Build COMPETENCY
across your TEAM

SYNERGETICS
GET IT RIGHT

Microsoft Partner
Gold Cloud Platform
Silver Learning

Logging

Google Cloud Platform

Windows Azure

amazon web services

APACHE kafka
A distributed streaming platform

Apache Software Foundation

APACHE Spark

Java

hadoop

Logging

SYNERGETICS GET IT RIGHT | Microsoft Partner Gold Cloud Platform Silver Learning

Log4J Concepts

Installation of Log4J

Configuring Log4J

Introduction Log4j

- log4j is a reliable, fast and flexible logging framework (APIs) written in Java, which is distributed under the Apache Software License.
- log4j is a popular logging package written in Java. log4j has been ported to the C, C++, C#, Perl, Python, Ruby, and Eiffel languages.

- Log4j API follows a layered architecture where each layer provides different objects to perform different tasks.
- This layered architecture makes the design flexible and easy to extend in future.

There are two types of objects available with log4j framework.

- **Core Objects:** These are mandatory objects of the framework. They are required to use the framework.
- **Support Objects:** These are optional objects of the framework. They support core objects to perform additional but important tasks.

Log4j Components

Core Objects

- Logger Object
- Layout Object
- Appender Object

Support Objects

- Level Object
- Filter Object
- ObjectRenderer
- LogManager

Logger Object

- The top-level layer is the Logger which provides the Logger object.
- The Logger object is responsible for capturing logging information and they are stored in a namespace hierarchy.

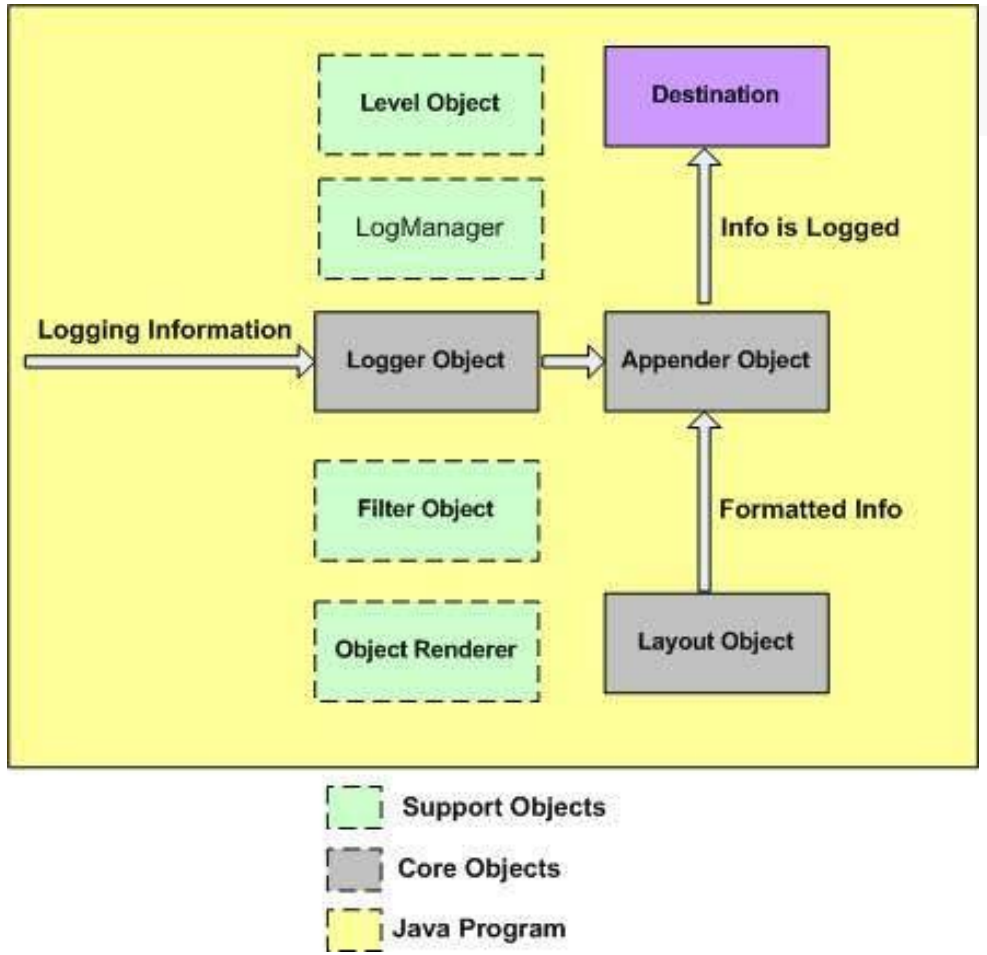
Layout Object

- The layout layer provides objects which are used to format logging information in different styles.
- It provides support to appender objects before publishing logging information.
- Layout objects play an important role in publishing logging information in a way that is human-readable and reusable.

