Lire Developer’s Manual

Joost van Baal

Egon L. Willighagen

Francis J. Lacoste

Lire Developer's Manual

by Joost van Baal, Egon L. Willighagen, and Francis J. Lacoste

Revision History

Revision 2.0 $Date: 2004/09/04 11:38:27 $
$Id: dev-manual.dbx,v 1.85 2004/09/04 11:38:27 vanbaal Exp $

Table of Contents

List of Tables

List of Figures

List of Examples

Preface

Log file analysis is both an essential and tedious part of system administration. It is essential because it's the best way of profiling the usage of the service installed on the network. It's tedious because programs generate a lot of data and tools to report on this data are often unavailable or incomplete. When such tools exist, they are generally specific to one product, which means that you can't compare e.g. your Qmail and Exim mail servers.

Lire is a software package developed by the Stichting LogReport Foundation to generate useful reports from raw log files of various network programs. Multiple programs are supported for various types of network services. Lire also supports various output formats for the generated reports.

What This Book Contains

This book is the *Lire Developer's Manual*. Its purpose is to present Lire as a log analysis framework. To this ends, it describes the architecture and design of Lire and contains comprehensive instructions on how to use it. Its intended audience is system administrators or programmers who want to extend Lire or want to understand its internals.

There is another book, the *Lire User's Manual* which describes how to install, configure and use Lire, as a "off-the-shelf" log analyzer. Its intended audience is system administrators who want to install and use Lire to gather information about the services operating on their network.

How Is This Book Organized?

This book is divided in five parts. Part I gives an overview of the architecture and design of Lire.

You will find in Part II information on extending Lire. In this part, you will learn how to add a new DLF format to Lire, write log file converters and add reports for a superservice.

Part III is a reference section which gives comprehensive details about the various XML formats used by Lire and gives in-depth descriptions of its various APIs.

Part IV is targeted at developers who want to participate in Lire's development. It contains information about CVS access, coding conventions, tools needed to build from CVS, release management and other aspects important to those part of the Lire development team. Furthermore, it gives some information on how to contribute code to Lire, as an external party.

Finally, Part V contains various implementation details that may be interesting to people wanting to learn more about Lire internals.

Conventions Used

If You Don't Find Something In This Manual

You can report typos, incorrect grammar or any other editorial problems to <bugs@logreport.org>. We welcome reader's feedback. If you feel that certain parts of this manual aren't clear, are missing information or lacking in any other aspect, please tell us. Of course, if you feel like writing the missing information yourself, we'll very happily accept your patch. We will make our best effort to improve this manual.

Remember, that there is another manual, the *Lire User's Manual* which contains comprehensive information on how to install, use and configure Lire. It also contains reference information about all of Lire's standard reports and supported services.

There are various public mailing lists for Lire's users. There is a general users' discussion list where you can find help on how to install and use Lire. You can subscribe to this list by sending an empty email with a subject of *subscribe* to <`questions-request@logreport.org`>. Email for the list should be sent to <`questions@logreport.org`>.

You can keep track of Lire's new release by subscribing to the announcement mailing list. You can subscribe yourself by sending an empty email with a subject of *subscribe* to <`announcement-request@logreport.org`>.

Finally, if you're interested in Lire's development, there is a development mailing list to which you can subscribe by sending an empty email with a subject of *subscribe* to <`development-request@logreport.org`>. Email to the list should be sent to <`development@logreport.org`>.

All posts on these lists are archived on a public website.

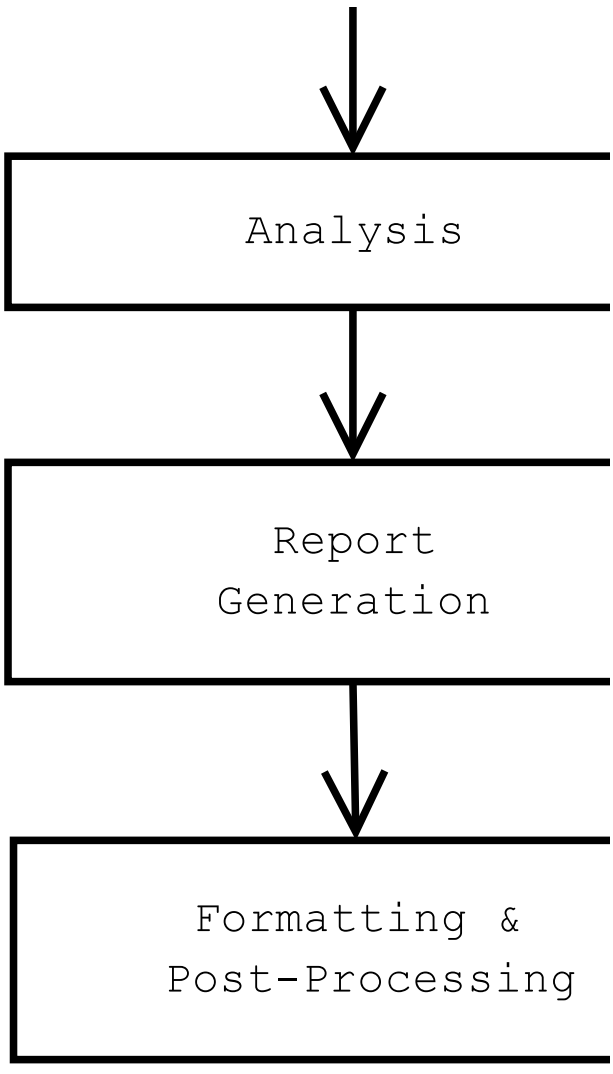
I. Lire Architecture

Chapter 1. Architecture Overview

From a developer's point of view, Lire intends to be the universal log analysis framework. To this end, it provides a reliable, complete, framework upon which to build log analysis and reporting solution. Lire, the tool, is a proof of the versality and extendability of the framework as it is able to produce reports for many of the services that run in today's heterogeneous networks in a variety of output formats.

As a framework, Lire is the best choice to replace all those home-grown scripts developed to produce reports from all the log files from the little-known products or custom-developed programs that run on your system. Leveraging Lire framework will make those scripts a lot more versatile while not being really more complicated to develop. It will be easier to add new reports or to support multiple report formats.

Figure 1-1. Log Processing in the Lire's Framework



The Lire's framework divides log analysis in four different processes. The figure Figure 1-1 shows those four processes:

1. **Log Normalisation.** The first process normalise logs from different products into a generic format that can be shared by all products that have similar functionality. For example, log files from products as different as Apache and Microsoft Internet Information Server will be transformed into an identical format.
2. **Log Analysis.** In the analysis process, other information is created, inferred or extracted from the normalised data. For example, an anlyser in the www superservice infers the browser used by the client from the referrer information.
3. **Report Generation.** The third process generates a report from the normalised and analysed data. This process is done by a generic report engine that computes the report based on specifications describing what and how the information should appear in the report. The report is generated in a generic XML format.
4. **Report Post-processing and Formatting.** The last process converts the generic report into a specific format like ASCII, PDF, HTML but other kind of post-processing (like charts generation) can also be accomplished in this stage.

Before going into a more detailed description of each of these procesesses, we'll introduce some of the common design's patterns that you'll find throughout the Lire's framework.

Lire's Design Patterns

At the center of each of these processes is an XML based file format. Having things specified in data files makes it easier to extend. For example, the reports are built using a generic report builder which finds the instructions on how to build the reports in XML files. So this makes it easy to add new information to a report: you just have to write an XML file. The fact that there are a lot of tools to process XML files is also an interesting aspect. For example, emacs lovers will appreciate the help that its psgml module gives them in writing report specifications.

Another important aspects is that we tried to interoperate and to build upon other standards while defining our XML formats . The best illustration of this is that in all the XML file formats that Lire use, a DocBook subset is used for all elements related to narrative descriptions.

Another common aspect you'll encounter is that each of these processes and XML file formats come with an API to manipulate them, making it easy to add functionalities at each processing stage. APIs are also a good thing because, even if in theory an open file format somewhat constitutes an API, having libraries that provide convenient access to the file formats makes it a lot easier to write new components providing new functionalities.

Log File Normalisation

Figure 1-2. The Log Normalisation Process

The first process of the Lire log analysis framework is the log file normalisation process. That process is summarized in the Figure 1-2 figure. This process is centered around the *DLF* concept which is kind of a universal log format. DLF stands for Distilled Log Format. The concept is that each product specific log file is transformed into a log format that can be common to all the products providing similar functionalities. In Lire's terminology, a class of applications providing similar functionality (e.g. MTA's supplying email) is called a *superservice*. Still in Lire's terminology, the *service* from which the super is derived (e.g. postfix or sendmail) refers to the native log format that is converted in the superservice's DLF. One can view the DLF as a table where the rows are the logged events and the fields are logged information related to each event.

Since the information logged by an email server is totally different from a web server, each superservice should have its own data models. In Lire, the data model is called a DLF *schema*. The DLF schemas are defined in XML files using the DLF Schema Markup Language. The schema describes what fields are available for each logged events.

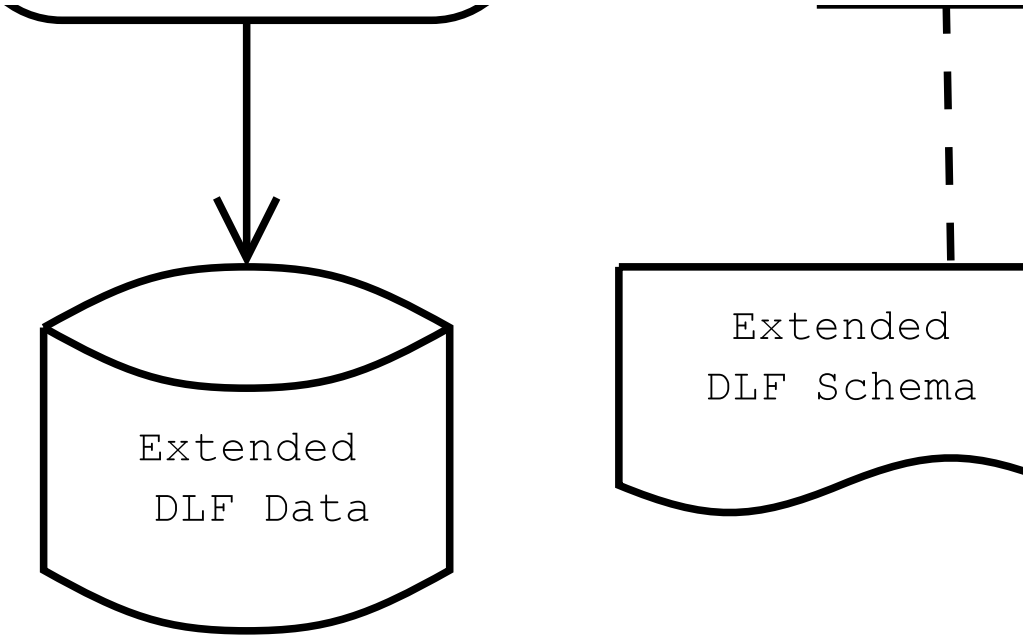
One interesting aspect of Lire, is that altough the email DLF is used by all email servers, the email DLF data model isn't restricted to the lowest common denominator across the log formats supported by each email servers. In the Lire's architecture, the superservice's schema can represent the information logged by the most sophisticated product. When some part of the information isn't available in one log format, the DLF log file will contain this information and the reports that needs this information won't be included.

This architecture means that to support a new service, i.e. a new log format, in Lire you just need to write a plugin, called a DLF converter. This is just a simple perl module that parses the native log format and maps the information according to the schema.

Log Analysis

After normalisation, comes the analysis process. The analysis process responsability is to extracts, infers or derives other information from the logged data. Since the superservice's logged data is in a standard format, the analysers are generic in the sense that they can operate for all the superservice's supported log formats, if the product's was clever enough to log the information required by the analyser. The analysis process is shown in the Figure 1-3 figure.

Figure 1-3. The Log Analysis Process



Since each analyser can add information to or create a new DLF, each analyser will generate data according to special kind of schemas.

Lire's framework include two kind of analysers. The difference between the two resides in the mapping between the source data and the new data they generate. Extended analysers generate new data for each DLF record whereas derived analysers are used when the new data doesn't have a one-to-one mapping with the source data.

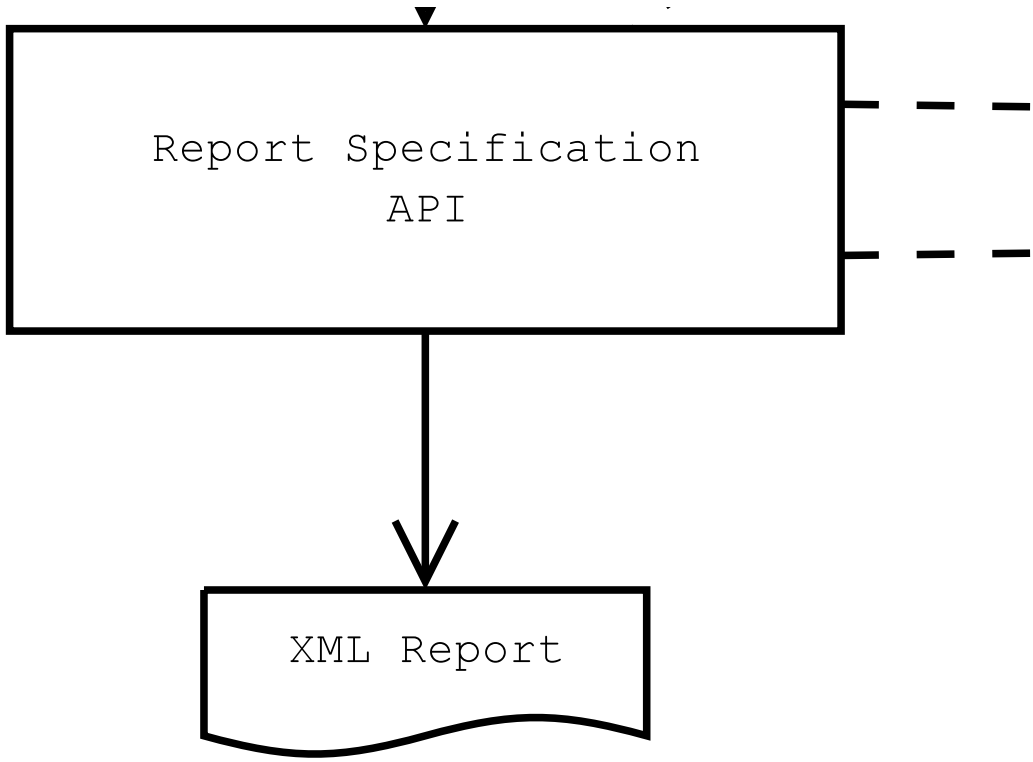
The analysers produce data according to a data model which is specified in other DLF schemas. There are *extended* schemas and *derived* schemas. An extended schema simply adds new fields to the base superservice's schema. For example, in the web superservice's schema, a lot of information can be obtained from the referer field. From this information, it is possible to guess the user's browser, language or operating system. Those fields are specified in the www-referer extended schema; one analyser is responsible for extracting this information from the referer field.

