PROJETO FRAMEWORK - CELEPAR

CONTROLE DE LOG EMAPLICAÇÕES JAVA PARA INTERNET

Fevereiro – 2005

Sumário de Informações do Documento

Tipo do Documento: Definição		
Título do Documento: Processo de Desenvolvimento de Software		
Estado do Documento: EB (Elaboração)		
Responsáveis: Grupo Framework		
Palavras-Chaves: log4j, log, auditoria		
Resumo: Demostração da instalação e configuração para utilização do Log4j.		
Número de páginas: 21		
Software utilizados:		
Versão	**Data**	**Mudanças**
1.0		

Sumário

1 CONTROLE DE LOG EMAPLICAÇÕES JAVA PARA INTERNET

1.1 Introdução

Para geração e gravação de log nas aplicações web desenvolvidas pela empresa adouto-se a utilização do framework chamado log4j. A utilização deste framework visa eliminar a utilização de System.out dentro do código das aplicações.

Log4j é um projeto de código aberto (open Source). Ele permite ao desenvolvedor controlar, de maneira flexível, cada saída de log. Com o ele, podemos ativar ou desativar o log em tempo de execução sem modificar os binários da aplicação, esse comportamento pode ser controlado apenas editando um arquivo de configuração.

Uma das características do Log4j é a herança em loggers. Usando uma hierarquia de loggers é possível controlar quais instruções de log serão usadas. Isto ajuda a reduzir o volume de saídas e minimizar seu custo.

A saída poderá ser um arquivo, um OutputStream, um java.io.Writer, um e-mail, um banco de dados, um arquivo XML, um arquivo HTML, ou mesmo o console.

1.2 Instalação

A instalação do Log4j ocorre em dois passos. O primeiro é colocar no classpath de sua aplicação o arquivo log4j-x.x.x.jar (onde x.x.x é a versão). O segundo passo é a configuração.

Para se usar o Log4j deve-se ter um considerável planejamento e esforço, pois após colocarmos instruções de log dentro do código de um sistema, é importante que não seja preciso modificá-las manualmente.

Esse planejamento é feito na configuração, esta é, talvez, a parte mais importante ao utilizar o Log4j. A configuração pode ser feita por programação, entretanto, fica claro que é obtida uma flexibilidade muito maior ao usarmos outras duas formas possíveis de configuração: um arquivo xml ou um arquivo de propriedades (no formato chave=valor). Aqui será mostrada a configuração utilizando

um arquivo de propriedades (properties) feita através do arquivo *"log4j.properties"*, que também deve estar no classpath da aplicação.

Configurar o Log4j é de fato definir os loggers para mostrar as mensagens de log nos appenders escolhidos usando layouts determinados pelo desenvolvedor. Os loggers podem ser criados e configurados em qualquer ordem, em particular um logger irá encontrar e agrupar seus descendentes mesmo que ele seja instanciado depois deles.

Isto tudo pode ser feito em um arquivo que deve ser chamado de log4j.properties, desta forma a configuração é feita apenas uma vez e é carregada na inicialização da aplicação. Este arquivo deve ser colocado no classpath da aplicação.

Veja um exemplo:

```
#### Usando 2 appenders, 1 para logar no console, outro para um arquivo
log4j.rootCategory=INFO, stdout, fileOut

#### O primeiro appender escreve no console
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

# Pattern que mostra o nome do arquivo e numero da linha.
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] (%F:%L) - %m%n

#### O segundo appender escreve em um arquivo
log4j.appender.fileOut =org.apache.log4j.RollingFileAppender
log4j.appender.fileOut.File=aplicacao.log

# Controla o tamanho maximo do arquivo
log4j.appender.fileOut.MaxFileSize=500KB

# Faz backup dos arquivos de log (apenas 1)
log4j.appender.fileOut.MaxBackupIndex=1

log4j.appender.fileOut.layout=org.apache.log4j.PatternLayout
log4j.appender.fileOut.layout.ConversionPattern=%5p [%t] (%F:%L) - %m%n
```

1.3 Componentes

O Log4j possui três componentes principais: loggers, appenders e layouts. Eles trabalham juntos para que os desenvolvedores façam o log de suas mensagens de acordo com o tipo e o nível delas, e controle em tempo de execução como estas mensagens são formatadas e onde são reportadas. O Logger tem a função de

receber as mensagens e incluí-las ao output de acordo com a configuração, Appender é o output do log. A forma como o log é exibido é definida pelo Layout. Nos próximos itens está uma explicação mais detalhada sobre cada um.

