



About effective logging

by Dino Celovic and Nader Soukouti

Abstract

Powerful logging frameworks, such as Log4J, exist on the market. They provide implementation for common logging concepts, such as logging levels, dynamic level switching, handling of multiple destinations. Many technical articles about configuring and using those tools have been written, but few give practical advice on how to write effective logs that contain *useful information* for the *intended audience*. The aim of this article is *not* to explain how to use a particular logging framework. Instead, we will try to bring attention to the purpose of logging and clarify what an effective log should contain, in the context of an enterprise application.

What is logging?

Logging is a way your application speaks to the application developers, the support team and sometimes the product analysts. Logs are *not aimed at end-users*. The objective is to communicate useful information to *an engineer*, about important processing steps the application is undertaking at run-time. In the other hand, we would like the application to report all necessary information about difficulties it encounters (warnings, errors).

From the end-user's point of view, logs do not directly add value to the application. Indirectly, they do, because logs can help faster achieve higher levels of reliability and performance, thus participating to the effort to increase the comfort of use and the confidence in the application.

The language used for logging can therefore be more technical and less user friendly, given that the intended audience comes from a *technical background*.

NOTE: examples in this article are based on Log4J since that is the most commonly used logging framework in the Java world and is a result of community efforts. The reflection presented here is however applicable to other logging frameworks.

Why use logging?

Logs help the project/product team in their effort to achieve higher quality of service for their application. Some of the reasons why we would consult logs are:

- ✚ **Problem solving** – to locate origins and better *understand the context* in which errors occur. Logs can help to faster locate and fix problems.
- ✚ **Stats and usage patterns** – to build statistics on how, by whom and when your application is used. Common for web applications.
- ✚ **Resource Optimisation** – you can trace when, where and how your resources, such as database connections are created, used, released and destroyed.
- ✚ **Load balancing** – how requests are distributed across your cluster.

An important argument for logging is that in case of a defect, info necessary to understand the problem might already be found in the logs. Without logs, we will need to reproduce the problem (in a debugger...), which is not always possible.

Logs should be written to be useful. If you think logs would not add any value to your application (very unlikely though), or you log only because everyone logs, then do not log.

Note

Logging goes hand in hand with **exception handling**. They have complementary roles and together, they help achieve more effective problem handling and reporting. *Exceptions carry information* on where in your application a problem has occurred, along with the nature of the problem. Logging *reports* it, and gives further information about the run time application context in which the exception was thrown.

Who is responsible for writing logs?

Logging API is used by **application developers** when the application is written. Developer needs to understand the purpose of logging in the context of the given application, and use *logging policy* and his *common sense* to decide what and how to log.

The **logging policy** needs to be clarified and communicated in early project phases, *before* the development starts. The responsibility of specifying the clearest possible logging strategy is up to the *application analysts* and the *architecture team* together. You can also have constraints imposed by the *production environment*.

Logging policy should clarify:

- ✚ What kind of information to log, and at which logging level. Note that this is application specific.
- ✚ Who logs: which software components in your application architecture can, and which should not log.
- ✚ How to *format* log messages.
- ✚ Clarify *technical concerns* about performance and other constraints imposed on your application.
- ✚ Specify *coding standards* as far as logging is concerned.

Think about logging from the start. It is unlikely that you will come back to your code later in order to improve logging. And even if you do so, you wouldn't have taken full advantage of logging from the beginning, which is probably where you need it the most.

To recap...:

- ✚ Think about logging before you start coding.
- ✚ Application analysts and the project architecture team draw a logging policy.
- ✚ Developers understand the logging policy and use their common sense to log effectively.
- ✚ Production environment can impose certain constraints. Those would probably be more of a technical nature.

Basic logging concepts: Logging levels

All logging frameworks provide support for *logging levels*, sometimes referred to as *severity*. We find that "severity" is not the most appropriate term as it implies certain level of "gravity". Logging is not only used when things go wrong - it could be purely informational. In this article we use the term "Logging level".

In Log4J for example we have the following logging level *hierarchy*:

- ✚ FATAL
- ✚ ERROR
- ✚ WARNING
- ✚ INFO
- ✚ DEBUG

In the `java.util.logging` framework we find SEVERE, WARNING, INFO, CONFIG, FINE, FINER and FINEST.

Note

These logging levels have *no inherent meaning* - they are what you decide to make of them. There are however few commonly accepted usage patterns that we will try to present in this article. It is then up to each project team to compile its own logging policy to better suite their needs.

We will base our further reflection on the Log4J hierarchy, since that is the most commonly used one in the Java community. Same principles can be applied to other logging frameworks.

At run time, the application is usually configured to log *up to* a certain level. If WARNING is selected, FATAL, ERROR and WARNING logs will be generated. Logging frameworks also offer the possibility to dynamically activate/deactivate logging levels; it is possible to set different logging levels for different components of your application; some allow to selectively switch off a specific logging level; or to redirect specific logging levels to different destinations... Refer to your logging framework documentation for more detail.

In different project phases, different logging levels would be activated. This can also give an indication on what should be logged at which logging level.

