Fixing liquibase logging (in Spring) with SLF4J and Log4J

by Benny Bottema - Sunday, December 29, 2013

http://www.bennybottema.com/2013/12/29/fixing-liquibase-logging-in-spring/

It took me awhile to figure it out, but I figured out how to run the liquibase logging through slf4j/log4j/logback etc, for Liquibase 3.0.8. There's an easy way and a hard way. The easy way is just the hard way pre-packaged as a jar for you. You'll understand.

The easy way

Drop in a jar called liquibase-slf4j, by Matt Bertolini, and configure its class logging through slf4j instead. In my case I'm using slf4j-log4j12 on top of that so I configure everything log4j style (make sure you have log4j on your classpath!).

```
<!-- your own standard logging dependencies -->
<dependency>
 <groupId>org.slf4j</groupId>
 <artifactId>slf4j-api</artifactId>
 <version>1.7.5</version>
</dependency>
<dependency>
 <groupId>org.slf4j</groupId>
 <artifactId>slf4j-log4j12</artifactId>
 <version>1.7.5</version>
</dependency>

<!-- special dependency to fix liquibase&apos;s logging fetish -->
<dependency>
 <groupId>com.mattbertolini</groupId>
 <artifactId>liquibase-slf4j</artifactId>
 <version>1.2.1</version>
</dependency>
```

Then to actually configure its output in log4j:

```
<category name="liquibase">
 <priority value="debug" />
</category>
```

There's also logging for "liquibase.integration.spring.SpringLiquibase" but that is only warn() and error(), so you should see that always anyway

The hard way

The difficulty in configuring Liquibase logging lies in that Liquibase invented its own 'independent' logging framework. The rationale is that in doing so it doesn't have 'an extra dependency', even though slf4j is meant to cut implementation dependencies so you're free to use whatever you want. The fact that Liquibase's logging approach changed drastically over several versions didn't help; the core Liquibase developer, Nathan Voxland, constantly reminds people he still needs to fix the logging plugins and tells people to do some hokey pokey magic code to make it work (and not every time the same hokey pokey magic code, mind you).

Then there's this little gem of SpringLiquibase javadoc in case you're using Liquibase as a Spring bean. It mentions how you can configure an *sqlOutputDir*, but that property doesn't exist anymore. Obsolete javadoc and no way to know how to do it now but to delve into the code. Oh and btw, Liquibase is doing its own magic generic class lookup with custom classloading and whatnot, so no, the code did not tell me properly what to do. But, we're getting very close to the solution now…

Finally, I found that this specific instruction to makes the magic happen with the current latest version 3.0.8:

> Basically you need to create a new class that extends liquibase.logging.core.AbstractLogger and overrides the getPriority method to return larger than 1 and the debug(), info() etc. methods. If you make the class in a sub-package of liquibase.ext it will automaticaly be picked up by liquibase and used.

So. A class extending AbstractLogger that *has* to be in a package called "liquibase.ext.logging". Nice! I love it! Nothing says self-explanatory package structure like a random plugin extension folder for liquibase with one class to circumvent its own logging fetish. Awesome also that if they do change any of this, this approach will silently fail and fall back on Liquibase's own default logger.

```
package liquibase.ext.logging;

import liquibase.changelog.ChangeSet;
import liquibase.changelog.DatabaseChangeLog;
import liquibase.logging.core.AbstractLogger;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```
/**
 * Stupid class, to enable liquibase logging.
 *

 * Liquibase finds this class by itself by doing a custom component sc
an (they though sl4fj wasn't generic enough).
 */
public class LiquibaseLogger extends AbstractLogger {
 private static final Logger LOGGER = LoggerFactory.getLogger(Liquibas
eLogger.class);
 private String name = "";

 @Override
 public void setName(String name) {
  this.name = name;
 }

 @Override
 public void severe(String message) {
  LOGGER.error("{} {}", name, message);
 }

 @Override
 public void severe(String message, Throwable e) {
  LOGGER.error("{} {}", name, message, e);
 }

 @Override
 public void warning(String message) {
  LOGGER.warn("{} {}", name, message);
 }

 @Override
 public void warning(String message, Throwable e) {
  LOGGER.warn("{} {}", name, message, e);
 }

 @Override
 public void info(String message) {
  LOGGER.info("{} {}", name, message);
 }

 @Override
 public void info(String message, Throwable e) {
  LOGGER.info("{} {}", name, message, e);
 }
```

```
 @Override
 public void debug(String message) {
  LOGGER.debug("{} {}", name, message);
 }

 @Override
 public void debug(String message, Throwable e) {
  LOGGER.debug("{} {}", message, e);
 }

 @Override
 public void setLogLevel(String logLevel, String logFile) {
 }

 @Override
 public void setChangeLog(DatabaseChangeLog databaseChangeLog) {
 }

 @Override
 public void setChangeSet(ChangeSet changeSet) {
 }

 @Override
 public int getPriority() {
  return Integer.MAX_VALUE;
 }
}
```

And this is exactly what Matt Bertolini did in liquibase-slf4j.
