
MyBatis Guice
v. 3.4
Project Documentation

Table Of Content

1. Table Of Content	i
2. Introduction	1
3. Getting Started	2
4. Core components	5
5. Data Sources	12
6. Injectons	13
7. @Transactional	14
8. JDBC Helper	19
9. Samples	23

1 Introduction

1.1 Why MyBatis-Guice - Motivation

In our daily work we've been strongly using both [MyBatis Sql Mapper](#) and [Google Guice](#) frameworks and once noticed we'd been continuously repeating the same code snippets in different projects, according to the DRY *don't repeat yourself* principle, we started realizing something that alleviate us the task to create our stuff.

Indeed, this small library intends to create the missing perfect glue between the two popular frameworks, reducing the boilerplate and redundant code that users have to write to configure and use MyBatis into a Google Guice context.

1.1.1 A little bit of history

The mybatis-guice library was born during the Christmas' vacation in the far December 2009, with the name of [iBaGuice](#) on Google Code, created and maintained by the [99soft.org](#) folks Marco Speranza and Simone Tripodi.

Since the two communities where strictly collaborating, Clinton Begin, the creator of the MyBatis project, invited Simone Tripodi to join the MyBatis team, bringing with him the *iBaGuice* code base to become an official MyBatis subproject.

By that day, the mybatis-guice project is maintained by the MyBatis.org team.

1.2 Requirements

Before starting reading the manual, it is very important you're familiar with both MyBatis and Google Guice framework and therminology.

Like MyBatis and Google Guice, mybatis-guice requires Java 5 or higher.

1.3 Acknowledgements

A special thanks goes to all the special people who made the Google Guice integration a reality, above all Clinton Begin, who strongly believed on that MyBatis sub-project, and Marco Speranza, without him that project wouldn't exist.

Special acknowledgments go to Stephen Friedrich for providing an amazing patch, Poitras Christian who made the XML module a reality and Marzia Forli for the JRS330 feedbacks.

2 Getting Started

2.1 Getting started

MyBatis-Guice helps you integrate your MyBatis code seamlessly with Google Guice. Using the classes in this library, Google Guice will load the necessary MyBatis classes for you. This library also provides an easy way to inject MyBatis data mappers and `SqlSessions` into your application beans. Finally, MyBatis-Guice will let you demarcate transactions declaratively so you won't need to commit/rollback/close them by hand.

2.2 Installation

Installing the mybatis-guice module it is very easy, just put the `mybatis-guice-X.X.jar` and dependencies in the classpath!

Apache Maven users instead can easily adding the following dependency in their POMs :

```
<dependencies>
...
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis-guice</artifactId>
  <version>X.X</version>
</dependency>
...
</dependencies>
```

2.3 Quick Setup

To use MyBatis with Guice you need to set up a `MyBatisModule`, with a `DataSource` at least one data mapper class and a transactional bean to be injected with the mapper.

To setup a `DataSource` you can simply use the `JdbcHelper` (please see the related appendix for more informations) module to build the URL needed for your database and provide the connection properties:

```
Properties myBatisProperties = new Properties();
myBatisProperties.setProperty("mybatis.environment.id", "test");
myBatisProperties.setProperty("JDBC.schema", "mybatis-guice_TEST");
myBatisProperties.setProperty("derby.create", "true");
myBatisProperties.setProperty("JDBC.username", "sa");
myBatisProperties.setProperty("JDBC.password", "");
myBatisProperties.setProperty("JDBC.autoCommit", "false");

Injector injector = Guice.createInjector(
    JdbcHelper.HSQLDB_Embedded,
    new Module() {
        public void configure(Binder binder) {
            Names.bindProperties(binder, myBatisProperties);
        }
    }
);
```

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

```
public interface UserMapper {

    @Select("SELECT * FROM user WHERE id = #{userId}")
    User getUser(@Param("userId") String userId);

}
```

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.