Note

Logging levels are complementary to each other: there is no need to log in DEBUG what has been logged in INFO.

FATAL: System Unusable

Context

A problem that makes the *system unusable* has occurred. As a consequence, the application would probably shut down or fail over. Without appropriate logs we would have difficulties to find out what has happened.

Expectations

At this level we would log information that would help us understand why and how our system has reached such a condition.

Examples

- ✚ One of core application services, such as calculation engine or workflow service for example, does not start up or crashes. System is unusable without those services.
- ✚ Your application runs out of memory for some reason. An emergency procedure of saving live data is triggered before the system starts the failover procedure.
- ✚ Database is unavailable. No much point to carry on.

ERROR: Limited Functionality

Context

An error that prevents the application from *completing the given task* has occurred. The same task might succeed or not by trying again later. In any case, the system carries on working, *possibly with limited functionality*. An alert needs to be triggered and support is needed immediately.

Expectations

By analysing the application logs, together with related application data, support team should be able to localise, understand and fix/escalate the problem.

Examples

- ✚ The flight booking application fails to complete a booking process for a customer because the AmEx online payment service is unavailable. The VISA payments might be working... By trying again later, the same process might succeed. With all respect to AmEx...
- ✚ In a pension fund management application, an inconsistency is encountered in a pensioner's data. As a result, the calculation engine fails to calculate the pension for the given pensioner. The reason could be a defect in the data migration procedure that was run during the night, or a bug elsewhere in the application, which left pensioner's data in an inconsistent state. The same process works however for other pensioners ...

FATAL or ERROR?

The decision between logging at FATAL or ERROR level could be ambiguous. The difficulty is in the fact that the decision needs to be taken at construction time (when code is written), whereas the problem is dependent on a real run time situation. If for example AmEx payments do not go through, the reason could be a general failure of all payments (FATAL, system unusable), or a failure of AmEx payments only (ERROR, system in limited functionality).

In such cases, I would take the *optimistic approach* and log ERROR. Logging is closely related to exception handling, and as a consequence, with emergency procedures that are triggered in case of failures. Optimistic approach avoids triggering of system rescue procedures if the defect is not FATAL. In our case, even if the cause was a general failure with payments (FATAL), the support would start getting frequent ERROR alerts and quickly realise that an emergency procedure needs to be triggered.

Note

This ambiguity is however avoidable with a clean layered architecture and a good exception handling. If, in the above example, the *communication layer* is well designed, it would make difference between a communication failure with VISA only, and a general communication failure. It will throw the appropriate exception in consequence. As a result, the exception handling logic will decide how serious the problem is, report the problem appropriately, and issue the suitable alert.

WARNING: Limited Capacity

Context

The system works, but due to a *non critical* error the request was *processed with difficulties* (had to wait for resource to become available), or only *partially processed*. The fact that the request was only partially processed is not business critical, because in the given case, manual or automated procedures have been designed in order to complete unfinished process later.

Expectations

By analysing application logs we will try to find out why, in which context and how often such difficulties occur. The log should help us improve the resource availability and the overall quality of service.

Examples

- ✚ Banking application needs to deposit £100 on a customer's account and then produce a confirmation letter. The first step completes, but the document generation service is too busy or unavailable. This is not critical to the process as the document generation can be scheduled for asynchronous processing during the night. The confirmation letter will be sent the following day...

A WARNING could say:

Log

```
[2005 Jan 14, 14:57] [foo.DocumentBuilder.java:74] [WARN] Failed to generate document for customer. Document generation has been scheduled for asynchronous processing; customerRef=CU5427, documentType=DOC0056, schedulerJobRef=JOB78889987
```

- In a security critical banking application you might WARN about too many attempts to log onto an account with a wrong password...

INFO

Context

The system works normally. This level traces *major* processing steps the application is undertaking in order to complete required tasks.

If we think of our application as an aggregation of services that collaborate together in order to satisfy a request, then those services would log in INFO the important processing steps they are undertaking. System components could also inform us about status changes in their life time (service starts up, connection pool invalidated, ...).

Expectations

Logs at this level will help us understand the context in which an error or a warning has occurred.

Note

INFO and DEBUG logging levels are complementary. They will both be used during debugging and testing phases, and once the system (or a component) is "stable", debug might be switched off. This obviously does not mean that the application is clean of defects, so we should make sure the INFO level still logs enough to help us faster eradicate them.

Examples

Following type of events could be logged in INFO:

- Banking application creates a new account.

Log

```
[2005 Jan 14, 14:57] [foo.AccountMgr.java:74] [INFO] Creating new bank account; customerRef="C24435", accType="SAVINGS", accRef="ACC455776", openingBalance=100.00
```

- Payment manager requests validation of a credit card payment to a remote payment system.
- Workflow engine starts or completes a process.
- Calculation service starts up.
- Web application creates a new user session.
- On-line flight booking application has locked the seat while the payment is being validated.
- ...

Again, *there's no rule of thumb*. In one application the payment validation might be an important processing step, whereas in another it might not be. You know your application the best.