Appender Object

- This is a lower-level layer which provides Appender objects.
- The Appender object is responsible for publishing logging information to various preferred destinations such as a database, file, console, UNIX Syslog, etc.

components of a log4J framework



Support Objects

- There are other important objects in the log4j framework that play a vital role in the logging framework:

Level Object

- The Level object defines the granularity and priority of any logging information.
- There are seven levels of logging defined within the API: OFF, DEBUG, INFO, ERROR, WARN, FATAL, and ALL.

Filter Object

- The Filter object is used to analyze logging information and make further decisions on whether that information should be logged or not.

- An Appender objects can have several Filter objects associated with them.
- If logging information is passed to a particular Appender object, all the Filter objects associated with that Appender need to approve the logging information before it can be published to the attached destination.

ObjectRenderer

- The ObjectRenderer object is specialized in providing a String representation of different objects passed to the logging framework.
- This object is used by Layout objects to prepare the final logging information.

LogManager

- The LogManager object manages the logging framework.
- It is responsible for reading the initial configuration parameters from a system-wide configuration file or a configuration class.

log4j - Configuration

- The log4j.properties file is a log4j configuration file which keeps properties in key-value pairs.
- By default, the LogManager looks for a file named log4j.properties in the CLASSPATH.

log4j - Configuration

- The level of the root logger is defined as **DEBUG**. The **DEBUG**attaches the appender named X to it.
- Set the appender named X to be a valid appender.
- Set the layout for the appender X.

Log4j.properties Syntax

- # Define the root logger with appender X
- log4j.rootLogger = DEBUG, X

- # Set the appender named X to be a File appender
- log4j.appender.X=org.apache.log4j.FileAppender

- # Define the layout for X appender
- log4j.appender.X.layout=org.apache.log4j.PatternLayout
- log4j.appender.X.layout.conversionPattern=%m%n

- Using the above syntax, we define the following in log4j.properties file:
- The level of the root logger is defined as DEBUG, The DEBUG appender named FILE to it.
- The appender FILE is defined as org.apache.log4j.FileAppender. It writes to a file named log.out located in the log directory.
- The layout pattern defined is %m%n, which means the printed logging message will be followed by a newline character.

```
# Define the root logger with appender file
log4j.rootLogger = DEBUG, FILE

# Define the file appender
log4j.appender.FILE=org.apache.log4j.FileAppender
log4j.appender.FILE.File=${log}/log.out

# Define the layout for file appender
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.layout.conversionPattern=%m%n
```

Debug Level

- We have used DEBUG with both the appenders. All the possible options are:
- TRACE
- DEBUG
- INFO
- WARN
- ERROR
- FATAL
- ALL

Appenders

- Apache log4j provides Appender objects which are primarily responsible for printing logging messages to different destinations such as consoles, files, sockets, NT event logs, etc.
- Each Appender object has different properties associated with it, and these properties indicate the behavior of that object.

Appenders

Property	Description
layout	Appender uses the Layout objects and the conversion pattern associated with them to format the logging information.
target	The target may be a console, a file, or another item depending on the appender.
level	The level is required to control the filtration of the log messages.
threshold	Appender can have a threshold level associated with it independent of the logger level. The Appender ignores any logging messages that have a level lower than the threshold level.
filter	The Filter objects can analyze logging information beyond level matching and decide whether logging requests should be handled by a particular Appender or ignored.

Appenders

- We can add an Appender object to a Logger by including the following setting in the configuration file with the following method:

log4j.logger.[logger-name]=level, appender1,appender..n

You can write same configuration in XML format as follows:

```
<logger name="com.apress.logging.log4j" additivity="false">
   <appender-ref ref="appender1"/>
   <appender-ref ref="appender2"/>
</logger>
```

- If you are willing to add Appender object inside your program then you can use following method:

- public void addAppender(Appender appender);

- The addAppender() method adds an Appender to the Logger object.
- As the example configuration demonstrates, it is possible to add many Appender objects to a logger in a comma-separated list, each printing logging information to separate destinations.