Assume you also have a transactional service bean that uses your mapper:

```
public class FooServiceMapperImpl implements FooService {

    @Inject
    private UserMapper userMapper;

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

}
```

Setup a `MyBatisModule`, add your mapper to it and bind also your transactional service interface `FooService` to its implementation.

```
public interface UserMapper {
    ...
}

Injector injector = Guice.createInjector(
    new MyBatisModule() {

        @Override
        protected void initialize() {
            install(JdbcHelper.HSQLDB_Embedded);

            bindDataSourceProviderType(PooledDataSourceProvider.class);
            bindTransactionFactoryType(JdbcTransactionFactory.class);
            addMapperClass(UserMapper.class);

            Names.bindProperties(binder, createTestProperties());
            bind(FooService.class).to(FooServiceMapperImpl.class);
        }

    }

);
```

This is all you need, you can now get an instance of your service. It will be automatically injected with the MyBatis mapper. With the mapper calling MyBatis data methods is only one line of code. Besides, all operations will be transactional, that means that you will not need to commit or rollback any connection.

```
FooService fooService = this.injector.getInstance(FooService.class);  
fooService.doSomeBusinessStuff("data");
```


3 Core components

3.1 Introduction

Core components are contained in the *org.mybatis.guice.** package, providing a set of reusable Google Guice *javax.inject.Providers* and *com.google.inject.Modules* that alleviate users the task to create MyBatis objects.

3.2 MyBatis Bootstrap

MyBatis offers an excellent APIs layer for the Bootstrap configuration that makes it easy to write custom bootstrap - by default MyBatis comes with the XML loader - and integrating 3rd part components.

The core component of the Guice approach is represented by the *org.mybatis.guice.MyBatisModule* that's able to create the core MyBatis *org.apache.ibatis.session.SqlSessionFactory*, *org.apache.ibatis.session.SqlSessionManager* and the user defined *Mappers*.

The best way to start is just adding the *org.mybatis.guice.MyBatisModule* into the *com.google.inject.Injector* as shown in the example below and explain details step by step:

```
Class<? extends Provider<DataSource>> dataSourceProviderType = [...];

Class<? extends TransactionFactory> txFactoryClass = [...];

Injector injector = Guice.createInjector(
    new MyBatisModule() {

        @Override
        protected void initialize() {
            environmentId("development");
            bindDataSourceProviderType(dataSourceProviderType);
            bindTransactionFactoryType(txFactoryClass);
        }

    },
    ...
);

SqlSessionFactory sessionFactory = injector.getInstance(SqlSessionFactory.class);
SqlSessionManager sessionManager = injector.getInstance(SqlSessionManager.class);
MyMapper mapper = injector.getInstance(MyMapper.class);
```

Let's have a look now at the MyBatis module components and features:

3.3 MyBatis properties

By design, we choose to reuse the default configuration properties provided by Guice to let users feel free to read and set them in any way you prefer; we suggest to put it in a properties file, maybe filtered and set depending on which environment users are building the application.

By default, if a configuration property is not specified, it will be ignored and MyBatis will take care about proper default initialization. Users can initialize properties using the proper setters OR by following method (please don't bind properties twice!):

```
binder.bindConstant( )
    .annotatedWith(Names.named( "mybatis.configuration.XXX" ))
    .to(XXXvalue);
```

The MyBatis module supports the following parameters:

Property	Setter	Default
mybatis.environment.id	environmentId(String)	Not set, it is required
mybatis.configuration.lazyLoadingEn	lazyLoadingEnabled(boolean)	false
mybatis.configuration.aggressiveLazy	aggressiveLazyLoading(boolean)	true
mybatis.configuration.multipleResultS	multipleResultSetsEnabled(boolean)	true
mybatis.configuration.useGeneratedId	useGeneratedKeys(boolean)	false
mybatis.configuration.useColumnLab	useColumnLabel(boolean)	true
mybatis.configuration.cacheEnabled	useCacheEnabled(boolean)	true
mybatis.configuration.defaultExecuto	executorType(ExecutorType)	ExecutorType.SIMPLE
mybatis.configuration.autoMappingBe	autoMappingBehavior(AutoMappingE	AutoMappingBehavior.PARTIAL
mybatis.configuration.failFast	failFast(boolean)	false

org.mybatis.guice.MyBatisModule properties

3.4 The DataSource Provider

The *javax.sql.DataSource* Provider is one of the two required providers that takes care about building and injecting the used *javax.sql.DataSource*.

The mybatis-guice framework comes with some providers that support the MyBatis-native Data Sources and other well known Data Sources, *C3P0*, *Apache Commons DBCP* and *BoneCP* but users are free to implement their own *javax.sql.DataSource* Provider and reference it in the *org.mybatis.guice.MyBatisModule*.