DEBUG

Context

The system works normally. If we think that additional information would be useful in case we were chasing the cause of a defect, we would log it at this level. This makes DEBUG level complementary to INFO.

Note

INFO logs **inform** us about important processing steps. DEBUG logs **describe** them in further detail. There is a kind of *what* and *how* relation between the two.

Expectations

DEBUG logs would help us to faster eradicate problems during application development and testing phases. They can also be used in a "mature" application when hunting a specific, 'hard to find' defect.

The extra information logged at this level should allow us to do so, but shouldn't bring our application to its knees in terms of performance. Personally, I would leave the DEBUG level active all through development and testing phases of a component, and then lower logging to INFO forever. When an application considerably slows down with DEBUG activated, the problem is often in the misuse of logging.

In any case, we should only switch to lower logging levels once the confidence in our application or a component reaches a *satisfactory* level and we need to, and know that, performance will increase as a result. Sounds ambiguous, but *satisfactory* would mean:

Note

"I know that without DEBUG logs I will need greater effort in order to diagnose a problem, but considering that the MTBF (Mean Time Between Failures) of my application (or a component) has reached satisfactory levels, that extra effort is justified by the gain in performance due to switching to a lower logging level."

Do **not** use DEBUG logging for logs of type:

```
tmp=15.3333334
name="John"
entering getEmployeeInfo()
...
```

Even if you need that kind of traces while you are still in the "code and try" early development phase, you can use System.out, or any other mechanism you can find useful, but **remove** those lines **before** you check-in your code.

Warning

DEBUG logs stay in your application code *forever*. They should therefore be professional and useful even in later project or product phases.

What should be logged?

There is no miracle answer to the question. Use your common sense (again...). Think about the purpose of your logs and what you need them for. How would you like your logs to help you? Think about different logging levels described above.

An Internet application would probably report about usage habits of its users: what kind of content is read and by whom, the time content is accessed, length of a user session... A banking application would log start of a transfer operation, debiting of the source account, crediting of the destination account, end of transfer..

In any case, all types of applications would probably like to report errors at least, but keep in mind that it is probably insufficient. An error always occurs in a particular usage context and logs could help you understand that context. That's where INFO and DEBUG come into play.

Technical Considerations

"Logging slows down my application, need to lower logging level/switch it off"...

Is your logging policy too verbose? Is it correctly applied? Are you using logging levels correctly? Have you configured your logging framework optimally? Are you logging to a remote machine, whereas during the development process you do not really need that functionality? Are you using your logging API the optimal way? See Coding considerations below.

Which components should log?

Good application design eases effective logging. If you have a clear *layered architecture*, and you have distributed *horizontal responsibilities* to your components, it will be more obvious what and where to log as well (see the class diagram in the example presented below). Few tips:

- ✚ Try to log errors on *service boundaries* only. A service has a functional contract with its customer. Part of that contract being exceptions, the service boundary should catch the exceptions, *log* them, and translate them to ones that are part of the service contract.
- ✚ Try to log info on *layer boundaries* only.
- ✚ Log in objects that contain logic only. Do not log in passive objects, and certainly not in those that are designed to carry data across tier boundaries: value objects, collections. Instead, override toString() method in objects that carry data and call it from objects that "do things".
- ✚ It seems useless to say, but do **not** log in exceptions. This is an example from a real world, wide scale distributed application. The exception was supposed to be the root for *all* "fatal" exceptions in the system, whatever that means...:

Code

```
public class FatalException extends Exception{

    /** CONSTRUCTOR */
    public FatalException(Exception rootCause){
        super(rootCause);
        logger.error(rootException);
    }
}
```

Note

It is important to remember that the "thrower" of the exception does not know how serious the problem is. The gravity depends on the business process in question. The "catcher" of the exception (service boundary) probably knows.

The thrower of the exception says: "I failed to complete the task you asked me to do, and here's the reason (Exception). Up to you to decide how serious that is...". This is why in the above example it does not make sense to call an exception "Fatal" - only the component (or layer) that catches the exception is able to decide whether it is fatal or not... And if a

component is not able to decide what to do with an exception, it should not catch it, but let it go. Higher layers will probably know.

The same stands for logging: the exceptions should be logged where you catch them, not where you throw them...

Wrapping the logging framework with proprietary API

The technique of wrapping a third party logging framework by a "simpler", proprietary one is often used. The argument is not to become dependent of a specific logging framework.

In early project phases we could think that we do not need all the functionality provided by the logging framework we have chosen. We then try to map it to a simpler interface. In other words, we hide functionality that we find unnecessary by wrapping it. Later on as the project grows, we start to complicate our wrapper more and more. In the best case, we end up with a wrapper that does almost the same as the logging framework: a proxy. In the worst case, we are stuck with design limitations of our wrapper, and have to refactor our code in order to improve logging. Or we just forget about effective logging...

What we are trying to say, is that the risk of imposing yourself limitations due to your wrapper design are far greater than the probability that one day you would move to a different logging framework; especially if you chose a widely accepted one.

Note

We advise **not** to wrap the logging framework, especially if you are using widely accepted frameworks such as Log4J. Unless you have a very good reason that we cannot think of...

