

Introduction à Log4J

par [Sébastien Le Ray](#)

Date de publication : 18/08/2005

Dernière mise à jour : 13/09/2005

Cet article est une introduction au système de journalisation pour java de Jakarta : log4j. L'accent est mis sur la façon de le configurer.

- I - Introduction
- II - Principes généraux
 - II.A - Logger
 - II.B - Niveaux de journalisation
 - II.C - Appenders
 - II.D - Layouts
- III - Configuration
 - III.A - Principes de configuration
 - III.A.1 - Héritage de niveau
 - III.A.2 - Additivité des Appenders
 - III.A.3 - Organisation de la configuration
 - III.B - Format des fichiers de configuration
 - III.B.1 - Mécanisme de recherche
 - III.B.2 - Fichiers properties
 - III.B.2.a - Configuration des Appenders
 - III.B.2.b - Configuration des Loggers
 - III.B.3 - Fichiers XML
 - III.B.3.a - Configuration des Appenders
 - III.B.3.b - Configuration des Loggers
- IV - Propriétés utiles
 - IV.A - Appenders
 - IV.A.1 - Seuils
 - IV.A.2 - ConsoleAppender
 - IV.A.3 - FileAppender
 - IV.A.4 - JDBCAppender
 - IV.A.5 - DailyRollingFileAppender
 - IV.A.6 - RollingFileAppender & ExternallyRolledFileAppender
 - IV.A.7 - AsyncAppender
 - IV.B - Layouts
 - IV.B.1 - PatternLayout
 - IV.B.2 - HTMLLayout & XMLLayout
- V - Un petit mot sur les nouveautés de la version 1.3
 - V.A - ErrorHandlers
 - V.B - WatchDogs
- VI - Conclusion
- VII - Remerciements
- VIII - Téléchargements

I - Introduction

Log4j est une API de journalisation très répandue dans le monde Java. Il s'agit d'un système hautement configurable, que ce soit au niveau de ce qui doit être enregistré ou de la destination des enregistrements (serveur de logging, fichiers tournants, *etc.*). Pour cet article, je me suis appuyé sur la version 1.2.11, la version 1.3 devrait voir le jour en octobre 2005, elle introduit certains changements au niveau de l'API ; néanmoins, les recommandations concernant la compatibilité avec la future API ont été appliquées.

Cet article insiste plus particulièrement sur la configuration de log4j, en effet, cette API est fréquemment utilisée comme système de journalisation sous-jacent en combinaison avec l'API [commons logging](#). De plus, son utilisation directe est relativement simple.

II - Principes généraux

II.A - Logger

Comme son nom l'indique, le *Logger* est l'entité de base pour effectuer la journalisation, il est mis en oeuvre par le biais de la classe **org.apache.log4j.Logger**. L'obtention d'une instance de *Logger* se fait en appelant la méthode statique **Logger.getLogger** :

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

public class MaClasse {
    private static final Logger logger = Logger.getLogger(MaClasse.class);
    // suite
}
```

Il est possible de donner un nom arbitraire au *Logger*, cependant, comme nous le verrons lorsque nous parlerons des hiérarchies de *Loggers* et de la configuration, il est préférable d'utiliser le nom de la classe pour des raisons de facilité.



Dans les version antérieures à la 1.2, l'entité utilisée pour la journalisation était la **Category**, elle a été remplacée par la classe **Logger** (consultez la [documentation](#) pour des informations sur les pratiques à adopter en vue de la parution de la version 1.3).

II.B - Niveaux de journalisation

Si vous avez déjà utilisé des systèmes de journalisation comme les commons logging, vous êtes familiers avec la notion de niveau de journalisation ou de priorité d'un message. Il s'agit d'une entité représentant l'importance du message à journaliser, elle est représentée par la classe **org.apache.log4j.Level**. Un message n'est journalisé que si sa priorité est supérieure ou égale à la priorité du *Logger* effectuant la journalisation. L'API Log4j définit 5 niveaux de logging présentés ici par gravité décroissante :

- **FATAL** : utilisé pour journaliser une erreur grave pouvant mener à l'arrêt prématuré de l'application ;
- **ERROR** : utilisé pour journaliser une erreur qui n'empêche cependant pas l'application de fonctionner ;
- **WARN** : utilisé pour journaliser un avertissement, il peut s'agir par exemple d'une incohérence dans la configuration, l'application peut continuer à fonctionner mais pas forcément de la façon attendue ;
- **INFO** : utilisé pour journaliser des messages à caractère informatif (nom des fichiers, etc.) ;
- **DEBUG** : utilisé pour générer des messages pouvant être utiles au débogage.


Deux niveaux particuliers, **OFF** et **ALL** sont utilisés à des fins de configuration. La version 1.3 introduira le niveau **TRACE** qui représente le niveau le plus fin (utilisé par exemple pour journaliser l'entrée ou la sortie d'une méthode). Il va de soi que plus l'on descend dans les niveaux, plus les messages sont nombreux.

Si vous avez besoin de niveaux supplémentaires, vous pouvez créer les vôtres en sous-classant **org.apache.log4j.Level**, néanmoins, ceux qui sont proposés devraient être suffisants. La journalisation d'un message à un niveau donné se fait au moyen de la méthode **log(Priority, String)**, il existe diverses variantes permettant par exemple de passer un **Throwable** dont la trace sera enregistrée. Pour les niveaux de base, des méthodes raccourcies sont fournies, elle portent le nom du niveau :

```
try {
    // équivaut à logger.info("Message d'information");
    logger.log(Level.INFO, "Message d'information");
    // Code pouvant soulever une Exception
    //...
} catch (UneException e) {
```