Please read the following chapter *Data Source Providers* to obtain more informations about natively supported providers.

3.5 The Transaction Factory

Users are free to plug their preferred *org.apache.ibatis.transaction.TransactionFactory*:

```
Class<? extends org.apache.ibatis.transaction.TransactionFactory> txFactoryType = .

Module module = new MyBatisModule() {

    @Override
    protected void initialize() {
        ...
        bindTransactionFactoryType(txFactoryType);
        ...
    }

}
```

3.6 Configuring aliases

Once users create the *org.mybatis.guice.MyBatisModule.Builder*, it's quite easy plugging optional MyBatis components, like aliases: here users can define simple aliases, for example *Foo* that stands for *com.acme.Foo*, or custom aliases, for example *MyFoo* that stands for *com.acme.Foo*.

We found it very useful to add simple aliases because it helped us reduce errors during development; just call:

```
MyBatisModule module = new MyBatisModule() {  
  
    @Override  
    protected void initialize() {  
        ...  
        addSimpleAlias(com.acme.Foo.class);  
        addSimpleAlias(com.acme.Bar.class);  
        addSimpleAlias(...);  
        ...  
    }  
  
}
```

If you prefer custom aliases, just invoke:

```
MyBatisModule module = new MyBatisModule() {  
  
    @Override  
    protected void initialize() {  
        ...  
        addAlias("MyFoo").to(com.acme.Foo.class);  
        addAlias("MyBar").to(com.acme.Bar.class);  
        ...  
    }  
  
}
```

3.7 Configuring Type Handlers

Users can also configure type handlers: given the *com.acme.Foo* type, that has to be handled by the type handler *com.acme.dao.FooHandler*, just invoke

```
MyBatisModule module = new MyBatisModule() {  
  
    @Override  
    protected void initialize() {  
        ...  
        handleType(com.acme.Foo.class).with(com.acme.dao.FooHandler.class);  
        handleType(com.acme.Bar.class).with(com.acme.dao.BarHandler.class);  
        ...  
    }  
  
}
```

and let Google Guice create the handlers instances and bind them to be injected to MyBatis components.

3.8 Configuring Interceptor Plugins

Users can easily add their preferred *org.apache.ibatis.plugin.Interceptor* by invoking:

```
MyBatisModule module = new MyBatisModule() {  
  
    @Override  
    protected void initialize() {  
        ...  
        addInterceptorClass(com.acme.dao.FooInterceptor.class);  
        addInterceptorClass(com.acme.dao.BarInterceptor.class);  
        ...  
    }  
  
}
```

and let Google Guice create the interceptors instances and bind them to be injected to MyBatis components.

3.9 Configuring Mappers

Users can add *Mapper* classes to the module by invoking:

```
MyBatisModule module = new MyBatisModule() {  
  
    @Override  
    protected void initialize() {  
        ...  
        addMapperClass(com.acme.dao.FooMapper.class);  
        addMapperClass(com.acme.dao.BarMapper.class);  
        ...  
    }  
  
};
```

and let Google Guice create the mappers instance and bind them to be injected to MyBatis components.

3.10 Configuring the Object Factory

Simply define your own *org.apache.ibatis.reflection.factory.ObjectFactory* and communicate it to the module and let Google Guice create it:

```
MyBatisModule module = new MyBatisModule() {  
  
    @Override  
    protected void initialize() {  
        ...  
        bindObjectFactoryType(com.acme.MyObjectFactory.class);  
        ...  
    }  
  
};
```

3.11 Multiple Datasources

It often happens that users need to interact with multiple schemas in the same application, that means to have separate MyBatis configurations.

Fortunately, the Google Guice *com.google.inject.PrivateModule* comes to help us in a very simple and smart way, that will be shown in the following example.