Log4j Appenders

- AppenderSkeleton
- AsyncAppender
- ConsoleAppender
- DailyRollingFileAppender
- ExternallyRolledFileAppender
- FileAppender
- JDBCAppender
- JMSAppender
- LF5Appender
- NTEventLogAppender
- NullAppender
- RollingFileAppender
- SMTPAppender
- SocketAppender
- SocketHubAppender
- SyslogAppender
- TelnetAppender
- WriterAppender

Layout

We have used PatternLayout with our appender. All the possible options are:

- DateLayout
- HTMLLayout
- PatternLayout
- SimpleLayout
- XMLLayout

Using HTMLLayout and XMLLayout, you can generate log in HTML and in XML format as well.

Example - log4j.properties

```
# Define the root logger with appender file
log = /usr/home/log4j
log4j.rootLogger = DEBUG, FILE

# Define the file appender
log4j.appender.FILE=org.apache.log4j.FileAppender
log4j.appender.FILE.File=${log}/log.out

# Define the layout for file appender
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.layout.conversionPattern=%m%n
```

Example – Java Application

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

import java.io.*;
import java.sql.SQLException;
import java.util.*;

public class log4jExample{

   /* Get actual class name to be printed on */
   static Logger log = Logger.getLogger(log4jExample.class.getName());

   public static void main(String[] args)throws IOException,SQLException{
      log.debug("Hello this is a debug message");
      log.info("Hello this is an info message");
   }
```

- Make sure you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.
- All the libraries should be available in CLASSPATH and your log4j.properties file should be available in PATH. Follow the steps give below –
- Create log4j.properties as shown above.
- Create log4jExample.java as shown above and compile it.
- Execute log4jExample binary to run the program.
- You would get the following result inside /usr/home/log4j/log.out file –

Logging Methods

- Logger class provides a variety of methods to handle logging activities. The
- Logger class does not allow us to instantiate a new Logger instance but it provides two static methods for obtaining a Logger object –

```
public static Logger getRootLogger();
public static Logger getLogger(String name);
```

- The first of the two methods returns the application instance's root logger and it does not have a name.

- Any other named Logger object instance is obtained through the second method by passing the name of the logger.
- The name of the logger can be any string you can pass, usually a class or a package name

```
static Logger log = Logger.getLogger(log4jExample.class.getName());
```

Logging Methods

- Once we obtain an instance of a named logger, we can use several methods of the logger to log messages.
- The Logger class has the following methods for printing the logging information.

#	Methods and Description
1	**public void debug(Object message)**It prints messages with the level Level.DEBUG.
2	**public void error(Object message)**It prints messages with the level Level.ERROR.
3	**public void fatal(Object message)**It prints messages with the level Level.FATAL.
4	**public void info(Object message)**It prints messages with the level Level.INFO.
5	**public void warn(Object message)**It prints messages with the level Level.WARN.
6	**public void trace(Object message)**It prints messages with the level Level.TRACE.

- All the levels are defined in the org.apache.log4j.Level class and any of the above mentioned methods can be called as follows

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

public class LogClass {
   private static org.apache.log4j.Logger log = Logger.getLogger(LogClass.class);

   public static void main(String[] args) {

      log.trace("Trace Message!");
      log.debug("Debug Message!");
      log.info("Info Message!");
      log.warn("Warn Message!");
      log.error("Error Message!");
      log.fatal("Fatal Message!");
   }
}
```

Logging Levels

- The org.apache.log4j.Level levels. You can also define your custom levels by sub-classing the Level class.

Level	Description
ALL	All levels including custom levels.
DEBUG	Designates fine-grained informational events that are most useful to debug an application.
ERROR	Designates error events that might still allow the application to continue running.
FATAL	Designates very severe error events that will presumably lead the application to abort.
INFO	Designates informational messages that highlight the progress of the application at coarse-grained level.
OFF	The highest possible rank and is intended to turn off logging.
TRACE	Designates finer-grained informational events than the DEBUG.
WARN	Designates potentially harmful situations.

How do Levels Works

- A log request of level p in a logger with level q is enabled if p >= q. This rule is at the heart of log4j. It assumes that levels are ordered. For the standard levels, we have ALL < DEBUG < INFO < WARN < ERROR < FATAL < OFF.