But sometimes the analysis cannot just simply add information to each event record, an altogether different schema is needed then. For those cases, there is the derived schema. An example of the use of such a schema in the current Lire distribution is the analyser which creates user sessions based on the logged client IP address and user agent. This analyser defines the www-session derived schema.

Report Generation

Once you have all this data, it's time to generate some useful reports out of it. Lire's framework includes a generic report builder. What Lire calls a *report* is actually a *collection* of what one may understand as reports; Lire however speaks about a *subreports*. For example, the proxy's superservice report will contain subreports about the top visited sites, another subreport on the cache hit ratio, as well as several others. The subreports are defined using the *Report Specification Markup Language*. This markup language contains elements for several things: information regarding the schema on which it operates; descriptions that should be included in the generated report to help in the interpretation of the data; parameters that can be used to modify the generated report (for example, to generate a top 20 subreport instead of a top 10); a filter that selects the records that will be used for the subreport; and finally the operations that make up the subreport: grouping, summing, counting, etc. The report markup language covers most simple needs and there is an extension element as well as an API that can be used to hook in more fancy computations. There are no subreport specifications in the current distribution that make use of this feature yet, however. You can see an overview of this process in the Figure 1-4 figure.

Figure 1-4. Report Generation Process

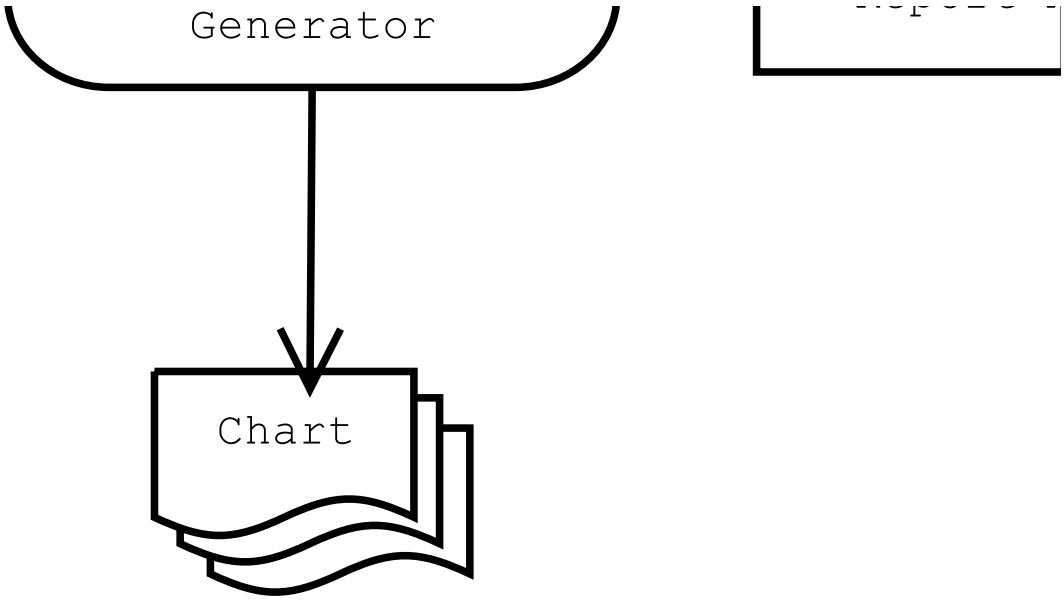


The generated report is another XML file that uses another markup language, this time called the Lire's Report Markup Language. An actual report contains the help descriptions from the report specifications, information on the subreport specifications used, as well as the actual subreport's data.Using another intermediary XML file as output format makes all sort of things possible in the formatting and post-processing stage.

Report Formatting and Other Post-Processing

The last process works with the generic XML report. Using a domain-specific XML format for the generated format makes it easy for the framework to support multiple different formats. Supporting a new output format is just a matter of writing a new module that processes the XML report file.

Figure 1-5. Processing of the XML Report Using The APIs



As shown in the Figure 1-5 figure, you can also process the XML files using the APIs to the XML report format.

Going Further

As you can see form this overview, the Lire framework provides a powerful architecture to use for your log analysis needs. The architecture provides extensibility from log normalisation to post-processing of the reports. Exactly how to use the framework is the topic of the next part.

II. Using the Lire Framework

In this part, you'll learn how to leverage the Lire's framework for your own log analysis need. The most common use cases are developing a converter for a new log format and developping new reports.

The first chapter Chapter 2 explains how to write a converter for a new log format.

The responsibility of the converter is to map the information contained in a log file to the data model of a specific DLF schema. When developping a converter for a log format which doesn't fall in the domain one of the existing DLF schema, you'll need to write a new one. This is the topic of the following chapter Chapter 3.

The chaper Chapter 4 gives information on how to write DLF analysers that can adds data to the base log information.

The chapter Chapter 5 this part gives some notes on how to develop new reports.

Chapter 2. Writing a New DLF Converter

Before Lire can do various analysis and generate reports on the data contained in your various log files, it must first be converted to a common data model. This is specifically the job of the DLF converter.

So if you want to generate the same reports for your RealServer log files (currently unsupported) than for you web server, you only need to develop a DLF converter which maps the RealServer content to the www DLF schema.

> **Note:** If no existing DLF schemas represent correctly the domain of your application log file, it is easy to develop a new one. Consult the chapter Chapter 3 for the whole story.

This chapter will show you through an example how to develop a new DLF converter for a kind of useless log format: the common log format encapsulated in syslog. (It is useless because there is not many reasons to make your web server logs it requests through syslog. And it would be probably be simpler to just use the **cut** command to remove the syslog header.)

> **Note:** The `doc/examples` in the source distribution contains another commented example which could serve as a starting point for your converters.

Prerequisites

Developing a new DLF converter requires some basic programming skills in perl. Altough not strictly necessarily, you should be familiar with perl object-oriented programming model. If you aren't, you should read perltoot(1) before continuing.

The common_syslog Log Format

The log format supported by our DLF converter is simply the standard Common Log Format supported by most web servers with a syslog header prepended to each line. Here is an example of what such a log file might contain:

```
May 10 11:13:10 hibou httpd[12344]: Apache/1.3.26 (Unix) Debian GNU/Linux Embperl/1.3.3 PHP/4.1.2
May 10 11:13:11 hibou httpd[12345]: 192.168.250.10 - - \
  [10/May/2003:11:13:11 +0200] "GET /" HTTP/1.1 200 1523
May 10 11:13:12 hibou httpd[12346]: 192.168.250.10 - - \
  [10/May/2003:11:13:11 +0200] "GET /images/logo.png" HTTP/1.1 200 1201
May 10 11:13:12 hibou httpd[12348]: 192.168.250.10 - - \
  [10/May/2003:11:13:11 +0200] "GET /images/corner.png" HTTP/1.1 200 1021
```

Remember that the other layer is a syslog log file and could contains other things than only the web server's requests. The first line in the example isn't a request record but really what usually ends up in the "error_log" and is a message about the server starting.

Creating the DLF Converter Skeleton

Put simply, a DLF converter is a perl object which implements a set of predefined methods (aka an “interface” in the object-oriented jargon).

Since a DLF converter is a perl object, it must be instantiated from a class. Classes in perl are defined in packages. We’ll name the package which implements our converter `MyConverters::SyslogCommonConverter`. To create such a package, you need to create a file named `MyConverters/SyslogCommonConverter.pm` in a directory searched by perl.

- You can obtain perl's default search list by running the command **`$ perl -V`**.
- This search list can be modified by setting the PERL5LIB environment variables.

Here is a first cut of our DLF converter:

```
package MyConverters::SyslogCommonConverter;

use base qw/Lire::DlfConverter/;

1;
```

The first line declare that the code is in the `MyConvertersw::SyslogCommonConverter` package. The second one specifies that objects in this package are subclasses of the Lire::DlfConverter packages. The last line fullfill perl’s requirement that package returns a true value once they are initialized.

This is a complete DLF, altough useless, DLF Converter. In fact, it isn’t complete because if you tried to register an instance of that class, you’ll get “unimplemented method” errors. Besides, we don’t even yet have a formal way to create instance of our converter. This is our next task.

Adding a Constructor

The Lire framework doesn’t place any restrictions on your DLF converter constructor. In fact, the constructor isn’t used by the framework at all, it will only be used by your DLF converter registration script (the Section called *Registering Your DLF Converter with the Lire Framework*).

We will follow perl’s convention of using a method named `new` for our constructor and of using an hash reference to hold our object’s data.

Here is our complete constructor:

```
use Lire::Syslog;

sub new {
    my $pkg = shift;

    my $self = bless {}, $pkg;

    $self->{syslog_parser} = new Lire::Syslog();
```

```
    return $self;
}
```

Since our log format is based on syslog, we will reuse the syslog parsing code included in Lire. This is the reason we instantiate a `Lire::Syslog` object and save a reference to it in our constructor.

The Meta-Data Methods

The Lire::DlfConverter interface requires two kinds of methods. First, it requires methods which provide information to the framework on your converter. Second, it requires methods which will actually implement the conversion process. It this the format that this section documents.

The DLF Converter Name

The method `name()` should returns the name of our DLF converter. It is this name that is passed to the **lr_log2report** command. This name must be unique among all the converters registered and it should be restricted to alphanumerical characters (hyphens, period and underscores can also be used).

We will name our converter `common_syslog`:

```
sub name {
    return "common_syslog";
}
```

Providing Information To Users

The next two required methods are used to give more verbose information on your converter to the users. The converter's `title()` and `description()` can be use to display information about your converter from the user interface or to generate documentation.

The `title()` should simply returns a string:

```
sub title {
    return "Common Log Format embedded in Syslog DLF Converter";
}
```

The `description()` method should returns a DocBook fragment describing your converter and the log formats it support. If you don't know DocBook just restrict yourself to using the `para` elements to make paragraphs:

```
sub description {
   return <<EOD;
<para>This DLF Converter extracts web server’s requests and error
information from a syslog file.
</para>
<para>The requests and errors should be logged under the
<literal>httpd</literal> program name. The errors are mapped to the
<type>syslog</type> schema, the requests are mapped to the
<type>www</type> schema.
</para>
<para>Syslog records from another program than
<literal>httpd</literal> are ignored.
</para>
EOF
}
```

Providing Information to the Framework

Two other meta-data methods are used by the framework itself. The first one specifies to what DLF schemas your DLF converter is converting to:

```
sub schemas {
    return ( "www", "syslog" );
}
```

In our case, we are converting to the syslog and www schemas. Like we described it in our converter’s description, we will map the web server’s error message to the syslog schema and the request logs to the www schema. Other alternatives would have been to only map the requests information to www schema or map all the non-request records to the syslog schema. The rationale behind the current choice (besides this being an example) is that it make it convenient to process one log file to obtain a report containing the requests and errors from our web server. For that use case, it is best to ignore the non-web server related stuff.

The other method affects how the conversion process will be handled. Lire offers two mode of conversion, the line oriented one and the file oriented one. (Both will be described in the next section). If your log file is line-oriented (each lines is one log record) like most log files are, you should use the line-oriented conversion mode:

```
sub handle_log_lines {
    return 1;
}
```

The Conversion Methods

The actual conversion process is handled through three methods: `init_dlf_converter`, `finish_conversion()` and either `process_log_file()` or `process_log_line()` depending on the conversion mode (as determined by `handle_log_lines()`'s return value.

Conversion Initialization

The method `init_dlf_converter()` will be called once before the log file is processed. It should be use to initialize the state of your converter. Since our DLF Converter doesn't need any initialization and doesn't need any configuration, the method is simply empty:

```
sub init_dlf_converter {
    my ( $self, $process ) = @_;

    return;
}
```

The `$process` parameter which is passed to all the processing methods is an instance of `Lire::DlfConverterProcess`. This is the object which is driving the conversion process and it defines several methods which you will use in the actual conversion process.

Conversion Finalization

The method `finish_conversion()` will be called once after the log file has been completely processed. This method will be mostly of use to stateful converter, that is DLF converters which generates DLF records from more than one line. Since this is not our case, we simply leave the method empty:

```
sub finish_conversion {
    my ( $self, $process ) = @_;

    return;
}
```

The DLF Conversion Process

Whether you are using the file-oriented or line-oriented conversion mode, the principles are the same. You extract information from the log file and creates DLF records from it. Your DLF converter communicates with the framework by calling methods on the `Lire::DlfConverterProcess` object which is passed as parameter to your methods.

Here is the complete code of our conversion method:

```
use Lire::Apache qw/parse_common/;
```

```
sub process_log_line {
    my ( $self, $process, $line ) = @_;

    my $sys_rec = eval { $self->{syslog_parser}->parse( $line ) };
    if ( $@ ) {
        $process->error( $@, $line );
        return;
    } elsif ( $sys_rec->{process} ne 'httpd' ) {
        $process->ignore_log_line( $line, "not an httpd record" );
        return;
    } else {
        my $common_dlf = {};
        eval { parse_common( $sys_rec->{content}, $common_dlf ) };
        if ( $@ ) {
            $sys_rec->{message} = $sys_rec->{content};
            $process->write_dlf( "syslog", $sys_rec );
        } else {
            $process->write_dlf( "www", $common_dlf );
        }
    }

}
```

The first thing that should be noted is that in the line-oriented conversion mode, the method `process_log_line()` will be called once for each line in the log file.

Secondly, the actual parsing of the line is done using two functions: `parse_common` and `Lire::Syslog`'s `parse`. These methods simply uses regular expressions to extract the appropriate information from the line and put it in an hash reference. What is important is that these methods already uses as key names the schema's field names.

Finally, you can see that there are four different methods used on the `$process` object to report different kind of information:

Reporting Error

> The example uses the `eval` statement to trap errors during the syslog record parsing. If the line cannot be parsed as a valid syslog record, it is an error and it is reported through the `error()` method. The first parameter is the error message and the second one is the line to which the error is associated. This last parameter is optional.

Ignoring Information

> When the syslog event doesn't come from the **httpd** process, we ignore the line. Ignored line are reported to the framework by using the `ignore_log_line()` method. The first parameter is the line which is ignored. The second optional parameter gives the reason why the line was ignored.

