Improved Java Logging with Mapped Diagnostic Context (MDC)

Last updated: May 11, 2024

Written by: baeldung (https://www.baeldung.com/author/baeldung)

Logging (https://www.baeldung.com/category/logging)

Log4j2 (https://www.baeldung.com/tag/log4j2)

Logback (https://www.baeldung.com/tag/logback)

SLF4J (https://www.baeldung.com/tag/slf4j)

1. Overview

In this tutorial, we will explore the use of *Mapped Diagnostic Context* (MDC) to improve the application logging.

Mapped Diagnostic Context provides a way to enrich log messages with information that could be unavailable in the scope where the logging actually occurs but that can be indeed useful to better track the execution of the program.

Further reading:

Creating a Custom Log4j2 Appender (/log4j2-custom-appender)

Learn how to create a custom logging appender for Log4j2.
Read more (/log4j2-custom-appender) →

Java Logging with Nested Diagnostic Context (NDC) (/java-logging-ndc-log4j)

Distinguish log messages from different sources with the Nested Diagnostic Context.
Read more (/java-logging-ndc-log4j) →

Creating a Custom Logback Appender (/custom-logback-appender)

Learn how to implement a custom Logback appender.
Read more (/custom-logback-appender) →

2. Why Use MDC

Let's suppose we have to write software that transfers money.

We set up a *Transfer* class to represent some basic information — a unique transfer id and the name of the sender:

```java
public class Transfer {
    private String transactionId;
    private String sender;
    private Long amount;

    public Transfer(String transactionId, String sender, long amount) {
        this.transactionId = transactionId;
        this.sender = sender;
        this.amount = amount;
    }

    public String getSender() {
        return sender;
    }

    public String getTransactionId() {
        return transactionId;
    }

    public Long getAmount() {
        return amount;
    }
}
```

To perform the transfer, we need to use a service backed by a simple API:

```
public abstract class TransferService {

    public boolean transfer(long amount) {
        // connects to the remote service to actually transfer money
    }

    abstract protected void beforeTransfer(long amount);

    abstract protected void afterTransfer(long amount, boolean outcome);
}
```

The *beforeTransfer()* and *afterTransfer()* methods can be overridden to run custom code right before and right after the transfer completes.

We're going to leverage *beforeTransfer()* and *afterTransfer()* to **log some information about the transfer.**

Let's create the service implementation:

```
import org.apache.log4j.Logger;
import com.baeldung.mdc.TransferService;

public class Log4JTransferService extends TransferService {
    private Logger logger = Logger.getLogger(Log4JTransferService.class);

    @Override
    protected void beforeTransfer(long amount) {
        logger.info("Preparing to transfer " + amount + "$.");
    }

    @Override
    protected void afterTransfer(long amount, boolean outcome) {
        logger.info(
          "Has transfer of " + amount + "$ completed successfully ? " +
outcome + ".");
    }
}
```

The main issue to note here is that **when the log message is created, it is not possible to access the *Transfer* object** — only the amount is accessible, making it impossible to log either the transaction id or the sender.

Let's set up the usual *log4j.properties* file to log on the console:

```
log4j.appender.consoleAppender=org.apache.log4j.ConsoleAppender
log4j.appender.consoleAppender.layout=org.apache.log4j.PatternLayout
log4j.appender.consoleAppender.layout.ConversionPattern=%-4r [%t] %5p %c
%x - %m%n
log4j.rootLogger = TRACE, consoleAppender
```

Finally, we'll set up a small application that is able to run multiple transfers at the same time through an *ExecutorService*:

```
public class TransferDemo {

    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3);
        TransactionFactory transactionFactory = new TransactionFactory();
        for (int i = 0; i < 10; i++) {
            Transfer tx = transactionFactory.newInstance();
            Runnable task = new Log4JRunnable(tx);
            executor.submit(task);
        }
        executor.shutdown();
    }
}
```

Note that in order to use the *ExecutorService*, we need to wrap the execution of the *Log4JTransferService* in an adapter because *executor.submit()* expects a *Runnable*:

```
public class Log4JRunnable implements Runnable {
    private Transfer tx;

    public Log4JRunnable(Transfer tx) {
        this.tx = tx;
    }

    public void run() {
        log4jBusinessService.transfer(tx.getAmount());
    }
}
```

When we run our demo application that manages multiple transfers at the same time, we quickly see that **the log is not as useful as we would like it to be.**

It's complex to track the execution of each transfer because the only useful information being logged is the amount of money transferred and the name of the thread that is running that particular transfer.