1.3.1 Logger

A primeira e principal vantagem de qualquer API de log sobre o System.out.println está em sua habilidade de desabilitar certos níveis de log (quando estes não são necessários) enquanto os demais níveis continuam funcionando normalmente.

O Logger é responsável por receber a mensagem e torná-la disponível para o Appender. Ao utilizar o Log4j é necessário configurar o nível hierárquico desejado para a exibição das mensagens. Como visto na listagem anterior, o logger está configurado como INFO, portanto, serão exibidos todos os logs FATAL, WARN, ERROR e INFO. O logger segue a hierarquia DEBUG, INFO, WARN, ERROR e FATAL . Se for setado logger = ERROR, serão exibidas as mensagens ERROR e FATAL, seguindo a ordem decrescente da hierarquia.

A classe org.apache.log4j.Logger é a classe central do Log4j, a maioria das operações de log, exceto configuração, são feitas através desta classe.

Para exemplificar a utilização considere a seguinte classe para teste:

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

public class Log4jTeste {

   static Logger logger = Logger.getLogger(Log4jTeste.class);

   public static void main(String[] args) {

        logger.debug("DEBUG MSG");
        logger.info("INFO MSG");
        logger.warn("WARN MSG");
        logger.error("ERROR MSG");
        logger.fatal("FATAL MSG");

   }
}
```

De acordo com a configuração onde INFO é o nível de log a ser exibido, ao ser executada a classe o resultado obtido tanto no console quanto no arquivo aplicacao.log é o seguinte:

```
 INFO [main] (Log4jTeste.java:11) - INFO MSG
 WARN [main] (Log4jTeste.java:12) - WARN MSG
ERROR [main] (Log4jTeste.java:13) - ERROR MSG
FATAL [main] (Log4jTeste.java:14) - FATAL MSG
```

1.3.2 Appender

Um appender é uma saída onde são mostradas as mensagens, o console, algum arquivo, um servidor remoto de sockets, etc. Um logger pode ter mais de um appender. Por exemplo, podemos mandar as mensagens de log para o console e para um arquivo ao mesmo tempo. Além disso podemos mostrar níveis diferentes de mensagens nos appenders diferentes, ou seja, mandar para o console todas as mensagens enquanto que para o arquivo podemos salvar apenas as mensagens de nível WARN, por exemplo.

Appenders:

- ConsoleAppender - envia as mensagens de log para o System.out (default) ou System.err (mudado com a propriedade Target);
- FileAppender - envia para um arquivo;
- RollingFileAppender - é subclasse de FileAppender, sendo que este pode fazer um backup do arquivo sempre que ele atingir um determinado tamanho;
- DailyRollingFileAppender - é subclasse de FileAppender, este appender pode fazer um backup de tempos em tempos (definido pelo desenvolvedor, a cada semana por exemplo), para setar o tempo, basta usar o mesmo pattern da classe SimpleDateFormat na propriedade DatePattern;
- SMTPAppender - appender para enviar o log para um destinatário de e-mail;
- SocketAppender - envia os eventos de log para um servidor de logs remoto através do protocolo TCP;
- NTEventLogAppender - envia para o sistema de log de uma máquina com plataforma Windows;

- SyslogAppender - envia os logs para um daemon (monitor de logs) remoto;
- JMSAppender - envia os logs como uma mensagem JMS;
- JDBCAppender – salva os logs em uma tabela de um banco de dados;
- AsyncAppender - possibilita que os logs sejam coletados em um buffer e depois enviados para um ou mais appender anexados a ele;
- NullAppender - meramente existe mas não manda mensagens para nenhum dispositivo.

Para filtrar os tipos de logs, basta setar a propriedade Threshold=<nivel minimo>, desta forma serão mostradas todas as mensagens que tiverem prioridade igual ou maior do que o nível setado, seguindo a ordem: DEBUG < INFO < WARN < ERROR < FATAL.

Adicionar a seguinte linha no arquivo log4j.properties,