Creating DLF Records

> Finally, DLF records are created by using the `write_dlf()` method. Its first parameter is the schema to which the DLF record complies. This schema must be one that is listed by your converter's `schemas()` method. The second parameter is the DLF data contained in an hash reference. The DLF record will be

created by taking for each field in the schema the value under the same name in the hash. (Since in the syslog schema, the field which contains the actual log message is called `message`, this is the reason we are assigning the content value to the message key.) Missing fields or fields whose value is `undef` will contains the special `LR_NA` missing value marker. Keys in the hash that don't map to a schema's field are simply ignored.

In our example, we distinguish between the server's error message (mapped to the syslog schema) and the request information (mapped to the www schema) based on whether `parse_common` succeeded in parsing the line.

Saving Log Line

Another possibility, not shown in our example, is to ask that the line be saved for a later processing. This is mostly of use to converters who maitains state between lines. In the cases, it is quite the case that there are related lines that are missing from the end of the log file. In that case, you save the line and they will automatically seen by the next run of your converter on the same DLF store. This option is only available in the line-oriented mode of conversion.

File-Oriented Conversion

The same principles apply when you are using the file-oriented mode of conversion. This mode will usually be used for binary log formats or format which aren't line-oriented like XML.

For demonstration purpose, the following code could be added to transform our line-oriented converter into a file-oriented one:

```
sub handle_log_lines {
    return 0;
}

sub process_log_file {
    my ( $self, $process, $fh ) = @_;

    my $line;
    while ( defined( $line = <$fh> ) {
        chomp $line;
        $self->process_log_line( $process, $line );
    }
}
```

The difference between the above code and using the line oriented mode is that the framework won't be aware of the number of log lines processed and your converter might have troubles when processing log files which uses a different line-ending convention than the host you are runnig on. Bottom line is that you should use the line-oriented conversion mode when your log format is line oriented.

Registering Your DLF Converter with the Lire Framework

We first said that DLF converters are perl *objects* which implements the Lire::DlfConverter interface. What we did is write a *class* which implements the said interface. Creating the object from that class is the responsability of the *DLF converter registration script*. This is simply a snippet of perl code which instantiates your object and registers it with the Lire::PluginManager:

```
use Lire::PluginManager;
use MyConverters::SyslogCommonConverter;

Lire::PluginManager->register_plugin(
            MyConverters::SyslogCommonConverter->new() );
```

That's all there is to it, really. You put this snippet in a file named `syslog_common_init` in one of the directories listed in the `plugins_init_path` configuration variable.

> **Note:** Some other notes on this topic:
>
> 1. The file can actually be named anything you want, the name *`service`*`_init` just make it clear what is the purpose of the file.
> 2. The initial value of the `plugins_init_path` contains the directories *`sysconfdir`*`/lire/plugins` and *`HOME`*`/.lire/plugins`. You can change this list by using the **lire** tool.
> 3. Your registration script can create and register more than one object.

You can now generate a www report for log files in that format using the command **`lr_log2report common_syslog < file.log`**.

DLF Converter API

The complete DLF Converter API documentation is included in POD format in the Lire distribution. It is usually formatted as man pages. You can alway read it using the **perldoc** command.

The following packages documentation should be consulted: Lire::DlfConverter(3), Lire::DlfConverterProcess(3) and Lire::PluginManager(3).

Chapter 3. Writing a DLF Schema

If you want to develop a DLF converter for an application whose logging data model isn't adequately represented by one of the existing DLF schema, you'll need to develop a new one.

If you are familiar with SQL, a DLF schema is similar to a table schema description. A DLF file can be seen as a table, where each log record is represented by a table row. Each log record in the same DLF schema shares the same fields.

Designing the ftpproto schema

In this chapter, we will create a new schema for logging of FTP session. That DLF schema could serve for an improved DLF converter for log files generated by Microsoft Internet Information Server. Lire currently has a DLF converter for these log files but the current ftp DLF schema is modelled after the xferlog log file which only represents file transfers whereas the log generated by Microsoft Internet Information Server contains more detailed information on the ftp session.

Here is an example of such a log file:

```
#Software: Microsoft Internet Information Server 4.0
#Version: 1.0
#Date: 2001-11-29 00:01:32
#Fields: time c-ip cs-method cs-uri-stem sc-status
00:01:32 10.0.0.1 [56]created spacedat/091001092951LGW_Data.zip 226
00:01:32 10.0.0.1 [56]created spacedat/html/bx01g01.gif 226
00:01:32 10.0.0.1 [56]created spacedat/html/catlogo.gif 226
00:01:32 10.0.0.1 [56]QUIT - 226
00:03:32 10.0.0.1 [58]USER badm 331
00:03:32 10.0.0.1 [58]PASS - 230
```

As you can see, this log file contains other information beyond the simple upload/download represented in the standard FTP schema. It a session identifier, the command executed, as well as the result code of the action. Our new schema should be able to represent these things.

Creating The Schema File

To create a DLF schema, you have to create a XML file named after your schema identifier: `ftpproto.xml`. Schema name should be made of alphanumeric characters. This schema identifier is case sensitive. You schema identifer shouldn't contains hyphens (`-`) or underscore characters (_). (The hyphen is used for a special purpose).

All DLF schemas starts and ends the same way:

```
<?xml version="1.0" encoding="ascii"?>
<!DOCTYPE lire:dlf-schema PUBLIC
  "-//LogReport.ORG//DTD Lire DLF Schema Markup Language V1.1//EN"
  "http://www.logreport.org/LDSML/1.1/ldsml.dtd">
<lire:dlf-schema xmlns:lire="http://www.logreport.org/LDSML/"

              superservice="ftpproto"
              timestamp="time"

              >
```

```
<!-- Other elements will go here -->
</lire:dlf-schema>
```

The first lines contains the usual XML declaration and DOCTYPE declarations, you'll find in many XML documents. The real stuff starts at the `lire:dlf-schema`. What is important for your schema are the value of the `superservice` and `timestamp` attributes. The first one contains your schema identifier. It is called "superservice" for historical reasons. The other one should contains the name of the field which order the record by their event type. (See the Section called *The Field Types* for more information.)

The last line in the above excerpt would be the last thing in the file and closes the `lire:dlf-schema` element.

Adding the Schema's Description

The next things that goes into the schema file are the schema's title and description. Both are intended for developers to read and should be informative of the scope of the schema:

```
<!-- Starting lire:dlf-schema element was omitted -->

 <lire:title>DLF Schema for FTP Protocol</lire:title>

 <lire:description>
   <para>This DLF schema should be used for FTP servers that have
         detailed information on the FTP connection in their log
         files.
   </para>
   <para>Each record represents a command done by the client during
    the FTP session.
   </para>
 </lire:description>
```

The content of the `lire:description` elements are DocBook elements. If you don't know DocBook, you just need to know that paragraphs are delimited using the `para` elements.

Defining the Schema's Fields

The only remaining things in the schema definitions are the field specifications. Here is the definition of the first one:

```
 <lire:field name="time" type="timestamp" label="Timestamp">
   <lire:description>
     <para>This field contains the timestamp at which the command was
             issued.
     </para>
   </lire:description>
 </lire:field>
```

As you can see, the fields are defined using the `lire:field` element which has three attributes:

name

This attribute contains the name of the field. This name should contains only alphanumeric characters. It can also make use of the underscore character.

type

This attribute contains the type of the field. The available types will described shortly.

label

This should contains the column label that should be used by default in your report for data coming from this field. This label should be short but descriptive.

The field's description is held in the `lire:description` element which contains DocBook markup. The field's description should be descriptive enough so that someone implementing a DLF converter for this schema knows what goes where.

The Field Types

The main types available for fields are:

timestamp

This should be use for field which contains a value to indicate a particular point in time. All timestamp values are represented in the usual UNIX convention: number of seconds since January 1st 1970.

Each DLF schema must contains at least one field of this kind and its name should be in the `lire:dlf-schema`'s `timestamp` attribute.

hostname

This type should be used for fields which contains an hostname *or* IP address.

It is important to mark such fields, because it will possible eventually to resolve automatically IP addresses to hostname.

bool

Type for boolean values.

number

Type for numeric values.

> **Important:** You shouldn't use this type when the values are limited in number and are semantically related to an enumeration like result code. You should use the string type for this.
>
> You should only use the number type for values which you'll want to report in classes instead on the individual values.

bytes

This type should be use for numeric values which are quantities in bytes. The more specific typing is useful for display purpose.

duration

This type should be use for numeric values which are quantities of time. The more specific typing is useful for display purpose.

string

This is the type which can be use for all other purpose.

Note: If you read the specifications, you'll find other types which are used. These additional types don't bring anything over the basic ones defined above and you shouldn't use them.

In addition to the time field defined above, here are the remaining field defintions which make our complete ftpproto schema:

```
<lire:field name="sessid" type="string" label="Session">
  <lire:description>
   <para>This field should contains an identifier that can used
   to related the commands done in the same FTP session. This
   identifier can be reused, but shouldn't be while the FTP session
   isn't closed.
   </para>
  </lire:description>
</lire:field>

<lire:field name="command" type="string" label="Command">
  <lire:description>
   <para>This field contains the FTP command executed. The FTP
    protocol command names (STOR, RETR, APPE, USER, etc.) should be used.
   </para>
  </lire:description>
</lire:field>

<lire:field name="result" type="string" label="Result">
  <lire:description>
   <para>This should contains the FTP result code after executing
   the command.
   </para>
  </lire:description>
</lire:field>

<lire:field name="cmd_args" type="string" label="Argument">
  <lire:description>
   <para>This field should contains the parameters to the FTP command.
   </para>
```

```
  </lire:description>
</lire:field>

<lire:field name="size" type="bytes" label="Bytes Transferred">
  <lire:description>
   <para>When the command involves a transfer like for the RETR or STOR
    command, it should contains the number of bytes transferred.
   </para>
  </lire:description>
</lire:field>

<lire:field name="elapsed" type="duration" label="Elasped">
  <lire:description>
   <para>This field contains the number of seconds executing the
         command took.
   </para>
  </lire:description>
</lire:field>
```

Installing The Schema

Making available the new schema to the Lire framework is pretty easy: just copy the file to one of the directories set in the `lr_schemas_path` configuration variable. By default, this variable contains the directories *`datadir`*`/lire/schemas` and *`HOME`*`/.lire/schemas`. Like all other configuration variables, its value can be changed using the **lire** tool.

Since we want our schema to be available for other users as well, we will install it in the system directory:

```
&root-prompt; install -m 644 ftproto.xml /usr/local/share/lire/schemas
```

(In this case, Lire was installed under `/usr/local`.

Chapter 4. Writing a New DLF Analyser

In Lire, a DLF Analyser is a plugin that can extract or derived data from other DLF data. The idea is that these analysis do not depends on the underlying log format but that it can be found simply by using the data normalised in the DLF schema.

For example, an analyser could assign category based on the url that was visited (like assigning the 'Public' or 'Private' category). This categorising operation doesn't depends on the log format but only on the presence of the *`requested_page`* field in the schema. This would be an example of a special kind of analyser, a Lire DLF Categoriser. This is a simpler analyser that can create new fields based on one DLF record.

> **Note:** The `doc/examples` in the source distribution contains the complete code for this categoriser.

There is a more generic kind of analysers that create data in another dlf streams based on arbitrary queries on the source DLF schema. An example of this kind is an analyser that construct session summary from the www requests. It reads the DLF records of the www DLF schema and creates www-user_session DLF records from that.

Writing an analyser is similar to writing a DLF converter, so consult Chapter 2 for the details converning registration and using configuration.

Writing a Categoriser

The simplest form of analyser are categorisers. In this section, we will show an example of how to write a categoriser that can assign categories using regular expressions to each www requested page.

Defining The Extended Schema

A categoriser writes DLF in an extended schema. An extended schemas is an extension of a base schema. If you are familiar with SQL you can see it as an inner join with the main schema. That is each fields in the main schema will have the extension fields of the extended schema.

In our case our extended schema is very simple, it only adds one *`category`* field to the www schema.

Defining an extended schema is identical to writing a DLF Schema with exception that we use a different top-level element. You should consult Chapter 3 for all the details. Here is the extended schema that our categoriser will use:

```
<?xml version="1.0"?>
<!DOCTYPE lire:extended-schema PUBLIC
  "-//LogReport.ORG//DTD Lire DLF Schema Markup Language V1.1//EN"
  "http://www.logreport.org/LDSML/1.1/ldsml.dtd">
<lire:extended-schema id="www-category" base-schema="www"
 xmlns:lire="http://www.logreport.org/LDSML/">

 <lire:title>Category Extended Schema for WWW service</lire:title>

 <lire:description>
  <para>This is an extended schema for the WWW service which adds a
    category field based on the regexp matched by the requested_page.
  </para>
```

```
 </lire:description>

 <lire:field name="category" type="string" label="Category">
  <lire:description>
   <para>This fields contain the page category.</para>
  </lire:description>
 </lire:field>
</lire:extended-schema>
```

The difference with a regular DLF schema is that it starts with the `extended-schema` tag which has a `base-schema` attribute which should contain the DLF schema or derived DLF schema that is extended.

Defining the Categoriser

Like a DLF Converter, the categoriser s an object deriving from a base class which defines the categoriser interface. In the categoriser case, that interface is Lire::DlfCategoriser. The categoriser also has to provide some meta-information to the framework. Here is the code for all of this:

```
package MyAnalysers::PageCategoriser;

use base qw/Lire::DlfCategoriser/;

sub new {
    return bless {}, shift;
}

sub name {
    return 'page-categoriser';
}

sub title {
    return "A page categoriser";
}

sub description {
    return "<para>A categoriser that assigns categories based on a map
    of regular expressions to categories.</para>";
}

sub src_schema {
    return "www";
}

sub dst_schema {
    return "www-category";
}
```

The methods different from the DLf converter case are the `src_schema` which specifies the schema which to which fields are added and the `dst_schema` which gives the schema specifying the fields that will be added.

Categoriser Configuration

Our categoriser will assign categories based on a mapping from regular expression to category names. To be useful, this mapping should be configurable. Like all plugins in Lire, DLF categorisers can use the Lire Configuration Specification Markup Language to defines the configuration data they use (see Chapter 8 for the full details). The convention is that if there is a parameter named *`yourname`*`_propeties`, this is considered the configuration specification for the plugin `yourname`. This will mean that a little button will appear in the **lire** user interface so that the user can configure your plugin data.

