Log4Delphi User Guide

Table of contents

1. About This Guide

The Log4Delphi User Guide is a document with the goal of assisting users in getting acquianted with the Log4Delphi Logging Suite, its components and how it can be employed in applications. This guide will attempt to provide a more rounded and in depth discussion on the various components of Log4Delphi as well as how to use it.

Warning:

This guide is still a work in progress, and far from complete!

2. Logging With Log4Delphi

2.1. What is Log4Delphi?

Log4Delphi is a logging suite for Borland's Delphi based on the Log4J package from the Apache Software foundation. It is not an exact port of Log4J, but rather a similar package based on the principles of simplicity and focus.

Logging is often used in conjunction with testing in order to debug applications, especially when debugging tools are not available, as is the case with distributed applications. Log4Delphi is a suite that can perform the logging function in your applications.

Log4Delphi is an open source project that aims to produce a high quality and usable logging suite. It is a native 32bit Borland Delphi Package developed with Borland Delphi version 6 on Windows XP.

2.2. What Can Log4Delphi Do For Me?

Logging is considered a low level debugging technique but is not meant to replace a traditional debugger, in fact the two can go hand in hand. When debugging an application it is good to have a clear picture of what the application does at every step of the way, especially large applications that have many things going on in the background. Logging is a way to find out what the application is doing, how and when.

Log4Delphi is a suite that provides logging capabilities for Borland's Delphi. It can be used to perform logging of almost any kind of application, especially those that are often difficult to debug like distributed systems.

2.3. How Log4Delphi Works

The application developer decides to generate a message that needs to be logged whenever something important in the application occurs, for example: the click of a button. The developer then decides where this message should go, its *target*, typically a file, and the developer may control the message's *format*.

An important aspect is the ability to assign different priority levels to the message and further to only log messages with a certain priority level. This affords the developer a large amount of control over how logging gets performed, for example, messages of a certain priority can be logged to a file and other messages with a different priority to another target.

3. Log4Delphi Basic Concepts

3.1. Installing Log4Delphi

There are two ways in which you can install and use Log4Delphi. The first is to install the package and the second is to simply include the source code in your application's code base. Both have pros and cons and should be carefully considered before making the decision.

3.1.1. Installing The Package

Installing the Log4Delphi package will make it available to be used in all your projects.

Using a binary distribution (`log4delphi-`*version* `-bin.`*archive*), the steps are:

1. Launch Delphi.
2. Select `Component > Install Packages` from the main menu.
3. Click the `Add` button.
4. Browse to where you extracted the archive and select the file named `log4delphi.bpl`, should be in the `bin/` directory.
5. You may need to add the compilation units to your library path. To do so, select `Tools > Environment Options` from the main menu.
6. Select the Library Tab and add the folder containing `log4delphi.bpl` to the library path.
7. You are ready to go!

Note:

You may have to create a package for your version of Delphi. To find out how to do this, see here.

3.1.2. Using The Source Code

This option is best for people who wish to add to or extend Log4Delphi or people who are

busy developing Log4Delphi or people who wish to include Log4Delphi into their existing project's code base.

Using a source distribution (`log4delphi-`*version*`-src.`*archive*), the steps are:

1. Copy all the *.pas files from the `log4delphi\src\delphi` directory to your project's source directory (include the `util`) directory.
2. Launch Delphi and Open your project.
3. Select `Project > Add To Project` in the main menu.
4. In the dialog that opens, select all the Log4Delphi source files `*.pas` that you added earlier.
5. Click the `Open` button.
6. Now build your project.
7. You are ready to go!

3.2. Upgrading Log4Delphi

You can upgrade from a previous version in one of two ways depending on how you installed Log4Delphi.

3.2.1. Binary Package

If you installed the Log4Delphi binary package, you will need to first uninstall this package before installing the newer version package. This is done in Delphi.

1. First launch Delphi.
2. Select *Install Packages* from the *Component* menu.
3. Select the package you wish to uninstall, in this case Log4Delphi and click the *Remove* button. This will remove the package.
4. Now install the newer Log4Delphi package.

3.2.2. Source Code

If you copied the source code into your project's code base, you will need to copy the newer source code into your code base.

1. Copy the new Log4Delphi source files into your code base, overwriting the old source files.
2. You may need to add some of the new files to your project.
3. Select *Add To Project* from the *Project* menu.
4. Select all the source code (`*.pas`) files and click the *Open* button.
5. Click the *OK* button.