What's more, it's impossible to distinguish between two different transactions of the same amount run by the same thread because the related log lines look essentially the same:

```
...
519  [pool-1-thread-3]  INFO Log4JBusinessService
  - Preparing to transfer 1393$.
911  [pool-1-thread-2]  INFO Log4JBusinessService
  - Has transfer of 1065$ completed successfully ? true.
911  [pool-1-thread-2]  INFO Log4JBusinessService
  - Preparing to transfer 1189$.
989  [pool-1-thread-1]  INFO Log4JBusinessService
  - Has transfer of 1350$ completed successfully ? true.
989  [pool-1-thread-1]  INFO Log4JBusinessService
  - Preparing to transfer 1178$.
1245 [pool-1-thread-3]  INFO Log4JBusinessService
  - Has transfer of 1393$ completed successfully ? true.
1246 [pool-1-thread-3]  INFO Log4JBusinessService
  - Preparing to transfer 1133$.
1507 [pool-1-thread-2]  INFO Log4JBusinessService
  - Has transfer of 1189$ completed successfully ? true.
1508 [pool-1-thread-2]  INFO Log4JBusinessService
  - Preparing to transfer 1907$.
1639 [pool-1-thread-1]  INFO Log4JBusinessService
  - Has transfer of 1178$ completed successfully ? true.
1640 [pool-1-thread-1]  INFO Log4JBusinessService
  - Preparing to transfer 674$.
...
```

Luckily, *MDC* can help.

3. MDC in Log4j

MDC in Log4j allows us to fill a map-like structure with pieces of information that are accessible to the appender when the log message is actually written.

The MDC structure is internally attached to the executing thread in the same way a *ThreadLocal* variable would be.

Here's the high-level idea:

1. Fill the MDC with pieces of information that we want to make available to the appender
2. Then log a message
3. And finally clear the MDC

The pattern of the appender should be changed in order to retrieve the variables stored in the MDC.

So, let's change the code according to these guidelines:

```
import org.apache.log4j.MDC;

public class Log4JRunnable implements Runnable {
    private Transfer tx;
    private static Log4JTransferService log4jBusinessService = new
Log4JTransferService();

    public Log4JRunnable(Transfer tx) {
        this.tx = tx;
    }

    public void run() {
        MDC.put("transaction.id", tx.getTransactionId());
        MDC.put("transaction.owner", tx.getSender());
        log4jBusinessService.transfer(tx.getAmount());
        MDC.clear();
    }
}
```

MDC.put() is used to add a key and a corresponding value in the MDC, while *MDC.clear()* empties the MDC.

Let's now change the *log4j.properties* to print the information that we've just stored in the MDC.

It is enough to change the conversion pattern, using the *%X{}* placeholder for each entry contained in the MDC we want to be logged:

```
log4j.appender.consoleAppender.layout.ConversionPattern=
  %-4r [%t] %5p %c{1} %x - %m - tx.id=%X{transaction.id}
tx.owner=%X{transaction.owner}%n
```

Now if we run the application, we'll note that each line also carries the information about the transaction being processed, making it far easier for us to track the execution of the application:

```
638  [pool-1-thread-2]  INFO Log4JBusinessService
  - Has transfer of 1104$ completed successfully ? true. - tx.id=2
tx.owner=Marc
638  [pool-1-thread-2]  INFO Log4JBusinessService
  - Preparing to transfer 1685$. - tx.id=4 tx.owner=John
666  [pool-1-thread-1]  INFO Log4JBusinessService
  - Has transfer of 1985$ completed successfully ? true. - tx.id=1
tx.owner=Marc
666  [pool-1-thread-1]  INFO Log4JBusinessService
  - Preparing to transfer 958$. - tx.id=5 tx.owner=Susan
739  [pool-1-thread-3]  INFO Log4JBusinessService
  - Has transfer of 783$ completed successfully ? true. - tx.id=3
tx.owner=Samantha
739  [pool-1-thread-3]  INFO Log4JBusinessService
  - Preparing to transfer 1024$. - tx.id=6 tx.owner=John
1259 [pool-1-thread-2]  INFO Log4JBusinessService
  - Has transfer of 1685$ completed successfully ? false. - tx.id=4
tx.owner=John
1260 [pool-1-thread-2]  INFO Log4JBusinessService
  - Preparing to transfer 1667$. - tx.id=7 tx.owner=Marc
```

4. MDC in Log4j2

The very same feature is available in Log4j2 too, so let's see how to use it.