```
// équivaut à logger.log(Level.FATAL, "Une exception est survenue", e);
logger.fatal("Une exception est survenue", e);
}
```

Il est possible d'effectuer une journalisation avec des messages localisés au moyen des méthodes `l7dlog(Priority, String cle, [Object[],] Throwable)`. `cle` correspond à l'identifiant du message dans le `ResourceBundle` positionné *via* la méthode `setResourceBundle`. Notez que pour ces méthodes, il n'existe pas de raccourci.

 Dans les version antérieures à la 1.2, le niveau du message était représenté par l'entité **Priority**, elle a été remplacée par la classe **Level** (consultez la [documentation](#) pour des informations sur les pratiques à adopter en vue de la parution de la version 1.3).

II.C - Appenders

Bien que vous ne devriez pas avoir à manipuler les *Appenders* directement en Java, à moins que vous ne créiez les vôtres, il est nécessaire de connaître leur fonctionnement afin de configurer correctement Log4j. Les *Appenders*, représentés par l'interface `org.apache.log4j.Appender`, sont le moyen utilisé par log4j pour enregistrer les événements de journalisation. Chaque *Appender* a une façon spécifique d'enregistrer ces événements. Log4j vient avec une série d'*Appenders* qu'il est utile de décrire, puisqu'ils seront repris dans la configuration :

- `org.apache.log4j.jdbc.JDBCAppender` : Effectue la journalisation vers une base de données ;
- `org.apache.log4j.net.JMSAppender` : Utilise JMS pour journaliser les événements ;
- `org.apache.log4j.nt.NTEventLogAppender` : Journalise *via* le journal des événements de Windows (NT/2000/XP) ;
- `org.apache.log4j.lf5.LF5Appender` : Journalise les événements vers une console basée sur Swing, celle-ci permet de trier ou de filtrer les événements ;
- `org.apache.log4j.varia.NullAppender` : N'effectue aucune journalisation ;
- `org.apache.log4j.net.SMTPAppender` : Envoie un email lorsque certains événements surviennent (à ne pas activer avec un niveau de journalisation **DEBUG...**) ;
- `org.apache.log4j.net.SocketAppender` : Envoie les événements de journalisation vers un serveur de journalisation ;
- `org.apache.log4j.net.SyslogAppender` : Journalise les événements vers un daemon *Syslog* (distant ou non) ;
- `org.apache.log4j.net.TelnetAppender` : Journalise les événements vers un socket auquel on peut se connecter *via* telnet ;
- `org.apache.log4j.ConsoleAppender` : Effectue la journalisation vers la console ;
- `org.apache.log4j.FileAppender` : Journalise dans un fichier ;
- `org.apache.log4j.DailyRollingFileAppender` : Journalise dans un fichier qui tourne régulièrement (contrairement à ce que son nom suggère, ce n'est pas forcément tous les jours) ;
- `org.apache.log4j.RollingFileAppender` : Journalise dans un fichier, celui-ci est renommé lorsqu'il atteint une certaine taille et la journalisation reprend dans un nouveau fichier (par exemple, on aura le fichier `logfile` dans lequel s'effectue la journalisation et `logfile.1` qui contient les événements antérieurs).

Les paramètres nécessaires à certains de ces *Appenders* sont détaillés dans la partie configuration. Notez cependant qu'il est possible d'affecter un niveau seuil (`threshold`) à tous les *Appenders* étendant la classe `org.apache.log4j.AppenderSkeleton` (ce qui est le cas de tous les *Appenders* fournis avec log4j). Dans ce cas, un message n'est journalisé par un *Appender* donné que si son niveau est supérieur ou égal à celui du *Logger* et qu'il est supérieur ou égal au seuil de l'*Appender* considéré.

II.D - Layouts

Les *Layouts* sont utilisés pour mettre en forme les différents événements de journalisation avant qu'ils ne soient

enregistrés. Ils sont utilisés en conjugaison avec les *Appenders*. Bien que tous les *Appenders* acceptent un *Layout*, ils ne sont pas forcés de l'utiliser (les *Appenders* utilisant un *Layout* sont repérables au fait que leur méthode **requiresLayout** renvoie true).

Les *Layouts* fournis par log4j sont les suivants, l'existence du **PatternLayout** permet de formater les événements d'à peu près n'importe quelle façon :

- **org.apache.log4j.SimpleLayout** : Comme son nom l'indique, il s'agit du *Layout* le plus simple, les événements journalisés ont le format **Niveau - Message[Retour à la ligne]** ;
- **org.apache.log4j.PatternLayout** : *Layout* le plus flexible, le format du message est spécifié par un motif (pattern) composé de texte et de séquences d'échappement indiquant les informations à afficher. Reportez vous à la JavaDoc pour une description complète des séquences d'échappement existantes. Par défaut, les événements sont journalisés au format **Message[Retour à la ligne]** ;
- **org.apache.log4j.XMLLayout** : Comme son nom l'indique, formate les données de l'événement de journalisation en XML (à utiliser en conjugaison avec un *Appender* de la famille des *FileAppenders*) ;
- **org.apache.log4j.HTMLLayout** : Les événements sont journalisés au format HTML. Chaque nouvelle session de journalisation (réinitialisation de Log4j) donne lieu à un document HTML complet (*ie.* préambule DOCTYPE, <html>, *etc.*).

III - Configuration

III.A - Principes de configuration

III.A.1 - Héritage de niveau

Le principe central de la configuration est sa hiérarchisation, c'est-à-dire que chaque *Logger* va hériter du niveau de son plus proche parent pour lequel on en a défini un, à moins qu'on ne lui attribue explicitement un niveau. Un *Logger* est parent d'un autre si son nom complet est le préfixe du second. Par exemple, le *Logger* **com.developpez.beuss.java** sera le parent du *Logger* **com.developpez.beuss.java.log4j**, il lui transmettra donc son niveau. Il existe un *Logger* particulier appelé *rootLogger* (*Logger* racine) qui est l'ancêtre de tous les *Loggers* existants.

L'intérêt de ce mécanisme est évident lorsque l'on utilise différentes bibliothèques au sein d'un projet qui se servent de Log4j pour journaliser les messages. Il est possible de compartimenter la configuration pour, par exemple, avoir tous les messages de débogage des classes que l'on développe mais uniquement les messages d'erreur des bibliothèques utilisées.

Prenons par exemple un projet qui utiliserait certaines des bibliothèques communs de Jakarta. Il est fréquent que ces bibliothèques utilisent les communs logging pour journaliser les événements. Les communs logging utilisent Log4j comme système de journalisation si celui-ci est disponible. on pourra donc avoir une hiérarchie ressemblant à cela (la syntaxe utilisée n'est pas celle des fichiers de configuration log4j, celle-ci sera présentée plus loin) :

Logger	Niveau
root	error (assigné)
org	error (hérité)
org.apache	error (hérité)
org.apache.commons	error (hérité)
org.apache.catalina	info (assigné)
org.apache.catalina.core	info (hérité)
com.developpez.beuss.java	debug (assigné)
com.developpez.beuss.java.log4j	debug (hérité)