Performance

Logging could slow down your application. Be reasonable with logging and try to configure your logging framework the optimal way to best suite your needs.

In a production environment you might have a requirement to redirect your logs to a remote server, in addition to writing a log file on the local disk. If performance is an issue, and you feel that you cannot lower the amount of logging, it might be acceptable to log more to the local log file, and send only lower logging levels to the remote machine.

Do you need a formatted time stamp for your DEBUG logs? Date formatting can be time consuming, so you can do it for lower logging levels only.

Another technique that can help in performance critical applications is to check the log level before logging. That might seem useless, but consider the example below:

Code

```
logger.debug("Calculating interests. accountRef=" + accRef + ", startDate=" + startDate + ", endDate=" + endDate);
```

The string argument will be evaluated *before* the debug method is called. If DEBUG level is switched off, the log will not be written because the debug method will verify it, but the string will be evaluated anyway, including formatting of two dates. If you do this often, that can be expensive, including the work of the garbage collector who will need to clean up all those unused strings.

Instead, you can do the following:

Code

```
if(logger.isDebugEnabled()){
    logger.debug("Calculating interests. accountRef=" + accRef + ", startDate=" + startDate + ", endDate=" + endDate);
}
```

In the latter case, the string will be evaluated only if it is needed. This however pollutes your application code, so you might choose to do it only for DEBUG logs, as the other logging levels will probably always be activated. Again, it depends on the application.

Code readability

Code that produces logs does not participate in application logic – if we removed it the application would still work the same way. There is a risk that imposing logging code decreases readability of the application code.

Few tips I use:

- 🔧 Do not log from anywhere... See the "Which components should log?" section above.
- 🔧 Even if my logging line is well over 80 characters long, I don't split it up on next lines. What I want to see clearly when I read my code is the application logic, not code that logs.
- 🔧 Try to put logs at the beginning and at the end of methods. Using private methods to implement bits of logic helps in that. In any case, private methods are a good practice to make your code more comprehensible, even if you are not logging.
- 🔧 If you check the logging level before you log, as we suggested in the paragraph above, it might not be necessary to do it for lower logging levels (ERROR, etc.).

toString()

Overwriting the `toString()` method of your objects can help to easier debug the application. You can then use `toString()` when you are logging:

```
Code
if(logger.isDebugEnabled()){
    logger.debug("Calculating interests. calculationRequest= " + calculationRequest;
}
```

If `toString()` method of the `CalculationRequest` class was overwritten, the code above would produce the following log:

```
Code
[2004 Jan 14, 14:57] [com/sanabel/bank/AccountMgr.java:74] [DEBUG] Calculating interests. calculationRequest= CalculationRequest:
accountRef=ACC1003, startDate=20040101, endDate=20041231.
```

This makes the code more readable.

Internationalisation

Internationalising log messages is usually not necessary. The intended audience is technical, so logging in the same language in which you write your Java doc for example is a reasonable choice.

The exception to that could be ERROR messages. You might have a requirement to catalogue all your error messages and identify them with a unique reference. This can help the support team to write a manual of actions to be taken when an error occurs.

An example to that are database exceptions for major databases: errors are identified by a unique key, and you can set your ODBC or JDBC driver to report errors in a language of your choice.

Log message format

A typical log message would look something like:

```
Code
[2004 Jan 14, 14:57] [com/sanabel/workflow/WorkflowMgr.java:74] [ERROR] Failed to instantiate workflow process.
processRef='bank.moneyTransfer'.
```

You can identify the following elements in the above message:

- 🚩 The time stamp.
- 🚩 The component that logged the message (file name and line number).
- 🚩 Logging level.
- 🚩 *Application message*. Described in a separate section below.

Logging frameworks allow you to configure the format and the elements of your log messages. In addition to ones presented above, commonly used message elements are logger name, category, exception (stack trace), name of the thread, etc. Consult you logging framework documentation for more details.

Note

There is no best logging format to use – configure your logging framework to log information that is useful to you, depending on the phase your project is in.

Remember that:

- 🚩 In *different project phases* you might need different logging formats.
 - Example: During debugging, it is useful to know the name of the java class and the line number which produced the log. The time stamp might not be important. Later in production, you might need the time stamp, but not the line number...
- 🚩 If you consider using a *log viewer* application to consult your log files, then format your log messages the way you would take full advantage of your viewer.
 - Example: You might not need to format the time stamp. Instead, you log the long representation of the time (Java way), and your log viewer will format it for you when you examine the log file. Thus at run time your application logs faster...

Application message

Application message is probably the most *delicate* element of the log message. That is *the only* part of the log message where the developer who writes the code intervenes. All other parts of the log message are configurable, but this one is not: it is fixed, compiled. Good application messages can make the whole difference, and that is why the logging policy and common sense are important.

Here in bold is an example of application message:

Log

```
[2004 Jan 14, 14:57] [com/sanabel/bank/AccountMgr.java:74] [INFO] Debiting account. accountRef="ACC1006", amount=500.00
```

Notice the structure of the message: first it says *what* the application is doing, and then gives more details about the activity (which process).

