MyBatis-Spring 1.0.1 - Reference Documentation

The MyBatis Community (MyBatis.org)

Chapter 1. Introduction

1.1. What is MyBatis-Spring?

MyBatis-Spring helps you integrate your MyBatis code seamlessly with Spring. Using the classes in this library, Spring will load the necessary MyBatis factory and session classes for you. This library also provides an easy way to inject MyBatis data mappers and `SqlSessions` into your service beans. It will also handle transactions and translate MyBatis exceptions into Spring `DataAccessExceptions`. Finally, it will let you build your application code free of dependencies on MyBatis, Spring or MyBatis-Spring.

1.2. Motivation

Spring version 2 only supports iBATIS version 2. An attempt was made to add MyBatis 3 support into Spring 3.0 (see the Spring Jira issue). Unfortunately, Spring 3.0 development ended before MyBatis 3.0 was officially released. Because the Spring team did not want to release with code based on a non-released version of MyBatis, official Spring support would have to wait. Given the interest in Spring support for MyBatis, the MyBatis community decided it was time to reunite the interested contributors and add Spring integration as a community sub-project of MyBatis instead.

1.3. Requirements

Before starting with MyBatis-Spring integration, it is very important that you are familiar with both MyBatis and Spring terminology. This document does not attempt to provide background information or basic setup and configuration tutorials for either MyBatis or Spring.

Like MyBatis and Spring 3.0, MyBatis-Spring requires Java 5 or higher.

1.4. Acknowledgements

A special thanks goes to all the special people who made this project a reality (in alphabetical order): Eduardo Macarron, Hunter Presnall and Putthibong Boonbong for the coding, testing and documentation; Andrius Juozapaitis, Giovanni Cuccu, Raj Nagappan and Tomas Pinos for their contributions; and Simone Tripodi for finding everyone and bringing them all back to the project under MyBatis ;) Without them, this project wouldn't exist.

Chapter 2. Getting Started

This chapter will show you in a few steps how to install and setup MyBatis-Spring and how to build a simple transactional application.

2.1. Installation

To use the MyBatis-Spring module, you just need to include the `mybatis-spring-1.0.1.jar` file and its dependencies in the classpath.

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

```
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis-spring</artifactId>
  <version>1.0.1</version>
</dependency>
```

2.2. Quick Setup

To use MyBatis with Spring you need at least two things defined in the Spring application context: an `SqlSessionFactory` and at least one data mapper class.

In MyBatis-Spring, an `SqlSessionFactoryBean` is used to create an `SqlSessionFactory`. To configure the factory bean, put the following in the Spring XML configuration file:

```
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
</bean>
```

Notice that the `SqlSessionFactory` requires a `DataSource`. This can be any `DataSource` and should be configured just like any other Spring database connection.

Assume you have a data mapper class defined like the following:

```
public interface UserMapper {
  @Select("SELECT * FROM users WHERE id = #{userId}")
  User getUser(@Param("userId") String userId);
}
```

This interface is added to Spring using a `MapperFactoryBean` like the following:

```
<bean id="userMapper" class="org.mybatis.spring.mapper.MapperFactoryBean">
  <property name="mapperInterface" value="org.mybatis.spring.sample.mapper.UserMapper" />
  <property name="sqlSessionFactory" ref="sqlSessionFactory" />
</bean>
```

Note that the mapper class specified *must* be an interface, not an actual implementation class. In this example, annotations are used to specify the SQL, but a MyBatis mapper XML file could also be used.

Once configured, you can inject mappers directly into your business/service objects in the same way you inject any other Spring bean. The `MapperFactoryBean` handles creating an `SqlSession` as well as closing it. If there is a Spring transaction in progress, the session will also be committed or rolled back when the transaction completes. Finally, any exceptions will be translated into Spring `DataAccessException`s.

Calling MyBatis data methods is now only one line of code:

```
public class FooServiceImpl implements FooService {

  private UserMapper userMapper;

  public void setUserMapper(UserMapper userMapper) {
    this.userMapper = userMapper;
  }

  public User doSomeBusinessStuff(String userId) {
    return this.userMapper.getUser(userId);
  }
}
```

Chapter 3. SqlSessionFactoryBean

In base MyBatis, the session factory can be built using `SqlSessionFactoryBuilder`. In MyBatis-Spring, `SqlSessionFactoryBean` is used instead.