Avec une configuration de ce type, seuls les messages d'erreur des APIs communs seront journalisés alors que l'on aura les messages d'information provenant de Tomcat (**org.apache.catalina**) et les messages de débogage de l'application en cours de développement.

C'est là qu'apparaît l'utilité de l'utilisation des noms complets des classes comme identifiant pour les *Loggers*. En procédant de cette façon, le contrôle du niveau de journalisation peut s'effectuer au niveau des packages ou même des classes.

Si vous n'affectez pas de niveau de journalisation au *Logger* racine, il prend automatiquement la valeur **DEBUG**, cela peut conduire à l'affichage de nombreux messages, aussi, il est préférable de définir un niveau manuellement.

III.A.2 - Additivité des Appenders

De prime abord, le fonctionnement de la configuration des *Appenders* peut sembler similaire à celui des niveaux, mais ce n'est pas le cas. En effet, il ne s'agit plus d'héritage proprement dit mais d'additivité. C'est-à-dire qu'un *Logger* d'un niveau donné va bénéficier de tous les *Appenders* de ses ancêtres en plus de ceux qui lui sont

éventuellement affectés. Il est possible de "couper" la chaîne à un niveau quelconque en indiquant que l'on ne veut pas que l'additivité s'applique. Un exemple vaut mieux qu'un long discours :

Logger	Appender(s)	Additivité
root	RootAppender	Non Pertinent
org	RootAppender (hérité), OrgAppender (assigné)	Oui
org.apache	RootAppender (hérité), OrgAppender (hérité)	Oui
org.apache.commons	RootAppender (hérité), OrgAppender (hérité)	Oui
org.apache.catalina	CatalinaAppender (assigné)	Non
org.apache.catalina.core	CatalinaAppender (hérité)	Oui
com.developpez.beuss.java	RootAppender (hérité), BeussJavaAppender (assigné)	Oui
com.developpez.beuss.java.log4j	RootAppender (hérité), BeussJavaAppender (assigné)	Oui

Avec ce type de configuration, on peut décider par exemple que le *Logger* racine enverra ses messages vers la console et que certains *Loggers* journaliseront dans un fichier. Le *Logger* **org.apache.catalina** introduit une rupture dans l'additivité, seuls les *Appenders* définis à partir de son niveau seront affectés à ses descendants.

III.A.3 - Organisation de la configuration

Quel que soit le format de la configuration, le procédé est toujours le même, seule la forme change. Il faut tout d'abord configurer les différents *Appenders*. Ensuite vient la configuration des *Loggers*. Vous n'êtes bien sûr pas obligé de configurer tous les *Loggers* de façon individuelle, utilisez les mécanismes d'héritage de niveau et d'*Appender*. Le minimum requis pour que Log4j fonctionne correctement est d'attribuer un *Appender* correctement configuré au *Logger* racine.

Si log4j n'est pas correctement configuré, cela n'empêchera pas l'application de fonctionner, simplement, les messages de journalisation peuvent être perdus.

III.B - Format des fichiers de configuration

III.B.1 - Mécanisme de recherche

La configuration de log4j peut se faire *via* un fichier properties classique (clé=valeur(s)) ou un fichier XML. La seconde solution offre plus de possibilités et, de par son format, est plus structurée ce qui permet de s'y retrouver plus facilement. Une troisième méthode est de configurer le système de journalisation par programmation mais nous n'en parlerons pas ici en raison de son manque de flexibilité. Notez toutefois que si vous procédez de cette façon, vous devez appeler la méthode **activateOptions()** de chaque *Appender* une fois que vous lui avez fourni tous ses paramètres.

Pour déterminer le fichier de configuration ainsi que son format, log4j utilise le mécanisme de recherche suivant. La recherche des fichiers s'effectue *via* le *ClassLoader*, les fichiers de configuration doivent donc se trouver sur le classpath.

- Si la propriété système **log4j.defaultInitOverride** est définie à autre chose que false, le mécanisme de recherche n'est pas exécuté (à utiliser si vous configurez le système par programmation) ;

- Vous pouvez définir les propriétés système **log4j.configuration** et **log4j.configuratorClass** qui indiquent respectivement le fichier à utiliser et la classe servant à lire et à charger les propriétés (elle doit implémenter l'interface **org.apache.log4j.spi.Configurator**) ;
- En l'absence de propriété système indiquant le fichier à utiliser, log4j recherche un fichier nommé **log4j.xml**, si celui-ci n'existe pas, il tente d'obtenir le fichier **log4j.properties**. Quel que soit le cas (fichier de configuration personnalisé ou mécanisme de recherche), si le fichier est introuvable, le mécanisme est interrompu et log4j n'est pas configuré ;
- Si aucun *Configurator* n'est spécifié et que le fichier termine par l'extension **.xml**, log4j utilise un *DomConfigurator* (configuration XML), si l'instanciation est impossible (classe inexistante, qui n'implémente pas **Configurator**, etc.), le mécanisme est interrompu et log4j n'est pas configuré. Si aucune classe n'est précisée et que le fichier n'est pas un fichier XML, un *PropertyConfigurator* (configuration classique) est utilisé.

La méthode la plus simple consiste donc à laisser faire log4j en plaçant un fichier **log4j.xml** dans un répertoire situé sur le classpath.

III.B.2 - Fichiers properties

La configuration *via* des fichiers properties est la première à avoir été implémentée, sa structuration est basique et ses possibilités sont plus restreintes que celles des fichiers XML (certains *Appenders* ne peuvent pas être configurés *via* un fichier properties). Néanmoins, il est utile de connaître son organisation puisqu'elle reste très répandue.

L'ordre de configuration présenté ici (*Appenders* puis *Loggers*) est purement arbitraire (elle suit en fait l'ordre utilisé dans la DTD des fichiers XML), vous pouvez faire l'inverse ou même mélanger les deux, log4j s'y retrouve parfaitement.

Vous pouvez définir un niveau minimal pour tous les *Loggers* au moyen de la clé **log4j.threshold**. Cela signifie que tous les messages en dessous de ce niveau seront ignorés par tous les *Loggers* **quel que soit leur niveau** (mais les messages au dessus de ce niveau sont traités normalement, en fonction du niveau de chaque *Logger*).

Si vous voulez que log4j affiche des messages de débogage indiquant les opérations effectuées en interne, vous pouvez positionner à **true** la clé **log4j.debug**. Cela peut être utile lors de la mise au point d'une configuration ou d'un *Appender* personnalisé.

III.B.2.a - Configuration des Appenders

Il faut savoir qu'au niveau du fichier de configuration, chaque *Appender* doit avoir un nom afin de pouvoir y faire référence lors de la configuration des *Loggers*. Les paramètres de configuration concernant les *Appenders* sont préfixés par **log4j.appender**. La déclaration d'un *Appender* d'un nom donné se fait de la façon suivante :