- The Following example shows how we can filter all our DEBUG and INFO messages. This program uses of logger method setLevel(Level.X) to set a desired logging level:

```
import org.apache.log4j.*;

public class LogClass {
   private static org.apache.log4j.Logger log = Logger.getLogger(LogClass.class);

   public static void main(String[] args) {
      log.setLevel(Level.WARN);

      log.trace("Trace Message!");
      log.debug("Debug Message!");
      log.info("Info Message!");
      log.warn("Warn Message!");
      log.error("Error Message!");
      log.fatal("Fatal Message!");
   }
}
```

Setting Levels using Configuration File

- log4j provides you configuration file based level setting which sets you free from changing the source code when you want to change the debugging level.

- Following is an example configuration file which would perform the same task as we did using the log.setLevel(Level.WARN) method in the above example.

```
# Define the root logger with appender file
log = /usr/home/log4j
log4j.rootLogger = WARN, FILE

# Define the file appender
log4j.appender.FILE=org.apache.log4j.FileAppender
log4j.appender.FILE.File=${log}/log.out

# Define the layout for file appender
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.layout.conversionPattern=%m%n
```

```
import org.apache.log4j.*;

public class LogClass {

   private static org.apache.log4j.Logger log = Logger.getLogger(LogClass.class);

   public static void main(String[] args) {

      log.trace("Trace Message!");
      log.debug("Debug Message!");
      log.info("Info Message!");
      log.warn("Warn Message!");
      log.error("Error Message!");
      log.fatal("Fatal Message!");
   }
}
```

Log Formatting

- Apache log4j provides various Layout objects, each of which can format logging data according to various layouts.
- It is also possible to create a Layout object that formats logging data in an application-specific way.

- All Layout objects receive a LoggingEvent object from the Appender objects.
- The Layout objects then retrieve the message argument from the LoggingEvent and apply the appropriate ObjectRenderer to obtain the String representation of the message.

The Layout Types

- The top-level class in the hierarchy is the abstract class org.apache.log4j.Layout.
- This is the base class for all other Layout classes in the log4j API.
- The Layout class is defined as abstract within an application, we never use this class directly; instead, we work with its subclasses which are as follows:
  - DateLayout
  - HTMLLayout
  - PatternLayout.
  - SimpleLayout
  - XMLLayout

PatternLayout

- If you want to generate your logging information in a particular format based on a pattern, then you can use org.apache.log4j.PatternLayout to format your logging information.

- The PatternLayout class extends the abstract org.apache.log4j.Layout class and overrides the format() method to structure the logging information according to a supplied pattern.

- PatternLayout is also a simple Layout object that provides the following-Bean Property which can be set using the configuration file:

Sr.No.	Property & Description
1	**conversionPattern**Sets the conversion pattern. Default is %r [%t] %p %c %x - %m%n

Pattern Conversion Characters

Conversion Character	Meaning
c	Used to output the category of the logging event. For example, for the category name "a.b.c" the pattern %c{2} will output "b.c".
C	Used to output the fully qualified class name of the caller issuing the logging request. For example, for the class name "org.apache.xyz.SomeClass", the pattern %C{1} will output "SomeClass".
d	Used to output the date of the logging event. For example, %d{HH:mm:ss,SSS} or %d{dd MMM yyyy HH:mm:ss,SSS}.
F	Used to output the file name where the logging request was issued.
l	Used to output location information of the caller which generated the logging event.
L	Used to output the line number from where the logging request was issued.
m	Used to output the application supplied message associated with the logging event.

Pattern Conversion Characters

M	Used to output the method name where the logging request was issued.
n	Outputs the platform dependent line separator character or characters.
p	Used to output the priority of the logging event.
r	Used to output the number of milliseconds elapsed from the construction of the layout until the creation of the logging event.
t	Used to output the name of the thread that generated the logging event.
x	Used to output the NDC (nested diagnostic context) associated with the thread that generated the logging event.
X	The X conversion character is followed by the key for the MDC. For example, X{clientIP} will print the information stored in the MDC against the key clientIP.
%	The literal percent sign. %% will print a % sign.

Format Modifiers

- By default, the relevant information is displayed as output as is.
- However, with the aid of format modifiers, it is possible to change the minimum field width, the maximum field width, and justification.