3.1. Setup

To create the factory bean, put the following in the Spring XML configuration file:

```
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
</bean>
```

Note that `SqlSessionFactoryBean` implements Spring's `FactoryBean` interface (see section 3.8 of the Spring documentation). This means that the bean Spring ultimately creates is *not* the `SqlSessionFactoryBean` itself, but what the factory returns as a result of the `getObject()` call on the factory. In this case, Spring will build an `SqlSessionFactory` for you at application startup and store it with the name `sqlSessionFactory`. In Java, the equivalent code would be:

```
SqlSessionFactoryBean factoryBean = new SqlSessionFactoryBean();
SqlSessionFactory sessionFactory = factoryBean.getObject();
```

In normal MyBatis-Spring usage, you will not need to use `SqlSessionFactoryBean` or the corresponding `SqlSessionFactory` directly. Instead, the session factory will be injected into `MapperFactoryBean`s or other DAOs that extend `SqlSessionDaoSupport`.

3.2. Properties

`SqlSessionFactory` has a single required property, the JDBC `DataSource`. This can be any `DataSource` and should be configured just like any other Spring database connection.

One common property is `configLocation` which is used to specify the location of the MyBatis XML configuration file. One case where this is needed is if the base MyBatis configuration needs to be changed. Usually this will be `<settings>` or `<typeAliases>` sections.

Note that this config file does *not* need to be a complete MyBatis config. Specifically, any environments, data sources and MyBatis transaction managers will be *ignored*. `SqlSessionFactoryBean` creates its own, custom MyBatis `Environment` with these values set as required.

Another reason to require a config file is if the MyBatis mapper XML files are not in the same classpath location as the mapper classes. With this configuration, there are two options. This first is to manually specify the classpath of the XML files using a `<mappers>` section in the MyBatis config file. A second option is to use the `mapperLocations` property of the factory bean.

The `mapperLocations` property takes a list of resource locations. This property can be used to specify the location of MyBatis XML mapper files. The value can contain Ant-style patterns to load all files in a directory or to recursively search all paths from a base location. For example:

```
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="mapperLocations" value="classpath*:sample/config/mappers/**/*.xml" />
</bean>
```

This will load all the MyBatis mapper XML files in the sample.config.mappers package and its sub-packages from the classpath.

One property that may be required in an environment with container managed transactions is `transactionFactoryClass`. Please see the relevant section in the Transactions chapter.

Chapter 4. Transactions

One of the primary reasons for using MyBatis-Spring is that it allows MyBatis to participate in Spring transactions. Rather than create a new transaction manager specific to MyBatis, MyBatis-Spring leverages the existing `DataSourceTransactionManager` in Spring.

Once a Spring transaction manager is configured, you can configure transactions in Spring as you normally would. Both `@Transactional` annotations and AOP style configurations are supported. A single `SqlSession` object will be created and used for the duration of the transaction. This session will be committed or rolled back as appropriate when then transaction completes.

MyBatis-Spring will transparently manage transactions once they are set up. There is no need for additional code in your DAO classes.

4.1. Standard Configuration

To enable Spring transaction processing, simply create a `DataSourceTransactionManager` in your Spring XML configuration file:

```
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource" />
</bean>
```

The `DataSource` specified can be any JDBC `DataSource` you would normally use with Spring. This includes connection pools as well as `DataSource`s obtained through JNDI lookups.

Note that the `DataSource` specified for the transaction manager *must* be the same one that is used to create the `SqlSessionFactoryBean` or transaction management will not work.

4.2. Container Managed Transactions

If you are using a JEE container and would like Spring to participate in container managed transactions (CMT), then Spring should be configured with a `JtaTransactionManager` or one of its container specific subclasses. The easiest way to do this is to use the Spring transaction namespace:

```
<tx:jta-transaction-manager />
```

In this configuration, MyBatis will behave like any other Spring transactional resource configured with CMT. Spring will automatically use any existing container transaction and attach an `SqlSession` to it. If no transaction is started and one is needed based on the transaction configuration, Spring will start a new container managed transaction.

Note that if you want to use CMT and do *not* want to use Spring transaction management, you *must not* configure any Spring transaction manager and you *must* also configure the `SqlSessionFactoryBean` to use the base MyBatis `ManagedTransactionFactory`:

```
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="transactionFactory">
    <bean class="org.apache.ibatis.transaction.managed.ManagedTransactionFactory" />
  </property>
</bean>
```

4.3. Programmatic Transaction Management

MyBatis `SqlSession` provides you with specific methods to handle transactions programmatically. But when using MyBatis-Spring your beans will be injected with a Spring managed `SqlSession` or a Spring managed mapper. That means that Spring will *always* handle your transactions.