```
log4j.appender.NomAppender=ClasseAppender
```

Où *NomAppender* est remplacé par le nom que l'on souhaite attribuer à l'*Appender* et *ClasseAppender* la classe d'implémentation de l'*Appender*. Si l'*Appender* n'a besoin d'aucun autre paramètre pour fonctionner (pas de *Layout* ni de paramètre particulier), il peut être utilisé directement par un *Logger*. Voici un exemple de déclaration de l'*Appender* pour LogFactor5 (LF5) :

```
log4j.appender.AppenderLogFactor5=org.apache.log4j.lf5.LF5Appender
```

Vous pouvez remplacer *AppenderLogFactor5* par ce que vous voulez, cela changera simplement l'identifiant utilisé

lorsque vous ferez référence à cet *Appender* lors de la configuration des *Loggers*.

Si l'*Appender* nécessite des paramètres (comme par exemple un *Layout*), il faut les lui passer en utilisant la syntaxe :

```
log4j.appender.NomAppender.NomOption=ValeurOption
```

Les options disponibles pour un *Appender* sont parfois décrites dans sa documentation mais vous pouvez toujours les retrouver en regardant les setters de cet *Appender* (méthodes **setXXX**). Le nom du paramètre est alors le nom du setter sans **set** et avec la première lettre en minuscule (par exemple **setLayout** correspond à la propriété *Layout*). Par exemple, si l'on veut configurer le layout pour un **ConsoleAppender**, on procédera de la façon suivante :

```
log4j.appender.Console=org.apache.log4j.ConsoleAppender
log4j.appender.Console.layout=org.apache.log4j.SimpleLayout
```

Une fois les *Appenders* configurés, il vous reste à configurer les *Loggers* pour qu'ils les utilisent.

III.B.2.b - Configuration des Loggers

La forme de la configuration des *Loggers* est similaire à celle des *Appenders*, néanmoins, les seules choses à configurer sont les *Appenders* et, éventuellement, le niveau et l'additivité des *Appenders*. Aussi la configuration d'un *Logger* a l'aspect suivant :

```
log4j.logger.nom.du.logger=[niveau], appender1, appender2
```

Où :

- **nom.du.logger** est le nom du *Logger* lorsqu'il est demandé via **Logger.getLogger** (vous pouvez ne spécifier qu'une partie du nom du *Logger* et vous reposer sur les mécanismes d'héritage de propriétés) ;
- **niveau** est le nom du niveau à attribuer au *Logger*, s'il n'est pas précisé, le niveau est hérité du parent (ou positionné à **DEBUG** pour le *Logger* racine) ;
- **appender*n*** est le nom d'un *Appender* tel qu'il a été déclaré par **log4j.appender.nomAppender**.



Le paramètre **niveau** est facultatif mais **pas** la virgule qui le suit !

La seule particularité concerne la configuration du *Logger* racine. En effet, celui-ci n'a pas de nom attribué puisqu'il n'est accessible que par la méthode **Logger.getRootLogger()**. Aussi, il existe une clé particulière servant à sa configuration qui est **log4j.rootLogger**

Le paramétrage de l'additivité d'un *Logger* en ce qui concerne les *Appenders* se fait de façon quelque peu contre-intuitive. Alors que l'on pourrait penser que l'additivité est un paramètre du *Logger* et qu'à ce titre elle pourrait être configurée selon le même schéma que les paramètres des *Appenders*, l'équipe de log4j n'a pas fait ce choix. La configuration de l'additivité se fait de la façon suivante :

```
log4j.additivity.nom.du.logger=true|false
```

Par défaut, l'additivité est active.

Ainsi, vous pouvez configurer le *Logger* **com.developpez.beuss.java.log4j.MonLogger** de plusieurs façons :

Configuration directe

```
log4j.logger.com.developpez.beuss.java.log4j.MonLogger=INFO, UnAppender
```


Héritage (tous les loggers du paquetage)

