



9 Logging Sins in Your Java Applications

EUGEN PARASCHIV | MAY 30, 2017 |

DEVELOPER TIPS, TRICKS & RESOURCES

([HTTPS://STACKIFY.COM/DEVELOPERS/](https://stackify.com/developers/)). |

🗨️ LEAVE A COMMENT ([HTTPS://STACKIFY.COM/9-LOGGING-SINS-JAVA/#RESPOND](https://stackify.com/9-logging-sins-java/#respond)).

Logging runtime information in your Java application is critically useful for understanding the behavior of any app, especially in cases when you encounter unexpected scenarios, errors or just need track certain application events (<https://stackify.com/error-monitoring/>).

In a real-world production environment, you usually don't have the luxury of debugging. And so, logging files can be the only thing you have to go off of when attempting to diagnose an issue that's not easy to reproduce.



Done properly, log files can also save you a lot of time by providing clues into the cause of the problem, and into the state of the system at the time it happened. Also, logging can be useful for auditing purposes, gathering statistics, extracting business intelligence and a variety of other tasks.

Overall, logging is certainly a foundational practice that provides significant benefits during the lifetime of the application – so it can be tempting to start recording as much log data as you can.

However, improper use of logging can have significant drawbacks as well.

In the following sections, we'll take a look at some of the most common and most detrimental practices that you can run into when making use of logging in an application.

All examples and configuration are using the popular [log4j 2](https://logging.apache.org/log4j/2.x/) (<https://logging.apache.org/log4j/2.x/>). [Logback](#) ([/logging-logback/](#)) is another great option, also [well-supported by Stackify](https://docs.stackify.com/v1/docs/errors-and-logs-logback). (<https://docs.stackify.com/v1/docs/errors-and-logs-logback>).

9 Java Logging Problems & How To Avoid Them

1. Logging Sensitive Information

To start with, probably the most damaging logging practice brought on by the “log as much as possible just in case” approach is displaying sensitive information in the logs.

Most applications handle data that should remain private, such as user credentials or financial information. The danger of having this type of information logged into a plain text file is clear – log files will very likely

What's more, logging some categories of data, such as financial information, is also heavily regulated and can have serious legal implications.

The best way to avoid this is simply to make sure you never log this kind of sensitive information.

There are alternatives, such as encrypting the log files, but that generally makes these files a lot less usable overall, which is not ideal.

Before we move on, here's [a more comprehensive list](https://www.owasp.org/index.php/Logging_Cheat_Sheet#Data_to_exclude) (https://www.owasp.org/index.php/Logging_Cheat_Sheet#Data_to_exclude) of the types of information that you need to be very careful logging.

Logging Plain User Input

Another common security issue in Java applications is **JVM Log Forging**.

Simply put, **log forging can happen when data from an external source like user input or another untrusted source is written directly to the logs**. A malicious attacker can enter input that simulates a log entry such as "*\n\nweb – 2017-04-12 17:47:08,957 [main] INFO Amount reversed successfully*" which can result in corrupted log data.

There are various ways to handle this kind of vulnerability:

- don't log any user input – not always possible, since the user data can be critical for getting to the root cause of some problems
- use validation before logging – this solution can impact performance, as well as forego logging important information
- log to a database – more secure but costly regarding performance, and can introduce another vulnerability – SQL injection (https://www.owasp.org/index.php/SQL_Injection).

- use a tool like the Enterprise Security API from OWASP

Using *ESAPI* is definitely a good way to go; this open-source security library from OWASP

(https://www.owasp.org/index.php/Main_Page) can encode data before writing it to the logs:

```
message = message.replace( '\n' , '_' ).replace( '\r' , '_' )  
            .replace( '\t' , '_' );  
message = ESAPI.encoder().encodeForHTML( message );
```

2. Excessive Logging

Another practice to be avoided is logging too much information. This can happen in an attempt to capture all potentially relevant data.

One possible and very real issue with this approach is decreased performance. However, with the evolution of logging libraries, you now have the tools to make this less of a concern.