3.3. Loggers, Appenders and Layouts

The basic idea is that a `Logger` is responsible for actually performing the logging in an application by handling the log operations, the `Appender` is responsible for appending the message to its *target* and controlling that output, and the `Layout` is responsible for formatting the logging message.

These and other core components are discussed in depth in the next seciton.

4. Five Core Components

The Log4Delphi suite consists of five core components, namely: `Level`s, `LoggingEvent`s, `Layout`s, `Appender`s and `Logger`s that all work together to provide the functinality of the package. Understanding these components, how they are related and how they work is essential to using Log4Delphi effectively.

4.1. Levels

Every event that is logged has an assigned priority level. Fatal errors would naturally be more severe than a simple warning and thus would have a higher priority level. In this regard, each level has an integer value assigned to it so that it can be compared to other levels.

There are five levels by default, namely (lowest prioirty to highest): `DEBUG`, `INFO`, `WARN`, `ERROR` and `FATAL`. Users can quite easily add another level as they see fit. Using these levels it is possible to only log events of a certain priority level, say for example: error, which results in only that level and any level with a higher priority being logged. This means that all error and fatal messages will be logged while all debug, info and warn messages will be ignored.

Two special priority levels exist, these are `ALL` and `OFF`, all is used to log all messages of any priority level, even user defined ones while off is used to ignore all messages regardless of their priority level. In essence, during debugging, all should be used while production code should use off.

Typical usage is by simply using the level constants defined in the `TLevelUnit` :

```
TLevelUnit.OFF
TLevelUnit.FATAL
TLevelUnit.ERROR
TLevelUnit.WARN
TLevelUnit.INFO
TLevelUnit.DEBUG
TLevelUnit.ALL
```

4.2. LoggingEvents

Logging events represent the actual event that gets logged and an instance is created whenever the Log4Delphi suite makes an affirmitive decision to log something. Logging events contain vital information such as the time it was created, the message itself, any exception if an exception is raised and an associated priority level.

The LoggingEvent is passed around to the various Log4Delphi components that use it and the information contained in the logging event is used to format and print the information to the desired target so that it can be analyzed at a later stage.

LoggingEvents are not usually used by application developers but rather only internally by Log4Delphi.

4.3. Layouts

Layouts provide the functionality of formatting logging events into any number of appropriate formats. The layout takes the Logging Event and formats it appropriately into a string so that it can be written to the `target` by the `Appender`. `TLayout` is abstract and subclassed by a number of concrete layout classes that provide an implementation of the `format(event : TLoggingEvent)` method.

A layout may also provide a header and footer as is the case in an HTML layout where an HTML document usually contains a header and a footer. These are returned by the `getHeader` and `getFooter` methods respectively. The default implementation returns empty strings for these. A layout can also choose to ignore any exceptions that are contained in the Logging Event, this can be determined from the `ignoresException` method which returns `false` by default. Naturally any subclasses that handle exceptions will override this method to return true such as in the XML layout.

The `TLayout` class exposes the following:

```
function format(event : TLoggingEvent) : String; Virtual; Abstract;
function getContentType() : String; Virtual;
function getHeader() : String; Virtual;
function getFooter() : String; Virtual;
function ignoresException() : Boolean; Virtual;
```

4.3.1. TSimpleLayout

Simple layout provides basic formatting, this output from this layout is the level of the log message followed by a dash (`-`) and the log message itself. It returns the default implementation for all other methods defined in the `TLayout` class, thus empty header and footer strings and it ignores exceptions.

Example output:

```
DEBUG - Button Clicked
```

4.3.2. THTMLLayout

The HTML Layout is used to format messages into an HTML format putting them into an HTML document structuring the log messages into a table. The resulting HTML is 4.01 compliant and should have no trouble rendering in any browser. The layout overrides all methods defined in `TLayout` thus it provides a header and footer and handles exceptions.

The `getContentType` method returns `text/html` and the layout provides a `setTite(title : String)` method in which it is possible to set the title of the resulting HTML document. It also provides a `getTitle` method to determine what the title is currently set to.