```
log4j.logger.com.developpez.beuss.java.log4j=, UnAppender
```

Notez que le niveau n'est pas précisé, cela signifie qu'il est hérité du *Logger* parent le plus proche.

Héritage (logger racine)

```
log4j.logger.rootLogger=INFO, UnAppender
```

 Si vous avez défini vos propres *Levels* vous pouvez configurer les *Loggers* pour qu'ils les utilisent en spécifiant un niveau de la forme **NomNiveau#ClasseNiveau**. **NomNiveau** sera passé à la méthode statique **toLevel(String)** qui est chargée de renvoyer un **Level** correspondant à celui décrit par la chaîne.

III.B.3 - Fichiers XML

Comme nous l'avons déjà vu, les fichiers XML sont beaucoup plus structurants que les fichiers properties. Ils obligent à une certaine disposition des différents éléments puisque le fichier de configuration XML est validé vis-à-vis de sa DTD lorsqu'il est chargé. La structure (simplifiée) d'un fichier de configuration log4j au format XML est la suivante :

- Configuration des *Appenders* ;
- Configuration des *Loggers* ;
- Configuration du *Logger* racine.

Les différentes parties de la configuration sont détaillées dans les sections suivantes. La syntaxe globale du fichier de configuration est la suivante :

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/" [threshold="all"]
[debug="false"]>
  <!-- Déclaration des différents Appenders et Loggers -->
</log4j:configuration>
```

Les propriétés **threshold** et **debug** sont identiques aux clés correspondantes dans la configuration *via* un fichier properties, elles permettent respectivement d'indiquer le niveau minimum que doit avoir un message pour être transmis aux *Loggers* et de demander à log4j d'afficher des informations sur le déroulement des opérations.

III.B.3.a - Configuration des Appenders

La configuration d'un *Appender* se fait au moyen de la balise **appender** qui prend la forme suivante :

```
<appender name="NomAppender" class="ClasseImplementation">
  [<param name="ParamètreAppender1" value="ValeurParamètreAppender1"/>]
  [<layout class="ClasseLayout"<
    [<param name="ParamètreLayout1" value="ValeurParamètreAppender1"/>]
  </layout>]
</appender>
```

Les balises **param** (que ce soit pour l'*Appender* ou le *Layout*) et **layout** sont facultatives. Ainsi, si l'on reprend la configuration de l'*Appender* LF5 présentée dans la configuration *via* fichiers properties, elle aura l'aspect suivant en

XML :

```
<appender name="console" class="org.apache.log4j.ConsoleAppender">
  <layout class="org.apache.log4j.SimpleLayout"/>
</appender>
```

Les *Appenders* acceptent d'autres balises (notamment *filter*) mais celles-ci n'interviennent que dans le cas de configurations plus avancées. Si vous voulez en savoir plus, quelques pistes sont donnés dans les sections "Seuils" et "Un petit mot sur les ErrorHandlers".

III.B.3.b - Configuration des Loggers

On ne change pas une équipe qui gagne, la configuration des *Loggers* se fait au moyen de la balise... **logger**. Sa forme est la suivante :

```
<logger name="nomLogger" [additivity="false"]>
  [<level value="NomLevel"/>]
  [<appender-ref ref="NomAppender1"/>]
</logger>
```


- Le nom du *Logger* (**name**) est celui utilisé lors de l'appel à **getLogger**, comme pour la configuration *via* les fichiers propriétés, vous pouvez utiliser l'héritage et l'additivité ;
- Le paramètre **additivity** indique si l'additivité des *Appenders* s'applique ou non ;
- La balise **level** permet d'indiquer le niveau minimum qu'un message doit avoir pour être pris en compte par le *Logger* ;
- L'attribut **ref** de la balise **appender-ref** doit renvoyer au nom d'un *Appender* déclaré précédemment. Il peut y avoir plusieurs balises **appender-ref** pour un *Logger*.

Le *Logger* racine doit être configuré d'une façon particulière. Contrairement à la configuration *via* un fichier propriétés, il n'a pas de nom spécial mais reçoit sa propre balise : **root**. Sa configuration suit le même modèle que les autres *Loggers*. Si vous observez attentivement la DTD, vous constaterez que d'autres balises sont disponibles pour **root**, elles ne sont là que pour assurer la compatibilité et ne doivent pas être utilisées.

 Comme pour la configuration *via* un fichier propriétés, si vous avez défini vos propres *Levels* vous pouvez configurer les *Loggers* pour qu'ils les utilisent en spécifiant un niveau de la forme **NomNiveau#ClasseNiveau**. **NomNiveau** sera passé à la méthode statique **toLevel(String)** qui est chargée de renvoyer un **Level** correspondant à celui décrit par la chaîne. Si vos niveaux utilisent des paramètres particuliers, vous devez définir des *setters* (méthodes **setXXX**) permettant de les positionner puis placer des balises **<param name="nom" value="valeur">** au sein de la balise **Level**.

IV - Propriétés utiles

Les deux sections qui suivent présentent une partie des propriétés de certains *Appenders* et *Layouts*. Elles n'ont pas pour vocation d'être exhaustives mais simplement d'aider à la configuration.

 **Rappel** : l'attribution de valeurs aux propriétés d'un *Appender* se fait en utilisant la syntaxe **log4j.appender.nomAppender.nomPropriété=valeur** dans une configuration via un fichier propriétés et par le biais de la balise **param** imbriquée dans la balise **appender** adéquate pour une configuration XML. Pour les *layouts*, on utilisera la syntaxe **log4j.appender.nomAppender.layout.nomPropriété** ou une balise **param** imbriquée dans la balise **layout** idoine. Pour plus de détails sur les deux formats de configuration, reportez vous à la section *Format des fichiers de configuration*.