In our categoriser case, we will define a list of records which will enable the user to define many pairs of regular expression and category name:

```
<?xml version="1.0"?>
<!DOCTYPE lrcsml:config-spec PUBLIC
  "-//LogReport.ORG//DTD Lire Report Configuration Specification Markup Language V1.0//EN"
  "http://www.logreport.org/LRCSML/1.1/lrcsml.dtd">
<lrcsml:config-spec xmlns:lrcsml="http://www.logreport.org/LRCSML/"
                    xmlns:lrcml="http://www.logreport.org/LRCML/">

 <lrcsml:list name="page-categoriser_properties">
  <lrcsml:summary>Page Categoriser Configuration</lrcsml:summary>

  <lrcsml:description>
   <para>This is a list of regexp that will be apply in this order
    along the category that should be applied when the regexp match.
   </para>
  </lrcsml:description>

  <lrcsml:record name="regex2category">
   <lrcsml:summary>The Regexp-Category Association</lrcsml:summary>
   <lrcsml:string name="regex">
    <lrcsml:summary>Regex</lrcsml:summary>
    <lrcsml:description>
     <para>The regular expression to test.</para>
    </lrcsml:description>
   </lrcsml:string>

   <lrcsml:string name="category">
    <lrcsml:summary>Category</lrcsml:summary>
    <lrcsml:description>
     <para>This field contains the category that should be assigned.</para>
    </lrcsml:description>
   </lrcsml:string>
  </lrcsml:record>
 </lrcsml:list>
p <lrcml:param name="page-categoriser_properties">
  <lrcml:param name="regex2category">
   <lrcml:param name="regex">.*</lrcml:param>
   <lrcml:param name="category">Unknown</lrcml:param>
  </lrcml:param>
 </lrcml:param>
 </lrcsml:list>
```

```
</lrcsml:config-spec>
```

This specification also sets a list containing one catchall regex with the category 'Uknown'. The user could add other values before that. An alternative implementation could define a field specifying the default category to assign when no regular expression matches.

Categoriser Implementation

Two methods are needed to implement the categoriser. The first is an initialisation method called `initialise`. This method receives as parameter the configuration data entered by the user.

In our case, we will compile the regular expressions for faster processing later on :

```
sub initialise {
    my ( $self, $config ) = @_;

    foreach my $map ( @$config ) {
        $map->[0] = qr/$map->[0]/;
    }

    $self->{'categories'} = $config;
    return;
}
```

The categorising is made in the `categorise` method. This method receives as parameter the DLF record to which the extended fields should be added. This DLF record is an hash reference containing one key for each of the fields defined in the source DLF schema. We simply assign the extended fields by adding new keys to the hash reference :

```
sub categorise {
    my ( $self, $dlf ) = @_;

    foreach my $map ( @{$self->{'categories'}} ) {
        if ( $dlf->{'requested_page'} =~ /$map->[0]/ ) {
            $dlf->{'category'} = $map->[1];
            return;
        }
    }
    return;
}
```

That's all. Like for the DLF converter you'll need to register this analyser with the `Lire::PluginManager` (see the Section called *Registering Your DLF Converter with the Lire Framework* in Chapter 2 for more information.

Writing an Analyser

When a categoriser isn't sufficient for your needs, you can write an `Lire::DlfAnalyser` which gets complete control on the analysis process. The main difference with at categoriser is that the `dst_schema` method will contain refer to a derived schema instead of an extended schema.

The core of the analyser is done in the `analyse` method that takes a reference to the store onto which data will be analysed and to a `Lire::DlfAnalyserProcess` callback object which should be use to write new DLF records and report errors. The method also receives the plugin configuration data. The analyser should create a `Lire::DlfQuery` to select the records necessary for its analysis.

The `doc/examples` in the source distribution contains the a boiler plate for witing an Analyser.

DLF Analyser API

The complete DLF Analyser API documentation is included in POD format in the Lire distribution. It is usually formatted as man pages. You can alway read it using the **perldoc** command.

The following packages documentation should be consulted: Lire::DlfAnalyser(3), Lire::DlfAnalyserProcess(3), Lire::DlfCategoriser(3), Lire::DlfQuery(3) and Lire::PluginManager(3).

Chapter 5. Writing a New Report

Writing a new report involves writing a report specification, e.g. `/service/<superservice>/reports/top-foo-by-bar.xml`, and adding this report along with possible configuration parameters to `<service>.cfg`. E.g., to create a new report, based upon `email/from-domain.xml`: copy the file `/usr/local/etc/lire/email.cfg` to `~/.lire/etc/email.cfg`. Copy the file `/usr/local/share/lire/reports/email/top-from-domain.xml` to e.g. `~/.lire/reports/reports/email/from-domain.xml`. Edit the last file to your needs, and enable it by listing it in your `~/.lire/etc/email.cfg`.

Beware! The name of the report generally consists of alphanumerics and '-', but the name of parameters may *not* contain any '-' characters. It generally consists of alphanumerics and '_' characters.

Filter Specification

For now, you'll have to refer to the example filters as found in the current report specification files. We'll give one other example here: specifying a time range.

Suppose you want to be able to report on only a specific time range. You could build a (possibly global and reused) filter like:

```
<lire:filter-spec>
  <lire:and>
    <lire:ge arg1="$timestamp" arg2="$period-start"/>
    <lire:le arg1="$period-end" arg2="$timestamp"/>
  </lire:and>
</lire:filter-spec>
```

When trying your new filter, you could install it in `~/.lire/filters/your-filter-name.xml`. When lr_dlf2xml looks up a filter which was mentioned in the report configuration file, it looks first in `~/.lire/filters/`, and then in `.../share/lire/filters/`.

III. Developer’s Reference

Chapter 6. Lire Data Types

Lire Textual Elements

This DTD module defines elements related that contains human-readable content in all the Lire DTDs.

This module will also imports some DocBook XML V4.1.2 elements for richer semantic tagging.

This module is also namespace aware and will honor the setting of *`LIRE.pfx`* to scope its element

The latest version of that module is 2.0 and its public identifier is -//LogReport.ORG//ELEMENTS Lire Textual Elements V2.0//EN.

```
<!--
    Make sure LIRE.pfx is defined. This declaration will be
    ignored if it was already defined.
                                                                -->
<!ENTITY % LIRE.pfx          "lire:"                              >

<!ENTITY % LIRE.title        "%LIRE.pfx;title"                    >
<!ENTITY % LIRE.description  "%LIRE.pfx;description"              >
```

`title` element

The `title` element contains a descriptive title.

This element represent some title in Lire. It can be used to give a title to a report specification or to specifify the title of a report or subreport.

The content of this element should be localized.

This element doesn't have any attribute.

```
<!ELEMENT %LIRE.title; (#PCDATA)                                  >
```

DocBook Elements

The standard `para`, `formalpara` and admonition elements (`note`, `tip`, `warning`, `important` and `caution`) are used as well as their content may be used.

```
<!ENTITY % docbook-block.mix
   "para|formalpara|warning|tip|important|caution|note">

<!ENTITY % DocBookDTD PUBLIC
  "-//OASIS//DTD DocBook XML V4.1.2//EN"
  "http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dtd">
```

```
%DocBookDTD;
```

`description` element

The `description` element is used to describe an element. It can be used to describe DLF fields, describe a report specification or include descriptions in the generated reports.

This element can contains one or more of the block-level DocBook elements we use.

The content of this element should be localized.

This element doesn’t have any attributes.

```
<!ELEMENT %LIRE.description; (%docbook-block.mix;)+>
```

Chapter 7. Common Textual Elements to All XML Formats

Lire Data Types Parameter Entities

This module contains the parameter entity declarations for the data types used by all Lire DTDs.

All defined data types have a `<type>.type` parameter entity which defines their type as an XML type valid in an attribute declaration and a `<type>.name` parameter entity that declare their name.

Additionally, this module declares <name>.types parameter entities that group related types together.

The latest version of that module is 1.0 and its public identifier is -//LogReport.ORG//ENTITIES Lire Data Types V1.0//EN.

Boolean Type

The bool type. It contains a boolean value, either `0`, `1`, `f`, `t`, `false` or `true`.

```
<!ENTITY % bool.type         "0 | 1 | f | t | false | true | yes | no">
<!ENTITY % bool.name         "bool"                                    >
```

Integer Type

The int type can contains positive or negative 32 bits integer.

```
<!ENTITY % int.type          "CDATA"                                   >
<!ENTITY % int.name          "int"                                     >
```

Number Type

The number type can contains any number either integral or floating point.

```
<!ENTITY % number.type       "CDATA"                                   >
<!ENTITY % number.name       "number"                                  >
```

String Type

The string type contains any displayable text string.

```
<!ENTITY % string.type       "CDATA"                                  >
<!ENTITY % string.name       "string"                                 >
```

Timestamp type

The timestamp type contains a time representation which contains the date and time informations. It can be represented in UNIX epoch time.

```
<!ENTITY % timestamp.type    "CDATA"                                  >
<!ENTITY % timestamp.name    "timestamp"                              >
```

Time Type

The time type contains a time representation which contains only the time of the day, not the date. For example, this data type can represent 12h00, 15:13:10, etc.

```
<!ENTITY % time.type         "CDATA"                                  >
<!ENTITY % time.name         "time"                                   >
```

Date Type

The date type contains a time representation which contains only a date.

```
<!ENTITY % date.type         "CDATA"                                  >
<!ENTITY % date.name         "date"                                   >
```

Duration Type

The duration type contains a quantity of time. For example : `5s`, `30h`, `2days`, `3w`, `2M`, `1y`. (The authoritive list of supported duration types is coded in `Lire::DataTypes::duration2sec`.)

```
<!ENTITY % duration.type     "CDATA"                                  >
<!ENTITY % duration.name     "duration"                               >
```

IP Type

The ip type contains an IPv4 address.

```
<!ENTITY % ip.type           "CDATA"                                >
<!ENTITY % ip.name           "ip"                                   >
```

Port Type

The port type contains a port as used in the TCP to name the ends of logical connections. See also RFC 1700 and http://www.iana.org/numbers.htm. Commonly found in /etc/services on Unix systems.

```
<!ENTITY % port.type         "CDATA"                                >
<!ENTITY % port.name         "port"                                 >
```

Hostname Type

The hostname type contains an DNS hostname. (It can also contains the IPv4 address of the host).

```
<!ENTITY % hostname.type     "NMTOKEN"                              >
<!ENTITY % hostname.name     "hostname"                             >
```

URL Type

The url type represents URL.

```
<!ENTITY % url.type          "CDATA"                                >
<!ENTITY % url.name          "url"                                  >
```

Email Type

The email type can be used to represent an email address.

```
<!ENTITY % email.type        "CDATA"                                >
<!ENTITY % email.name        "email"                                >
```

Bytes Type

The bytes type can be used to represent quantity of data. (`5m`, `1.2g`, 300bytes, etc.)

```
<!ENTITY % bytes.type        "CDATA"                                >
<!ENTITY % bytes.name        "bytes"                                >
```

Filename Type

The filenametype can be used to Represent the name of a file or directory.

```
<!ENTITY % filename.type     "CDATA"                                >
<!ENTITY % filename.name     "filename"                             >
```

Field Type

> **Important:** This type should be considered internal to Lire and shouldn't be used as a parameter or DLF field type.

The field type can contains a DLF field name. It is used in the parameter specification to represent a choice of sort field for example.

```
<!ENTITY % field.type        "NMTOKEN"                              >
<!ENTITY % field.name        "field"                                >
```

Superservice Type

Important: This type should be considered internal to Lire and shouldn't be used as a parameter or DLF field type.

```
<!ENTITY % superservice.type "NMTOKEN"                              >
<!ENTITY % superservice.name "superservice"                         >
```

Related Types

```
<!ENTITY % basic.types       "%bool.name; | %int.name; |
                              %number.name; | %string.name;"        >
<!ENTITY % internet.types    "%email.name; | %url.name; |
                              %ip.name; | %hostname.name; |
                              %port.name;"                          >
<!ENTITY % misc.types        "%filename.name; | %bytes.name; "      >
<!ENTITY % time.types        "%date.name; | %time.name; |
                              %timestamp.name; | %duration.name;"   >

<!ENTITY % lire.types        "%basic.types; | %time.types; |
                              %internet.types; | %misc.types;"      >
```

Chapter 8. The Lire Report Configuration Specification Markup Language

The Lire Report Configuration Specification Markup Language

Document Type Definition for the Lire Report Configuration Specification Markup Language.

This DTD defines a grammar that is used to specify the configuration parameters used by the Lire framework. Besides the framework parameters, this DTD can be used by extensions writers to register their parameters with the framework. The configuration specifications are usually stored in *`prefix`*`/share/lire/config-spec`.

Currently, Lire's configuration namespace is flat, which means that two different specification documents cannot define parameters of the same names.

Elements of this DTD uses the http://www.logreport.org/LRCSML/ namespace that is usually mapped to the `lrcsml` prefix.

The latest version of that DTD is 1.1 and its public identifier is -//LogReport.ORG//DTD Lire Report Specification Markup Language V1.1//EN. Its canonical system identifier is http://www.logreport.org/LRCSML/1.1/lrcsml.dtd.

```
<!--
                                                                    -->

<!--                    Namespace prefix for validation using the
                        DTD                                         -->
<!ENTITY % LRCSML.xmlns.pfx    "lrcsml"                                 >
<!ENTITY % LRCSML.pfx          "%LRCSML.xmlns.pfx;:"                    >
<!ENTITY % LRCSML.xmlns.attr.name "xmlns:%LRCSML.xmlns.pfx;"            >
<!ENTITY % LRCSML.xmlns.attr
  "%LRCSML.xmlns.attr.name; CDATA #FIXED 'http://www.logreport.org/LRCSML/'">

<!ENTITY % LRCML.xmlns.pfx    "lrcml"                                   >
<!ENTITY % LRCML.pfx          "%LRCML.xmlns.pfx;:"                      >
<!ENTITY % LRCML.xmlns.attr.name "xmlns:%LRCML.xmlns.pfx;">
<!ENTITY % LRCML.xmlns.attr
  "%LRCML.xmlns.attr.name; CDATA #FIXED 'http://www.logreport.org/LRCML/'">

<!-- For the modules which we are including                          -->
<!ENTITY % LIRE.pfx           "%LRCSML.pfx;"                          >
```

This DTD uses the common lire-desc.mod module which is used to include a subset of DocBook in description and text elements.

```
<!ENTITY % lire-desc.mod PUBLIC
    "-//LogReport.ORG//ELEMENTS Lire Description Elements V2.0//EN"
    "lire-desc.mod">
%lire-desc.mod;
```