```
log4j.appender.stdout.Threshold=WARN
```

significa dizer que somente as mensagens com nível WARN, ERROR e FATAL serão mostradas no console.

1.3.3 Layout

Layout é o formato como a mensagem vai ser mostrada. Este formato pode incluir a data da mensagem, qual o método que contém o log, qual linha, classe, arquivo, etc.

Log4j fornece as seguintes classes de Layout:

```
org.apache.log4j.Layout (abstrata)
        -org.apache.log4j.SimpleLayout
        -org.apache.log4j.PatternLayout
        -org.apache.log4j.HTMLLayout
        -org.apache.log4j.XMLLayout
        org.apache.log4j.DateLayout(abstrata)
                -org.apache.log4j.TTCCLayout
```

A utilização de um destes layouts é simples, veja este exemplo que irá criar a saída em uma tabela HTML:

```
log4j.rootCategory=DEBUG, htmlOut
!--------------------------HTMLLayout OPTIONS--------------------------!
log4j.appender.htmlOut=org.apache.log4j.RollingFileAppender

log4j.appender.htmlOut.File=logging/HTMLLayout.html
log4j.appender.htmlOut.layout=org.apache.log4j.HTMLLayout

log4j.appender.htmlOut.layout.LocationInfo=true
log4j.appender.htmlOut.layout.Title=Log gerado pelo Log4j
```

Isto irá criar um arquivo com o nome HTMLLayout.html, e neste arquivo terá uma tabela com o tempo decorrido em milisegundos desde quando o programa foi iniciado, a thread que disparou o log, o nível de prioridade, a classe, o nome do arquivo *.java desta classe e o número da linha (se LocationInfo = true) e a mensagem de log.

Ao enviar o log para o console ou para um arquivo por exemplo, podemos mostrar tais informações na ordem em que escolhermos e com o formato que desejarmos, um ótimo exemplo é o formato da data, podemos mostrar "dd/mm/aaaa" ou "aaaa-mm-dd" com hora ou não. Isto depende do pattern usado. Veja este exemplo:

```
Pattern: "%r [%t] %-5p %c{2} - %m%n"


176 [main] INFO  examples.Sort - Populating an array of 2 elements in
reverse order
```

r - número de milissegundos transcorridos desde o inicio do programa;

t - nome da thread que gerou o evento de log;

p - prioridade (o -5 indica que deve alinhar a direita se o número de caracteres for menor que cinco);

c - nome da classe (2 indica que se o nome completo da classe for "a.b.c" por exemplo, deverá ser mostrado apenas "b.c");

m - é a mensagem (não pode faltar !);

n - é o separador de linhas padrão do sistema operacional - "\n" ou "\r\n".

Para usar os patterns devemos dizer que o layout será da classe org.apache.log4j.PatternLayout. Ela serve justamente para formatar um evento de log e retornar uma string que será mostrada no appender escolhido, para fazer a formatação usamos uma propriedade de PatternLayout chamada ConversionPattern.

Veja como esta configuração é simples:

```
log4j.rootCategory=DEBUG, stdOut

# stdOut usa PatternLayout
log4j.appender.stdOut.layout=org.apache.log4j.PatternLayout
log4j.appender.stdOut.layout.ConversionPattern=%-2d{dd/MM/yy HH:mm} [%t]
%5p %c:%L - %m%n
```

Isto irá mostrar

```
22/01/04 13:07 [main]  INFO logging.UseLogger :12 - Iniciando a aplicacao
22/01/04 13:07 [main]  INFO logging.UseLogger :14 - Fim da aplicacao
```

Para ver a lista completa dos patterns, consulte a documentação da classe PatternLayout na versão on-line da documentação em http://logging.apache.org/log4j/docs/index.html

1.4 Customizando a saída

Como observado anteriormente na instanciação de um Logger o parâmetro passado é um “class” ou ainda pode ser uma String, a prática adotada é a de cada classe possuir um logger como atributo estático instanciado com a própria classe e depois apenas invocar o método apropriado de acordo com a categoria de log que se pretende (DEBUG, INFO, WARN, ERROR, FATAL) como segue:

```
package log.exemplo.configuracao;

import org.apache.log4j.Logger;

public class SuaClasse {

   private static Logger logger = Logger.getLogger(SuaClasse.class);

   ....

