(/)

Improved Java Logging with Mapped Diagnostic Context (MDC)

Last modified: November 4, 2018

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

The basic idea of *Mapped Diagnostic Context* is to provide a way to enrich log messages with pieces of information that could be not available in the scope where the logging actually occurs, but that can be indeed useful to better track the execution of the program.

2. Why Use MDC

Let's start with an example. Let's suppose we have to write a 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:

```
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** – just the amount is accessible making 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
```

Let's finally set up a small application that is able to run multiple transfer at the same time through an *ExecutorService*:

```
1  public class TransferDemo {
2
3      public static void main(String[] args) {
4          ExecutorService executor = Executors.newFixedThreadPool(3);
5          TransactionFactory transactionFactory = new TransactionFactory();
6          for (int i = 0; i < 10; i++) {
7              Transfer tx = transactionFactory.newInstance();
8              Runnable task = new Log4JRunnable(tx);
9              executor.submit(task);
10         }
11         executor.shutdown();
12     }
13 }
```

We 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*:

```
1  public class Log4JRunnable implements Runnable {
2      private Transfer tx;
3
4      public Log4JRunnable(Transfer tx) {
5          this.tx = tx;
6      }
7
8      public void run() {
9          log4jBusinessService.transfer(tx.getAmount());
10     }
11 }
```

When we run our demo application that manages multiple transfers at the same time, we very quickly discover that **the log is not 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 executing that particular transfer.

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

Let's introduce *MDC*.

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.

And so, the high level idea is:

1. to 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 obviously changed in order to retrieve the variables stored in the MDC.

So let's then 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();
    }
}
```

Unsurprisingly *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 store in the MDC. It is enough to change the conversion pattern, using the *%X{}* placeholder for each entry contained in the MDC we would like 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 carries also the information about the transaction being processed making far more 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.

Let's firstly 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, that 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 than 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{transacti
        </Console>
    </Appenders>
    <Loggers>
        <Logger name="com.baeldung.log4j2" level="TRACE" />
        <AsyncRoot level="DEBUG">
            <AppenderRef ref="stdout" />
        </AsyncRoot>
    </Loggers>
</Configuration>
```

Again, let's execute 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 supports 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:

```
1  import org.slf4j.Logger;
2  import org.slf4j.LoggerFactory;
3
4  final class Slf4TransferService extends TransferService {
5      private static final Logger logger = LoggerFactory.getLogger(Slf4TransferService.class);
6
7      @Override
8      protected void beforeTransfer(long amount) {
9          logger.info("Preparing to transfer {}$.", amount);
10     }
11
12     @Override
13     protected void afterTransfer(long amount, boolean outcome) {
14         logger.info("Has transfer of {}$ completed successfully ? {}.", amount, outcome);
15     }
16 }
```

Let's now use the SLF4J's flavor of MDC. In this case, the syntax and semantic is the same than log4j's:

```
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
    </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 in case we set up the SLF4J back-end to a logging system that does not support MDC all the related invocations will be simply skipped without side effects.

6. Conclusion

MDC has lots of applications, mainly in scenarios in which execution of 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, you'll find the sources over on GitHub (https://github.com/eugenp/tutorials/tree/master/logging-modules/log-mdc).

newest oldest most voted

Guest

Nikunj Jain

Nice Post

+ 0 −

2 years ago

Guest

Grzegorz Piwowarek

Thanks, we are glad you like it!

+ 2 −

2 years ago

CATEGORIES

SPRING (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)
HTTP CLIENT (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (/ABOUT)

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

CONSULTING WORK (/CONSULTING)

META BAELDUNG (HTTP://META.BAELDUNG.COM/)

THE FULL ARCHIVE (/FULL_ARCHIVE)

WRITE FOR BAELDUNG (/CONTRIBUTION-GUIDELINES)

EDITORS (/EDITORS)

OUR PARTNERS (/PARTNERS)

ADVERTISE ON BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)

PRIVACY POLICY (/PRIVACY-POLICY)

COMPANY INFO (/BAELDUNG-COMPANY-INFO)

CONTACT (/CONTACT)