Each configuration specification is a XML document which has one `config-spec` as its root element.

```
<!ENTITY % LRCSML.config-spec     "%LRCSML.pfx;config-spec"             >
<!ENTITY % LRCSML.summary         "%LRCSML.pfx;summary"                 >
<!ENTITY % LRCSML.boolean         "%LRCSML.pfx;boolean"                 >
<!ENTITY % LRCSML.integer         "%LRCSML.pfx;integer"                 >
<!ENTITY % LRCSML.string          "%LRCSML.pfx;string"                  >
<!ENTITY % LRCSML.dlf-schema      "%LRCSML.pfx;dlf-schema"              >
<!ENTITY % LRCSML.dlf-streams     "%LRCSML.pfx;dlf-streams"             >
<!ENTITY % LRCSML.dlf-converter   "%LRCSML.pfx;dlf-converter"           >
<!ENTITY % LRCSML.command         "%LRCSML.pfx;command"                 >
<!ENTITY % LRCSML.file            "%LRCSML.pfx;file"                    >
<!ENTITY % LRCSML.executable      "%LRCSML.pfx;executable"              >
<!ENTITY % LRCSML.directory       "%LRCSML.pfx;directory"               >
<!ENTITY % LRCSML.select          "%LRCSML.pfx;select"                  >
<!ENTITY % LRCSML.option          "%LRCSML.pfx;option"                  >
<!ENTITY % LRCSML.list            "%LRCSML.pfx;list"                    >
<!ENTITY % LRCSML.object          "%LRCSML.pfx;object"                  >
<!ENTITY % LRCSML.output-format   "%LRCSML.pfx;output-format"           >
<!ENTITY % LRCSML.plugin          "%LRCSML.pfx;plugin"                  >
<!ENTITY % LRCSML.record          "%LRCSML.pfx;record"                  >
<!ENTITY % LRCSML.reference       "%LRCSML.pfx;reference"               >
<!ENTITY % LRCSML.report-config   "%LRCSML.pfx;report-config"           >

<!ENTITY % LRCML.param            "%LRCML.pfx;param"                    >

<!ENTITY % LRCSML.summary         "%LRCSML.pfx;summary"                 >
<!ENTITY % types-spec           "%LRCSML.boolean;|%LRCSML.integer;|
                                 %LRCSML.string;|%LRCSML.dlf-schema;|
                                 %LRCSML.dlf-converter;|%LRCSML.dlf-streams;|
                                 %LRCSML.command;|%LRCSML.file;|
                                 %LRCSML.executable;|%LRCSML.directory;|
                                 %LRCSML.select;|%LRCSML.list;|%LRCSML.object;|
                                 %LRCSML.output-format;|
                                 %LRCSML.plugin;|%LRCSML.record;|%LRCSML.reference;
                                 |%LRCSML.report-config;
                                ">
<!ENTITY % common.mix           "(%LRCSML.summary;)?,(%LIRE.description;)?">
<!ENTITY % default              "(%LRCML.param;)?"                  >
<!ENTITY % common.mix.default   "(%common.mix;, %default;)"         >

<!ELEMENT %LRCML.param; (#PCDATA|%LRCML.param;)*                       >
<!ATTLIST %LRCML.param;
             name      NMTOKEN                            #REQUIRED
             value     CDATA                              #IMPLIED >
```

`config-spec` element

Root element of a configuration specification document. It contains a list of parameter specifications..

This element doesn't have any attributes.

```
<!ELEMENT %LRCSML.config-spec; ((%types-spec;)+)                        >
<!ATTLIST %LRCSML.config-spec;
             %LRCSML.xmlns.attr;
             %LRCML.xmlns.attr;                                         >
```

`summary` element

This element is used for a short one description of the parameter's purpose. Use the `description` element for longer help text.

This element doesn't have any attribute.

```
<!ELEMENT %LRCSML.summary;    (#PCDATA)                                >
```

Parameter Specifiations Elements

Common Attributes

These attributes are common to all parameters specification elements:

name

: Contains the name of the parameter to which this specification apply.

required

: Determines if a valid value is required to make the container validates. Defaults to true.

section

: This attribute can be used to set a menu section which can be used by configuration frontends to group parameters together.

summary

: This attribute is equivalent to the `summary` element.

obsolete

: This attribute can be used to mark a parameter as obsolete. Obsolete parameters will be removed from the specification in a future Lire release.

```
<!ENTITY % common.attr "
```

```
        name     NMTOKEN                                    #REQUIRED
        required NMTOKEN                                    '1'
        section  CDATA                                      #IMPLIED
        summary  CDATA                                      #IMPLIED
        obsolete NMTOKEN                                    '0'">
```

`boolean` element

This element is used to define a boolean parameter which can takes a `yes` or `no` value.

This element doesn't have any specific attributes.

```
<!ELEMENT %LRCSML.boolean; (%common.mix.default;)                    >
<!ATTLIST %LRCSML.boolean;
 %common.attr;
                                                                     >
```

`integer` element

This element is used to define an integer parameter.

This element doesn't have any specific attributes.

```
<!ELEMENT %LRCSML.integer; (%common.mix.default;)                    >
<!ATTLIST %LRCSML.integer;
 %common.attr;
                                                                     >
```

`string` element

This element is used to define an string parameter. These parameters can contains any value.

This can have a `valid-re` attribute which specify a regular expression that the value must match.

```
<!ELEMENT %LRCSML.string; (%common.mix.default;)                    >
<!ATTLIST %LRCSML.string;
 %common.attr;
 valid-re CDATA                                          #IMPLIED
                                                                     >
```

`dlf-converter` element

This element is used to select a registered DlfConverter.

This element doesn't have any specific attributes.

```
<!ELEMENT %LRCSML.dlf-converter; (%common.mix.default;)              >
<!ATTLIST %LRCSML.dlf-converter;
 %common.attr;
                                                                     >
```

`dlf-schema` element

This element is used to select an available DlfSchema.

If this element has the `superservices` set, only superservices can be selected.

```
<!ELEMENT %LRCSML.dlf-schema; (%common.mix.default;)                 >
<!ATTLIST %LRCSML.dlf-schema;
 %common.attr;
 superservices NMTOKEN                                            '0'
                                                                     >
```

`dlf-streams` element

This element is used to configure Lire::DlfStream in Lire::DlfStore.

This element has no attribute.

```
<!ELEMENT %LRCSML.dlf-streams; (%common.mix.default;)                >
<!ATTLIST %LRCSML.dlf-streams;
 %common.attr;
                                                                     >
```

`command` element

This element is used to define a command parameter. To be accepted as valid the parameter's value must point to an executable file or an executable file with the specified value must exist in a directory of the PATH environment variable.

This element doesn't have any specific attributes.

```
<!ELEMENT %LRCSML.command; (%common.mix.default;)                    >
<!ATTLIST %LRCSML.command;
 %common.attr;
```

```
                                                                    >
```

`file` element

This element is used to define a file parameter. To be accepted as valid, the parameter's value must point to an existing file.

This element doesn't have any specific attributes.

```
<!ELEMENT %LRCSML.file; (%common.mix.default;)                      >
<!ATTLIST %LRCSML.file;
 %common.attr;
                                                                    >
```

`directory` element

This element is used to define a directory parameter. To be accepted as valid, the parameter's value must point to an existing directory.

This element doesn't have any specific attributes.

```
<!ELEMENT %LRCSML.directory; (%common.mix.default;)                 >
<!ATTLIST %LRCSML.directory;
 %common.attr;
                                                                    >
```

`executable` element

This element is used to define an executable parameter. To be accepted as valid, the parameter's value must point to an existing executable file.

This element doesn't have any specific attributes.

```
<!ELEMENT %LRCSML.executable; (%common.mix.default;)                >
<!ATTLIST %LRCSML.executable;
 %common.attr;
                                                                    >
```

`select` element

This element is used to define a parameter for which the value is selected among a set of options. The allowed set of options is specified using `option` elements.

This element doesn't have any specific attributes.

```
<!ELEMENT %LRCSML.select;     (%common.mix;,(%LRCSML.option;)+, %default;) >
<!ATTLIST %LRCSML.select;
 %common.attr;
                                                                          >
```

`option` element

This element is used to define the valid values for a `select` parameter.

This element doesn't have any specific attributes.

```
<!ELEMENT %LRCSML.option;     (%common.mix;)                                >
<!ATTLIST %LRCSML.option;
 %common.attr;
                                                                          >
```

`list` element

This element is used to define a parameter that can contains an ordered set of values. The type of values which can be contained is specified using other parameters elements. Any number of parameters of the type specified by the children elements can be contained by the defined parameter.

This element doesn't have any specific attributes.

```
<!ELEMENT %LRCSML.list;       (%common.mix;,(%types-spec;)+,%default;) >
<!ATTLIST %LRCSML.list;
 %common.attr;
                                                                          >
```

`object` element

This element is used to define a parameter that will instantiate an object. The object will be instantiated by calling the "new_from_config()" class method defined in the package specified by the element's `class` attribute. The constructor will receive the hash instantiated from the parameter's components as parameter.

The `label` attribute can be used to specify the contained element that should be used to represent this object in lists.

```
<!ELEMENT %LRCSML.object;       (%common.mix;,(%types-spec;)+,%default;) >
<!ATTLIST %LRCSML.object;
 %common.attr;
        class    NMTOKEN                                    #REQUIRED
        label    NMTOKEN                                    #IMPLIED
                                                                    >
```

`output-format` element

This element is used to select an available OutputFormat.

This element doesn't have any specific attributes.

```
<!ELEMENT %LRCSML.output-format; (%common.mix.default;)             >
<!ATTLIST %LRCSML.output-format;
 %common.attr;
                                                                    >
```

`record` element

This element is used to define a parameter that holds record-like data.

The `label` attribute can be used to specify the contained element that should be used to represent this record in lists.

```
<!ELEMENT %LRCSML.record;       (%common.mix;,(%types-spec;)+, %default;) >
<!ATTLIST %LRCSML.record;
 %common.attr;
        label    NMTOKEN                                    #IMPLIED
                                                                    >
```

`record` element

This element is used to define a parameter that holds record-like data.

The `label` attribute can be used to specify the contained element that should be used to represent this record in lists.

```
<!ELEMENT %LRCSML.record;       (%common.mix;,(%types-spec;)+,%default;) >
<!ATTLIST %LRCSML.record;
 %common.attr;
        label    NMTOKEN                                    #IMPLIED
                                                                    >
```

`reference` element

This element is used to select from an index. The index in which the available values is taken is specified in the `index` attribute.

```
<!ELEMENT %LRCSML.reference; (%common.mix.default;)                  >
<!ATTLIST %LRCSML.reference;
 %common.attr;
 index  CDATA                                              #REQUIRED
                                                                     >
```

`report-config` element

This element is used to configure a report configuration.

This element doesn't have any attribute. Each superservice can define a default report configuration using this element with a name of *`superservice`*`_default`.

```
<!ELEMENT %LRCSML.report-config; (%common.mix.default;)              >
<!ATTLIST %LRCSML.report-config;
 %common.attr;
                                                                     >
```

`plugin` element

This element is used to define a parameter for which the value is selected among a set of options. The allowed set of options is specified using `option` elements. The element will also contain additional parameters based on the selected value. The available paramaters should be defined in a `record` or similar specification named *`name`*`_properties`. For example, the additional parameters when the `option_1` option is selected will be found in the specification named `option_1_properties`.

This element doesn't have any specific attributes.

```
<!ELEMENT %LRCSML.plugin;     (%common.mix;,(%LRCSML.option;)+, %default;) >
<!ATTLIST %LRCSML.plugin;
 %common.attr;
                                                                     >
```

Chapter 9. The Lire Report Configuration Markup Language

The Lire Report Configuration Markup Language

Document Type Definition for the Lire Report Configuration Markup Language.

This DTD defines a grammar that is used to store the Lire configuration. The configuration is stored in one or more XML files. Parameters set in later configuration files override the ones set in the formers. The valid parameter names as well as their description and type are specified using configuration specification documents.

Elements of this DTD use the http://www.logreport.org/LRCML/ namespace, which is usually mapped to the `lrcml` prefix.

The latest version of the DTD is 1.0 and its public identifier is -//LogReport.ORG//DTD Lire Report Specification Markup Language V1.0//EN. Its canonical system identifier is http://www.logreport.org/LRCML/1.0/lrcml.dtd.

```
<!--
                                                                    -->

<!--                    Namespace prefix for validation using the
                        DTD                                         -->
<!ENTITY % LRCML.xmlns.pfx    "lrcml"                                  >
<!ENTITY % xmlns.colon       ":"                                       >
<!ENTITY % LRCML.pfx          "%LRCML.xmlns.pfx;%xmlns.colon;"         >
<!ENTITY % LRCML.xmlns.attr.name "xmlns%xmlns.colon;%LRCML.xmlns.pfx;" >
<!ENTITY % LRCML.xmlns.attr
  "%LRCML.xmlns.attr.name; CDATA #FIXED 'http://www.logreport.org/LRCML/'">

<!-- For the module which we are including                          -->
<!ENTITY % LIRE.pfx           "%LRCML.pfx;"                          >
```

Each configuration specification is an XML document which has one `config` as its root element.

```
<!ENTITY % LRCML.config          "%LRCML.pfx;config"                  >
<!ENTITY % LRCML.global          "%LRCML.pfx;global"                  >
<!ENTITY % LRCML.param           "%LRCML.pfx;param"                   >
```

`config` element

Root element of a configuration document. It contains presently only one `global` element which is used to hold the global configuration parameters.

This element doesn't have any attributes.

```
<!ELEMENT %LRCML.config; (%LRCML.global;)                                >
<!ATTLIST %LRCML.config;
             %LRCML.xmlns.attr;                                         >
```

`global` element

This element starts the global configuration data. (This is the only scope currently defined). It contains a list of `param` elements.

```
<!ELEMENT %LRCML.global; (%LRCML.param;)+                                >
```

`param` element

This element contains the parameter's value. The parameter's name is defined in the `name` attribute.

The `value` attribute can be used to store scalar's value.

When the parameter's type is a list, the values are stored in children `param` elements.

> **Warning**
>
> This element has a mixed content type. We should probably use a `value` attribute to hold scalar values.

```
<!ELEMENT %LRCML.param; (#PCDATA|%LRCML.param;)*                          >
<!ATTLIST %LRCML.param;
             name      NMTOKEN                              #REQUIRED
             value     CDATA                                #IMPLIED >
```

Chapter 10. The Lire DLF Schema Markup Language

The Lire DLF Schema Markup Language

The Lire DLD Schema Markup Language (LDSML) is used describe the fields used by DLF records of a specific schema like www, email or msgstore.

DLF schemas are defined in one XML document that should be installed in one of the directories that is included in the schema path (usually *HOME*`/.lire/schemas` and *prefix*`/share/lire/schemas`). This document must conforms to the LDSML DTD which is described here. Elements of that DTD are defined in the namespace http://www.logreport.org/LDSML/ which will be usually mapped to the lire prefix (altough other prefixes may be used).