Let's take in consideration, to simplify the example, we have only two datasources (but the same concept can be extended for an arbitrary data sources number) one for the *contacts* schema and another one for the *companies* schema. So, all it has to do is installing the *org.mybatis.guice.MyBatisModule* modules into the Google Guice *com.google.inject.PrivateModule* as shown below:

```

Injector injector = Guice.createInjector(
    new PrivateModule() {
        @Override
        protected void configure() {
            install(new MyBatisModule() {

                @Override
                protected void initialize() {
                    bindDataSourceProviderType(PooledDataSourceProvider.class);
                    bindTransactionFactoryType(JdbcTransactionFactory.class);
                    addMapperClass(ContactMapper.class);
                    addSimpleAlias(Contact.class);
                }

            });
            Names.bindProperties(this.binder(),
                getConnectionProperties("contacts"));
            // binds Mappers/DAOs here
            bind(ContactDao.class).to(ContactDaoImpl.class);
            ...

            // exposes Mappers/DAOs here
            expose(ContactDao.class);
            ...
        }
    }, new PrivateModule() {
        @Override
        protected void configure() {
            install(new MyBatisModule() {

                @Override
                protected void initialize() {
                    bindDataSourceProviderType(PooledDataSourceProvider.class);
                    bindTransactionFactoryType(JdbcTransactionFactory.class);
                }

            });
            Names.bindProperties(this.binder(),
                getConnectionProperties("trades"));
            // binds Mappers/DAOs here
            bind(CompanyDao.class).to(CompanyDaoImpl.class);
            ...

            // exposes Mappers/DAOs here
            expose(CompanyDao.class);
            ...
        }
    }
);

```

The example shows how to use the *org.mybatis.guice.MyBatisModule* to create two different MyBatis configurations in the same context. Feel free to implement the *getConnectionProperties()* method in the way you prefer! It could be, for example:

```
private final static Properties getConnectionProperties(String schema) {
    final Properties myBatisProperties = new Properties();

    myBatisProperties.setProperty("mybatis.environment.id", "test");
    myBatisProperties.setProperty("JDBC.driver",
        "org.apache.derby.jdbc.EmbeddedDriver");
    myBatisProperties.setProperty("JDBC.url",
        "jdbc:mysql://localhost:3306/" + schema);
    myBatisProperties.setProperty("JDBC.username", "mybatis-user");
    myBatisProperties.setProperty("JDBC.password", "changeme");
    myBatisProperties.setProperty("JDBC.autoCommit", "true");

    return myBatisProperties;
}
```

3.12 MyBatis XML Bootstrap

Users that want configure the MyBatis via the XML configuration, without loosing any single feature of the *org.mybatis.guice.MyBatisModule*, can create their Injector using the *org.mybatis.guice.XMLMyBatisModule*.

XMLMyBatisModule clients have just to instantiate it specifying

1. the MyBatis XML configuration file, located in the classpath, by default the module will look for *mybatis-config.xml* in the root in the classpath;
2. the optional MyBatis *environmentId*, *development* by default;
3. the optional *java.util.Properties* to fill placeholders in the MyBatis XML configuration, empty by default.

A typical use case could be identified in the following code snippet:

```
Properties props = new Properties();
props.setProperty("JDBC.username", "mybatis-user");
props.setProperty("JDBC.password", "changeme");

Injector injector = Guice.createInjector(
    new XMLMyBatisModule() {

        @Override
        protected void initialize() {
            setEnvironmentId("test");
            setClassPathResource("my/path/to/mybatis-config.xml");
            addProperties(props);
        }

    },
    ...
);
```

Important Google Guice will inject dependencies, if required, in the *TypeHandlers* and *Interceptors*.

4 Data Sources

4.1 DataSource setup

The `org.mybatis.guice.datasource` package contains an easy-to-use set of classes that makes easier the `javax.sql.DataSource` creation using Guice, through configurable Data Source Providers.

Important `org.mybatis.guice.XMLMyBatisModule` users can skip this section.

Configurable means that users are free to bind `com.google.inject.name.Named` Data Source properties and let Guice injects them.

