts. Notice the XML header, required to validate the XML document. The body of the environment element contains the environment configuration for transaction management and connection pooling. The mappers element contains a list of mappers – the XML files and/or annotated Java interface classes that contain the SQL code and mapping definitions.

Building SqlSessionFactory without XML

If you prefer to directly build the configuration from Java, rather than XML, or create your own configuration builder, MyBatis provides a complete Configuration class that provides all of the same configuration options as the XML file.

```
DataSource dataSource = BlogDataSourceFactory.getBlogDataSour
ce();
TransactionFactory transactionFactory = new JdbcTransactionFa
ctory();
Environment environment = new Environment("development", tran
sactionFactory, dataSource);
Configuration configuration = new Configuration(environment);
configuration.addMapper(BlogMapper.class);
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBu
ilder().build(configuration);
```

Notice in this case the configuration is adding a mapper class. Mapper classes are Java classes that contain SQL Mapping Annotations that avoid the need for XML. However, due to some limitations of Java Annotations and the

complexity of some MyBatis mappings, XML mapping is still required for the most advanced mappings (e.g. Nested Join Mapping). For this reason, MyBatis will automatically look for and load a peer XML file if it exists (in this case, BlogMapper.xml would be loaded based on the classpath and name of BlogMapper.class). More on this later.

Acquiring a SqlSession from SqlSessionFactory

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

```
SqlSession session = sqlSessionFactory.openSession();
try {
  Blog blog = session.selectOne("org.mybatis.example.BlogMapp
er.selectBlog", 101);
} finally {
  session.close();
}
```

While this approach works, and is familiar to users of previous versions of MyBatis, there is now a cleaner approach. Using an interface (e.g. BlogMapper.class) that properly describes the parameter and return value for a given statement, you can now execute cleaner and more type safe code, without error prone string literals and casting.

For example:

```
SqlSession session = sqlSessionFactory.openSession();
try {
  BlogMapper mapper = session.getMapper(BlogMapper.class);
  Blog blog = mapper.selectBlog(101);
} finally {
  session.close();
}
```

Now let's explore what exactly is being executed here.

Exploring Mapped SQL Statements

At this point you may be wondering what exactly is being executed by the SqlSession or Mapper class. The topic of Mapped SQL Statements is a big one, and that topic will likely dominate the majority of this documentation. But to give you an idea of what exactly is being run, here are a couple of examples.

In either of the examples above, the statements could have been defined by either XML or Annotations. Let's take a look at XML first. The full set of features provided by MyBatis can be realized by using the XML based mapping language that has made MyBatis popular over the years. If you've used MyBatis before, the concept will be familiar to you, but there have been numerous improvements to the XML mapping documents that will become clear later. Here is an example of an XML based mapped statement that would satisfy the above SqlSession calls.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="org.mybatis.example.BlogMapper">
  <select id="selectBlog" resultType="Blog">
    select * from Blog where id = #{id}
  </select>
</mapper>
```

While this looks like a lot of overhead for this simple example, it is actually very light. You can define as many mapped statements in a single mapper XML file as you like, so you get a lot of mileage out of the XML header and doctype declaration. The rest of the file is pretty self explanatory. It defines a name for the mapped statement “selectBlog”, in the namespace “org.mybatis.example.BlogMapper”, which would allow you to call it by specifying the fully qualified name of “org.mybatis.example.BlogMapper.selectBlog”, as we did above in the following example:

```
Blog blog = session.selectOne("org.mybatis.example.BlogMapper
.selectBlog", 101);
```

Notice how similar this is to calling a method on a fully qualified Java class, and there's a reason for that. This name can be directly mapped to a Mapper class of the same name as the namespace, with a method that matches the name, parameter, and return type as the mapped select statement. This allows you to very simply call the method against the Mapper interface as you saw above, but here it is again in the following example:

```
BlogMapper mapper = session.getMapper(BlogMapper.class);
Blog blog = mapper.selectBlog(101);
```

The second approach has a lot of advantages. First, it doesn't depend on a string literal, so it's much safer. Second, if your IDE has code completion, you can leverage that when navigating your mapped SQL statements.

NOTE **A note about namespaces.**

Namespaces were optional in previous versions of MyBatis, which was confusing and unhelpful. Namespaces are now required and have a purpose beyond simply isolating statements with longer, fully-qualified names.

Namespaces enable the interface bindings as you see here, and even if you don’t think you’ll use them today, you should follow these practices laid out here in case you change your mind. Using the namespace once, and putting it in a proper Java package namespace will clean up your code and improve the usability of MyBatis in the long term.

Name Resolution: To reduce the amount of typing, MyBatis uses the following name resolution rules for all named configuration elements, including statements, result maps, caches, etc.

- Fully qualified names (e.g. “com.mypackage.MyMapper.selectAllThings”) are looked up directly and used if found.
- Short names (e.g. “selectAllThings”) can be used to reference any unambiguous entry. However if there are two or more (e.g. “com.foo.selectAllThings and com.bar.selectAllThings”), then you will receive an error reporting that the short name is ambiguous and therefore must be fully qualified.

There's one more trick to Mapper classes like BlogMapper. Their mapped statements don't need to be mapped with XML at all. Instead they can use Java Annotations. For example, the XML above could be eliminated and replaced with:

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

The annotations are a lot cleaner for simple statements, however, Java Annotations are both limited and messier for more complicated statements. Therefore, if you have to do anything complicated, you're better off with XML mapped statements.

It will be up to you and your project team to determine which is right for you, and how important it is to you that your mapped statements be defined in a consistent way. That said, you're never locked into a single approach. You can very easily migrate Annotation based Mapped Statements to XML and vice versa.

Scope and Lifecycle

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

NOTE Object lifecycle and Dependency Injection Frameworks

Dependency Injection frameworks can create thread safe, transactional SqlSessions and mappers and inject them directly into your beans so you can just forget about their lifecycle. You may want to have a look at MyBatis-Spring or MyBatis-Guice sub-projects to know more about using MyBatis with DI frameworks.

SqlSessionFactoryBuilder

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

SqlSessionFactory

Once created, the SqlSessionFactory should exist for the duration of your application execution. There should be little or no reason to ever dispose of it or recreate it. It's a best practice to not rebuild the SqlSessionFactory multiple times in an application run. Doing so should be considered a “bad smell”. Therefore the best scope of SqlSessionFactory is application scope. This can be achieved a number of ways. The simplest is to use a Singleton pattern or Static Singleton pattern.

SqlSession

Each thread should have its own instance of SqlSession. Instances of SqlSession are not to be shared and are not thread safe. Therefore the best scope is request or method scope. Never keep references to a SqlSession instance in a static field or even an instance field of a class. Never keep references to a SqlSession in any sort of managed scope, such as HttpSession of the Servlet framework. If you're using a web framework of any sort, consider the SqlSession to follow a similar scope to that of an HTTP request. In other words, upon receiving an HTTP request, you can open a SqlSession, then upon returning the response, you can close it. Closing the session is very important. You should always ensure that it's closed within a finally block. The following is the standard pattern for ensuring that SqlSessions are closed:

```
SqlSession session = sqlSessionFactory.openSession();
try {
  // do work
} finally {
  session.close();
}
```

Using this pattern consistently throughout your code will ensure that all database resources are properly closed.

Mapper Instances

Mappers are interfaces that you create to bind to your mapped statements. Instances of the mapper interfaces are acquired from the SqlSession. As such, technically the broadest scope of any mapper instance is the same as the SqlSession from which they were requested. However, the best scope for mapper instances is method scope. That is, they should be requested within the method that they are used, and then be discarded. They do not need to be closed explicitly. While it's not a problem to keep them around throughout a request, similar to the SqlSession, you might find that managing too many resources at this level will quickly get out of hand. Keep it simple, keep Mappers in the method scope. The following example demonstrates this practice.

```
SqlSession session = sqlSessionFactory.openSession();
try {
  BlogMapper mapper = session.getMapper(BlogMapper.class);
  // do work
} finally {
  session.close();
}
```