IV.A - Appenders

IV.A.1 - Seuils

Comme nous l'avons vu lors de l'explication du concept d'*Appender*, il est possible de définir un seuil au niveau de l'*Appender* en dessous duquel les messages ne seront pas journalisés. Il vient en complément du seuil global et du niveau propre du chaque *Logger*. Ce seuil est positionnable *via* la propriété **threshold** :

Appender journalisant les messages INFO et supérieurs

```
<appender name="console" class="org.apache.log4j.ConsoleAppender">
  <param name="threshold" value="INFO"/>
  <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="%-5p %c{1} - %m%n" />
  </layout>
</appender>
```

Ce paramètre est pratique lorsque l'on veut que l'*Appender* ne prenne en compte que les messages au dessus d'un certain niveau. Néanmoins, il peut être souhaitable de réellement séparer les différents niveaux pour, par exemple, avoir un fichier contenant les messages de débogage, un autre les messages d'information, *etc*. Dans ce cas, le paramètre **threshold** n'est plus suffisant. Pour cela, il faut utiliser un autre mécanisme de log4j : les filtres. Les filtres ne sont supportés que dans la configuration XML, ils se placent au sein de la balise **appender**. Dans notre cas, on peut utiliser le **LevelMatchFilter** ou le **LevelRangeFilter** selon que l'on veut ne journaliser que les messages d'un niveau donné ou ceux compris dans un intervalle de niveau. Voici un exemple de configuration :

```
<!-- Premier Appender, ne journalise que les messages DEBUG -->
<appender name="filedebug" class="org.apache.log4j.FileAppender">
  <param name="file" value="debug.log"/>
  <layout class="org.apache.log4j.SimpleLayout"/>
  <filter class="org.apache.log4j.varia.LevelMatchFilter">
    <param name="levelToMatch" value="DEBUG"/>
  </filter>
  <filter class="org.apache.log4j.varia.DenyAllFilter"/>
</appender>
<!-- Second Appender, ne journalise que les messages INFO à ERROR (exclu donc FATAL) -->
<appender name="fileinfoup" class="org.apache.log4j.FileAppender">
  <param name="file" value="infoup.log"/>
  <layout class="org.apache.log4j.SimpleLayout"/>
  <filter class="org.apache.log4j.varia.LevelRangeFilter">
    <param name="levelMin" value="INFO"/>
    <param name="levelMax" value="ERROR"/>
  </filter>
</appender>
<root>
  <appender-ref ref="filedebug" />
  <appender-ref ref="fileinfoup"/>
</root>
```

Le **LevelMatchFilter** prend comme paramètre **levelToMatch** qui est le niveau auquel doit correspondre le message pour ne pas être rejeté (on peut éventuellement lui passer **acceptOnMatch** avec la valeur **false** pour qu'au contraire le message soit rejeté). Il est nécessaire d'appliquer un second filtre qui rejete tous les messages qu'il rencontre. C'est dû au comportement du **LevelMatchFilter** : si le niveau du message correspond au niveau défini, le message est accepté sans passer par les autres filtres. Par contre, si le niveau ne correspond pas, le filtre est dit *neutre*, c'est-à-dire qu'il laisse le soin aux filtres suivants de décider ; s'il n'y a pas d'autres filtres le message est accepté ce qui n'est pas le comportement attendu dans notre exemple.

Le second *Appender* est configuré pour accepter les message de INFO à ERROR (inclus), grâce à un **LevelRangeFilter** qui prend en paramètre **levelMin** (niveau minimum que le message doit avoir pour être accepté) et **levelMax** (niveau maximum que le message peut avoir pour être accepté). Ce filtre accepte lui aussi un paramètre **acceptOnMatch**, mais celui-ci a une sémantique un peu différente du **LevelMatchFilter** : s'il est positionné à **false** le filtre rejette le message s'il est à l'extérieur des bornes et reste neutre s'il est à l'intérieur ; si **acceptOnMatch** est à **true** (par défaut) les autres filtres ne sont pas consultés et le message est accepté directement s'il est dans l'intervalle.

Les filtres sont des outils très puissants, cependant, il est recommandé d'utiliser en priorité les mécanismes de filtrage des *Loggers* et des *Appenders* avant d'y avoir recours.

IV.A.2 - ConsoleAppender

Le **ConsoleAppender** journalise, comme son nom l'indique, vers la console. Il est cependant possible de choisir si les messages sont envoyés vers la sortie standard ou vers la sortie des erreurs. Ce comportement est contrôlé par le paramètre **target** qui prend les valeurs **System.out** (sortie standard) ou **System.err** (sortie des erreurs).

IV.A.3 - FileAppender

Le **FileAppender** est la classe de base pour tous les *Appenders* de type *RollingFile*, les propriétés décrites dans cette section s'appliquent donc également à ces *Appenders*. Notez que certaines propriétés proviennent en fait du **WriterAppender**

- **append** : indique si l'écriture doit se faire à la suite dans le fichier (**true**, valeur par défaut) ou si le contenu doit être écrasé (**false**) ;
- **encoding** : définit l'encodage du fichier produit (utilisation des propriétés système si aucun encodage n'est précisé) ;
- **bufferedIO** : indique si un tampon doit être utilisé pour l'écriture (**false** par défaut) ;
- **bufferSize** : dans le cas où un tampon est utilisé pour la sortie, indique la taille du buffer en octets (8Ko par défaut) ;
- **file** : définit le nom de base du fichier où effectuer la journalisation (des suffixes peuvent y être ajoutés dans le cas des *RollingFileAppenders*).



Si vous êtes sous Windows, n'oubliez pas de doubler les slashes dans la déclaration du chemin du fichier (par exemple : **C:\Vogs\debug.log**).