As an example of improved performance, the 2.x version of *log4j* uses asynchronous logging, which means executing I/O operations in a separate thread.

Too many log messages can also lead to difficulty in reading a log file and identifying the relevant information when a problem does occur.

One way to reduce the number of log lines of code is by logging important information across cross-cutting concerns in the system.

For example, if you want to log the start and end of particular methods, you can add an *Aspect* that will do this for every method that has a specified custom annotation:




```
@Aspect
public class MyLogger {

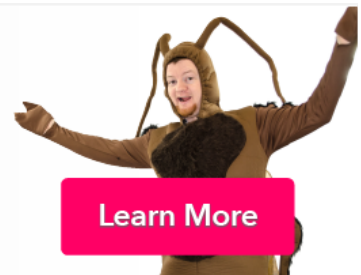
    private static final Logger logger = LogManager.getLogger(MyLogger.class);

    @Around("execution(* *(..)) && @annotation(LogMethod)")
    public Object around(ProceedingJoinPoint joinPoint) throws Throwable {
        logger.info("Starting method execution: " + joinPoint.getSignature().getName() +
            " in class:" + joinPoint.getSignature().getDeclaringTypeName());
        Object result = joinPoint.proceed();
        logger.info("Exiting method execution: " + joinPoint.getSignature().getName() +
            " in class:" + joinPoint.getSignature().getDeclaringTypeName());
        return result;
    }
}
```

With the help of the custom aspect, **we can now be very selective and pick the exact areas of the application** where we actually need that information in the logs. And, as a result, we can significantly decrease the overall logging footprint of the system.

Stackify Developers

That's why we created  Retrace
as a key APM solution for developers



https://info.stackify.com/cs/c/?cta_guid=c6b2e255-e511-4668-9529-58041fb5c42a&placement_guid=c00f5072-32fb-4332-a4d7-60765e6305d0&portal_id=207384&canon=https%3A%2F%2Fstackify.com%2F9-logging-sins-

3. Cryptic Log Messages

When parsing log files, encountering a line that doesn't provide sufficient information can be frustrating. **A common pitfall is the lack of specificity or context in log messages.**

To illustrate the problem, let's have a look at a **log message lacking specificity**:

```
Operation failed.
```

Instead, you can add more specific and identifiable information:

```
File upload picture.jpg failed.
```

Always keep in mind that your logs will most certainly be read by a different developer or system administrator, and they need to understand what has happened in the application.

A good way to add context in log messages is by including the timestamp, log level, thread name and fully qualified class name of the event. This way, you can more easily identify when and where specific events occur at runtime.

To add this information when using *log4j 2*, you can configure a *Pattern Layout* with the options *%d* for the date, *%p* for log level, *%t* for thread name and *%c* for class name:

```
<PatternLayout>  
  <Pattern>%d [%t] %p %c - %m%n</Pattern>  
</PatternLayout>
```



A log message using the above layout will look like this:



```
2017-05-11 22:51:43,223 [main] INFO com.stackify.service.MyService -  
User info updated
```

4. Using a Single Log File

The downside of only using one log file for the application is, that this will, over time, become quite large and difficult to work with.

A good practice for quickly finding relevant information is to create a new log file each day, with the date as part of the file name.

Let's take a look at an example of how to create a log file with the name equal to the current date if using the *log4j2* library:

```
SimpleLayout layout = new SimpleLayout();  
FileAppender appender = new FileAppender(layout, LocalDate.now().toString(), false);  
logger.addAppender(appender);
```

The same library also provides the option to configure a *Rolling File Appender* that will create new log files at certain time intervals:

```
<RollingFile name="RollingFile" fileName="logs/app.log"  
  filePattern="logs/${date:yyyy-MM}/app-%d{MM-dd-yyyy}-%i.log.gz">  
  <PatternLayout>  
    <Pattern>%d [%t] %p %c - %m%n</Pattern>  
  </PatternLayout>  
  <Policies>  
    <TimeBasedTriggeringPolicy />  
    <SizeBasedTriggeringPolicy size="250 MB"/>  
  </Policies>  
  <DefaultRolloverStrategy max="20"/>  
</RollingFile>
```



The configuration above will result in a one or more files created for each day up to 250 MB per file with the current date as the file name, placed in folders with names of the form year-month.