As previously said, the mybatis-guice framework comes with some providers that support the MyBatis [built-in Data Sources](#), moreover we added the support for the popular:

1. Apache Commons [DBCP](#);
2. [C3P0](#);
3. [BoneCP](#).

5 Injections

5.1 Requesting Injections

5.1.1 Getting 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, get mappers, commit or rollback connections and finally, when it is no longer needed, you close the session. With MyBatis-Guice 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 `@Transactional` annotation.

```
public class UserDaoImpl implements UserDao {

    @Inject
    private SqlSession sqlSession;

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

}
```

5.1.2 Getting Mappers

Rather than code data access objects (DAOs) manually using `SqlSession`, Mybatis-Guice is able to 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-Guice will create a proxy for you. When using mappers you will not even see the inner `SqlSession`, but not worries, it will just work.

```
@Singleton
public class FooServiceMapperImpl implements FooService {

    @Inject
    private UserMapper userMapper;

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

}
```

6 @Transactional

6.1 @Transactional

6.1.1 Introduction

Thanks to the excellent combination between AOP and Google Guice, users can drastically reduce the boilerplate code into their DAOs.

Let's take in consideration the following code snippet, written without introducing mybatis-guice:

```
package com.acme;

import org.apache.ibatis.session.*;
import org.mybatis.guice.transactional.*;

public final class FooDAO {

    private final SqlSessionManager sessionManager;

    public FooDAO(SqlSessionManager sessionManager) {
        this.sessionManager = sessionManager;
    }

    public void doFooBar() throws MyDaoException {
        // Starts a new SqlSession
        this.sessionManager.startManagedSession(ExecutorType.BATCH,
            TransactionIsolationLevel.READ_UNCOMMITTED);
        try {
            // Retrieve the FooMapper and execute the doFoo() method.
            FooMapper fooMapper = this.sessionManager.getMapper(FooMapper.class);
            fooMapper.doFoo();

            // Retrieve the BarMapper and execute the doBar() method.
            BarMapper barMapper = this.sessionManager.getMapper(BarMapper.class);
            barMapper.doBar();

            // If everything gone fine, commit the open session.
            this.sessionManager.commit();
        } catch (Throwable t) {
            // If something gone wrong, rollback the open session.
            this.sessionManager.rollback();
            // Optionally, throw a proper DAO layer Exception
            throw new MyDaoException("Something went wrong", t);
        } finally {
            // Close the session.
            this.sessionManager.close();
        }
    }
}
```

Users can easily note that this is a recursive and redundant code pattern that mybatis-guice will help to simplify introducing a special AOP interceptor.

6.1.2 The @Transactional annotation

Annotating methods with the `org.mybatis.guice.transactional.Transactional` annotation, users can eliminate recursive code patterns.

First of all, let's have a look at the injector that will create the previous `FooDAO` instance:

```
Class<? extends Provider<DataSource>> dataSourceProviderClass = [...];
Class<? extends Provider<TransactionFactory>> txFactoryProviderClass = [...];

Injector injector = Guice.createInjector(new MyBatisModule() {

    @Override
    protected void initialize() {
        environmentId("test");
        bindDataSourceProviderType(dataSourceProviderType);
        bindTransactionFactoryType(txFactoryClass);
        addMapperClass(FooMapper.class);
        addMapperClass(BarMapper.class);
    }

});

FooDAO fooDAO = injector.getInstance(FooDAO.class);
```

Where `FooDAO` definition is:

```

package com.acme;

import javax.inject.*;
import org.apache.ibatis.session.*;
import org.mybatis.guice.transactional.*;

@Singleton
public final class FooDAOImpl {

    @Inject
    private FooMapper fooMapper;

    @Inject
    private BarMapper barMapper;

    // let's assume setters here

    @Transactional(
        executorType = ExecutorType.BATCH,
        isolation = Isolation.READ_UNCOMMITTED,
        rethrowExceptionsAs = MyDaoException.class,
        exceptionMessage = "Something went wrong"
    )
    public void doFooBar() {
        this.fooMapper.doFoo();
        this.barMapper.doBar();
    }

}

```

Users can now simply read how the code can be reduced, delegating to the interceptor the session management!