        logger.debug("DEBUG MSG");
   ...
}
```

Para a customização é de fundamental importância o nome dado ao logger, neste caso o nome acaba sendo o nome da própria classe acrescido do seu pacote completo (log.exemplo.configuracao.SuaClasse).

Então no arquivo log4j.properties poderia ter a seguinte configuração:

```
log4j.rootLogger=INFO, stdout

log4j.logger.log.exemplo.configuracao=INFO, fileOut

#### O primeiro appender escreve no console
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%5p [%t] (%F:%L) - %m%n

#### O segundo appender escreve em um arquivo
log4j.appender.fileOut =org.apache.log4j.RollingFileAppender
log4j.appender.fileOut.File=aplicacao.log

# Controla o tamanho maximo do arquivo
log4j.appender.fileOut.MaxFileSize=100KB

# Faz backup dos arquivos de log (apenas 1)
log4j.appender.fileOut.MaxBackupIndex=1

log4j.appender.fileOut.layout=org.apache.log4j.PatternLayout
log4j.appender.fileOut.layout.ConversionPattern=%5p %d{ABSOLUTE} %c{1}:%L - %m%n
```

No qual a segunda linha de configuração (log4j.logger.log.exemplo.configuracao=INFO, fileOut) define que todos os loggers do pacote log.exemplo.configuracao e seus sub-pacotes terão suas mensagens com a categoria a partir de INFO tratadas pelo appender fileOut, o qual vai gravar a mensagem em um arquivo.

Podem ainda haver casos em que se queira fazer uma auditoria de uma aplicação fazendo com que essas mensagens fiquem isoladas das demais mensagens de log, para isso é utilizada a instanciação do logger passando como parâmetro uma String, que terá um nome que facilite a configuração posteriormente. Ainda haverá casos em que determinada classe possa gerar ambas mensagens de log, as de auditoria e as demais mensagens padrões da aplicação. Neste caso a solução a ser adotada seria de a classe possuir dois atributos logger, um padrão e o customizável. Essa solução poderia ainda ser utilizada caso mais loggers fossem necessários. Veja o exemplo:

```
package log.exemplo.configuracao;

import org.apache.log4j.Logger;

public class SuaClasse {

   private static Logger logger = Logger.getLogger(SuaClasse.class);
   private static Logger audit= Logger.getLogger("AUDTORIA_APLICACAO");

   ....

        logger.debug("DEBUG MSG");
   ...
        audit.error("ERRO ao ...");
}
```

Então o arquivo log4j.properties poderia passar a ter o seguinte conteúdo:

```
log4j.rootLogger=INFO, stdout

log4j.logger.log.exemplo.configuracao=INFO, fileOut

log4j.logger.AUDITORIA_APLICACAO=INFO, fileAuditoria


#### O primeiro appender escreve no console
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%5p [%t] (%F:%L) - %m%n

#### O segundo appender escreve em um arquivo
log4j.appender.fileOut =org.apache.log4j.RollingFileAppender
log4j.appender.fileOut.File=aplicacao.log

# Controla o tamanho maximo do arquivo
log4j.appender.fileOut.MaxFileSize=100KB

# Faz backup dos arquivos de log (apenas 1)
log4j.appender.fileOut.MaxBackupIndex=1

log4j.appender.fileOut.layout=org.apache.log4j.PatternLayout
log4j.appender.fileOut.layout.ConversionPattern=%5p %d{ABSOLUTE} %c{1}:%L - %m%n

#### O terceiro appender escreve em um outro arquivo as mensagens especificas
log4j.appender.fileAuditoria =org.apache.log4j.RollingFileAppender
log4j.appender.fileAuditoria.File=auditoria_aplicacao.log