We'll first set up a *TransferService* subclass that logs using Log4j2:

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class Log4J2TransferService extends TransferService {
    private static final Logger logger = LogManager.getLogger();

    @Override
    protected void beforeTransfer(long amount) {
        logger.info("Preparing to transfer {}$.", amount);
    }

    @Override
    protected void afterTransfer(long amount, boolean outcome) {
        logger.info("Has transfer of {}$ completed successfully ? {}.", 
amount, outcome);
    }
}
```

Let's then change the code that uses the MDC, which is actually called *ThreadContext* in Log4j2:

```
import org.apache.log4j.MDC;

public class Log4J2Runnable implements Runnable {
    private final Transaction tx;
    private Log4J2BusinessService log4j2BusinessService = new 
Log4J2BusinessService();

    public Log4J2Runnable(Transaction tx) {
        this.tx = tx;
    }

    public void run() {
        ThreadContext.put("transaction.id", tx.getTransactionId());
        ThreadContext.put("transaction.owner", tx.getOwner());
        log4j2BusinessService.transfer(tx.getAmount());
        ThreadContext.clearAll();
    }
}
```

Again, *ThreadContext.put()* adds an entry in the MDC and *ThreadContext.clearAll()* removes all the existing entries.

We still miss the *log4j2.xml* file to configure the logging.

As we can note, the syntax to specify which MDC entries should be logged is the same as the one used in Log4j:

```
<Configuration status="INFO">
    <Appenders>
        <Console name="stdout" target="SYSTEM_OUT">
            <PatternLayout
              pattern="%-4r [%t] %5p %c{1} - %m -
tx.id=%X{transaction.id} tx.owner=%X{transaction.owner}%n" />
        </Console>
    </Appenders>
    <Loggers>
        <Logger name="com.baeldung.log4j2" level="TRACE" />
        <AsyncRoot level="DEBUG">
            <AppenderRef ref="stdout" />
        </AsyncRoot>
    </Loggers>
</Configuration>
```

Again, let's run the application, and we'll see the MDC information being printed in the log:

```
1119 [pool-1-thread-3]  INFO Log4J2BusinessService
  - Has transfer of 1198$ completed successfully ? true. - tx.id=3
tx.owner=Samantha
1120 [pool-1-thread-3]  INFO Log4J2BusinessService
  - Preparing to transfer 1723$. - tx.id=5 tx.owner=Samantha
1170 [pool-1-thread-2]  INFO Log4J2BusinessService
  - Has transfer of 701$ completed successfully ? true. - tx.id=2
tx.owner=Susan
1171 [pool-1-thread-2]  INFO Log4J2BusinessService
  - Preparing to transfer 1108$. - tx.id=6 tx.owner=Susan
1794 [pool-1-thread-1]  INFO Log4J2BusinessService
  - Has transfer of 645$ completed successfully ? true. - tx.id=4
tx.owner=Susan
```

5. MDC in SLF4J/Logback

MDC is available in SLF4J too, under the condition that it is supported by the underlying logging library.

Both Logback and Log4j support MDC, as we've just seen, so we need nothing special to use it with a standard set up.

Let's prepare the usual *TransferService* subclass, this time using the Simple Logging Facade for Java:

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

final class Slf4TransferService extends TransferService {
    private static final Logger logger = 
LoggerFactory.getLogger(Slf4TransferService.class);

    @Override
    protected void beforeTransfer(long amount) {
        logger.info("Preparing to transfer {}$.", amount);
    }

    @Override
    protected void afterTransfer(long amount, boolean outcome) {
        logger.info("Has transfer of {}$ completed successfully ? {}.", 
amount, outcome);
    }
}
```

Let's now use the SLF4J's flavor of MDC.

In this case, the syntax and semantics are the same as in log4j:

```
import org.slf4j.MDC;

public class Slf4jRunnable implements Runnable {
    private final Transaction tx;

    public Slf4jRunnable(Transaction tx) {
        this.tx = tx;
    }

    public void run() {
        MDC.put("transaction.id", tx.getTransactionId());
        MDC.put("transaction.owner", tx.getOwner());
        new Slf4TransferService().transfer(tx.getAmount());
        MDC.clear();
    }
}
```

We have to provide the Logback configuration file, *logback.xml*:

```
<configuration>
    <appender name="stdout" class="ch.qos.logback.core.ConsoleAppender">
        <encoder
class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
            <pattern>%-4r [%t] %5p %c{1} - %m - tx.id=%X{transaction.id}
tx.owner=%X{transaction.owner}%n</pattern>
        </encoder>
    </appender>
    <root level="TRACE">
        <appender-ref ref="stdout" />
    </root>
</configuration>
```

