IBM

Effective logging practices ease enterprise development

Establish a logging plan up front and reap rewards later in the development process

Level: Intermediate

Charles Chan (chancharles@gmail.com), Principal Consultant, Ambrose Software Inc.

09 Aug 2005

Bumps along the road are almost inevitable in enterprise development, and if you want to effectively tackle bugs in the late stages of the development process, you're going to need an effective logging strategy. But logging effectively in an enterprise application requires planning and discipline. In this article, consultant Charles Chan guides you through some best practices to help you write useful logging code from the beginning of your project.

If you're a developer, you've almost certainly had this experience: You've developed your code and your test cases. Your apps have gone through rigorous QA testing, and you're confident that your code is going to fulfill the business requirements. When the application finally reaches the end users' hands, however, there are unexpected problems. Without proper log messages, it could take days to diagnose these problems. Unfortunately, most projects do not have a clear strategy on logging. Without one, the log messages produced by the system may not be useful for problem resolution. In this article, I discuss the various aspects of logging for enterprise applications. You get an overview of the APIs available for logging with the Java™ platform, learn some best practices for writing logging code, and see how to adapt if you need to resort to detailed logging in a production environment.

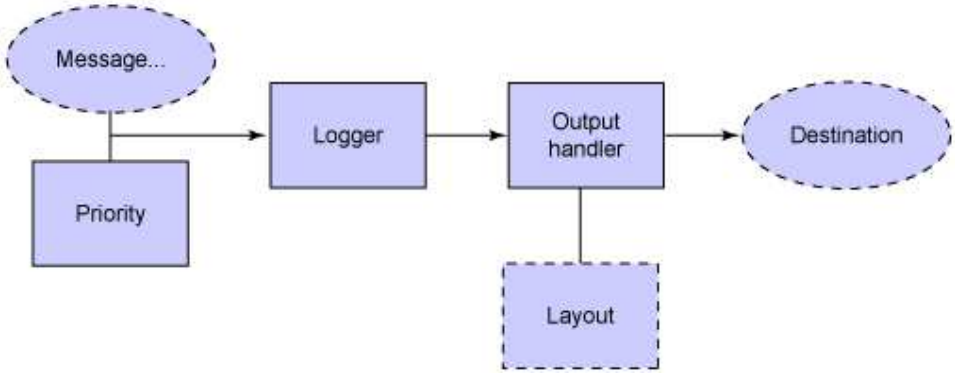
Choosing a logging API

When you develop using the Java platform, you have two major logging APIs to choose from: Apache Log4J and the Java Logging API that comes with version 1.4 and above of the Java platform. Log4J is more mature and feature-rich than the Java Logging API. Both logging implementations have a similar design pattern (as shown in Figure 1). Unless your company puts restrictions on the use of third-party libraries, I strongly recommend that you use Log4J. If you cannot decide on an API to use, you can use the Apache Commons Logging API, which acts as a wrapper to any underlying logging implementation. This, in theory, lets you switch the logging implementations without changing your code. In practice, however, you seldom need to switch logging implementations; thus, I would not recommend using the Apache Commons Logging API because its additional complexities do not buy you any additional features.

Logging overview

Both Log4J and the Java Logging API have a similar design and usage pattern (as shown in Figure 1 and Listing 1). Messages are first created and passed into a logger object with a specific priority. Then the destination and the format of these messages are determined by the output handler and its layout.