# Controla o tamanho maximo do arquivo
log4j.appender.fileAuditoria.MaxFileSize=100KB

# Faz backup dos arquivos de log (apenas 1)
log4j.appender.fileAuditoria.MaxBackupIndex=1

log4j.appender.fileAuditoria.layout=org.apache.log4j.PatternLayout
log4j.appender.fileAuditoria.layout.ConversionPattern=%5p %d{ABSOLUTE} %c{1}:%L - %m%n
```

No qual a terceira linha de configuração (log4j.logger.AUDITORIA_APLICACAO=INFO, fileAuditoria) define que todos os

loggers “AUDITORIA_APLICACAO” independente da classe de onde foram invocados terão suas mensagens com a categoria a partir de INFO tratadas pelo appender fileAuditoria, o qual vai gravar a mensagem em um outro arquivo.

Como observado a API oferece diversas maneira de se configurar e utilizar o log, a qual exige um pequeno planejamento inicial pelo fato de ser ou não necessário a instanciação de mais de um logger.

1.5 Mecanismo simples de log

Aqui será demonstrado um mecanismo simples de log que estará disponível na API de componentes do Framework Celepar, o qual poderá ser reutilizado caso sua funcionalidade seja suficiente (já que é simples) ou ainda ser utilizado como referência para criação ou customização de um mecanismo próprio. O mecanismo a ser demonstrado é baseado nas facilidades disponíveis através da utilização do Hibernate como framework de persistência.

O propósito da solução é registrar operações (INSERT, UPDATE e DELETE) nas tabelas do banco de dados utilizadas pela aplicação, operações que realizadas pelo Hibernate vão registrar o autor, data/hora, tipo de operação, tabela e os dados no momento.

A idéia inicial da solução baseia-se na utilização de um DAO básico (BaseHibernateLogDAO) do qual as demais classes DAO da aplicação devem estender.

1.5.1 gov.pr.celepar.framework.database.log.BaseHibernateLogDAO

Classe DAO Básica (Abstrata) a ser estendida pelas demais classes DAO da aplicação a qual é sub-classe de gov.pr.celepar.framework.database.BaseHibernateDao e possui os seguintes métodos de acesso:

- protected void salvarWithLog(Object obj) – método que deve ser utilizado para realizar o INSERT de determinado objeto e registrar o log da operação;
- protected alterarWithLog(Object obj) – método que deve ser utilizado para realizar

o UPDATE de determinado objeto e registrar o log da operação;

- protected excluirWithLog(Object obj) – método que deve ser utilizado para realizar o DELETE de determinado objeto e registrar o log da operação.
- public abstract String getUser() - método abstrato o qual deve ser implementado na sub-classe e que é responsável por retornar o usuário logado no sistema que está realizando a operação.

1.5.2 Exemplo

O exemplo aqui demonstra a criação de uma classe DAO e de um POJO para manipulação dos dados de um Aluno (tb_aluno) utilizando-se das classes apresentadas anteriormente.

A classe Aluno (POJO) tem o seguinte código:

```
public class Aluno implements Serializable {
    private String nome;
    private String sexo;
    ...
    //Métodos get/set para todas a propriedades
}
```

Como observado é um classe POJO normal com propriedade e métodos get/set para todas elas.

Já a classe AlunoDao tem o seguinte código:

```
import gov.pr.celepar.framework.database.log.BaseHibernateLogDAO;
import javax.servlet.http.HttpServletRequest;
import org.apache.log4j.Logger;

public class AlunoDao extends BaseHibernateLogDAO {
private String user;
private static Logger logger = Logger.getLogger(AlunoDao.class);
   public AlunoDao(HttpServletRequest request) throws Exception{
       super();
       try{
              user = (String)request.getSession().getAttribute("usuario");
       }catch(Exception e){
              logger.error("Erro ao obter o Usuário da Sessão...",e);
       }
   }
   public String getUser() {
       return user;
   }
   public java.util.List listar() throws Exception{
       return listaGenerica(Aluno.class);
   }
   public void salvar(Object obj) throws Exception{
       this.salvarWithLog(obj);
   }
   public void alterar(Object obj) throws Exception{
       this.alterarWithLog(obj);
   }
   public void excluir(Object obj) throws Exception{
       this.excluirWithLog(obj);
   }
}
```

Observe o seguinte:

- a classe estende de BaseHibernateLogDAO;
- a classe possui um atributo String user o qual vai guardar o nome do usuário logado;
- a recuperação do usuário está sendo feita no construtor da classe já que ela está recebendo o HttpServletRequest como parâmetro, então o valor é recuperado da

Sessão;

- a classe implementa o método public String getUser() que acabando retornando o nome do usuário logado na aplicação, o qual foi recuperado anteriormente;
- a classe implementa os demais métodos de manipulação de dados os quais se utilizam de métodos disponibilizados tanto pela classe BaseHibernateLogDAO quanto da classe BaseHibernateDAO.

Para exemplificar, suponhamos uma chamada ao método alterar, o qual se utiliza do método alterarWithLog da classe BaseHibernateLogDAO, a qual irá produzir a seguinte mensagem de log:

```
INFO [OPERACOES_HIBERNATE]:(BaseHibernateLogDAO.java:157) - anonimo -
16/02/2005 10:57:27 - UPDATE - tb_aluno - cgmaluno=923852603#nome=VICTOR
KEVIN KENZO CURCINO KUEKE #dataNasc=1995-06-13#sexo=M#rg=
#ufRg=null#certidaoNasc=092127 #livroCertidaoNasc=A 254
#folhaCertidaoNasc=107 #naturalidade=LONDRINA
#ufNaturalidade=PR#nacionalidade=BRASILEIRA
#cpf=null#rgEstrangeiro=null#tituloEleitor=null#zonaEleitoral=null#secaoEl
eitoral=null#nomeMae=MARIA CLEUDES CURCINO KUEKE #rgMae=
#ufRgMae=null#nomePai=MILTON SHIGUERU KUEKE #rgPai=
#ufRgPai=null#nomeResponsavel=MARIA CLEUDES CURCINO KUEKE
#dataInclusaoAluno=null#dataGeracaoHistorico=null#nomeLogradouro=RUA QUATA
#nomeBairro=JARDIM HEDI #numEnd=795 #complementoEnd=
#numCep=86062320#codEstabInclusao=null#ufEndereco=null#caixaPostal=null#id
entificacaoCopel=null#foneResidencial= 33288878
#foneComercial=null#ramalFoneComercial=null#email=null#foneCelular=null#ob
servacao=null#codigoNis= #codigoLocalidade=null#nomeLocalidade=LONDRINA
#inscricaoImobiliaria=null#carteiraReservista= #indNomeDiferenteAbc=null -
```

Como pode-se observar a mensagem gerada possui as informações configuradas pelo Layout esperado (categoria, logger, classe java, linha e a mensagem propriamente dita) no qual a mensagem propriamente dita possui os seguintes dados:

- usuário;
- data/hora;
- operação realizada;
- tabela do banco de dados;
- dados da tabela.

A mensagem de log mostra os valores atuais de todas as propriedades do objeto (tabela), valores esse inclusive que são recuperados baseados nos arquivos de mapeamento do Hibernate.

1.5.3 Conclusão

O exemplo apresentado demonstra a utilização de um mecanismo simples de log, que se utiliza da API Log4j, que pode ser utilizado para geração de log de operações em bando de dados de determinada aplicação caso a implementação demostrada aqui seja suficiente, ou ainda a idéia aqui demonstrada pode servir como base para a criação de um outro mecanismo mais específico de log que determinada aplicação necessite. Como mencionado anteriormente a implementação básica do mecanismo de log aqui descrita está disponibilizada na API de componentes do Framework Celepar.

2 ANEXOS

2.1 Exemplo configuração abrangente

Veja abaixo um arquivo com muitas opções de configuração do Log4j o qual pode servir como base de documentação na hora da configuração de log de determinada aplicação.

```
#log4j.debug=true
#log4j.disable=fatal
#log4j.additivity.TestLogging=false

log4j.rootLogger=, dest1
log4j.logger.TestLogging=DEBUG, dest1
log4j.appender.dest1=org.apache.log4j.ConsoleAppender
#log4j.appender.dest1.layout=org.apache.log4j.SimpleLayout
log4j.appender.dest1.layout=org.apache.log4j.PatternLayout
#log4j.appender.dest1.layout.ConversionPattern=%-5p %l %x: %m%n


!----------------------####### END OF PROPERTIES #######----------------------!


################################################################################
# Below I document in more detail how to write a log4j configuration file.     #
# SELECTIVELY copy lines beginning with #, paste and uncomment them above.     #
################################################################################

!------------------------------------------------------------------------------!
! PLACE THIS FILE ANYWHERE IN CLASSPATH                                        !
! Appenders are additive by default.                                           !
! Priorities are inherited until overridden in a category.                     !
! In ${property_key}, the value of the key can be defined as a system          !
! property or in this file itself.  System properties are searched first and   !
! then this file.                                                              !
!------------------------------------------------------------------------------!