When we read the log file, we first need to *find* the message we are interested in, and once we have found it, get *more details* about it. Saying clearly what the message is about, at the beginning of the message, can help in that.

Note also that it is *not* enough to say only *what* the application is doing ("Debiting account."), especially when we do **loop** processing. If we are debiting five accounts in a loop, it is useful to log the account reference, which will indicate *where* in the loop we are.

Note

The application message is *written by the developer*. This is where the *clear logging policies* and the *common sense* come into play, in order to log useful information without being unnecessarily verbose.

Think about what information you give in the application messages, but also how you format it. With an unstructured message format it might take you five minutes to analyse 100 lines of logs, whereas with a structured format, it might take you 1 minute instead.

Note

If your brain spots formatting patterns in a log file, it might not need to *read all* 100 messages in order to find the information you are looking for. Patterns ease "speed reading".

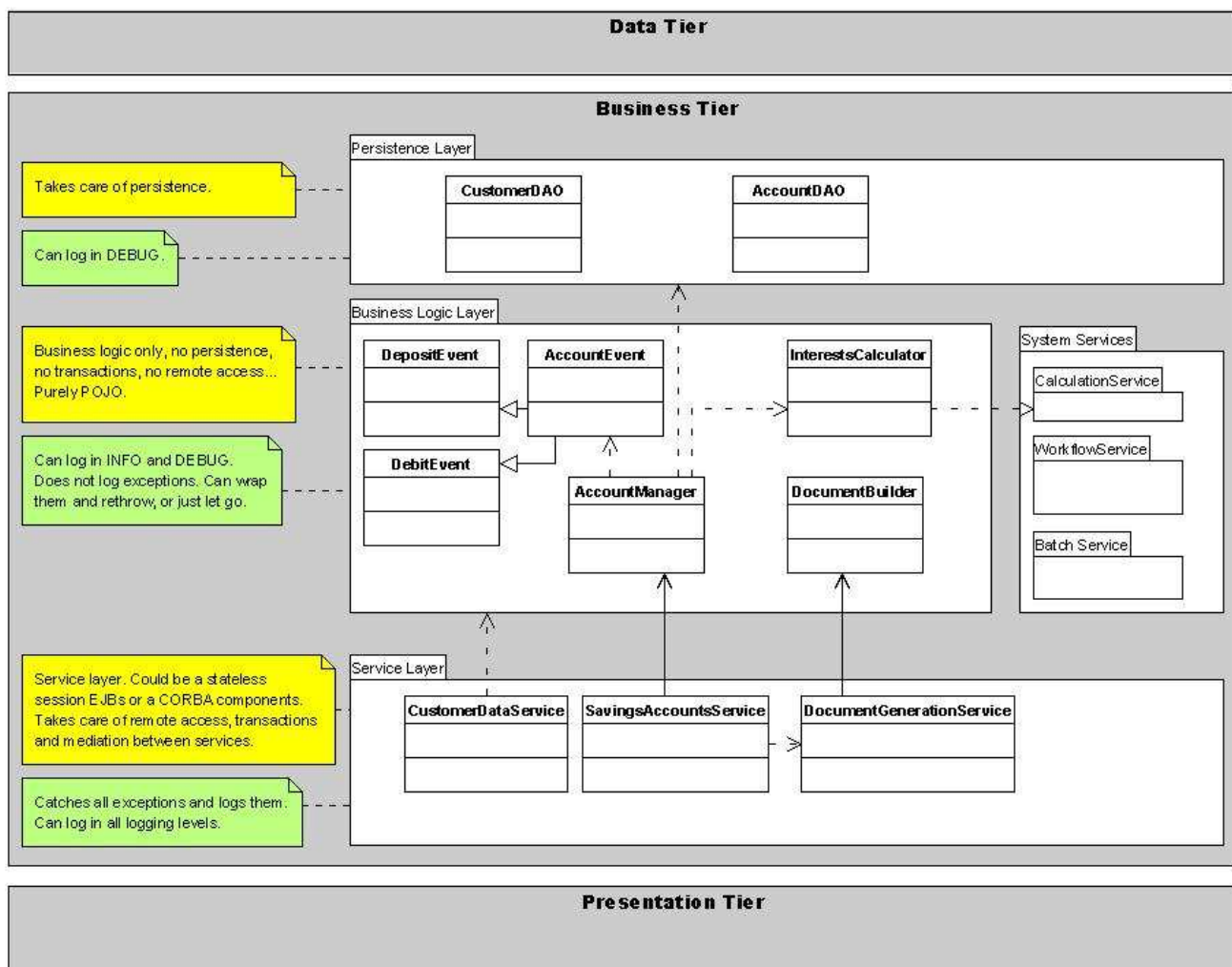
Example

"An example is better than precept." - Rajaram Ramachadran

The sample application

We will try to illustrate the concepts presented in this article on a simplified banking application. The application manages personal accounts for a high street bank. The application is able to perform standard banking operations, such as deposits, withdrawals, money transfers, interest calculations for savings accounts... It has a documents generation service able to format bank statements, confirmation letters, etc. Interests are calculated and deposited on the account in a nightly batch process that runs annually.