The `org.mybatis.guice.transactional.Transactional` annotation supports the following parameters:

Property	Default	Description
<code>executorType</code>	<code>ExecutorType.SIMPLE</code>	the MyBatis executor type
<code>isolation</code>	<code>Isolation.DEFAULT</code>	the transaction isolation level. The default value will cause MyBatis to use the default isolation level from the data source.
<code>force</code>	<code>false</code>	Flag to indicate that MyBatis has to force the transaction <code>commit()</code>
<code>rethrowExceptionsAs</code>	<code>Exception.class</code>	rethrow caught exceptions as new Exception (maybe a proper layer exception)

exceptionMessage	empty string	A custom error message when throwing the custom exception; it supports <code>java.util.Formatter</code> place holders, intercepted method arguments will be used as message format arguments.
rollbackOnly	false	If true, the transaction will never committed, but rather the rollback will be forced. That configuration is useful for testing purposes.

org.mybatis.guice.transactional.Transactional properties

When specifying `rethrowExceptionsAs` parameter, it is required that the target exception type has the constructor with `Throwable` single argument; when specifying both `rethrowExceptionsAs` and `exceptionMessage` parameters, it is required that the target exception type has the constructor with `String`, `Throwable` arguments; specifying the `exceptionMessage` parameter only doesn't have any effect.

6.1.3 Nested transactions

The `org.mybatis.guice.transactional.Transactional` annotation is nicely handled to support inner transactional methods; given the following simple MyBatis clients:

```
class ServiceA {
    @Transactional
    public void method() {
        ...
    }
}

class ServiceB {
    @Transactional
    public void method() {
        ...
    }
}
```

That in a certain point are involved in another one in the same transaction:

```
class CompositeService {  
  
    @Inject  
    ServiceA serviceA;  
  
    @Inject  
    ServiceB serviceB;  
  
    @Transactional  
    public void method() {  
        ...  
        this.serviceA.method();  
        ...  
        this.serviceB.method();  
        ...  
    }  
}
```

In this case, `ServiceA#method()` and `ServiceB#method` can be invoked as atomic transactions, the advantage is when `serviceA#method()` and `serviceB#method()` will be invoked inside the `CompositeService#method`, that the interceptor will take care to manage them in the same session, even if annotated to start a new transaction.

7 JDBC Helper

7.1 JDBC Helper

7.1.1 Bind automatically JDBC Connection URL and Driver

For those users (like me!!!) don't have a dictionary to quickly retrieve the JDBC url pattern and the driver class name for a specific DBMS, here it comes the `org.mybatis.guice.datasource.helper.JdbcHelper` Module.

It is an easy to use Google Guice module that makes easy the `JDBC.url` and `JDBC.driver` properties binding needed for the `DataSource` providers.

The `org.mybatis.guice.datasource.helper.JdbcHelper` provides a large number of *Enumeration* values that cover the most popular DBMS (both commercial and open source) that are provided/recommended by the actual database vendor, resumed in the table below.

Variable Format Patterns contain variables in the *Apache ANT* alike style, in the format `${name|defaultValue}`, where `name` is used as label which will looked for in the current Guice Injector, with `defaultValue` as its default value, replaced in the pattern if `name` is not found.

NOTE this feature doesn't work with XML Module!!!

7.1.2 Supported DBMS

DBMS	URL pattern
<code>JdbcHelper.Cache</code>	<code>jdbc:Cache://\${JDBC.host localhost}:\${JDBC.port 1972}/\${JDBC.schema}</code>
<code>JdbcHelper.Daffodil_DB</code>	<code>jdbc:daffodilDB://\${JDBC.host localhost}:\${JDBC.port 3456}/\${JDBC.schema}</code>
<code>JdbcHelper.DB2</code>	<code>jdbc:db2://\${JDBC.host localhost}:\${JDBC.port 50000}/\${JDBC.schema}</code>
<code>JdbcHelper.DB2_DataDirect</code>	<code>jdbc:datadirect:db2://\${JDBC.host localhost}:\${JDBC.port 50000}/\${JDBC.schema}</code> <code>DatabaseName=\${JDBC.schema}</code>
<code>JdbcHelper.DB2_AS400_JTOpen</code>	<code>jdbc:as400://\${JDBC.host localhost}</code>
<code>JdbcHelper.Firebird</code>	<code>jdbc:firebirdsql:\${JDBC.host localhost}/\${JDBC.port 3050}:\${JDBC.schema}</code>
<code>JdbcHelper.FrontBase</code>	<code>jdbc:FrontBase://\${JDBC.host localhost}/\${JDBC.schema}</code>
<code>JdbcHelper.HP_Neoview</code>	<code>jdbc:hpt4jdbc://\${neoview.system}:\${JDBC.port}/:</code>
<code>JdbcHelper.HSQLDB_Server</code>	<code>jdbc:hsqldb:hsql://\${JDBC.host localhost}:\${JDBC.port 9001}/\${JDBC.schema}</code>
<code>JdbcHelper.HSQLDB_Embedded</code>	<code>jdbc:hsqldb:\${JDBC.schema}</code>
<code>JdbcHelper.H2_IN_MEMORY_PRIVATE</code>	<code>jdbc:h2:mem"</code>
<code>JdbcHelper.H2_IN_MEMORY_NAMED</code>	<code>jdbc:h2:mem:\${JDBC.schema}"</code>