Example output:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<html>
    <head>
        <title>Log4Delphi Log Messages</title>
        <style type="text/css">
        <!--
            body {background: XFFFFFF; margin: 6px; font-family:
arial,sans-serif; font-size: small;}
            table {font-family: arial,sans-serif; font-size: 9pt;}
            th {background: #336699; color: #FFFFFF; text-align: left;}
            td.debug {color: #339933}
            td.warn {color: #FF9229}
            td.error {color: #CC0000}
            td.fatal {color: #FF0000}
        -->
        </style>
    </head>
    <body>
        <p>Log session start time Tue Sep 13 16:19:28 SAST 2005</p>
        <table cellspacing="0" cellpadding="4" width="100%">
            <thead>
                <tr>
                    <th>Time</th>
                    <th>Level</th>
                    <th>Message</th>
                </tr>
            </thead>
            <tbody>
```

```
                <tr>
                    <td title="Timestamp">863649565</td>
                    <td title="Level" class="DEBUG">DEBUG</td>
                    <td title="Message">Button Clicked!</td>
                </tr>

            </tbody>
        </table>
    </body>
</html>
```

4.3.3. TXMLLayout

XML layout is used to format log messages as XML. It does not output a complete well-formed XML file, rather the output is designed to be included as an external entity in a separate file to form a correct XML file. For example, if app.log is the name of the file where the XMLLayout ouput goes, then a well-formed XML file would be:

```
<?xml version="1.0" ?>
<!DOCTYPE log4delphi:eventSet SYSTEM "log4delphi.dtd" [<!ENTITY data SYSTEM
"app.log">]>
<log4delphi:eventSet xmlns:log4delphi="http://log4delphi.sourceforge.net/">
    &data;
</log4delphi:eventSet>
```

The `log4delphi.dtd` can be found in the `example/` directory in any distribution archive.

The layout handles exceptions and the `getContentType` returns `text/xml`. Example output is:

```
<log4delphi:event timestamp="863649814" level="DEBUG">
    <log4delphi:message><![CDATA[Button Clicked!]]></log4delphi:message>
</log4delphi:event>
```

4.3.4. TPatternLayout

The pattern layout provides the most control over formatting of log messages. The goal of this class is to format a LoggingEvent and return the results as a String, dependant on the conversion pattern supplied.

The conversion pattern is closely related to the format strings used in Delphi. A conversion pattern is composed of literal text and format control expressions called conversion specifiers. Literal text is copied verbatim to the resulting string and you are free to insert any literal text within the conversion pattern.

Each conversion specifier starts with a percent sign (%) and is followed by **optional** *format modifiers* and a *conversion character*. The conversion character specifies the type of data, e.g. logger, level priority, date and log message. The format modifiers control such things as field width, padding and left and right justification. The following is a simple example.

Let the conversion pattern be **"%d [%5p] %m%n"** and assume that Log4Delphi has been configured to use a TPatternLayout. Then the statements

```
logger.debug('Debug message');
logger.info('Info message');
logger.warn('Warn message');
logger.error('Error message');
logger.fatal('Fatal message');
```

would yield the output

```
12/20/2005 4:53:33 PM [DEBUG] Debug message
12/20/2005 4:53:33 PM [ INFO] Info message
12/20/2005 4:53:33 PM [ WARN] Warn message
12/20/2005 4:53:33 PM [ERROR] Error message
12/20/2005 4:53:33 PM [FATAL] Fatal message
```

Note that there is no explicit separator between literal text and conversion specifiers. The pattern parser knows when it has reached the end of a conversion specifier when it reads a conversion character. The recognized conversion characters are

Character	Effect
d	Outputs the date of the logging event. The date conversion specifier may be followed by an optional *date format specifier* enclosed in braces. This format specifier is the same as the Delphi Date-Time Format Strings, for example: **%d{dd mmm yyyy hh:nn:ss:zzz}**. For more information see the Delphi help files.
m	Used to output the supplied application log message. Standard format specifiers apply, for example: **%20.30m**.
n	Outputs a line separator character, typicaly mapping to ascii character 13.
p	Outputs the priority level of the log event. Stanard format specifiers apply, for example: **%-5p**.
e	Used to output an the associated exception's

	classname and message if there is an exception associated with the log event. If no exception is associated this simply outputs nothing, which means it is safe to use.
L	Output the logger's name. Note that this must be a captial L, not lowercase. Standard format specifiers apply.

Table 1: Conversion Characters

Note that the conversion string is case sensitive, such that "L" does not equal "l"!

By default the relevant information is 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. The optional format modifier is placed between the percent sign and the conversion character. For more information see the Delphi help files on "format strings".

4.4. Appenders

The ability to selectively enable or disable logging requests based on their logger, and to format those messages is only part of the picture. Log4Delphi allows logging requests to output to multiple destinations, a destination being a specific *target*.

An Appender is responsible for "appending" a log message to a specific target. Each appender specifies its own layout (the formatting it will use) as well as a threshold which determines the level priority that it will log. Some appenders inherently require a layout but others might not.

the `TAppender` class exposes:

```
    procedure SetLayout(ALayout : TLayout); Virtual;
    procedure SetName(AName : String);
    procedure SetThreshold(AThreshold : TLevel);
    procedure SetErrorHandler(AHandler : TErrorHandler);
    function GetLayout() : TLayout;
    function GetName() : String;
    function GetThreshold() : TLevel;
    function GetErrorHandler() : TErrorHandler;
    function IsAsSevereAsThreshold(ALevel : Tlevel) : Boolean;
    function RequiresLayout() : Boolean; Virtual;
```

4.4.1. TFileAppender

You can use the `TFileAppender` class to append log messages to a file. All that is required is to specify the filename of the file to log message to. Although the file appender

requires a layout, if one is not specified, `TSimpleLayout` is used by default.

4.4.2. TRollingFileAppender

The `TRollingFileAppender` allows you to specify a maximum log file size and allow you to perform log rotation of the log files. The rationale behind this is simple: it is easy to create a log file, managing a log file on the other hand is a difficult task. A log file will grow without bound unless some form of action is taken; large log files are difficult to manipulate and may lead to filesystems running out of space. Log rotation is the process where by a log file is periodically renamed to another file and a new empty file is used as the log. After a set number of file rotations, the oldest file will be removed.

As a practical example, consider a log file named `app.log`. Using a rolling file appender we decide to make the maximum size of this file 100Kb and that we rotate between two files. Once this file reaches 100Kb the file is renamed to `app.log.1` and a new log file is created named `app.log` which is empty. Once this new file reaches 100Kb in size the old file `app.log.1` is deleted, the file named `app.log` is renamed to `app.log.1` and a new file named `app.log` is created and used. This process repeats indefinitely.

`TRollingFileAppender` inherits from `TFileAppender` and exhibits its default behaviour unless modified by setting the maximum log file size and the backup index. The max log file size set how large the files should be before log rotation and the max backup index is the number of log files to rotate between. Setting the backup index to 0 has the effect of rotating in one file only, which means that upon log rotation, the log is destroyed and a new file is used (This should be used with caution since important log information may be lost).

4.4.3. TDBAppender

The Database Appender is used to send messages to a database. It is flexible enough to support Borland Database Engine (BDE), Interbase Express (IBX) and Database Express (DBX) through the use of `LogInserter`s.

The log message can be sent to any table or column in the relational database, using custom parameterized SQL which is given to the inserter to use. As an example, consider the IBXLogInserter (this code is similar if you use other database components / engines):

```
inserter := TIBXLogInserter.Create(IBDatabase1,
   'INSERT INTO Log (log_level, log_message, log_starttime, log_exception)'
   + 'VALUES (:_level,:_msg,:_startTime,:_exception)');
logger.addAppender(TDBAppender.Create(inserter));
```

All you need to do is to drop the appropriate component onto your form, or create it

dynamically and pass it to the `LogInserter`. Then use the inserter to create the database appender.

The various inserters are:

4.4.3.1. TBDELogInserter

For those using Borland Database Engine, simply use this inserter and pass it a pre-configured `TDatabase` component.

```
constructor Create(ADatabase: TDatabase; ASQL : String);
```

4.4.3.2. TDBXLogInserter

Used for Database Express Applications. Give this one a `TSQLConnection` component.

```
constructor Create(ASqlConnection: TSQLConnection; ASQL : String);
```

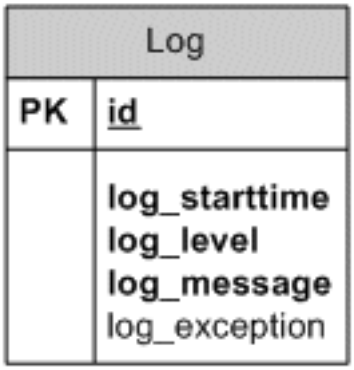
4.4.3.3. TIBXLogInserter

This is for Interbase Express applications. Hand this inserter a `TIBDatabase` component.