5. Choosing Incorrect Log Levels

Choosing an inadequate log level will lead to either missing significant events or being flooded with a lot of less important data.

Simply put, choosing the right log level for the different logs in your system is one of the core things you need to get right to have a good experience understanding your logs.

Most logging frameworks have a set of levels similar to ***FATAL, ERROR, WARN, INFO, DEBUG, TRACE***, ordered from highest to lowest.

Available log levels

Let's take a look at each of these and the type of log messages they should contain based on severity:

- ***FATAL*** should be reserved for errors that cause the application to crash or fail to start (ex: JVM out of memory)
- ***ERROR*** should contain technical issues that need to be resolved for proper functioning of the system (ex: couldn't connect to database)
- ***WARN*** is best used for temporary problems or unexpected behavior that does not significantly hamper the functioning of the application (ex: failed user login)
- ***INFO*** should contain messages that describe what is happening in the application (ex: user registered, order placed)
- ***DEBUG*** is intended for messages that could be useful in debugging an issue (ex: method execution started)
- ***TRACE*** is similar to ***DEBUG*** but contains more detailed events (ex: data model updated)

Controlling log levels

The log level of a message is set when it is written:

```
logger.info("Order ID:" + order.getId() + " placed.");
```

Logging APIs usually allow you to set the level up to which you want to see messages. What this means is that, if you set the log level for the application or certain classes to *INFO*, for example, you will only see messages at the levels *FATAL*, *ERROR*, *WARN* and *INFO*, while *DEBUG* and *TRACE* messages will not be included.

This is helpful as you would usually show *DEBUG* or lower messages in development, but not in production.

Here is how you can set the log level for a class, package or entire application in *log4j 2*, using a *log4j2.properties* file:

```
loggers=classLogger,packageLogger

logger.classLogger.name=com.stackify.service.MyService
logger.classLogger.level=info

logger.packageLogger.name=com.stackify.config
logger.packageLogger.level=debug
```

```
rootLogger.level=debug
```

To display the log level in the message, you can add the *%p* option in the *log4j2 PatternLayout*.

Before we move on, keep in mind that, if none of the existing levels are appropriate for your application needs, you have the possibility of defining a custom log level as well.



6. Tracking A Single Operation Across Multiple Systems and Logs



In distributed systems with multiple, independently deployed services that work together to process incoming requests, tracking a single request across all of these systems can be difficult.

A single request will very likely hit multiple of these services, and if a problem does occur, we'll need to corroborate all of the individual logs of these systems to get the full picture of what happened.

To address this kind of architecture, we now have a new generation of logging helper tools in the ecosystem, such as [Zipkin](http://zipkin.io/) (<http://zipkin.io/>) and [Spring Cloud Sleuth](https://cloud.spring.io/spring-cloud-sleuth/) (<https://cloud.spring.io/spring-cloud-sleuth/>).


Zipkin traces requests across microservice architectures to help you identify which application is causing the issue. It also comes with a helpful UI where you can filter traces based on application, length of the trace or timestamp.

And the *Spring Cloud Sleuth* project works by adding a unique 64-bit ID to each trace; a web request, for instance, can constitute a trace. This way, the request can be identified across multiple services.

These tools address the limitations of the core libraries and you navigate the new realities of the more distributed style of architectures.

7. Not Logging with JSON

While logging in a plaintext format is very common, the advent of log storage and data analysis systems has shifted that towards JSON.

:  JSON as the primary application log format has the advantage of being just as readable as plain text, while also being a lot easier to parse by automated processing tools.