The latest version of that DTD is 1.1 and its public identifier is -//LogReport.ORG//DTD Lire DLF Schema Markup Language V1.1//EN. Its canonical system identifier is http://www.logreport.org/LDSML/1.1/ldsml.dtd.

```
<!--                    Namespace prefix for validation using the
                        DTD                                        -->
<!ENTITY % LIRE.xmlns.pfx    "lire"                                  >
<!ENTITY % LIRE.pfx          "%LIRE.xmlns.pfx;:"                     >
<!ENTITY % LIRE.xmlns.attr.name "xmlns:%LIRE.xmlns.pfx;"             >
<!ENTITY % LIRE.xmlns.attr
  "%LIRE.xmlns.attr.name; CDATA #FIXED
                        'http://www.logreport.org/LDSML/'">
```

This DTD uses the common modules lire-types.mod which defines the data types recognized by Lire and lire-desc.mod which is used to include a subset of DocBook in description and text elements.

```
<!ENTITY % lire-types.mod PUBLIC
    "-//LogReport.ORG//ENTITIES Lire Data Types V1.0//EN"
    "lire-types.mod">
%lire-types.mod;

<!ENTITY % lire-desc.mod PUBLIC
    "-//LogReport.ORG//ELEMENTS Lire Description Elements V2.0//EN"
    "lire-desc.mod">
%lire-desc.mod;
```

The top-level element in XML documents describing a DLF schema will be either a `dlf-schema`, `extented-schema` or `derived-schema` depending on the schema's type. *DLF schemas* are used as base schema for one superservice. For example, the DLF schema of the www superservice is named www. An *extended schema* is used to define additional fields which values are to be computed by an *analyser*.

Extended schemas are named after the schema which they extend. For example, the www-attack extended schema adds an *`attack`* field which contains, if any, the "attack" that was attempted in that request.

Derived schemas are used by another type of analysers which defines an entirely different schema. Whereas in the extended schema the new fields will be added to all the DLF records of the base schema, the derived schema will create new DLF records based on the DLF records of the base schema. An example of this is the www-session schema which computes users' session information based on the web requests contained in the www schema. Like for the `extended-schema` case, derived schemas are named after the base schema from which they are derived.

The fields that makes each schema are defined using `field` elements.

```
<!-- Prefixed names declaration.                                    -->
<!ENTITY % LIRE.dlf-schema   "%LIRE.pfx;dlf-schema"                  >
<!ENTITY % LIRE.extended-schema "%LIRE.pfx;extended-schema"          >
<!ENTITY % LIRE.derived-schema  "%LIRE.pfx;derived-schema"           >
<!ENTITY % LIRE.field        "%LIRE.pfx;field"                       >
```

The `dlf-schema` element

The `dlf-schema` element is used to define the base schema of a superservice. It should contains optional `title` and `description` elements followed by `field` elements describing the schema structure.

The `title` is an optional text string that will be used to in the automatic documentation generation that can be extracted from the schema definition. The `description` element should describe what is represented by each DLF records (one web request, one email delivery, one firewall event, etc.)

`dlf-schema`'s attributes

superservice

> This required attribute contains the name of the superservice described by this schema. This will also be used as the base schema's identifier.

timestamp

> This required attribute contains the name of the field which contains the official event's timestamp. This field will be used to sort the DLF records for timegroup and timeslot report operations.

```
<!ELEMENT %LIRE.dlf-schema;  ( (%LIRE.title;)?, (%LIRE.description;)?,
                               (%LIRE.field;)+ )                     >
<!ATTLIST %LIRE.dlf-schema;
             superservice    %superservice.type;           #REQUIRED
             timestamp       IDREF                         #REQUIRED
             %LIRE.xmlns.attr;                                       >
```

`extended-schema` element

This is the root element of an extended DLF Schema. Extended-schema defines additional fields that will be added to the base schema. It contains an optional title, an optional description and one or more field specifications.

`dlf-schema`'s attributes

id

This required attribute contains the identifier of that schema. This identifier should be composed of the superservice's name followed by an hypen (`-`) and then an word describing the extended schema.

base-schema

This required attribute contains the identifier of the schema that is extended.

required-fields

This optional attribute contains a space delimited list of field names that must be available in the base schema for the analyser to do its job. If any of the listed field is missing in the DLF, extended fields for the base schema cannot be computed.

module

This required attribute contains the name of the analyser that is used to compute the extended fields. This is a perl module that should be installed in perl's library path.

```
<!ELEMENT %LIRE.extended-schema;
                            ( (%LIRE.title;)?, (%LIRE.description;)?,
                              (%LIRE.field;)+ )                    >
<!ATTLIST %LIRE.extended-schema;
             id              NMTOKEN                       #REQUIRED
             base-schema     NMTOKEN                       #REQUIRED
             module          NMTOKEN                       #REQUIRED
             required-fields NMTOKENS                      #IMPLIED
             %LIRE.xmlns.attr;                                      >
```

`derived-schema` element

This is the root element of a derived DLF Schema. The difference between a normal schema and a derived schema is that the data is generated from another DLF instead of a log file.

`derived-schema`'s attributes

id

This required attribute contains the identifier of that schema. This identifier should be composed of the superservice's name followed by an hypen (`-`) and then an word describing the derived schema.

base-schema

This required attribute contains the identifier of the schema from which this derived schema's data is derived.

required-fields

This optional attribute contains a space delimited list of field names that must be available in the base schema for the analyser to do its job. If any of the listed field is missing in the DLF, the derived records cannot be computed.

module

This required attribute contains the name of the analyser that is used to compute the derived records. This is a perl module that should be installed in perl's library path.

timestamp

This required attribute contains the name of the field which contains the official event's timestamp. This field will be used to sort the DLF records for timegroup and timeslot report operations.

```
<!ELEMENT %LIRE.derived-schema;
                            ( (%LIRE.title;)?, (%LIRE.description;)?,
                              (%LIRE.field;)+ )                    >
<!ATTLIST %LIRE.derived-schema;
             id              NMTOKEN                        #REQUIRED
             base-schema     NMTOKEN                        #REQUIRED
             module          NMTOKEN                        #REQUIRED
             required-fields NMTOKENS                       #IMPLIED
             timestamp       IDREF                          #REQUIRED
             %LIRE.xmlns.attr;                                        >
```

`field` element

The `field` is used to describe the fields of the schema. Each field is specified by its name and type. The field element may contain an optional `description` element which gives more information on the data contained in the field. Description should be used to give better information to the DLF converter implementors on what should appears in that field.

`field`'s attributes

name

This required attribute contains the name of the field.

type

This required attribute contains the the field's type.

default

> **Warning**
>
> This attribute is obsolete and will be removed in a future Lire release.

label

This optional attribute gives the label that should be used to display this field in reports. Defaults to the field's name when omitted.

```
<!ELEMENT %LIRE.field;       (%LIRE.description;)?                >
<!ATTLIST %LIRE.field;
             name            ID                          #REQUIRED
             type            (%lire.types;)              #REQUIRED
             default         CDATA                       #IMPLIED
             label           CDATA                       #IMPLIED  >
```

Chapter 11. The Lire Report Specification Markup Language

The Lire Report Specification Markup Language

Document Type Definition for the Lire Report Specification Markup Language.

This DTD defines a grammar that is used to specify reports that can be generated by Lire. Elements of this DTD uses the http://www.logreport.org/LRSML/ namespace that is usually mapped to the `lire` prefix.

The latest version of that DTD is 2.1 and its public identifier is -//LogReport.ORG//DTD Lire Report Specification Markup Language V2.1//EN. Its canonical system identifier is http://www.logreport.org/LRSML/2.1/lrsml.dtd.

```
<!--
                                                                    -->

<!--                    Namespace prefix for validation using the
                        DTD                                         -->
<!ENTITY % LIRE.xmlns.pfx    "lire"                                  >
<!ENTITY % LIRE.pfx          "%LIRE.xmlns.pfx;:"                     >
<!ENTITY % LIRE.xmlns.attr.name "xmlns:%LIRE.xmlns.pfx;"             >
<!ENTITY % LIRE.xmlns.attr
  "%LIRE.xmlns.attr.name; CDATA #FIXED 'http://www.logreport.org/LRSML/'">

<!ENTITY % LRCML.xmlns.pfx    "lrcml"                                 >
<!ENTITY % LRCML.pfx          "%LRCML.xmlns.pfx;:"                    >
<!ENTITY % LRCML.xmlns.attr.name "xmlns:%LRCML.xmlns.pfx;">
<!ENTITY % LRCML.xmlns.attr
  "%LRCML.xmlns.attr.name; CDATA #FIXED 'http://www.logreport.org/LRCML/'">
```

This DTD uses the common modules lire-types.mod which defines the data types recognized by Lire and lire-desc.mod which is used to include a subset of DocBook in description and text elements.

```
<!ENTITY % lire-types.mod PUBLIC
    "-//LogReport.ORG//ENTITIES Lire Data Types V1.0//EN"
    "lire-types.mod">
%lire-types.mod;

<!ENTITY % lire-desc.mod PUBLIC
    "-//LogReport.ORG//ELEMENTS Lire Description Elements V2.0//EN"
    "lire-desc.mod">
%lire-desc.mod;
```

Each report specification is a XML document which has one `report-spec` as its root element. This DTD can also be used for filter specification which have one `global-filter-spec` as root element.

```
<!ENTITY % LIRE.report-spec     "%LIRE.pfx;report-spec"              >
<!ENTITY % LIRE.global-filter-spec "%LIRE.pfx;global-filter-spec">
<!ENTITY % LIRE.display-spec    "%LIRE.pfx;display-spec"             >
<!ENTITY % LIRE.param-spec      "%LIRE.pfx;param-spec"               >
<!ENTITY % LIRE.param           "%LIRE.pfx;param"                    >
<!ENTITY % LIRE.chart-configs   "%LIRE.pfx;chart-configs"            >
<!ENTITY % LRCML.param          "%LRCML.pfx;param"                   >
<!ENTITY % LIRE.filter-spec     "%LIRE.pfx;filter-spec"              >
<!ENTITY % LIRE.report-calc-spec "%LIRE.pfx;report-calc-spec"        >

<!ELEMENT %LRCML.param; (#PCDATA|%LRCML.param;)*                     >
<!ATTLIST %LRCML.param;
             name      NMTOKEN                              #REQUIRED
             value     CDATA                                #IMPLIED  >
```

`report-spec` element

Root element of a report specification. It contains descriptive elements about the report specification (`title`, `description`). It contains the display elements that will be in the generated report (`display-spec`).

It contains specification for the parameters that can be used to customize the report generated from this specification (param-spec). Finally, it contains elements to specify a filter expression which can be used to select a subset of the records (`filter-spec`) and the expression to build the report (`report-calc-spec`).

`report-spec`'s attributes

id

the name of the superservice for which this report is available : i.e. email, www, dns, etc.

schema

The DLF schema used by the report. This defaults to the superservice's schema, but can be one of its derived or extended schema.

joined-schemas

A whitespace delimited list of additional schemas that will be joined for this report. This will make all fields define in these schemas available for the operators. The schemas that can be joined depends on the specification's schema.

id

An unique identifier for the report specification

```
<!ELEMENT %LIRE.report-spec;
                        (%LIRE.title;, %LIRE.description;,
                         (%LIRE.param-spec;)?, %LIRE.display-spec;,
                        (%LIRE.filter-spec;)?, (%LIRE.chart-configs;)?,
                        %LIRE.report-calc-spec;)
                                                                    >
<!ATTLIST %LIRE.report-spec;
             id             ID                                #REQUIRED
             superservice   %superservice.type;               #REQUIRED
```

```
              schema         NMTOKEN                          #IMPLIED
              joined-schemas NMTOKENS                         #IMPLIED
              %LIRE.xmlns.attr;
              %LRCML.xmlns.attr;                                        >
```

`global-filter-spec` element

Root element of a filter specification. It contains descriptive elements about the filter specification (`title`, `description`). It contains the display elements that will be used when that filter is used in a generated report (`display-spec`). It contains specification for the parameters that can be used to customize the filter generated from this specification (`param-spec`). Finally, it contains element to specify the filter expression which can be used to select a subset of the records (`filter-spec`).

`global-filter-spec`'s attributes

superservice

the name of the superservice for which this filter is available : i.e. email, www, dns, etc.

schema

the DLF schema used by the report. This defaults to the superservice's schema, but can be one of its derived or extended schema.

joined-schemas

A whitespace delimited list of additional schemas that will be joined for this report. This will make all fields define in these schemas available for the operators. The schemas that can be joined depends on the specification's schema.

id

An unique identifier for the filter specification

```
<!ELEMENT %LIRE.global-filter-spec;
                        (%LIRE.title;, %LIRE.description;,
                         (%LIRE.param-spec;)?, %LIRE.display-spec;,
                        (%LIRE.filter-spec;))
                                                                  >

<!ATTLIST %LIRE.global-filter-spec;
              id             ID                               #REQUIRED
              superservice   %superservice.type;             #REQUIRED
              schema         NMTOKEN                          #IMPLIED
              joined-schemas NMTOKENS                         #IMPLIED
              %LIRE.xmlns.attr;                                         >
```

`display-spec` element

This element contains the descriptive element that will appear in the generated report.

It contains one title and may contains one description which will be used as help message

This element has no attribute.

```
<!ELEMENT %LIRE.display-spec; (%LIRE.title;, (%LIRE.description;)?)  >
```

`param-spec` element

This element contains the parameters than can be customized in this report specification.

This element doesn't have any attribute.

```
<!ELEMENT %LIRE.param-spec; (%LIRE.param;)+                          >
```

`param` element

This element contains the specification for a parameter than can be used to customize this report.

This element can contains a `description` element which can be used to explain the parameter's purpose.

It is an error to define a parameter with the same name than one of the superservice's field.

`param`'s attributes

name

: the name of the parameter.

type

: the parameter's data type

default

: the parameter's default value

```
<!ELEMENT %LIRE.param; (%LIRE.description;)?                         >
<!ATTLIST  %LIRE.param;
             name       ID                                  #REQUIRED
             type       (%lire.types;)                      #REQUIRED
             default    CDATA                               #IMPLIED  >
```

`chart-configs` element

This element contains one or more chart configurations that should be copied to the generated subreport. These chart configurations are specified using the Lire Report Configuration Markup Language.

This element has no attribute.

```
<!ELEMENT %LIRE.chart-configs; (%LRCML.param;)+                    >
```

Filter expression elements

```
<!ENTITY % LIRE.eq      "%LIRE.pfx;eq"                            >
<!ENTITY % LIRE.ne      "%LIRE.pfx;ne"                            >
<!ENTITY % LIRE.gt      "%LIRE.pfx;gt"                            >
<!ENTITY % LIRE.ge      "%LIRE.pfx;ge"                            >
<!ENTITY % LIRE.lt      "%LIRE.pfx;lt"                            >
<!ENTITY % LIRE.le      "%LIRE.pfx;le"                            >
<!ENTITY % LIRE.and     "%LIRE.pfx;and"                           >
<!ENTITY % LIRE.or      "%LIRE.pfx;or"                            >
<!ENTITY % LIRE.not     "%LIRE.pfx;not"                           >
<!ENTITY % LIRE.match   "%LIRE.pfx;match"                         >
<!ENTITY % LIRE.value   "%LIRE.pfx;value"                         >

<!ENTITY % expr "%LIRE.eq; | %LIRE.ne; |
                 %LIRE.gt; | %LIRE.lt; | %LIRE.ge; | %LIRE.le; |
                 %LIRE.and; | %LIRE.or; | %LIRE.not; |
                 %LIRE.match; | %LIRE.value;"                     >
```

`filter-spec` element

This element is used to select the subset of the records that will be used to generate the report. If this element is missing, all records will be used to generate the report.