Figure 1. Major components of a logging implementation



Listing 1. Instantiate and using a logger object

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

public class MyClass {
  /*
   * Obtain a logger for a message category. In this case, the message
   * category is  the fully qualified class name of MyClass.
   */
  private static final Logger logger
        = Logger.getLogger(MyClass.class.getName());
  ...
  public void myMethod() {
    ...
    if (logger.isDebugEnabled()) {
      logger.debug("Executing with parameters:
        " + param1 + ":" + param2);
    }
  }
}
```

A good logging implementation comes with many different output handlers, the most common of which are the file output handler and the console output handler. Log4J also provides handlers to publish messages to a JMS topic or to insert messages into a database table. Although it is not difficult to write a custom appender, the total cost of writing and maintaining the code should not be underestimated. The format of a message is also configurable through the `Layout` object. The most common layout object is `PatternLayout`, which formats messages according to a supplied pattern.

Listing 2 shows a sample Log4J configuration file that configures a `FileAppender`. In this configuration, error messages under the `com.ambrosesoft.log.MyClass` category are sent to the `FileAppender`, which in turn writes them to a file called log.txt. The messages are formatted according to the layout (`PatternLayout`, in this case) associated with the appender.

Listing 2. A sample Log4J XML configuration file

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

  <appender name="fileAppender"
        class="org.apache.log4j.FileAppender">
    <param name="File" value="log.txt"/>
    <param name="Append" value="true"/>
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
        value="%d [%t] %p - %m%n"/>
    </layout>
  </appender>

  <category name="com.ambrosesoft.log.MyClass">
    <priority value="error"/>
```

```
    <appender-ref ref="fileAppender"/>
  </category>

  <root>
    <priority value="debug"/>
    <appender-ref ref="fileAppender"/>
  </root>

</log4j:configuration>
```

Logging best practices

Perhaps the most important choice you have to make about logging is to decide on a scheme that assigns each log message to a particular *category*. It is common practice to use the fully qualified name of each class whose activity is being logged as a message category (as we do in Listing 1) because this allows developers to fine-tune log settings for each class. This, however, will only work well if the log messages are intended for execution trace. In an enterprise application, there are many other kinds of log messages. For instance, one log message might be produced for security advisors, while another could be produced to aid performance tuning. If both messages concern the same class and are thus assigned the same category, it will be difficult to distinguish them in the log output.

To avoid this problem, an application should have a set of specialized loggers with distinct categories, as Listing 3 illustrates. Each of these loggers can be configured with its own priority and output handler. For example, the security logger can encrypt a message before it is written to the destination. From time to time, the application architect should review the log output with the intended consumers (such as the security advisors) to fine-tune the messages.

Listing 3. Special-purpose loggers

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

public interface Loggers {
  Logger performance = Logger.getLogger("performance");
  Logger security = Logger.getLogger("security");
  Logger business = Logger.getLogger("business");
}
...
public class MyClass {
  ....
  if (Loggers.security.isWarnEnabled()) {
    Loggers.security.warn
        ("Access denied: Username [" + userName + "] ...");
  }
  ...
}
```

Choosing logging levels

Messages that fall within a single *category* (security, say) can have different *priorities*. Some messages are produced for debugging, some for warnings, and some for errors. The different priorities of messages are represented by logging *levels*. The most common logging levels are:

- **Debug:** Messages in this level contain extensive contextual information. They are mostly used for problem diagnosis.
- **Info:** These messages contain some contextual information to help trace execution (at a coarse-grained level) in a production environment.
- **Warning:** A warning message indicates a potential problem in the system. For example, if the message category is related to security, a warning message should be produced if a dictionary attack is detected.

- **Error:** An error message indicates a serious problem in the system. The problem is usually non-recoverable and requires manual intervention.

Both the standard Java Logging API and Apache Log4J provide logging levels beyond these basics. The primary purpose of a logging level is to help you filter useful information out of the noise. To avoid using the wrong level and thus reducing the usefulness of log messages, developers must be given clear guidelines before they start coding.

Log message format

Once you've chosen the logger and established the logging level, you can start building your log message. When doing so, it is crucial to include as much contextual information as possible -- parameters supplied by users and other application state information, for example. One way to log objects is to convert them into XML. Third-party libraries like XStream (see Resources) can automatically convert Java objects into XML by introspection. Although this is a very powerful mechanism, the trade-off between speed and verbosity should be considered. Besides the typical application state information, the following information should also be logged:

- **Thread ID:** Enterprise applications are often executed in a multithreaded environment. With thread ID information, you can distinguish one request from another.

- **Caller identity:** The identity (or principal) of the caller is also an important piece of information. Because different users have different privileges, their execution paths could be very different. Putting the user's identity in the log messages can be a great help for a security-aware application.

- **Timestamp:** Generally, users can only approximate the time at which a problem occurred. Without a timestamp, it's difficult for the support personnel to identify the problem.

- **Source code information:** This includes class name, method name, and line number. Unless you are concerned about security, I recommend leaving the debug flag (`-g`) on all the time even when compiling for production. Without the debug flag, the Java compiler removes all line number information and significantly reduces the usefulness of your log messages.

All of the above information (except caller identity) is automatically retrieved by the logging implementation. To include it in your messages, you only need to configure an appropriate layout pattern for your output handler. To capture a caller's identity, you can make use of the diagnostic context feature in Log4J (see Resources for more information). Diagnostic context lets you associate contextual information with the currently running thread. This information can then be included in every message as it is being formatted for output.

In a J2EE Web application, the best place put the logic to store the user's identity into the diagnostic context is inside a servlet filter. Listing 4 shows the essential code required to achieve this. It uses the mapped diagnostic context class (`MDC`) available in Log4J 1.3 alpha. You can achieve the same thing with nested diagnostic context (`NDC`) in Log4J 1.2. For more information about servlet filters in general, refer to the Resources section.

Listing 4. Using diagnostic context in a servlet filter