This is a very simplified package diagram showing the architecture layers. For the purpose of this example only few components that participate in the process that handles account interests are presented. Presentation tier has not been detailed.



The Process

The application is calculating yearly interests 2004 for all customer accounts in a batch process. The batch in question implements the following algorithm:

1. Find the list of all savings accounts,
2. Iterate through the list and in each iteration:
 - 2.1. Handle interests for the given account
 - 2.1.1. Calculate interests,
 - 2.1.2. Generate interests event,
 - 2.1.3. Register the event on the account,
 - 2.2. Generate confirmation letter for the customer, stating how much interests the account has gained in 2004.

3. Generate a batch report.
4. End of processing.

Bad log:

Log
<pre>[2005 Jan 14, 14:57] [foo.AccountMgr.java:74] [DEBUG] entering calculateInterests() [2005 Jan 14, 14:57] [foo.AccountMgr.java:95] [DEBUG] exiting calculateInterests() [2005 Jan 14, 14:57] [foo.AccountMgr.java:155] [DEBUG] entering registerEvent () [2005 Jan 14, 14:57] [foo.AccountMgr.java:155] [DEBUG] exiting registerEvent () [2005 Jan 14, 14:57] [foo.AccountMgr.java:74] [DEBUG] entering calculateInterests() [2005 Jan 14, 14:57] [foo.AccountMgr.java:95] [DEBUG] exiting calculateInterests() [2005 Jan 14, 14:57] [foo.AccountMgr.java:155] [DEBUG] entering registerEvent () [2005 Jan 14, 14:57] [foo.BatchMgr.java:15] [ERROR] java.lang.NullPointerException foo.AccountDAO.saveEvent(AccountDAO.java:167) at foo.AccountMgr.registerEvent(AccountMgr.java:45) at foo.AccountMgr.handleYearlyInterests (AccountMgr.java:116) at foo.AccountMgr.<init>(AccountMgr.java:27) at foo.BatchMgr.runBatch (BatchMgr.java:215) at foo.BatchMgr.main(BatchMgr.java:318)</pre>

The purpose of this log was to help us understand the context in which a potential error occurs. The INFO logs above do not give any extra information compared to the one we can find in the exception stack trace.

Remember that we are in a **loop**, iterating through the collection of all savings accounts. From the log above **we cannot work out which account has failed**. The above INFO logs are pointless...

A Better log 1:

Log
<pre>[2005 Jan 14, 14:55] [foo.BatchMgr.java:55] [INFO] Starting batch. batchRef="account.yearlyInterests", yearStarting="20040101" [2005 Jan 14, 14:57] [foo.AccountMgr.java:74] [INFO] Handling account interests. accountRef=ACC1001, yearStarting="20040101" [2005 Jan 14, 14:57] [foo.AccountMgr.java:74] [INFO] Handling account interests. accountRef=ACC1002, yearStarting="20040101" [2005 Jan 14, 14:57] [foo.BatchMgr.java:15] [ERROR] Failed to process batch. batchRef="account.yearlyInterests", rootCause= java.lang.NullPointerException foo.AccountDAO.saveEvent(AccountDAO.java:167) at foo.AccountMgr.registerEvent(AccountMgr.java:45) at foo.AccountMgr.handleYearlyInterests (AccountMgr.java:116) at foo.AccountMgr.<init>(AccountMgr.java:27) at foo.BatchMgr.runBatch (BatchMgr.java:215) at foo.BatchMgr.main(BatchMgr.java:318) ...</pre>

In this log we can see the progress of the accounts iterator. If an unexpected exception occurs we can work out that the account in question is the account with reference **ACC1002**.

From the stack trace we can see that step "2.1.3: Register event on the account" (saveEvent()) failed with NullPointerException. We don't know why, but we could probably find out by examining the code and the data of the account in question.

Adding few DEBUG logs might help...

A Better log 2:

Log
<pre>... [2005 Jan 14, 14:55] [foo.BatchMgr.java:55] [INFO] Starting batch. batchRef="account.yearlyInterests", yearStarting="20040101" [2005 Jan 14, 14:57] [foo.AccountMgr.java:74] [INFO] Handling account interests. accountRef="ACC1001", yearStarting="20040101" [2005 Jan 14, 14:57] [foo.InterestsCalculator.java:95] [DEBUG] Calculating interests. startDate=20040101, endDate=20041231 [2005 Jan 14, 14:57] [foo.AccountMgr.java:155] [DEBUG] Registering account event. eventType="credit.interests", amount=24.33, valueDate=20041231 [2005 Jan 14, 14:57] [foo.AccountMgr.java:74] [INFO] Handling account interests. accountRef="ACC1002", yearStarting="20040101" [2005 Jan 14, 14:57] [foo.InterestsCalculator.java:95] [DEBUG] Calculating interests. startDate=20040101, endDate=20041117</pre>

```
[2005 Jan 14, 14:57] [foo.AccountMgr.java:155] [DEBUG] Registering account event. eventType="credit.interests", amount=24.33,
valueDate=20041231
[2005 Jan 14, 14:57] [foo.BatchMgr.java:15] [ERROR] Failed to process batch. batchRef="account.yearlyInterests", rootCause=
java.lang.NullPointerException
    foo.AccountDAO.saveEvent(AccountDAO.java:167)
    at foo.AccountMgr.registerEvent(AccountMgr.java:45)
    at foo.AccountMgr.handleYearlyInterests (AccountMgr.java:116)
    at foo.AccountMgr.<init>( AccountMgr.java:27)
    at foo.BatchMgr.runBatch (BatchMgr.java:215)
    at foo.BatchMgr.main(BatchMgr.java:318)
. . .
```