IV.A.4 - JDBCAppender

Le **JDBCAppender** enregistre les messages de journalisation dans une base de données. Il est donc nécessaire de lui fournir les informations nécessaires à cette insertion :

- **driver** : indique le *Driver* JDBC à utiliser pour la connexion ;
- **url** : URL JDBC utilisée pour se connecter à la base ;

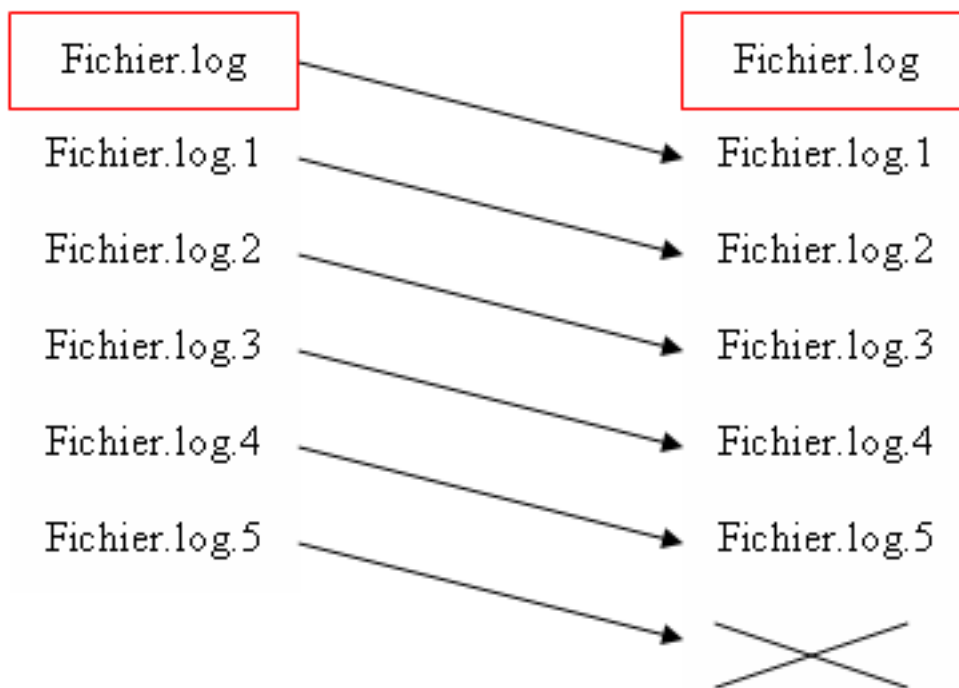
- **user, password** : nom d'utilisateur et mot de passe (propriétés distinctes), l'utilisateur doit avoir le droit d'insérer de nouvelles lignes ;
- **bufferSize** : nombre d'événements de journalisation à recevoir avant de les envoyer en base ;
- **sql** : requête d'insertion à utiliser, elle suit le format du *PatternLayout*, par exemple **"INSERT INTO journal(niveau, logger, message) VALUES('%p', '%c', '%m');"**.

IV.A.5 - DailyRollingFileAppender

Le paramètre important de cet *Appender* est **datePattern**, en effet, c'est lui qui va contrôler la fréquence du roll-over du fichier journal ; il détermine également le modèle du nom de fichier. Le motif utilisé suit les conventions de **SimpleDateFormat**, la précision va jusqu'à la minute (c'est-à-dire que le fichier est sauvegardé et renommé toutes les minutes). Consultez la Javadoc de **DailyRollingFileAppender** pour des exemples et une description de l'interprétation du format.

IV.A.6 - RollingFileAppender & ExternallyRolledFileAppender

Les *RollingFileAppender* utilisent un jeu de fichiers tournants pour effectuer leur journalisation. C'est-à-dire que lorsque le roll-over intervient, soit parce que le fichier atteint une certaine taille (**RollingFileAppender**) soit parce que cela a été demandé (**ExternallyRolledFileAppender**), le fichier journal courant est archivé en **nomdefichier.1** tandis que l'indice des autres archives est incrémenté de un. Si le suffixe fichier le plus ancien dépasse une certaine valeur, celui-ci est supprimé.



Fonctionnement des RollingFileAppenders

Les éléments configurables dans ce mécanisme sont l'indice maximal des fichiers de sauvegarde (par le biais de la propriété **maxBackupIndex**) et, uniquement pour le **RollingFileAppender**, la taille du fichier déclenchant le roll-over (via la propriété **maxFileSize**).

Le déclenchement du roll-over pour l'**ExternallyRolledFileAppender** se fait d'une façon similaire à la suivante :

```
Socket sock = new Socket([Adresse où tourne l'Appender], [port sur lequel écoute l'Appender]);
DataOutputStream dos = new DataOutputStream(sock.getOutputStream());
dos.writeUTF("RollOver");
dos.close();
sock.close();
```

Le port sur lequel écoute l'**ExternallyRolledFileAppender** est configurable *via* la propriété **port** (tout simplement), mais la chaîne déclenchant le roll-over ne l'est pas. Notez que la casse est prise en compte, si l'*Appender* reçoit autre chose que "RollOver", il renverra un message d'erreur *via* le *Socket*.

IV.A.7 - AsyncAppender

L'**AsyncAppender** fait suivre les messages aux *Appenders* qui lui sont attachés dans un *Thread* distinct. Cela permet de ne pas bloquer l'application effectuant la journalisation. Les événements sont journalisés un par un au fur et à mesure. Lorsque la file est pleine, l'**AsyncAppender** doit attendre qu'une place se libère avant de pouvoir continuer. Il n'est configurable que par un fichier XML.

Tout d'abord, il faut lui indiquer les *Appenders* auxquels il va devoir faire suivre les événements. Il ne s'agit pas à proprement parler de propriétés puisque ce sont les *Appenders* proprement dits qui sont utilisés et non pas une valeur quelconque. Aussi, il existe une balise spéciale : **appender-ref**. Celle-ci dispose d'un attribut **ref** qui doit prendre pour valeur le nom d'un *Appender*. Il est possible d'affecter plusieurs *Appenders* à un **AsyncAppender**.