```
import javax.servlet.Filter;
...
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import org.apache.log4j.MDC;

public class LoggerFilter implements Filter {

    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {

        // Retrieves the session object from the current request.
        HttpSession session = ((HttpServletRequest)request).getSession();

        // Put the username into the diagnostic context.
        // Use %X{username} in the layout pattern to
```

```
        // include this information.
        MDC.put("username",
                session.getAttribute("username"));

        // Continue processing the rest of the filter chain.
        chain.doFilter(request, response);

        // Remove the username from the diagnostic context.
        MDC.remove("username");
    }
    ...
}
```

Trace execution with AspectJ

When you are diagnosing a problem, it is often useful to trace your program's execution. How can you consistently put log statements at various points of execution, such as method entries and method exits? This is an age-old problem that didn't have a good solution until AspectJ arrived. With AspectJ, you can execute code snippets at various points in your application. In AspectJ, these points are called *point cuts,* and the code you execute at point cuts is called *advice.* The combination of a point cut and an advice is called an *aspect*.

What is so amazing about AspectJ is that an aspect can be applied to an entire application without a lot of effort. For more information about AspectJ, see Resources. Listing 5 shows an AspectJ source file used to enable logging on method entries and exits. In this example, the trace logger will log a message every time a public method in the `com.ambrosesoft` package is entered or exited.

Listing 5. Using AspectJ to log method entries and exits

```
import org.apache.log4j.Logger;
import java.lang.reflect.Field;

public aspect AutoTrace {

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

    pointcut publicMethods() :
        execution(public * com.ambrosesoft..*(..));

    pointcut loggableCalls() : publicMethods();

    /**
     * Inspect the class and find its logger object. If none is found,
     * use the one defined here.
     */
    private Logger getLogger(org.aspectj.lang.JoinPoint joinPoint) {
        try {
            /*
             * Try to discover the logger object.
             * The logger object must be a static field called logger.
             */
            Class declaringType
            = joinPoint.getSignature().getDeclaringType();
            Field loggerField
            = declaringType.getField("logger");
            loggerField.setAccessible(true);
            return (Logger)loggerField.get(null);
        } catch(NoSuchFieldException e) {
            /*
             * Cannot find a logger object, use the internal one.
             */
            return logger;
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
    }

    /**
     * An aspect to log method entry.
```

```
     */
    before() : loggableCalls(){
        getLogger(thisJoinPoint).debug("Entering.."
        + thisJoinPoint.getSignature().toString());
    }