The content of this element are expression element which defines an expression which will evaluate to true or false for each record. The subset used for to generate the report are all records for which the expression evaluates to true.

The value used to evaluate the expressions are either literal, value of parameter or value of one of the field of the record. Parameter and field starts with a $ followed by the name of the parameter or field. All other values are interpreted as literals.

This element doesn’t have any attribute.

```
<!ELEMENT %LIRE.filter-spec; (%expr;)                             >
```

`value` element

This expression element to false if the 'value' attribute is undefined, the empty string or 0. It evaluate to true otherwise.

`value`'s attributes

value

The value that should be evaluated for a boolean context.

```
<!ELEMENT %LIRE.value;  EMPTY                                   >
<!ATTLIST %LIRE.value;
             value      CDATA                           #REQUIRED >
```

`eq` element

```
<!ELEMENT %LIRE.eq;     EMPTY                                   >
<!ATTLIST %LIRE.eq;
             arg1       CDATA                           #REQUIRED
             arg2       CDATA                           #REQUIRED >
```

`ne` element

```
<!ELEMENT %LIRE.ne;     EMPTY                                   >
<!ATTLIST %LIRE.ne;
             arg1       CDATA                           #REQUIRED
             arg2       CDATA                           #REQUIRED >
```

`gt` element

```
<!ELEMENT %LIRE.gt;     EMPTY                                   >
<!ATTLIST %LIRE.gt;
             arg1       CDATA                           #REQUIRED
             arg2       CDATA                           #REQUIRED >
```

ge element

```
<!ELEMENT %LIRE.ge;     EMPTY                                   >
<!ATTLIST %LIRE.ge;
             arg1       CDATA                           #REQUIRED
             arg2       CDATA                           #REQUIRED >
```

lt element

```
<!ELEMENT %LIRE.lt;     EMPTY                                   >
<!ATTLIST %LIRE.lt;
             arg1       CDATA                           #REQUIRED
             arg2       CDATA                           #REQUIRED >
```

le element

```
<!ELEMENT %LIRE.le;     EMPTY                                   >
<!ATTLIST %LIRE.le;
             arg1       CDATA                           #REQUIRED
             arg2       CDATA                           #REQUIRED >
```

match element

The match expression element tries to match a POSIX 1003.2 extended regular expression to a value and return true if there is a match and false otherwise.

match's attributes

value

: the value which should matched

re

: A POSIX 1003.2 extended regular expression.

case-sensitive

: Is the regex sensitive to case. Defaults to true.

```
<!ELEMENT %LIRE.match;  EMPTY                                   >
<!ATTLIST %LIRE.match;
             value          CDATA                           #REQUIRED
```

```
              re              CDATA                          #REQUIRED
              case-sensitive (%bool.type;)                   'true'    >
```

`not` element

```
<!ELEMENT %LIRE.not;    (%expr;)                                        >
```

`and` element

```
<!ELEMENT %LIRE.and;    (%expr;)+                                       >
```

`or` element

```
<!ELEMENT %LIRE.or;     (%expr;)+                                       >
```

Report Calculation Elements

```
<!ENTITY % LIRE.timegroup   "%LIRE.pfx;timegroup"                   >
<!ENTITY % LIRE.group       "%LIRE.pfx;group"                       >
<!ENTITY % LIRE.rangegroup  "%LIRE.pfx;rangegroup"                  >
<!ENTITY % LIRE.timeslot    "%LIRE.pfx;timeslot"                    >
<!ENTITY % LIRE.field       "%LIRE.pfx;field"                       >
<!ENTITY % LIRE.sum         "%LIRE.pfx;sum"                         >
<!ENTITY % LIRE.avg         "%LIRE.pfx;avg"                         >
<!ENTITY % LIRE.min         "%LIRE.pfx;min"                         >
<!ENTITY % LIRE.max         "%LIRE.pfx;max"                         >
<!ENTITY % LIRE.first       "%LIRE.pfx;first"                       >
<!ENTITY % LIRE.last        "%LIRE.pfx;last"                        >
<!ENTITY % LIRE.count       "%LIRE.pfx;count"                       >
<!ENTITY % LIRE.records     "%LIRE.pfx;records"                     >

<!-- Empty group operator                                          -->
<!ENTITY % LIRE.empty-ops   "%LIRE.sum; | %LIRE.avg; | %LIRE.count; |
                             %LIRE.min; | %LIRE.max; | %LIRE.first; |
     %LIRE.last; | %LIRE.records;"           >

<!-- Group operations that are also aggregators                    -->
```

```
<!ENTITY % LIRE.nestable-aggr
                            "%LIRE.group; | %LIRE.timegroup; |
                             %LIRE.timeslot; | %LIRE.rangegroup;"    >

<!-- Group operations                                              -->
<!ENTITY % LIRE.group-ops   "%LIRE.empty-ops;| %LIRE.nestable-aggr;" >

<!-- Containers for group operations                               -->
<!ENTITY % LIRE.aggregator  "%LIRE.nestable-aggr;"                   >
```

`report-calc-spec` element

This element describes the computation needs to generate the report.

It contains one aggregator element.

This element doesn't have any attributes.

```
<!ELEMENT %LIRE.report-calc-spec; (%LIRE.aggregator;)                >
```

Common Attributes

All elements which will create a column in the resulting report have a label attribute that will be used as the column label. When this attribute is omitted, the name attribute content will be used as column label.

```
<!ENTITY % label.attr "label CDATA #IMPLIED">
```

All operation elements may have a name attribute which can be used to reference that column. (It is required in the case of aggrage functions). The primary usage is for controlling the sort order of the rows in the generated report.

```
<!ENTITY % name.attr   "name ID      #IMPLIED">
<!ENTITY % name.attr.req  "name ID      #REQUIRED">
```

`group` element

The group element generates a report where records are grouped by some field values and aggregate statistics are computed on those group of records.

It contains the field that should be used for grouping and the statistics that should be computed.

The sort order in the report is controlled by the 'sort' attribute.

`group`'s attributes

name

An identifier that can be used to reference this operation from other elements. This name will most often be used in the parent's sort attribute. If omitted a default name will be generated.

sort

whitespace delimited list of fields name that should used to sort the records. Field names can be prefixed by `-` to specify reverse sort order, otherwise ascending sort order is used. The name can also refer to the name attribute of the statistics element.

limit

limit the number of records that will be in the generated report. It can be either a positive integer or the name of a user supplied param.

```
<!ELEMENT %LIRE.group;      ((%LIRE.field;)+, (%LIRE.group-ops;)+)   >
<!ATTLIST %LIRE.group;
     %name.attr;
             sort       NMTOKENS                          #IMPLIED
             limit      CDATA                             #IMPLIED  >
```

`timegroup` element

The timegroup element generates a report where records are grouped by time range (hour, day, etc.). Statistics are then computed on these records grouped by period.

`timegroup`'s attributes

name

An identifier that can be used to reference this operation from other elements. This name will most often be used in the parent's sort attribute. If omitted a default name will be generated.

label

Sets the column label that will be used for column generated by this element. If omitted a default label will be generated.

field

the name of the field which is used to group records. This should be a field which is of one of the time types (timestamp, date, time). It defaults to the default timestamp field if unspecified.

period

This is the timeperiod over which records should be grouped. Valid period looks like (hour, day, 1h, 30m, etc). It can also be the name of a user supplied param.

```
<!ELEMENT %LIRE.timegroup;  (%LIRE.group-ops;)+                      >
<!ATTLIST %LIRE.timegroup;
     %name.attr;
```

```
%label.attr;
        field      NMTOKEN                          #IMPLIED
        period     CDATA                           #REQUIRED  >
```

`timeslot` element

The `timeslot` element generates a report where records are grouped according to a cyclic unit of time. The duration unit used won't fall over to the next higher unit. For example, this means that using a unit of 1d will generate a report where the stats will be by day of the week, 8h will generate a report by third of day, etc. The statistics are then computed over the records in the same timeslot.

Example 11-1. timeslot with 1d unit

Using a specification like:

```
<lire:timeslot unit="1d">
  ...
</lire:timeslot>
```

would generate a report like:

Table 11-1. weekly overview

Sunday	...
Monday	...
Tuesday	...
...	...
Saturday	...

where data will be summed over all Sunday's, Monday's, ..., and Saturdays found in the log.

Example 11-2. timeslot with 2m unit

Specifying **`unit="2m"`** would generate a line for each two months, giving a yearly view.

`timeslot`'s attributes

name

An identifier that can be used to reference this operation from other elements. This name will most often be used in the parent's sort attribute. If omitted a default name will be generated.

label

Sets the column label that will be used for column generated by this element. If omitted a default label will be generated.

field

the name of the field which is used to group records. This should be a field which is of one of the time types (timestamp, date, time). It defaults to the default 'timestamp' field if unspecified.

unit

This is the cyclic unit of time in which units the records are aggregated. It can be any duration value. (hour, day, 1h, 30m, etc). It can also be the name of a user supplied param.

```
<!ELEMENT %LIRE.timeslot;  (%LIRE.group-ops;)+                      >
<!ATTLIST %LIRE.timeslot;
     %name.attr;
     %label.attr;
             field      NMTOKEN                          #IMPLIED
             unit       CDATA                           #REQUIRED  >
```

`rangroup` element

The `rangegroup` element generates a report where records are grouped into distinct class delimited by a range. This element can be used to aggregates continuous numeric values like duration or bytes. Statistics are then computed on these records grouped in range class.

`rangegroup`'s attributes

name

An identifier that can be used to reference this operation from other elements. This name will most often be used in the parent's sort attribute. If omitted a default name will be generated.

label

Sets the column label that will be used for column generated by this element. If omitted a default label will be generated.

field

the name of the field which is used to group records. This should be a field which is of a continuous numeric type (bytes, duration, int, number). Time types aggregation should use the timegroup element or timeslot.

range-start

The starting index of the first class. Defaults to 0. This won't be used a the lower limit of the class. It is only used to specify relatively at which values the classes delimitation start. For example, if the range-start is 1, and the range-size is 5, a class ranging -4 to 0 will be created if values are in that range. It can be supplied in any continuous unit (i.e 10k, 5m, etc.) This can also be the name of a user supplied param.

range-size

This is the size of class. It can be supplied in any continuous unit (i.e 10k, 5m, etc.) It can also be the name of a user supplied param.

min-value

All value lower then this boundary value will be considered to be equal to this value. If this parameter isn't set, the ranges won't be bounded on the left side.

max-value

All value greater then this boundary value will be considered to be equal to this value. If this parameter isn't set, the ranges won't be bounded on the right side.

size-scale

The rate at which the size scale from one class to another. If it is different then 1, this will create a logarithmic distribution. For example, setting this to 2, each successive class will be twice larger then the precedent : 0-9, 10-29, 30-69, etc.

```
<!ELEMENT %LIRE.rangegroup;  (%LIRE.group-ops;)+                    >
<!ATTLIST %LIRE.rangegroup;
     %name.attr;
     %label.attr;
             field          NMTOKEN                         #REQUIRED
             range-start    CDATA                           #IMPLIED
             range-size     CDATA                           #REQUIRED
             min-value      CDATA                           #IMPLIED
             max-value      CDATA                           #IMPLIED
             size-scale     CDATA                           #IMPLIED   >
```

`field` element

This element reference a DLF field which value will be displayed in a separate column in the resulting report. Its used to specify the grouping fields in the `group` element and to specify the fields to output in the `records` element.

`field`'s attribute

name

The name of the DLF field that will be used as key for grouping.

label

Sets the column label that will be used for column generated by this element. If omitted a default label will be generated.

```
<!ELEMENT %LIRE.field;  EMPTY                                      >
<!ATTLIST %LIRE.field;
             name   NMTOKEN     #REQUIRED
     %label.attr;                                        >
```

`sum` element

The `sum` element sums the value of a field in the group.

`sum`'s attributes

name

An identifier that can be used to reference this operation from other elements. This name will most often be used in the parent's sort attribute.

label

Sets the column label that will be used for column generated by this element. If omitted a default label will be generated.

field

the field that should be summed.

ratio

This attribute can be used to display the sum as a ratio of the group or table total. If the attribute is set to `group` the resulting value will be the ratio on the group's total sum. If the attribute is set to `table`, it will be expressed as a ratio of the total sum of the table. The defaults is `none` which will not convert the sum to a ratio.

weight

This optional attribute can be used to create a weighted sum. It should contain a numerical DLF field name. The content of that field will be used to multiply each field value before summing them.

```
<!ELEMENT %LIRE.sum;    EMPTY                                          >
<!ATTLIST %LIRE.sum;
             %name.attr.req;
     %label.attr;
     ratio (none | group |table)             'none'
             field      NMTOKEN                          #REQUIRED
     weight NMTOKEN    #IMPLIED >
```

`avg` element

The avg element calculate average of all value of a field in the group. The average will be computed either on the number of records if the by-field attribute is left empty, or by the number of different values that there are in the by-fields.

avg's attributes

name

An identifier that can be used to reference this operation from other elements. This name will most often be used in the parent's sort attribute.

label

Sets the column label that will be used for column generated by this element. If omitted a default label will be generated.

field

the field that should be averaged. If left unspecified the number of record will be counted.

by-fields

the fields that will be used to dermine the count over which the average is computed.

weight

This optional attribute can be used to create a weighted average. It should contain a numerical DLF field name. The content of that field will be used to multiply each field value before summing them. Its that weighted sum that will be used to calculate the average.

```
<!ELEMENT %LIRE.avg;    EMPTY                                          >
<!ATTLIST %LIRE.avg;
             %name.attr.req;
     %label.attr;
             field      NMTOKEN                          #IMPLIED
             by-fields  NMTOKENS                         #IMPLIED
     weight NMTOKEN    #IMPLIED >
```

max element

The max element calculates the maximum value for a field in all the group's records.

max's attributes

name

An identifier that can be used to reference this operation from other elements. This name will most often be used in the parent's sort attribute.

label

Sets the column label that will be used for column generated by this element. If omitted a default label will be generated.

field

the field for which the maximum value should found.

```
<!ELEMENT %LIRE.max;    EMPTY                                          >
```

```
<!ATTLIST %LIRE.max;
             %name.attr.req;
     %label.attr;
             field      NMTOKEN                              #REQUIRED >
```

`min` element

The `min` element calculates the minimum value for a field in all the group's records.