You cannot call `SqlSession.commit()`, `SqlSession.rollback()` or `SqlSession.close()` over a Spring managed `SqlSession`. If you try to do so, a `UnsupportedOperationException` exception will be thrown. Note these methods are not exposed in injected mapper classes.

Regardless of your JDBC connection's autocommit setting, any execution of a `SqlSession` data method or any call to a mapper method outside a Spring transaction will be automatically committed.

If you want to control your transactions programmatically please refer to chapter 10.6 of the Spring reference manual. This code shows how to handle a transaction manually using the `PlatformTransactionManager` described in section 10.6.2.

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);

TransactionStatus status = txManager.getTransaction(def);
try {
  userMapper.insertUser(user);
}
catch (MyException ex) {
  txManager.rollback(status);
  throw ex;
}
txManager.commit(status);
```

Notice that this code uses a mapper, but it will also work with a `SqlSession`.

Chapter 5. Using an SqlSession

In MyBatis you use the `SqlSessionFactory` to create an `SqlSession`. Once you have a session, you use it to execute your mapped statements, commit or rollback connections and finally, when it is no longer needed, you close the session. With MyBatis-Spring you don't need to use `SqlSessionFactory` directly because your beans can be injected with a thread safe `SqlSession` that automatically commits, rollbacks and closes the session based on Spring's transaction configuration.

Note that it is usually not necessary to use a `SqlSession` directly. In most cases a `MapperFactoryBean` that will inject mappers into your beans, will be all that is needed. The `MapperFactoryBean` will be explained in detail in the next chapter.

5.1. SqlSessionTemplate

`SqlSessionTemplate` is the heart of MyBatis-Spring. This class is responsible for managing MyBatis `SqlSessions`, calling MyBatis SQL methods and translating exceptions. `SqlSessionTemplate` is thread safe and can be shared by multiple DAOs.

When calling SQL methods, including any method from Mappers returned by `getMapper()`, `SqlSessionTemplate` will ensure that the `SqlSession` used is the one associated with the current Spring transaction. In addition, it manages the session life-cycle, including closing, committing or rolling back the session as necessary.

`SqlSessionTemplate` implements `SqlSession` and is meant to be a drop-in replacement for any existing use of `SqlSession` in your code. `SqlSessionTemplate` should *always* be used instead of default MyBatis implementation `DefaultSqlSession` because the template can participate in Spring transactions and is thread safe for use by multiple injected mapper classes. Switching between the two classes in the same application can cause data integrity issues.

A `SqlSessionTemplate` can be constructed using an `SqlSessionFactory` as a constructor argument.

```
<bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate">
  <constructor-arg index="0" ref="sqlSessionFactory" />
</bean>
```

This bean can now be injected directly in your DAO beans. You need a `SqlSession` property in your bean like the following

```
public class UserDaoImpl implements UserDao {

  private SqlSession sqlSession;

  public void setSqlSession(SqlSession sqlSession) {
    this.sqlSession = sqlSession;
  }

  public User getUser(String userId) {
    return (User) sqlSession.selectOne("org.mybatis.spring.sample.mapper.UserMapper.getUser", userId);
  }
}
```

And inject the `SqlSessionTemplate` as follows

```
<bean id="userDao" class="org.mybatis.spring.sample.dao.UserDaoImpl">
  <property name="sqlSession" ref="sqlSession" />
</bean>
```

`SqlSessionTemplate` has also a constructor that takes an `ExecutorType` as an argument. This allows you to construct, for example, a batch `SqlSession` by using the following in Spring's configuration xml:

```
<bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate">
  <constructor-arg index="0" ref="sqlSessionFactory" />
  <constructor-arg index="1" value="BATCH" />
</bean>
```

Now all your statements will be batched so the following could be coded in a DAO

```
public void insertUsers(User[] users) {
   for (User user : users) {
           sqlSession.insert("org.mybatis.spring.sample.mapper.UserMapper.insertUser", user);
   }
 }
```

Note that this configuration style only needs to be used if the desired execution method differs from the default set for the `SqlSessionFactory`.

The caveat to this form is that there *cannot* be an existing transaction running with a different ExecutorType when this method is called. Either ensure that calls to `SqlSessionTemplates` with different executor types run in a separate transaction (e.g. with PROPAGATION_REQUIRES_NEW) or completely outside of a transaction.

5.2. SqlSessionDaoSupport

`SqlSessionDaoSupport` is an abstract support class that provides you with a `SqlSession`. Calling `getSqlSession()` you will get a `SqlSessionTemplate` which can then be used to execute SQL methods, like the following:

```
public class UserDaoImpl extends SqlSessionDaoSupport implements UserDao {
  public User getUser(String userId) {
    return (User) getSqlSession()
        .selectOne("org.mybatis.spring.sample.mapper.UserMapper.getUser", userId);
  }
}
```

Usually `MapperFactoryBean` is preferred to this class, since it requires no extra code. But, this class is useful if you need to do other non-MyBatis work in your DAO and concrete classes are required.

`SqlSessionDaoSupport` requires either an `sqlSessionFactory` or an `sqlSessionTemplate` property to be set. These can be set explicitly or autowired by Spring. If both properties are set, the `sqlSessionFactory` is ignored.

Assuming a class `UserDaoImpl` that subclasses `SqlSessionDaoSupport`, it can be configured in Spring like the following:

```
<bean id="userMapper" class="org.mybatis.spring.sample.mapper.UserDaoImpl">
  <property name="sqlSessionFactory" ref="sqlSessionFactory" />
</bean>
```

Chapter 6. Injecting Mappers

Rather than code data access objects (DAOs) manually using `SqlSessionDaoSupport` or `SqlSessionTemplate`, Mybatis-Spring provides a proxy factory: `MapperFactoryBean`. This class lets you inject data mapper interfaces directly into your service beans. When using mappers you simply call them as you have always called your DAOs, but you won't need to code any DAO implementation because MyBatis-Spring will create a proxy for you.

With injected mappers your code will have no direct dependencies on MyBatis, Spring or MyBatis-Spring. The proxy that `MapperFactoryBean` creates handles opening and closing the session as well as translating any exceptions into Spring `DataAccessExceptions`. In addition, the proxy will start a new Spring transaction if required or participate in an existing one if it a transaction is active.

6.1. MapperFactoryBean

A data mapper is added to Spring like the following:

```
<bean id="userMapper" class="org.mybatis.spring.mapper.MapperFactoryBean">
  <property name="mapperInterface" value="org.mybatis.spring.sample.mapper.UserMapper" />
  <property name="sqlSessionFactory" ref="sqlSessionFactory" />
</bean>
```

`MapperFactoryBean` creates a proxy class that implements `UserMapper` and injects it into the application. Because a proxy is created at runtime, the specified Mapper *must* be an interface, not an implementation class.

If the UserMapper has a corresponding MyBatis XML mapper file, it will be parsed automatically by the `MapperFactoryBean` if the XML file is in the same classpath location as the Mapper class. There is no need to specify the mapper in a MyBatis configuration file unless the mapper XML files are in a different classpath location. See the `SqlSessionFactoryBean's` `configLocation` property for more information.

Note that `MapperFactoryBean` requires either an `SqlSessionFactory` or an `SqlSessionTemplate`. These can be set through the respective `sqlSessionFactory` and `sqlSessionTemplate` properties, or they can be autowired by Spring. If both properties are set, the `SqlSessionFactory` is ignored. Since the `SqlSessionTemplate` is required to have a session factory set, that factory will be used by `MapperFactoryBean`.

You can inject mappers directly on your business/service objects in the same way you inject any other Spring bean:

```
<bean id="fooService" class="org.mybatis.spring.sample.mapper.FooServiceImpl">
  <property name="userMapper" ref="userMapper" />
</bean>
```

This bean can be used directly in application logic:

```
public class FooServiceImpl implements FooService {

  private UserMapper userMapper;

  public void setUserMapper(UserMapper userMapper) {
    this.userMapper = userMapper;
  }

  public User doSomeBusinessStuff(String userId) {
    return this.userMapper.getUser(userId);
  }
}
```

Notice that there are no `SqlSession` or MyBatis references in this code. Nor is there any need to create, open or close the session, MyBatis-Spring will take care of that.

6.2. MapperScannerConfigurer

There is no need to register all your mappers in the Spring XML file. Instead, you can use a `MapperScannerConfigurer` that will search the classpath for your mappers and set them up automatically as `MapperFactoryBeans`.

To set up a `MapperScannerConfigurer` add the following to the Spring configuration:

```
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
  <property name="basePackage" value="org.mybatis.spring.sample.mapper" />
</bean>
```

The `basePackage` property lets you set the base package for your mapper interface files. You can set more than one package by using a semicolon or comma as a separator. Mappers will be searched for recursively starting in the specified package(s).

Notice that there is no need to specify a `SqlSessionFactory` or `SqlSessionTemplate` because the `MapperScannerConfigurer` will autowire `MapperFactoryBeans`. But, if you are using more than one `DataSource` (thus, more than one SqlSessionFactory), autowiring will not work. In this case you can use the `sqlSessionFactory` or `sqlSessionTemplate` properties to set the right factory / template.

`MapperScannerConfigurer` supports filtering the mappers created by either specifying a marker interface or an annotation. The `annotationClass` property specifies an annotation to search for. The `markerInterface` property specifies a parent interface to search for. If both properties are specified, mappers are added for interfaces that match *either* criteria. By default, these two properties are null, so all interfaces in the given base package(s) will be loaded as mappers.

Discovered mappers will be named using Spring default naming strategy for autodetected components (see section 3.14.4 of the Spring manual). That is, if no annotation is found, it will use the uncapitalized non-qualified class name of the mapper. But if either a @Component or a JSR-330 @Named annotation is found it will get the name from the annotation. Notice that you can set the `annotationClass` property to `org.springframework.stereotype.Component`, `javax.inject.Named` (if you have JSE 6) or to your own annotation (that must be itself annotated) so the annotation will work both as a marker and as a name provider.

Chapter 7. Using the MyBatis API

With MyBatis-Spring, you can continue to directly use the MyBatis API. Simply create an `SqlSessionFactory` in Spring using `SqlSessionFactoryBean` and use the factory in your code.

```
public class UserMapperSqlSessionImpl implements UserMapper {
  // SqlSessionFactory would normally be set by SqlSessionDaoSupport
  private SqlSessionFactory sqlSessionFactory;

  public void setSqlSessionFactory(SqlSessionFactory sqlSessionFactory) {
    this.sqlSessionFactory = sqlSessionFactory;
  }

  public User getUser(String userId) {
    // note standard MyBatis API usage - opening and closing the session manually
    SqlSession session = sqlSessionFactory.openSession();

    try {
      return (User) session.selectOne("org.mybatis.spring.sample.mapper.UserMapper.getUser", userId);
    } finally {
      session.close();
    }
  }
}
```

Use this option *with care* because wrong usage may produce runtime errors or worse, data integrity problems. Be aware of the following caveats with direct API usage:

- It will *not* participate in any Spring transactions.
- If the `SqlSession` is using a `DataSource` that is also being used by a Spring transaction manager and there is currently a transaction in progress, this code *will* throw an exception.
- MyBatis' `DefaultSqlSession` is not thread safe. If you inject it in your beans you *will* get errors.
- Mappers created using `DefaultSqlSession` are not thread safe either. If you inject them it in your beans you *will* get errors.
- You must make sure that your `SqlSessions` are *always* closed in a finally block.

Chapter 8. Sample Code

You can check out sample code from the MyBatis repository on Google Code.

- Java code
- Config files

Any of the samples can be run with JUnit 4.

The sample code shows a typical design where a transactional service gets domain objects from a data access layer.

The service is composed by an interface `FooService.java` and an implementation `FooServiceImpl.java`. This service is transactional so a transaction is started when any of its methods is called and committed when the method ends without throwing an unchecked exception.

```
@Transactional
public interface FooService {
  User doSomeBusinessStuff(String userId);
}
```

Notice that transactional behaviour is configured with the `@Transactional` attribute. This is not required; any other way provided by Spring can be used to demarcate your transactions.

This service calls a data access layer built with MyBatis. This layer is composed by a MyBatis mapper interface `UserMapper.java` and a DAO composed by an interface `UserDao.java` and its corresponding implementation `UserMapperImpl.java`

In all the samples the mapper/Dao is injected into the service bean so the service code is just the following:

```
public class FooServiceImpl implements FooService {
  private UserMapper userMapper;

  public void setUserMapper(UserMapper userMapper) {
    this.userMapper = userMapper;
  }

  public User doSomeBusinessStuff(String userId) {
    return this.userMapper.getUser(userId);
  }
}
```

The database access layer has been implemented using the some of the different techniques explained in this manual.

Table 8.1. Sample test classes

Sample test	Description
`SampleBasicTest`	Shows you the recommended and simplest configuration based on a `MapperFactoryBean`
`SampleAutoTest`	Shows a minimal xml configuration based on component scanning and autowiring
`SampleSqlSessionTest`	Shows how to hand code a DAO using a Spring managed `SqlSession`

Sample test	Description
`SampleBatchTest`	Shows how to use a batch `SqlSession`

Please take a look at the different `applicationContext.xml` files to see MyBatis-Spring in action.