

MaduraUtils

User Guide



Table of Contents

1.Change Log.....	5
2.References.....	6
3.Overview.....	7
4.Locking and Database.....	8
4.1.SimpleLock.....	8
4.2.SQLLock.....	8
4.3.Other Locking Mechanisms.....	9
5.Two Phase Commit support.....	10
6.Asserts.....	11
A.License.....	12
B.Release Notes.....	13

1. Change Log

Author	Date	Comment
rogerparkinson	2016-08-13	Updated the release notes
rogerparkinson	2015-12-02	moved utils into madura-objects-parent and upgraded Spring and Hibernate to latest versions Also tidied up more of the managed dependencies. Build, including tests, works.

2. References

- [1] [Spring Framework](#)
- [2] [slf4j](#)
- [3] [Apache Licence 2.0](#)

3. Overview

The MaduraUtils project is a catch-all project for various miscellaneous features required by other Madura projects (or anyone else) but which aren't big enough to justify their own project.

- Locking: this provides a factory and template for using `java.util.concurrent.locks.Lock` implementations as well as two of those implementations. We re-interpret the `Lock` interface here because the original intention for that interface seems to be more about locking memory objects such as lists and queues, across threads. This re-interpretation is about pessimistic locks of abstract items that *might* be locked across multiple threads and multiple JVMs, depending on the implementation.
- Parser is a set of parsing tools which can be extended to support a specific grammar fairly simply.
- Schemaparser is a reader of XSD files which creates an object structure which can be easily queried for class names and fields etc.

4. Locking and Database

The Madura Utils project supplies the framework for a locking system, but the implementation of the locks are your choice.

To take out a lock you use the lock factory which is configured like this:

```
<bean id="lockFactory"
  class="nz.co.senanque.locking.simple.SimpleLockFactory" />
```

The lock factory is injected into the relevant objects and called like this:

```
Lock lock = getLockFactory().getWrappedLock(ObjectSpecificString,
  LockType.WRITE, comment);
```

The lock is actually a `java.util.concurrent.locks.Lock`, and you can use it to take out a lock on the `ObjectSpecificString`, which can represent any abstraction you can think of. It is helpful to use a template like this:

```
LockTemplate lockTemplate = new LockTemplate(lock, new LockAction() {

  public void doAction() {

    // some code that must happen in a lock
  }});
if (!lockTemplate.doAction()) {
  throw new RuntimeException("Failed to get a lock");
}
```

The template ensures that the lock is released when it is done no matter what happens. There is a variation that allows a timeout on the internal `trylock` call. It can accept a single lock as shown, or it can accept a list of locks which it locks in the order they appear in the list, and unlocks in the reverse order.

The factory configured earlier delivers a class with an interface called `nz.co.senanque.locking.LockFactory` and that is where it can be extended to use any locking mechanism you like. These are the ones available:

4.1. SimpleLock

This is the example already shown. It uses a memory array to manage the locks so it is not suitable to lock across different JVMs. This one is mostly used for testing since it needs no extra configuration. It does expose the lock table with JMX so that you can cancel locks manually on the fly. Exactly how much this really is remains to be seen.

4.2. SQLLock

This one keeps the locks on a database table so it requires a database and it does work across JVMs. This is the table definition required (for Oracle):

```
CREATE TABLE SQL_LOCK
(
  lockName  VARCHAR(255) NOT NULL,
  ownerName VARCHAR(100) NOT NULL,
  started   VARCHAR(255) NOT NULL,
  comments  VARCHAR(255),
  hostAddress VARCHAR(100) NOT NULL,
  CONSTRAINT sql_lock_name PRIMARY KEY (lockName)
);
```


The configuration looks like this:

```
<bean id="lockFactory" class="nz.co.senanque.locking.sql.SQLLockFactory">
  <property name="datasource" ref="datasource"/>
  <property name="prefix" value="" />
  <property name="maxRetries" value="-1" />
  <property name="sleepTime" value="1000" />
</bean>
```

The datasource must be a JDBC datasource and the other properties are optional, with the default values shown.

The prefix is used when you have multiple JVMs running on the same server (with the same ip address), setting different prefixes on each allows the lock manager to know which JVM owns which lock.

The maxRetries and sleepTime defines how many times to keep retrying the lock if we are locked out (-1 means infinite) and the sleep time is how long to wait between each try in milliseconds.

4.3. Other Locking Mechanisms

The obvious one to implement is Hazelcast which supports the `java.util.concurrent.locks.Lock` interface already so a simple factory that delivers a Hazelcast implementation would be easy to build. Terracotta can manage distributed locks as well.

5. Two Phase Commit support

Hibernate requires a support class to enable two phase commits under JTA using Atomikos. The resulting class is pretty trivial but it enables Spring to get the wiring right. The dependencies for this are only Hibernate and in this project it is marked as 'provided' in maven so you don't get it if you don't want it.

6. Asserts

The `MaduraAsserts` class roughly mimics JUnit's asserts but doesn't need the JUnit dependency in your production code. The asserts take optional format messages and even `RuntimeExceptions` if you want.

A. License

The code specific to MaduraUtils is licensed under the Apache Licence 2.0 [\[3\]](#).
The dependent products have compatible licenses as specified in their pom files.

B. Release Notes

3.0.1

Improved interface to schema parser. Specifically added a `traverse(visitor)` pattern and supporting classes to generate an example jdom document from the XSD.

3.0.0

Released to align version with Madura Objects.

1.0.0

Added asserts.

Some minor formatting.

0.0.3

Adjusted parser behaviour to support Eclipse plugin.

0.0.2

Upgraded Spring dependency.

0.1

Initial version.