JdbcHelper.H2_SERVER_TCP	jdbc:h2:tcp://\${JDBC.host localhost}:\${JDBC.port 9001}/ \${JDBC.schema}"
JdbcHelper.H2_SERVER_SSL	jdbc:h2:ssl://\${JDBC.host localhost}:\${JDBC.port 9001}/ \${JDBC.schema}"
JdbcHelper.H2_FILE	jdbc:h2:file://\${JDBC.schema}"
JdbcHelper.H2_EMBEDDED	jdbc:h2:\${JDBC.schema}"
JdbcHelper.Informix	jdbc:informix-sqli://\${JDBC.host localhost}:\${JDBC.port 1533}/ \${JDBC.schema}: informixserver= \${dbservername}
JdbcHelper.Informix_DataDirect	jdbc:datadirect:informix:// \${JDBC.host localhost}: \${JDBC.port 1533}; InformixServer= \${informixserver}; DatabaseServer= \${JDBC.schema}
JdbcHelper.Derby_Server	jdbc:derby://\${JDBC.host localhost}: \${JDBC.port 1527}/\${JDBC.schema}
JdbcHelper.Derby_Embedded	jdbc:derby:\${JDBC.schema}; create= \${derby.create false}
JdbcHelper.JDataStore	jdbc:borland:dslocal:\${JDBC.schema}
JdbcHelper.JDBC_ODBC_Bridge	jdbc:odbc:\${ODBC.datasource}
JdbcHelper.MaxDB	jdbc:sapdb://\${JDBC.host localhost}: \${JDBC.port 7210}/\${JDBC.schema}
JdbcHelper.McKoi	jdbc:mckoi://\${JDBC.host localhost}: \${JDBC.port 9157}/\${JDBC.schema}
JdbcHelper.Mimer	jdbc:mimer:\${mimer.protocol}:// \${JDBC.host localhost}:\${JDBC.port 1360}/\${JDBC.schema}
JdbcHelper.MySQL	jdbc:mysql://\${JDBC.host localhost}: \${JDBC.port 3306}/\${JDBC.schema}
JdbcHelper.Netezza	jdbc:netezza://\${JDBC.host localhost}:\${JDBC.port 5480}/ \${JDBC.schema}
JdbcHelper.Oracle_Thin	jdbc:oracle:thin:@\${JDBC.host localhost}:\${JDBC.port 1521}: \${oracle.sid ORCL}
JdbcHelper.Oracle_OCI	jdbc:oracle:oci:@\${JDBC.host localhost}:\${JDBC.port 1521}: \${oracle.sid ORCL}
JdbcHelper.Oracle_DataDirect	jdbc:datadirect:oracle:// \${JDBC.host localhost}: \${JDBC.port 1521}; ServiceName= \${oracle.servicename ORCL}
JdbcHelper.Pervasive	jdbc:pervasive://\${JDBC.host localhost}:\${JDBC.port }/ \${JDBC.schema}