Format Modifiers

SYNERGETICS GET IT RIGHT | Microsoft Partner Gold Cloud Platform Silver Learning

Format modifier	left justify	minimum width	maximum width	comment
%20c	false	20	none	Left pad with spaces if the category name is less than 20 characters long.
%-20c	true	20	none	Right pad with spaces if the category name is less than 20 characters long.
%.30c	NA	none	30	Truncate from the beginning if the category name is longer than 30 characters.
%20.30c	false	20	30	Left pad with spaces if the category name is shorter than 20 characters. However, if the category name is longer than 30 characters, then truncate from the beginning.
%-20.30c	true	20	30	Right pad with spaces if the category name is shorter than 20 characters. However, if category name is longer than 30 characters, then truncate from the beginning.

PatternLayout Example

- # Define the root logger with appender file
- log = /usr/home/log4j
- log4j.rootLogger = DEBUG, FILE

- # Define the file appender
- log4j.appender.FILE=org.apache.log4j.FileAppender
- log4j.appender.FILE.File=${log}/log.out

- # Define the layout for file appender
- log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
- log4j.appender.FILE.layout.ConversionPattern=%d{yyyy-MM-dd}-%t-%x-%-5p-%-10c:%m%n

PatternLayout Example

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

import java.io.*;
import java.sql.SQLException;
import java.util.*;

public class log4jExample{
   /* Get actual class name to be printed on */
   static Logger log = Logger.getLogger(log4jExample.class.getName());

   public static void main(String[] args)throws IOException,SQLException{
      log.debug("Hello this is an debug message");
      log.info("Hello this is an info message");
   }
}
```

Logging in Files

- To write your logging information into a file, you would have to use org.apache.log4j.FileAppender.

- FileAppender Configuration

Property	Description
immediateFlush	This flag is by default set to true, which means the output stream to the file being flushed with each append operation.
encoding	It is possible to use any character-encoding. By default, it is the platform-specific encoding scheme.
threshold	The threshold level for this appender.
Filename	The name of the log file.
fileAppend	This is by default set to true, which means the logging information being appended to the end of the same file.
bufferedIO	This flag indicates whether we need buffered writing enabled. By default, it is set to false.
bufferSize	If buffered I/O is enabled, it indicates the buffer size. By default, it is set to 8kb.

log4j.properties for FileAppender

```
# Define the root logger with appender file
log4j.rootLogger = DEBUG, FILE

# Define the file appender
log4j.appender.FILE=org.apache.log4j.FileAppender

# Set the name of the file
log4j.appender.FILE.File=${log}/log.out

# Set the immediate flush to true (default)
log4j.appender.FILE.ImmediateFlush=true
```

```
# Set the threshold to debug mode
log4j.appender.FILE.Threshold=debug

# Set the append to false, overwrite
log4j.appender.FILE.Append=false

# Define the layout for file appender
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.layout.conversionPattern=%m%n
```

Logging in Multiple Files

- You may want to write your log messages into multiple files for certain reasons, for example, if the file size reached to a certain threshold.

- To write your logging information into multiple files, you would have to use org.apache.log4j.RollingFileAppender class which extends the FileAppender class and inherits all its properties.

Logging in Multiple Files

- We have the following configurable parameters in addition to the ones mentioned above for FileAppender –

Property	Description
maxFileSize	This is the critical size of the file above which the file will be rolled. Default value is 10 MB.
maxBackupIndex	This property denotes the number of backup files to be created. Default value is 1.

log4j.properties for RollingFileAppender.

- Following is a sample configuration file log4j.properties for RollingFileAppender.

```
# Define the root logger with appender file
log4j.rootLogger = DEBUG, FILE

# Define the file appender
log4j.appender.FILE=org.apache.log4j.RollingFileAppender

# Set the name of the file
log4j.appender.FILE.File=${log}/log.out

# Set the immediate flush to true (default)
log4j.appender.FILE.ImmediateFlush=true

# Set the threshold to debug mode
log4j.appender.FILE.Threshold=debug
```

log4j.properties for RollingFileAppender.

```
# Set the append to false, should not overwrite
log4j.appender.FILE.Append=true

# Set the maximum file size before rollover
log4j.appender.FILE.MaxFileSize=5KB

# Set the the backup index
log4j.appender.FILE.MaxBackupIndex=2

# Define the layout for file appender
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.layout.conversionPattern=%m%n
```

SYNERGETICS GET IT RIGHT | Microsoft Partner
Gold Cloud Platform
Silver Learning

Q & A

Contact: amitmahadik@synergetics-india.com

SYNERGETICS GET IT RIGHT | Microsoft Partner
Gold Cloud Platform
Silver Learning

Thank You