!------------------------------------------------------------------------------!
! Configure log4j's operation at the meta level                                !
!------------------------------------------------------------------------------!
! Observe log4j parsing this file
#log4j.debug=true
! Set this to false for log4j to actually obey the log4j.disable property(next)
#log4j.disableOverride=false
! Disable all logging in all categories for messages with priority equal to
! or lower than the one given here
#log4j.disable=INFO



!------------------------------------------------------------------------------!
! Configure categories (loggers)                                               !
!------------------------------------------------------------------------------!

! ROOT CATEGORY (Usually sufficient to set this one only)
! Here, logs messages with priority DEBUG (default) or higher
#log4j.rootCategory=, dest1
! Or,
#log4j.rootCategory=debug, dest1, dest2
```

```
! YOUR CATEGORIES (to customize logging per class/pkg/project/etc)
! Here, overrides ancestor's priority and makes it WARN or higher for this cat.
#log4j.category.TestLogging=WARN, dest3
! Or,
#log4j.category.TestLogging=DEBUG, dest3

!--------DON'T DO THIS!!!  APPENDERS ARE ADDITIVE BY DEFAULT!!!---------------!
! It will write the same log message TWICE to dest1. Once for root, then for  !
! this category.                                                               !
!#log4j.category.TestLogging=DEBUG, dest1, dest3                              !
! If you DO NOT want additivity for this category, say so                     !
!#log4j.additivity.TestLogging=false                                          !
!------------------------------------------------------------------------------!

!------------------------------------------------------------------------------!
! Configure appenders (log destinations/targets) and their options             !
!------------------------------------------------------------------------------!

! WRITE TO CONSOLE (stdout or stderr)
#log4j.appender.dest1=org.apache.log4j.ConsoleAppender
#log4j.appender.dest1.ImmediateFlush=true

! WRITE LOG TO A FILE, ROLL THE FILE AFTER SOME SIZE
#log4j.appender.dest2=org.apache.log4j.RollingFileAppender
! This appender will only log messages with priority equal to or higher than
! the one specified here
#log4j.appender.dest2.Threshold=ERROR
! Specify the file name (${property_key} gets substituted with its value)
#log4j.appender.dest2.File=${java.home}/log4j.log
! Don't append, overwrite
#log4j.appender.dest2.Append=false
! Control the maximum log file size
#log4j.appender.dest2.MaxFileSize=100KB
! Keep backup file(s) (backups will be in filename.1, .2 etc.)
#log4j.appender.dest2.MaxBackupIndex=2

! WRITE LOG TO A FILE, ROLL THE FILE EVERY WEEK
#log4j.appender.dest3=org.apache.log4j.DailyRollingFileAppender
! Specify the file name
#log4j.appender.dest3.File=log4TestLogging2.html
! Control the maximum log file size
#log4j.appender.dest3.MaxFileSize=300KB
! Rollover log file at the start of each week
#log4j.appender.dest3.DatePattern='.'yyyy-ww



!------------------------------------------------------------------------------!
! Configure appender layouts (log formats) and their options                   !
!------------------------------------------------------------------------------!

! USE SIMPLE LOG FORMAT (e.g. INFO - your log message)
#log4j.appender.dest1.layout=org.apache.log4j.SimpleLayout

! USE A C PRINTF STYLE PATTERN TO FORMAT LOG MESSAGE
#log4j.appender.dest1.layout=org.apache.log4j.PatternLayout
! For a pattern layout, specify the pattern (Default is %m%n which is fastest)
#log4j.appender.dest1.layout.ConversionPattern=%-5p: %m%n
! Or,
#log4j.appender.dest1.layout.ConversionPattern=%-5p %6.10r[%t]%x(%F:%L) - %m%n

#log4j.appender.dest2.layout=org.apache.log4j.PatternLayout
#log4j.appender.dest2.layout.ConversionPattern=[%d{ISO8601}]%5p%6.6r[%t]%x(%F:%L) - %m%n
! Or, (the pattern below will slow down your app)
#log4j.appender.dest2.layout.ConversionPattern=[%d{yyyy-mm-dd hh:mm},%6.6r]%-5p[%t]%x(%F:%L)
- %m%n
```