    /**
     * An aspect to log method exit.
     */
    after() : loggableCalls(){
        getLogger(thisJoinPoint).debug("Exiting.."
        + thisJoinPoint.getSignature().toString());
    }
}
```

Logging in a production environment

Once your application is in production, you'll usually turn off debug or informational log messages to optimize run-time performance. However, when something bad happens and you cannot reproduce the problem in a development environment, your might need to turn on debug messages in production. It is important to be able to change log settings without bringing down the server. Diagnosing a production problem often requires hours if not days of careful investigation. During that period of time, developers need to turn on and off logging in different areas of the application. If you need to restart the production application every time the log settings are changed, the situation will become untenable quickly.

Fortunately, Log4J provides a simple mechanism to handle this problem. In Log4J 1.2, the method `configureAndWatch()` in `DOMConfigurator` configures Log4J and automatically monitors changes in the log configuration file. This is illustrated in Listing 6. (Note that `DOMConfigurator` is deprecated in Log4J 1.3 (currently in alpha) and will be replaced by a more flexible implementation, `JoranConfigurator`.)

To ensure that `configureAndWatch()` is called before Log4J initializes, you should invoke it in your startup class. Different application servers have different mechanisms to execute startup code (see Resources for more information). Check your application server's implementation for details. Some application servers may require you to put the Log4J library in the server classpath. The log configuration file should be stored in a location accessible by support personnel.

Listing 6. Configure Log4J with DOMConfigurator

```
/*
 * Configure Log4J library and periodically
 * monitor log4j.xml for any update.
 */
DOMConfigurator.configureAndWatch
        ("/apps/config/log4j.xml");
```

If your log configuration file is not easily accessible (if your production environment is maintained by a different organization, for instance), you must use a different tactic. A standard approach is to use JMX, which provides a standard API to manage your application settings. In modern JMX-compliant servers, you can extend the capabilities of the application server's administration console with managed beans, or *MBeans*. (See the Resources section for more on using JMX and doing this in WebSphere Application Server 6.0.) Because the JMX approach is fairly complex, you should only use this route if your circumstances require it.

Logging sensitive data

Besides technical challenges, there are also business issues that you need to overcome when logging in a production environment. For example, logging sensitive data can pose a security risk. There is nothing that prevents you from

logging a user's username and password in plain text. You must also protect other sensitive information, such as e-mail addresses, phone numbers, and account information. It is the responsibility of the security advisor and the architect to make sure that this information is not logged without masking. Using a security-specific logger for sensitive messages can help reduce risk. You can configure this logger with a special appender to store messages in an encrypted format or in a secured location. However, the best way to avoid security risk is to have proper coding guidelines before the project starts and enforce them during code review.

Making sense out of exceptions

When an unexpected exception occurs -- for instance, if a database connection is suddenly dropped, or system resources are running low -- it must be handled properly or else you will lose useful information that can help you diagnose the problem. First of all, the exception and its stack trace must be logged. Secondly, a user-friendly error page that is useful both to the end user and the support team should be presented.

One of the challenges support team members face when they receive a support call is correlating a user-reported problem with a specific logged exception. A simple technique that can help tremendously involves logging a unique ID with every exception. This ID can either be presented to the user or be included in the problem report form that users can fill out. Doing so removes any guessing by the support team members and lets them respond to the situation quickly. Consider recycling the ID periodically for readability.

Log file management

A busy application's log file can quickly become very large. A large log file is difficult to use because you need to filter out a lot of noise to find the useful signal. *Log rotation* is a common practice that can help you avoid this problem. Log rotation periodically archives old log messages so that new messages are always written to a relatively small file. Log messages cease to be useful relatively quickly; you'll very rarely need to refer to a log message that's more than a week old. In Log4J 1.2, the `DailyRollingFileAppender` appender can rotate log files based on the supplied date pattern. (In Log4J 1.3, the rolling file appender has been redesigned. You can now supply a policy that controls how rolling is performed. For example, a `TimeBasedRollingPolicy` defines a rotation scheme based on time and date.) Listing 7 shows the configuration snippets required to let Log4J rotate its log file daily at midnight.

Listing 7. Using a DailyRollingFileAppender to rotate a log file

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

  <appender name="fileAppender" class
        ="org.apache.log4j.DailyRollingFileAppender">
    <param name="File" value="log.txt"/>
    <param name="Append" value="true"/>
    <param name="DatePattern"
        value="'.'yyyy-MM-dd"/>
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
        value="%d [%t] %p - %m%n"/>
    </layout>
  </appender>
...
</log4j:configuration>
```