For example, *Log4j 2* offers the *JSONLayout* for this exact purpose:


```
<JSONLayout complete="true" compact="false"/>
```

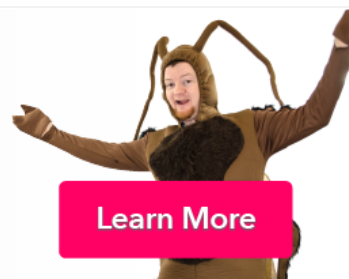
This will produce a well-formed JSON document:

```
[
  {
    "logger": "com.stackify.service.MyService",
    "timestamp": "1376681196470",
    "level": "INFO",
    "threadId": 1,
    "thread": "main",
    "threadPriority": 1,
    "message": "Order ID:1234 placed."
  },
  ...
]
```

As JSON, the log data will be semantically richer when it's processed by a log management system such as Retrace (<https://stackify.com/log-management/>)– which will immediately enable its powerful structured/semantic logging capabilities.

Stackify  Developers

That's why we created  Retrace
as a key APM solution for developers



Learn More

(https://info.stackify.com/cs/c/?cta_guid=c6b2e255-e511-4668-9529-58041fb5c42a&placement_guid=c00f5072-32fb-4332-a4d7-60765e6305d0&portal_id=207384&canon=https%3A%2F%2Fstackify.com%2F9-logging-

8. Logging Impact on Performance

Finally, let's consider an issue which is unavoidable when adding logging to an application: the impact on performance.

A small drop in performance is to be expected. However, it's important to track this so that you can minimize it and not slow down the system.

Some performance-related aspects to consider when choosing a logging API are:

- file I/O operations using a buffer – this is critical as file I/O is an expensive operation
- asynchronous logging – this should be considered so that logging doesn't block other application processes
- logging response time – the time it takes to write a log entry and return
- number of threads used for logging
- log level filtering – this is done to verify if the log level corresponding to a message is enabled, and can be done by traversing the hierarchy or having the *Logger* point directly to the *Logger* configuration; the latter approach is preferable regarding performance

Of course, if you need to keep the choice open and the system flexible, you can always use a higher level abstraction such as [slf4j](https://www.slf4j.org/) (<https://www.slf4j.org/>).



Before we move on, here are just a few steps you can take to improve logging performance of your system:



- tune the log level of the application for verbose packages
- avoid logging source location information at runtime, as looking up the current thread, file, a method is a costly operation
- avoid logging errors with long stack traces
- check if a specific log level is enabled before writing a message with that level – this way the message won't be constructed if it's not needed
- review the logs before moving to production to check if any logging can be removed

9. Honorable Mentions

Before we wrap up, let's have a look at one final practice that you should avoid – and that is **using standard out instead of logging**.

While *System.out()* can be a quick way to start very early on in the development cycle, it's definitely not a good practice to follow after that point.

Besides the fact that you lose all the powerful features of a dedicated logging API, this primary downside here is simply the fact that the logging data won't be persisted anywhere.

Finally, another honorable mention is a practice that can make reading and analyzing log data a lot easier – standardized log messages. Simply put, similar events should have similar messages in the log.

If you need to search for all instances of that particular event or extract meaningful insights out of your log data, standard log messages are quite important.



For example, if an upload operation fails – having these different messages in the log would be confusing:



```
Could not upload file picture.jpg
```

```
File upload picture.jpg failed.
```

Instead, whenever the file upload fails, you should consistently use one of these messages to log the failure.

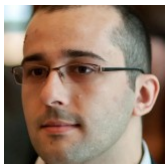
Conclusion

The use of logging has become ubiquitous in application development, due to the highly useful and actionable insights it brings into the runtime of the system.

However, to get the most out of your log data, it's important to go beyond the basics, develop a culture of logging and understand the finer points of operating with this data at scale and in production.

You also need the proper tools (<https://stackify.com/best-log-viewer-prefix/>), to help you extract that actionable insight out of your raw log files.

So, log freely, but log wisely.

[About the Author](#)[Latest Posts](#)

About Eugen Paraschiv

Eugen is a software engineer with a passion for Spring, REST APIs, Security and teaching, and the founder of [Baeldung](http://www.baeldung.com/) (<http://www.baeldung.com/>).