```
!------------------------------------------------------------------------------!
!                          PATTERN FORMATS GLOSSARY                            !
!------------------------------------------------------------------------------!
! %n - newline                                                                 !
! %m - your log message                                                        !
! %p - message priority (FATAL, ERROR, WARN, INFO, DEBUG or custom)            !
! %r - millisecs since program started running                                 !
! %% - percent sign in output                                                  !
!                                                                              !
!----------------------SOME MORE CLUTTER IN YOUR LOG-------------------------!
! %c - name of your category (logger), %c{2} will outputs last two components !
! %t - name of current thread                                                  !
! %x - Nested Diagnostic Context (NDC) (you supply it!)                        !
!                                                                              !
!------------------------SLOW PERFORMANCE FORMATS----------------------------!
! %d - date and time, also %d{ISO8601}, %d{DATE}, %d{ABSOLUTE},                !
!        %d{HH:mm:ss,SSS}, %d{dd MMM yyyy HH:mm:ss,SSS} and so on              !
! %l - Shortcut for %F%L%C%M                                                   !
! %F - Java source file name                                                   !
! %L - Java source line number                                                 !
! %C - Java class name, %C{1} will output the last one component               !
! %M - Java method name                                                        !
!                                                                              !
!-----------------------------FORMAT MODIFIERS---------------------------------!
! %-any_letter_above - Left-justify in min. width (default is right-justify)  !
! %20any_letter_above - 20 char. min. width (pad with spaces if reqd.)        !
! %.30any_letter_above - 30 char. max. width (truncate beginning if reqd.)    !
! %-10.10r - Example.  Left-justify time elapsed within 10-wide field.        !
!              Truncate from beginning if wider than 10 characters.            !
!------------------------------------------------------------------------------!
!------------------------------------------------------------------------------!
!                             OPTIONS GLOSSARY                                 !
!------------------------------------------------------------------------------!
!-------------------------OVERALL OPTIONS FOR log4j----------------------------!
! Specify as command line option: -Dlog4j.defaultInitOverride=false
! Specify as command line option: -Dlog4j.configuration=app_config.properties
!#log4j.debug=true
!#log4j.disable=INFO
!#log4j.disableOverride=false
!#log4j.additivity.your.category.name=false
!
!----------------------------NullAppender OPTIONS-----------------------------!
!#log4j.appender.dest1.Threshold=INFO
!
!---------------------------ConsoleAppender OPTIONS---------------------------!
!#log4j.appender.dest1.Threshold=INFO
!#log4j.appender.dest1.ImmediateFlush=true
!#log4j.appender.dest1.Target=System.err
!
!----------------------------FileAppender OPTIONS-----------------------------!
!#log4j.appender.dest2.Threshold=INFO
!#log4j.appender.dest2.ImmediateFlush=true
!#log4j.appender.dest2.File=mylog.txt
!#log4j.appender.dest2.Append=false
!
!-------------------------RollingFileAppender OPTIONS-------------------------!
!#log4j.appender.dest2.Threshold=INFO
!#log4j.appender.dest2.ImmediateFlush=true
!#log4j.appender.dest2.File=mylog.txt
!#log4j.appender.dest2.Append=false
!#log4j.appender.dest2.MaxFileSize=100KB
!#log4j.appender.dest2.MaxBackupIndex=2
!
!---------------------DailyRollingFileAppender OPTIONS----------------------!
!#log4j.appender.dest2.Threshold=INFO
```

```
!
!-----------------------------SimpleLayout OPTIONS-----------------------------!
!**None**
!
!-------------TTCCLayout OPTIONS (PatternLayout is more flexible)-------------!
!#log4j.appender.dest1.layout.DateFormat=ISO8601
!#log4j.appender.dest1.layout.TimeZoneID=GMT-8:00
!#log4j.appender.dest1.layout.CategoryPrefixing=false
!#log4j.appender.dest1.layout.ThreadPrinting=false
!#log4j.appender.dest1.layout.ContextPrinting=false
!
!----------------------------PatternLayout OPTIONS---------------------------!
!#log4j.appender.dest1.layout.ConversionPattern=%m%n
!
!-------------------------------HTMLLayout OPTIONS----------------------------!
!#log4j.appender.dest3.layout.LocationInfo=true
!#log4j.appender.dest3.layout.Title=My app title
!
!--------------------------------XMLLayout OPTIONS----------------------------!
!#log4j.appender.dest3.layout.LocationInfo=true
!-------------------------------------------------------------------------------!
```