```
constructor Create(AIBDataBase : TIBDataBase; ASQL : String);
```

4.4.3.4. The SQL

The SQL you give to the LogInserter needs to be parameterized. Consider the following entitiy in a relational database:



Log ERD

An example of the SQL that can be used for this table is as follows:

```
INSERT INTO Log (log_level, log_message, log_starttime,
log_exception) VALUES(:_level, :_msg, :_startTime, :_exception)
```

As you can see, parameters are designated with a semi-colon (`:`) and in Log4Delphi, all parameters begin with an unserscore (_). A list of the various parameters that you can use are:

Parameter	Description
`_msg`	This will resolve to the actual log message contained in the LoggingEvent.
`_exception`	This resolves to the Exception's message property.
`_exceptionclass`	Resolved to the Class Name of the Exception.
`_startTime`	The time the log message is created, as a `TDateTime` value.
`_level`	The priority level of the log message.
`_levelCode`	The priority level's actual integer value.

Table 1: SQL Parameters

Warning:

The database appender has not been tested with all databases. As of this writing, only Firbird / Interbase has been tested with IBX and DBX components.

4.5. Loggers

Loggers are named entities with their names being case sensistive and are responsible for performing the logging operations in an application. A root logger always exists and obeys two fundamental rules: 1) it always exists and 2) it cannot be retrieved by name. In order to access the root logger you use the static method `getLogger`. All other loggers are retrieved by name using `getLogger(name)`.

Loggers may be assigned a priority level, if a given logger is not assigned a priority level then it uses the default `ALL` priority level. A log request with a given assigned level say ***S*** in a logger with level ***T*** is enabled if and only if S is greater or equal to T: (**S>=T**).

5. Configuration

During configuration the following occurs:

1. Log4Delphi's internal logging is activated and sends output to a file named 'log4delphi.log'.
2. Each of the respective Levels is initialized using the Level unit's `initialize` method.
3. Finally the Logger interface is initialized by calling the `initialize` method of the Logger unit.

Configuration is performed using one of the methods in the Configurator Unit. Currently two methods exist, `doBasicConfiguration` and the `doPropertiesConfiguration` method. Both of these methods can be used. The first provides a minimalist configuration which is fine for basic logging, the second method allows you to control the logging using a properties file.

Configuration should be performed in the Application's code:

```
  Application.Initialize;
  TConfiguratorUnit.doBasicConfiguration; // ADD THIS!
  Application.CreateForm(TForm1, Form1);
  Application.Run;
```

alternately, specifiying the properties file to read configuration from:

```
  Application.Initialize;
  TConfiguratorUnit.doPropertiesConfiguration('log4delphi.properties'); //
ADD THIS!
  Application.CreateForm(TForm1, Form1);
  Application.Run;
```

5.1. The Configuration File

An example configuration file named `log4delphi.properties` is provided in both binary and source distributions. We will now discuss a few of the configuration details of this file.

First, it is important to note that this file may be named anything, but `log4delphi.properties` is a good convention to use.

Next, this file may be placed anywhere on the file system but it is a good idea to place the file in the same folder as your application's executable (`.exe`) and then initializing Log4Delphi appropriately:

```
TConfiguratorUnit.doPropertiesConfiguration(ExtractFileDir(Application.ExeName)
   + '\log4delphi.properties');
```

5.1.1. Controlling Log4Delphi's Internal Logging

Log4Delphi's internal logging is controlled through the `log4delphi.debug` directive. Those who are interested to see what Log4Delphi is doing can set this value to `true`. This results in a file named `log4delphi.log` in the same directory as your application executable. If you do not wish to have internal logging then set this value to `false`.

```
# Set this to true to turn on Log4Delphi's internal
# logging
log4delphi.debug=false
```

5.1.2. Configuring the Root Logger

The next thing to do is to set the root logger by specifying the priority level and an appender's name. The format is `log4delphi.rootLogger=LEVEL, AppenderName`. The *LEVEL* is any one of DEBUG, INFO, WARN, ERROR or FATAL and the *AppenderName* is any alpha-numeric string of no more than 255 characters. As an example, consider setting the root logger to debug and using a file appender:

```
# Set the root logger's priority threshold to DEBUG and assign an
# appender named "fileAppender" to it.
log4delphi.rootLogger=DEBUG, fileAppender
```

5.1.3. Configuring the Appender

The first step is to tell Log4Delphi which appender you wish to use by giving the appender class's name.

```
# Specify the appender class for fileAppender.
log4delphi.appender.fileAppender=TFileAppender
```

Some appender's require additional information, these are set using directives that match the appender's setter methods. The TFileAppender class has a `setFile` method that can be used as follows:

```
# Specify which file fileAppender should use.
log4delphi.appender.fileAppender.File=app.log
```

5.1.4. Setting the Layout

Some appenders require a layout. This is easily set using the `layout` directive and

specifying the layout class to use.

```
# Specify the layout class for fileAppender.
log4delphi.appender.fileAppender.layout=TSimpleLayout
```

You should look at the `log4delphi.properties` file for a complete example.

6. Summary

Log4Delphi is a easy to use but powerful and flexible logging suite that is open source and under constant development.