The two additional DEBUG lines describe in further details the processing step that we were informed about by the INFO log. The processing failed for the account ACC1002. By further examining DEBUG logs we can notice that the interests on that account were calculated until the 17.11.2004, *not* 31.12! A very likely reason would be that the account was closed in November. From the second DEBUG log we can see that the value date of the event that we are trying to write to the account is 31.12.2004 - posterior to the account closure date. We are trying to register an event on the account, at a date where the account does not exist!

Conclusion: our application does not handle properly yearly interests for accounts closed before the end of year. We can also see that the interest calculation logic (step 2.1.1: Calculate interests) seems to be OK – it has taken into account the account closure date.

New requirements: Multithreading

Interests calculation for different accounts is completely independent from each other– more that one account could be processed in parallel. This is a good case for multithreading.

After refactoring the batch manager to be multithreaded, the accounts are not processed sequentially any more. As a consequence the logs can get mixed up.

This is what we could have:

Multithreaded log 1:

```
Log
...
[2005 Jan 14, 14:55] [foo.BatchMgr.java:55] [INFO] Starting batch. batchRef="account.yearlyInterests", yearStarting="20040101"
[2005 Jan 14, 14:57] [foo.AccountMgr.java:74] [INFO] Handling account interests. accountRef="ACC1001", yearStarting="20040101"
[2005 Jan 14, 14:57] [foo.AccountMgr.java:74] [INFO] Handling account interests. accountRef="ACC1003", yearStarting="20040101"
[2005 Jan 14, 14:57] [foo.InterestsCalculator.java:95] [DEBUG] Calculating interests. startDate=20040101, endDate=20041231
[2005 Jan 14, 14:57] [foo.AccountMgr.java:155] [DEBUG] Registering account event. eventType="credit.interests", amount=24.33,
valueDate=20041231
[2005 Jan 14, 14:57] [foo.AccountMgr.java:74] [INFO] Handling account interests. accountRef="ACC1002", yearStarting="20040101"
[2005 Jan 14, 14:57] [foo.InterestsCalculator.java:95] [DEBUG] Calculating interests. startDate=20040101, endDate=20041117
[2005 Jan 14, 14:57] [foo.AccountMgr.java:155] [DEBUG] Registering account event. eventType="credit.interests", amount=24.33,
valueDate=20041231
[2005 Jan 14, 14:57] [foo.BatchMgr.java:15] [ERROR] FAILED to process batch. batchRef="account.yearlyInterests", rootCause=
java.lang.NullPointerException
    foo.AccountDAO.saveEvent(AccountDAO.java:167)
    at foo.AccountMgr.registerEvent(AccountMgr.java:45)
    at foo.AccountMgr.handleYearlyInterests (AccountMgr.java:116)
    at foo.AccountMgr.<init>( AccountMgr.java:27)
    at foo.BatchMgr.runBatch (BatchMgr.java:215)
    at foo.BatchMgr.main(BatchMgr.java:318)
[2005 Jan 14, 14:57] [foo.AccountMgr.java:155] [DEBUG] Registering account event. eventType="credit.interests", amount=24.33,
valueDate=20041231
. . .
```

Now, we cannot workout which account has failed any more... DEBUG logs do not log the account reference because we were trying to be briefest possible, and we do not inform about the success of processing.

Few additional logs would help:

```
Log
...
[2005 Jan 14, 14:57] [foo.AccountMgr.java:74] [INFO] Handling account interests. accountRef="ACC1001", yearStarting="20040101"
```

```
[2005 Jan 14, 14:57] [foo.AccountMgr.java:74] [INFO] Handling account interests. accountRef="ACC1003", yearStarting="20040101"
[2005 Jan 14, 14:57] [foo.InterestsCalculator.java:95] [DEBUG] Calculating interests. accountRef="ACC1001", startDate=20040101,
endDate=20041231
[2005 Jan 14, 14:57] [foo.AccountMgr.java:155] [DEBUG] Registering account event. accountRef="ACC1001", eventType="credit.interests",
amount=24.33, valueDate=20041231
[2005 Jan 14, 14:57] [foo.AccountMgr.java:74] [INFO] Handling account interests. accountRef="ACC1002", yearStarting="20040101"
[2005 Jan 14, 14:57] [foo.AccountMgr.java:74] [INFO] SUCCESSfully processed interests. accountRef="ACC1001"
...