Ensuite, il est possible de spécifier la taille de la file d'attente des messages. Dans une application envoyant beaucoup de messages à des *Appenders* assez lents, il peut être utile d'augmenter sa taille. Cela se fait *via* la propriété **bufferSize** (128 événements par défaut). Enfin, si vos *Appenders* affichent les informations sur l'endroit dans le code d'où est émis le message, il est utile de positionner la propriété **locationInfos** à **true**.

Exemple de configuration d'AsyncAppender

```
<appender class="org.apache.log4j.FileAppender" name="file">
  <param name="file" value="extern.log"/>
  <layout class="org.apache.log4j.SimpleLayout"/>
</appender>

<appender class="org.apache.log4j.AsyncAppender" name="async">
  <appender-ref ref="file"/>
</appender>
```

IV.B - Layouts


IV.B.1 - PatternLayout

Le **PatternLayout** est le *Layout* le plus souple et le plus configurable. Comme son nom l'indique, il utilise un motif pour mettre en forme les informations fournies par l'événement de journalisation. Sa seule propriété se nomme **conversionPattern**, elle permet d'indiquer la façon dont doivent être formatées les informations. Ce formatage s'effectue au moyen de séquences d'échappement. En voici quelques unes :

- **%c** : nom du logger ;
- **%m** : message de journalisation ;
- **%n** : caractère de nouvelle ligne spécifique à la plate-forme. N'oubliez pas de l'ajouter à la fin de la ligne ;
- **%p** : niveau de gravité de l'événement.

Il est possible de contrôler l'alignement des différentes informations au moyen de chiffres placés entre le signe %

et la lettre indiquant l'information à afficher. Reportez vous à la JavaDoc pour plus d'informations. Ce *Layout* est celui utilisé entre autres par le **JDBCAppender** pour la définition de la requête SQL.

 *Il existe toute une variété de séquences d'échappement permettant d'obtenir des informations sur la provenance du message de journalisation (classe, fichier, ligne, méthode), l'extraction de ces informations est relativement lente, il est donc préférable de ne pas utiliser ces options en environnement de production.*

IV.B.2 - HTMLLayout & XMLLayout

Les *Layouts* **HTMLLayout** et **XMLLayout** ont un paramètre en commun appelé **locationInfo** qui sert à demander au *Layout* d'inclure des informations sur la provenance du message dans le code. Le **HTMLLayout** indiquera le fichier et la ligne d'où provient le message tandis que le **XMLLayout** inclura également les noms de la classe et de la méthode.

Le **HTMLLayout** accepte aussi un paramètre **title** indiquant le titre que doit porter la page récapitulant les informations.

Log session start time Sun Jul 31 11:20:01 CEST 2005

Time	Thread	Level	Category	Message
0	main	DEBUG	com.developpez.beuss.log4j.tuto.Main	Tour de boucle n°1
10	main	DEBUG	com.developpez.beuss.log4j.tuto.Main	Tour de boucle n°2
10	main	DEBUG	com.developpez.beuss.log4j.tuto.Main	Tour de boucle n°3
10	main	DEBUG	com.developpez.beuss.log4j.tuto.Main	Tour de boucle n°4
10	main	DEBUG	com.developpez.beuss.log4j.tuto.Main	Tour de boucle n°5
10	main	DEBUG	com.developpez.beuss.log4j.tuto.Main	Tour de boucle n°6
10	main	DEBUG	com.developpez.beuss.log4j.tuto.Main	Tour de boucle n°7
10	main	DEBUG	com.developpez.beuss.log4j.tuto.Main	Tour de boucle n°8
10	main	DEBUG	com.developpez.beuss.log4j.tuto.Main	Tour de boucle n°9
10	main	DEBUG	com.developpez.beuss.log4j.tuto.Main	Tour de boucle n°10

Exemple de sortie du HTMLLayout

V - Un petit mot sur les nouveautés de la version 1.3

V.A - ErrorHandler

Vous aurez peut-être remarqué la présence d'une balise **errorHandler** qu'il est possible d'intégrer au sein de balises **appender** dans la configuration XML. Il s'agissait d'un mécanisme conçu pour pouvoir avoir un *Appender* de repli dans le cas où l'*Appender* spécifié n'aurait pas pu fonctionner. Néanmoins, en raison de la façon dont étaient implémentés les *ErrorHandlers*, il était très difficile de s'assurer que les *Appendes* fonctionnaient avec ce mécanisme. Aussi Ceki Güلكü (le créateur de log4j) a décidé de les supprimer en concertation avec l'équipe de développeurs :

"Given the extreme difficulty in testing generic error handlers, log4j 1.3 no longer supports ErrorHandler."

V.B - WatchDogs

Si vous avez déjà utilisé log4j auparavant et que vous initialisiez la configuration manuellement (en la programmant), peut-être vous êtes vous servi de la méthode **configureAndWatch** qui permettait de charger la configuration et de surveiller si un changement y était apporté pour la recharger périodiquement. Ces méthodes disparaissent dans la version 1.3 au profit des *WatchDogs* qui offrent un système plus souple au niveau des extensions possibles.

VI - Conclusion

Voilà, vous en savez maintenant un peu plus sur log4j. Sachez qu'il a été porté sur d'autres langages, pour plus d'informations, reportez vous au site [des services de journalisation Apache](#). La version 1.3 de log4j apportera son lot de nouveautés, nous en reparlerons quand une version stable sera sortie.

Si vous avez des corrections ou des suggestions pour améliorer cet article, vous pouvez me contacter par [MP](#) sur le forum

VII - Remerciements

Merci à Maximilian pour sa relecture et à Wookai pour ses corrections.

VIII - Téléchargements

- Le site de [log4j](#) ;
- L'article au format [HTML Zippé \(mirroir\)](#) ;
- L'article au format [PDF \(mirroir\)](#).