<code>JdbcHelper.Pointbase_Embedded</code>	<code>jdbc:pointbase:embedded: \${JDBC.schema},database.home= \${pointbase.home} ,create= \${pointbase.create false}</code>
<code>JdbcHelper.Pointbase_Server</code>	<code>jdbc:pointbase:server://\${JDBC.host localhost}:\${JDBC.port 9092}/ \${JDBC.schema} ,database.home= \${pointbase.home},create= \${pointbase.create false}</code>
<code>JdbcHelper.PostgreSQL</code>	<code>jdbc:postgresql://\${JDBC.host localhost}:\${JDBC.port 5432}/ \${JDBC.schema}</code>
<code>JdbcHelper.Progress</code>	<code>jdbc:jdbcProgress:T:\${JDBC.host localhost}:\${JDBC.port 2055}: \${JDBC.schema}</code>
<code>JdbcHelper.SQL_Server_DataDirect</code>	<code>jdbc:datadirect:sqlserver:// \${JDBC.host localhost}:\${JDBC.port 1433}; DatabaseName=\${JDBC.schema Northwind}</code>
<code>JdbcHelper.SQL_Server_jTDS</code>	<code>jdbc:jtds:sqlserver:// \${JDBC.host localhost}: \${JDBC.port 1433}; DatabaseName= \${JDBC.schema Northwind};domain= \${sqlserver.domain}</code>
<code>JdbcHelper.SQL_Server_MS_Driver</code>	<code>jdbc:microsoft:sqlserver:// \${JDBC.host localhost}:\${JDBC.port 1433}; DatabaseName=\${JDBC.schema Northwind}</code>
<code>JdbcHelper.SQL_Server_2005_MS_Driver</code>	<code>jdbc:sqlserver://\${JDBC.host localhost}:\${JDBC.port 1433};DatabaseName=\${JDBC.schema Northwind}</code>
<code>JdbcHelper.Sybase_ASE_jTDS</code>	<code>jdbc:jtds:sybase://\${JDBC.host localhost}:\${JDBC.port 5000};DatabaseName=\${JDBC.schema}</code>
<code>JdbcHelper.Sybase_ASE_JConnect</code>	<code>jdbc:sybase:Tds:\${JDBC.host localhost}:\${JDBC.port 5000}/ \${JDBC.schema}</code>
<code>JdbcHelper.Sybase_SQL_Anywhere_JConne</code>	<code>jdbc:sybase:Tds:\${JDBC.host localhost}:\${JDBC.port 2638}/ \${JDBC.schema}</code>
<code>JdbcHelper.Sybase_DataDirect</code>	<code>jdbc:datadirect:sybase:// \${JDBC.host localhost}:\${JDBC.port 2048};ServiceName=\${JDBC.schema}</code>

7.1.3 Usage Example

Using the helper is simpler than explain how it works, let's have a look at the code below that configures the `PooledDataSourceProvider` for *MySQL*:

```
final Properties myBatisProperties = new Properties();
myBatisProperties.setProperty("mybatis.environment.id", "test");
// configure the database host
myBatisProperties.setProperty("JDBC.host", "contacts.acme.db");
// configure the database port
myBatisProperties.setProperty("JDBC.port", "6969");
// configure the database schema
myBatisProperties.setProperty("JDBC.schema", "contacts_test");
myBatisProperties.setProperty("JDBC.username", "fooTest");
myBatisProperties.setProperty("JDBC.password", "barPWD");
myBatisProperties.setProperty("JDBC.autoCommit", "false");
// binds the JDBC connection URL and the Driver class name for MySQL
Injector injector = Guice.createInjector(JdbcHelper.MySQL,
    new MyBatisModule() {
        public void configure(Binder binder) {
            setDataSourceProviderType(PooledDataSourceProvider.class);
            addSimpleAliases(Contact.class);
            addMapperClasses(ContactMapper.class);
        }
    },
    new Module() {
        public void configure(Binder binder) {
            /* binds the properties configuration; JDBC.host,
             * JDBC.port and JDBC.schema will be used to be
             * replaced in the URL Pattern
             */
            Names.bindProperties(binder, myBatisProperties);
        }
    });
```

8 Samples

8.1 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 its method is called and committed when the method ends without throwing a unchecked exception.

```
public class FooServiceMapperImpl implements FooService {  
  
    @Inject  
    private UserMapper userMapper;  
  
    @Transactional  
    public User doSomeBusinessStuff(String userId) {  
        return this.userMapper.getUser(userId);  
    }  
  
}
```

Notice that transactional behaviour is configured with the `@Transactional` annotation.

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 its interface `UserDao.java` and its implementation `UserDaoImpl.java`

The database access layer has been implemented using a mapper and a Dao that internally uses a `SqlSession`.

Sample test	Description
<code>SampleBasicTest</code>	Shows you the recommended and simplest configuration based on a mapper.
<code>SampleSqlSessionTest</code>	Shows how to hand code a DAO using a managed <code>SqlSession</code>

Sample test classes

Please have a look and run the sample code to see MyBatis-Guice in action.