Again, we'll see that the information in the MDC is properly added to the logged messages, even though this information is not explicitly provided in the *log.info()* method:

```
1020 [pool-1-thread-3]  INFO c.b.m.s.Slf4jBusinessService
  - Has transfer of 1869$ completed successfully ? true. - tx.id=3
tx.owner=John
1021 [pool-1-thread-3]  INFO c.b.m.s.Slf4jBusinessService
  - Preparing to transfer 1303$. - tx.id=6 tx.owner=Samantha
1221 [pool-1-thread-1]  INFO c.b.m.s.Slf4jBusinessService
  - Has transfer of 1498$ completed successfully ? true. - tx.id=4
tx.owner=Marc
1221 [pool-1-thread-1]  INFO c.b.m.s.Slf4jBusinessService
  - Preparing to transfer 1528$. - tx.id=7 tx.owner=Samantha
1492 [pool-1-thread-2]  INFO c.b.m.s.Slf4jBusinessService
  - Has transfer of 1110$ completed successfully ? true. - tx.id=5
tx.owner=Samantha
1493 [pool-1-thread-2]  INFO c.b.m.s.Slf4jBusinessService
  - Preparing to transfer 644$. - tx.id=8 tx.owner=John
```

It is worth noting that if we set up the SLF4J backend to a logging system that does not support MDC, all the related invocations will simply be skipped without side effects.

6. MDC and Thread Pools

MDC implementations typically use *ThreadLocal*s to store the contextual information. That's an easy and reasonable way to achieve thread-safety.

However, we should be careful using MDC with thread pools.

Let's see how the combination of *ThreadLocal*-based MDCs and thread pools can be dangerous:

1. We get a thread from the thread pool.
2. Then we store some contextual information in MDC using *MDC.put()* or *ThreadContext.put()*.
3. We use this information in some logs, and somehow we forgot to clear the MDC context.
4. The borrowed thread comes back to the thread pool.
5. After a while, the application gets the same thread from the pool.
6. Since we didn't clean up the MDC last time, this thread still owns some data from the previous execution.

This may cause some unexpected inconsistencies between executions.

One way to prevent this is to always remember to clean up the MDC context at the end of each execution. This approach usually needs rigorous human supervision and is therefore error-prone.

Another approach is to use *ThreadPoolExecutor* hooks and perform necessary cleanups after each execution.

To do that, we can extend the *ThreadPoolExecutor* class and override the *afterExecute()* hook:

```
public class MdcAwareThreadPoolExecutor extends ThreadPoolExecutor {

    public MdcAwareThreadPoolExecutor(int corePoolSize,
      int maximumPoolSize,
      long keepAliveTime,
      TimeUnit unit,
      BlockingQueue<Runnable> workQueue,
      ThreadFactory threadFactory,
      RejectedExecutionHandler handler) {
        super(corePoolSize, maximumPoolSize, keepAliveTime, unit,
workQueue, threadFactory, handler);
    }

    @Override
    protected void afterExecute(Runnable r, Throwable t) {
        System.out.println("Cleaning the MDC context");
        MDC.clear();
        org.apache.log4j.MDC.clear();
        ThreadContext.clearAll();
    }
}
```

This way, the MDC cleanup would automatically happen after each normal or exceptional execution.

So, there is no need to do it manually:

```
@Override
public void run() {
    MDC.put("transaction.id", tx.getTransactionId());
    MDC.put("transaction.owner", tx.getSender());

    new Slf4TransferService().transfer(tx.getAmount());
}
```

Now we can re-write the same demo with our new executor implementation:

```
ExecutorService executor = new MdcAwareThreadPoolExecutor(3, 3, 0,
MINUTES,
  new LinkedBlockingQueue<>(), Thread::new, new AbortPolicy());

TransactionFactory transactionFactory = new TransactionFactory();

for (int i = 0; i < 10; i++) {
    Transfer tx = transactionFactory.newInstance();
    Runnable task = new Slf4jRunnable(tx);

    executor.submit(task);
}

executor.shutdown();
```

7. Conclusion

MDC has lots of applications, mainly in scenarios in which running several different threads causes interleaved log messages that would be otherwise hard to read.

And as we've seen, it's supported by three of the most widely used logging frameworks in Java.

As usual, the sources are available over on GitHub (https://github.com/eugenp/tutorials/tree/master/logging-modules/log-mdc).

THE COURSES PLATFORM (HTTPS://COURSES.BAELDUNG.COM)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON SERIES (/JACKSON)
APACHE HTTPCLIENT SERIES (/HTTPCLIENT-SERIES)
REST WITH SPRING SERIES (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE SERIES (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)
SPRING REACTIVE SERIES (/SPRING-REACTIVE-SERIES)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE FULL ARCHIVE (/FULL_ARCHIVE)
EDITORS (/EDITORS)
OUR PARTNERS (/PARTNERS/)
PARTNER WITH BAELDUNG (/PARTNERS/WORK-WITH-US)
EBOOKS (/LIBRARY/)
FAQ (HTTPS://WWW.BAELDUNG.COM/LIBRARY/FAQ)
BAELDUNG PRO (/MEMBERS/)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)