```

Other processes: reusing functionalities

In our case the interests are calculated in a yearly batch processing. But the interest calculation logic (step "2.1.1 Calculate interests") could be used elsewhere in the application. For example:

- ✚ We might have an interest simulator module, able to calculate interests up to any future date based on the current account situation. Even further, the customer could *simulate future payments* and see how much interest that would bring him... Almost the same processing steps could be applied as in our batch example, but without registering the interest event on the account.
- ✚ When an account is closed before the end of year, the interests might be calculated immediately and registered on the account. The only difference is in the calling code: instead of a batch, the calling process is "Close Account".

For the latter example, the resulting log would be:

```
Log
...
[2005 Jan 14, 14:57] [foo.AccountMgr.java:170] [INFO] Closing account. accountRef="ACC1023", closingDate="20041117"
[2005 Jan 14, 14:57] [foo.InterestsCalculator.java:95] [DEBUG] Calculating interests. accountRef="ACC1023", startDate=20040101,
endDate=20041117
[2005 Jan 14, 14:57] [foo.AccountMgr.java:155] [DEBUG] Registering account event. accountRef="ACC1023", eventType="credit.interests",
amount=24.33, valueDate=20041117
...

```

Note that only INFO logs differ from previous examples. This example is trying to illustrate how INFO logs are used to inform about *major* processing steps, whereas DEBUG logs describe them further.

Tolerance: WARNING or ERROR?

Our batch fails to calculate interests on one account. The batch carries on with processing of other accounts. As far as the batch is concerned, this is a WARNING, not an ERROR. The batch report will say which account has failed, and appropriate action will be taken to fix the problem.

The log:

```
Log
...
[2005 Jan 14, 14:55] [foo.BatchMgr.java:55] [INFO] Starting batch. batchRef="account.yearlyInterests", yearStarting="20040101"
[2005 Jan 14, 14:57] [foo.AccountMgr.java:74] [INFO] Handling account interests. accountRef="ACC1001", yearStarting="20040101"
[2005 Jan 14, 14:57] [foo.InterestsCalculator.java:95] [DEBUG] Calculating interests. startDate=20040101, endDate=20041231
[2005 Jan 14, 14:57] [foo.AccountMgr.java:155] [DEBUG] Registering account event. eventType="credit.interests", amount=24.33,
valueDate=20041231
[2005 Jan 14, 14:57] [foo.AccountMgr.java:74] [INFO] Handling account interests. accountRef="ACC1002", yearStarting="20040101"
[2005 Jan 14, 14:57] [foo.InterestsCalculator.java:95] [DEBUG] Calculating interests. startDate=20040101, endDate=20041117
[2005 Jan 14, 14:57] [foo.BatchMgr.java:15] [WARN] Failed to process interests for account. accountRef="ACC1002", rootCause=
java.lang.NullPointerException
    foo.AccountDAO.saveEvent(AccountDAO.java:167)
    at foo.AccountMgr.registerEvent(AccountMgr.java:45)
    at foo.AccountMgr.handleYearlyInterests (AccountMgr.java:116)
    at foo.AccountMgr.<init>( AccountMgr.java:27)
    at foo.BatchMgr.runBatch (BatchMgr.java:215)
    at foo.BatchMgr.main(BatchMgr.java:318)
[2005 Jan 14, 14:57] [foo.AccountMgr.java:74] [INFO] Handling account interests. accountRef="ACC1003", yearStarting="20040101"
[2005 Jan 14, 14:57] [foo.InterestsCalculator.java:95] [DEBUG] Calculating interests. startDate=20040101, endDate=20041117

```

```
[2005 Jan 14, 14:57] [foo.AccountMgr.java:155] [DEBUG] Registering account event. eventType="credit.interests", amount=24.33, valueDate=20041231
[2005 Jan 14, 14:57] [foo.BatchMgr.java:15] [WARN] Batch processing completed partially. batchRef="account.yearlyInterests", batchReportRef="BR200412316"
. . .
```

If the same processing step (interest calculation) fails in the process "Close Account", this might be considered as an ERROR.

This example shows the importance of placing the logging code to the right component.

Note

Catch exceptions and log them on service boundaries. Lower components do not have the context necessary to decide the right gravity for an exception. If the database throws DuplicateKeyException, the persistence layer will not be able to decide how severe that is. The gravity depends on the business process in question and the persistence layer does not have that context.

Conclusion

Every application has its own requirements in terms of logging. When used effectively, logging is an important aid to the effort of making robust and reliable applications. By clearly understanding *why* we are doing something, we can find a better solution for *how* to do it. We hope this article will help you to find the right reasons *why* your application should log.

About the authors

Dino Celovic and *Nader Soukouti* are senior software architects for enterprise applications at Sanabel Solutions in Geneva. *Dino* specialises in services such as workflow, calculation engines and batch management. He has previously worked on a variety of projects in the pension management, finance and insurance industries.

Nader has a long experience in building distributed enterprise applications in CORBA and J2EE. He has previously worked in domains of banking, pension fund management and telecoms and has co-authored the book "EJB 2.0 – Mise en Oeuvre" in French.