Logging in a clustered environment

More and more enterprise applications are deployed in clustered or distributed environments. Logging in a clustered environment, however, requires more planning because messages are generated from different sources (usually different machines). If logging is performed on different machines, the machines' timestamps must be synchronized,

or the log messages will be out of order. A simple way to synchronize clocks among machines is to use a time server. There are two ways to set one of these up. You could designate one of your internal machines as the time server. The other machines would then use the network time protocol (NTP) to synchronize their time stamps with the time server's. Alternately, you could use one of the time servers accessible on the Internet (see Resources). On AIX, the `xntpd` daemon is used to synchronize system time among machines. Once the machines have the same time, their log messages can be analyzed together.

Gathering log messages in a clustered environment also presents some challenges. A simple way to store log messages in this environment is to store them in their host-specific log files. This works very well when your cluster is configured with *session affinity* -- that is, if requests for a particular user session always comes to the same server and your EJBs are deployed locally. In such a setup, log files produced by the machines in the cluster can be analyzed independently. If this is not the case -- in other words, if any given request can potentially be handled by multiple machines -- analyzing the log messages in different log files becomes more difficult. In this case, a better approach is to manage the log messages with system management software, such as IBM Tivoli® software (see Resources for a link). Such software provides an integrated view of all of your log messages (*events,* in system management software terminology) for ease of administration. System management software can also trigger actions (such as sending an e-mail message or a pager message) based on the type of events it receives.

Conclusion

In this article, I've shown what needs to be considered when you are planning a logging strategy. As with just about anything in programming, you'll save work in the long run by having a well-thought-out plan from the beginning rather than making things up as you go along. A good logging strategy can be a great help in diagnosing problems. Ultimately, the end users will benefit from a better application and faster response time from the support team.

Resources

- The Apache Log4J library is the most feature-rich and mature logging API for the Java platform.
- The official short introduction to Log4J is essential for anyone who wants to use the Log4J library.
- The Log4J tutorial offers another good introduction to the logging package.
- The Java Logging API is the standard logging API shipped with Java 1.4 and above. This API was originally JSR 47.
- "An introduction to the Java Logging API," Brian Gilstrap (*OnJava.com,* June 2002) talks about the use of the standard logging API.
- "The Java Logging API," Stuart Dabbs Halloway (*JavaPro,* June 2002) is another good introduction to the API.
- The Apache Commons Logging API provides a wrapper to other logging implementations and thus allows you to switch among different implementations.
- Nested diagnostic context (NDC) stores contextual information in local thread storage that can be picked up when log messages are produced.
- The XStream library can serialize Java objects to XML (and back again) by introspection. It can be very useful for logging application states.
- Servlet filter intercepts requests and responses and allows you to set up contextual information for your log

messages.

- Sun's The Essentials of Filters is a good resource to learn servlet filters.
- Popular developerWorks author Sing Li offers his two cents on servlet filtering with "Filtering tricks for your Tomcat" (developerWorks, June 2001), which shows you how to make productive use of filters in your projects.
- *AOP@Work* series, penned by leading experts in the aspect-oriented development community, offers excellent insight on applied AOP. Of particular relevance to this article are "Introducing AspectJ 5" and "New AJDT releases ease AOP development."
- "Migrating WebLogic startup code to WebSphere Application Server V5," Wayne Beaton and Sree Anand Ratnasingh (developerWorks, January 2004) talks about the differences in startup code between two popular application servers.
- Check out the Java Management Extensions (JMX) home page.
- Popular author Sing Li provides some initial insight into JMX in his three-part series "Management, JXM 1.1 style."
- This help page discusses extending WebSphere Application Server administrative system with Java Management Extensions (JMX).
- The Network Time Protocol project homepage has a list of public time servers you can use to synchronize your machines.
- The `xntpd` manual page talks about the network time protocol daemon on AIX.
- You'll find articles about every aspect of Java programming in the developerWorks Java technology zone.

About the author

Charles Chan is an independent consultant working in Toronto, Canada. His interests include distributed systems, high-performance computing, internationalization, and software design patterns. In his spare time, he contributes to the open source community.

Share this....