`min`'s attributes

name

An identifier that can be used to reference this operation from other elements. This name will most often be used in the parent's sort attribute.

label

Sets the column label that will be used for column generated by this element. If omitted a default label will be generated.

field

the field for which the minimum value should found.

```
<!ELEMENT %LIRE.min;    EMPTY                                        >
<!ATTLIST %LIRE.min;
             %name.attr.req;
     %label.attr;
             field      NMTOKEN                              #REQUIRED >
```

`first` element

The `first` element will display the value of the value of one field of the first DLF record within its group. The sort order is controlled through the sort attribute..

`first`'s attributes

name

An identifier that can be used to reference this operation from other elements. This name will most often be used in the parent's sort attribute.

label

Sets the column label that will be used for column generated by this element. If omitted a default label will be generated.

field

the DLF field which will be displayed.

sort

whitespace delimited list of fields name that should used to sort the records. Field names can be prefixed by `-` to specify reverse sort order, otherwise ascending sort order is used. If this attribute is omitted, the records will be sort in ascending order of the default timestamp field.

```
<!ELEMENT %LIRE.first;    EMPTY                                          >
<!ATTLIST %LIRE.first;
             %name.attr.req;
     %label.attr;
             field      NMTOKEN                          #REQUIRED
             sort       NMTOKENS                         #IMPLIED
     >
```

`last` element

The `last` element will display the value of the value of one field of the last DLF record within its group. The sort order is controlled through the sort attribute..

`last`'s attributes

name

An identifier that can be used to reference this operation from other elements. This name will most often be used in the parent's sort attribute.

label

Sets the column label that will be used for column generated by this element. If omitted a default label will be generated.

field

the DLF field which will be displayed.

sort

whitespace delimited list of fields name that should used to sort the records. Field names can be prefixed by `-` to specify reverse sort order, otherwise ascending sort order is used. If this attribute is omitted, the records will be sort in ascending order of the default timestamp field.

```
<!ELEMENT %LIRE.last;    EMPTY                                          >
<!ATTLIST %LIRE.last;
             %name.attr.req;
     %label.attr;
             field      NMTOKEN                          #REQUIRED
             sort       NMTOKENS                         #IMPLIED
     >
```

`count` element

The `count` element counts the number of records in the group if the fields attribute is left empty. Otherwise, it will count the number of different values in the fields specified.

`count`'s attributes

name

: An identifier that can be used to reference this operation from other elements. This name will most often be used in the parent's sort attribute.

label

: Sets the column label that will be used for column generated by this element. If omitted a default label will be generated.

fields

: Which fields to count. If unspecified all records in the group are counted. If not, only different fields' value will be counted.

ratio

: This attribute can be used to display the frequency as a ratio of the group or table total. If the attribute is set to `group` the resulting value will be the ratio on the group's total frequency. If the attribute is set to `table`, it will be expressed as a ratio of the total frequency of the table. The defaults is `none` which will not convert the frequency to a ratio.

```
<!ELEMENT %LIRE.count;  EMPTY                                        >
<!ATTLIST %LIRE.count;
             %name.attr.req;
             %label.attr;
     ratio (none | group |table)             'none'
             fields     NMTOKENS                          #IMPLIED  >
```

`records` element

The `records` element will put the content of selected fields in the report. This can be used in reports that shows events matching certain criteria. The fields that will be included in the report for each record is specified by the `field` element.

`records`'s attribute

fields

: whitespace delimited list of fields name that should included in the report.

```
<!ELEMENT %LIRE.records; EMPTY                                       >
<!ATTLIST %LIRE.records;
             fields      NMTOKENS                          #REQUIRED >
```

Chapter 12. The Lire Report Markup Language

The Report Markup Language

Document Type Definition for the XML Lire Report Markup Language as generated by **lr_dlf2xml**.

Elements of that DTD are defined in the namespace http://www.logreport.org/LRML/ which will be usually mapped to the lire prefix.

The latest version of that DTD is 2.1 and its public identifier is -//LogReport.ORG//DTD Report Markup Language V2.1//EN. Its canonical system identifier is http://www.logreport.org/LRML/2.1/lrml.dtd (http://www.logreport.org/LDSML/2.1/lrml.dtd).

```
<!--                    Namespace prefix for validation using the
                        DTD                                        -->
<!ENTITY % LIRE.xmlns.pfx    "lire"                                  >
<!ENTITY % LIRE.pfx          "%LIRE.xmlns.pfx;:"                     >
<!ENTITY % LIRE.xmlns.attr.name "xmlns:%LIRE.xmlns.pfx;"             >
<!ENTITY % LIRE.xmlns.attr
  "%LIRE.xmlns.attr.name; CDATA #FIXED 'http://www.logreport.org/LRML/'">

<!ENTITY % LRCML.xmlns.pfx    "lrcml"                                >
<!ENTITY % LRCML.pfx          "%LRCML.xmlns.pfx;:"                   >
<!ENTITY % LRCML.xmlns.attr.name "xmlns:%LRCML.xmlns.pfx;"           >
<!ENTITY % LRCML.xmlns.attr
  "%LRCML.xmlns.attr.name; CDATA #FIXED 'http://www.logreport.org/LRCML/'">
```

This DTD uses the common modules lire-types.mod which defines the data types recognized by Lire and lire-desc.mod which is used to include a subset of DocBook in description and text elements.

```
<!-- Include needed modules -->
<!ENTITY % lire-types.mod PUBLIC
    "-//LogReport.ORG//ENTITIES Lire Data Types V1.0//EN"
    "lire-types.mod">
%lire-types.mod;

<!ENTITY % lire-desc.mod PUBLIC
    "-//LogReport.ORG//ELEMENTS Lire Description Elements V3.0//EN"
    "lire-desc.mod">
%lire-desc.mod;
```

Each report is an XML document of which the top-level element is the `report` element. The report's data is contained in `subreport` elements (these hold the results of each report specification that was used to generate the report).

```
<!--                    Parameter entities which defines qualified
                        names of the elements                      -->
<!ENTITY % LIRE.report         "%LIRE.pfx;report"                    >
```

```
<!ENTITY % LIRE.section         "%LIRE.pfx;section"                  >
<!ENTITY % LIRE.subreport       "%LIRE.pfx;subreport"                >
<!ENTITY % LIRE.missing-subreport "%LIRE.pfx;missing-subreport"      >
<!ENTITY % LIRE.table           "%LIRE.pfx;table"                    >
<!ENTITY % LIRE.table-info "%LIRE.pfx;table-info"               >
<!ENTITY % LIRE.group-info "%LIRE.pfx;group-info"               >
<!ENTITY % LIRE.column-info "%LIRE.pfx;column-info"              >
<!ENTITY % LIRE.group-summary   "%LIRE.pfx;group-summary"            >
<!ENTITY % LIRE.entry           "%LIRE.pfx;entry"                    >
<!ENTITY % LIRE.group           "%LIRE.pfx;group"                    >
<!ENTITY % LIRE.name            "%LIRE.pfx;name"                     >
<!ENTITY % LIRE.value           "%LIRE.pfx;value"                    >
<!ENTITY % LIRE.date            "%LIRE.pfx;date"                     >
<!ENTITY % LIRE.timespan        "%LIRE.pfx;timespan"                 >
<!ENTITY % LIRE.chart-configs   "%LIRE.pfx;chart-configs"            >
<!ENTITY % LRCML.param          "%LRCML.pfx;param"                   >

<!ELEMENT %LRCML.param; (#PCDATA|%LRCML.param;)*                       >
<!ATTLIST %LRCML.param;
             name      NMTOKEN                              #REQUIRED
             value     CDATA                                #IMPLIED >
```

`report` element

A report starts with the report's meta-informations: title, timespan and description.

The report's actual data is contained in one or more subreports.

`report`'s attributes

version

The version of the DTD to which this report complies. New report should use the `2.1` value.

```
<!ELEMENT %LIRE.report;      ((%LIRE.title;)?, (%LIRE.date;)?,
                              (%LIRE.timespan;)?, (%LIRE.description;)?,
                              (%LIRE.section;)+)                     >
<!ATTLIST %LIRE.report;
             version         %number.type;                #REQUIRED
             %LIRE.xmlns.attr;
             %LRCML.xmlns.attr;                                      >
```

Meta-information elements

`date` element

The `date` element contains the date on which the report was generated.

The content of this element should be the timestamp in a format suitable for display.

’s attribute

time

The date in epoch time.

```
<!ELEMENT %LIRE.date;    (#PCDATA)                                    >
<!ATTLIST %LIRE.date;
             time        %number.type;                       #REQUIRED>
```

`timespan` element

The `timespan` element contains the starting and ending date which delimits the period of the report.

The content of this element should be formatted for display purpose. The starting and ending time of the timespan can be read in epoch time in the attributes. The `period` attribute contains the timespan period.

`timespan`’s attributes

period

Optional attribute which contains the period for which the report was generated.

start

The start time of the timespan in epoch time.

end

The end time of the timespan in epoch time.

```
<!ELEMENT %LIRE.timespan;    (#PCDATA)                                >
<!ATTLIST %LIRE.timespan;
          period        (hourly|daily|weekly|monthly|yearly) #IMPLIED
          start         %number.type;                     #REQUIRED
          end           %number.type;                     #REQUIRED   >
```

`section` element

The `section` element group common subreports together. The section’s description will usually contains informations about the filters that were applied in this section.

It contains a `title`, a `description` if some global filters were applied and the section’s subreports.

This element doesn’t have any attribute.

```
<!ELEMENT %LIRE.section;      ( %LIRE.title;, (%LIRE.description;)?,
       (%LIRE.subreport;|%LIRE.missing-subreport;)*) >
```

`subreport` element

The `subreport` element contains data for a certain report.

It can contains meta-information elements, it they are different from the one of the report.

Example of subreports for the email superservice are :

- Message delay by relay in seconds.
- Per hour traffic summary.
- Top 10 messages delivery.
- etc.

The data is contains in a `table` element.

If charts should be generated from the table's data, their configuration is contained in the `chart-configs` element.

`subreport`'s attributes

id

A unique identifier that can be used to link to this element.

superservice

the name of the superservice from which the report's data comes from : i.e. email, www, dns, etc.

type

This is the name of the report specification that was used to generated this subreport.

schemas

A space delimited list of the schemas used by this subreport.

```
<!ELEMENT %LIRE.subreport;    ( %LIRE.title;, (%LIRE.description;)?,
                                %LIRE.table;, (%LIRE.chart-configs;)?) >
<!ATTLIST %LIRE.subreport;
     id     ID     #REQUIRED
             superservice   %superservice.type;            #REQUIRED
             type           CDATA                          #REQUIRED
             schemas        NMTOKENS                       #REQUIRED >
```

`missing-subreport` element

`missing-subreport`'s attributes

id

A unique identifier that can be used to link to this element.

superservice

the name of the superservice from which the report's data comes from : i.e. email, www, dns, etc.

type

This is the name of the report specification that was used to generated this subreport.

schemas

A space delimited list of the schemas used by this subreport.

reason

The reason why this subreport is missing.

```
<!ELEMENT %LIRE.missing-subreport; (EMPTY)                          >
<!ATTLIST %LIRE.missing-subreport;
     id     ID     #IMPLIED
             superservice   %superservice.type;            #REQUIRED
             reason         CDATA                          #IMPLIED
             type           CDATA                          #REQUIRED
             schemas        NMTOKENS                       #REQUIRED >
```

`table` element

The `table` element contains the data of the subreport. It starts by a `table-info` element which contains information on the columns defined in the subreport. Following the table structure, there is a `group-summary` element which contains values computed over all the records.

A `table` element can contains the subreport data directly or the data can be subdivided into groups.

An example of a subreport which would contains directly the data would be "messages per to-domain, top-10". This would contains ten entries, one for each to-domain.

An example of a subreport which would contains data in group would be "deliveries to users, per to-domain, top 30, top 5 users". It would contain 30 groups (one per to-domain) and each group would contain 5 entries (one per user).

Group can be nested to arbitrary depth (but logic don't recommend to nest too much).

`table`'s attributes

show

the number of entry to display. By default all entries should be displayed.

```
<!ELEMENT %LIRE.table;        (%LIRE.table-info;, %LIRE.group-summary;,
       (%LIRE.entry;)*) >
<!ATTLIST %LIRE.table;
              show         %int.type;                        #IMPLIED >
```

`table-info` element

The `table-info` element contains information on the table structure. It contains one `column-info` element for each columns defined. It will also contains one `group-info` element for every grouping operation used in the report specification.

This element doesn't have any attribute.

```
<!ELEMENT %LIRE.table-info;   (%LIRE.column-info;|%LIRE.group-info;)+ >
```

`group-info` element

The `group-info` element play a similar role to the `table-info` element. Its used to group the columns defined by particular subgroup.

`group-info`'s attribute

name

: This attribute holds the name of the operation in the report specification which was responsible for the creation of this group data.

row-idx

: Specify the row index of the table header in which this group's categorical labels should be displayed.

```
<!ELEMENT %LIRE.group-info;   (%LIRE.column-info;|%LIRE.group-info;)+ >
<!ATTLIST %LIRE.group-info;
     name     NMTOKEN            #REQUIRED
     row-idx  %int.type;    #REQUIRED >
```

`column-info` element

The `column-info` element describes a column of the table. It holds information related to display purpose (label, class, col-start, col-end, col-width) as well as information needed to use the content of the column as input to other computation (type, name).

The col-start, col-end and col-width can be used to render the data in grid.

`column-info`'s attributes

name

This attribute contains the name of the operation in the report specification which was used to generata data in this column.

type

The Lire data type of this column.

class

This attribute can either be `categorical` or `numerical`. Categorical data is held in `name` element and numerical data is held in `value` element. Also, numerical column will have `column-summary` element associated to them.

label

This optional attribute contains the column's label. If omitted, the name attribute's content will be used.

col-start

The column number in which this column start. The first column being column 0.

col-end

The column number in which this column ends. The first column being column 0. Spans are used to cover "padding columns" to indent grouped entries under their parent entry.

col-width

The suggested column width (in characters) to use for this column.

max-chars

The maximum entry's length in that column (this includes the label).

avg-chars

The average entry's length in that column (this includes the label). This value is rounded up to the nearest integer.

```
<!ELEMENT %LIRE.column-info;        EMPTY                          >
<!ATTLIST %LIRE.column-info;
             name    NMTOKEN #REQUIRED
     class    (categorical|numerical) #REQUIRED
     type    (%lire.types;) #REQUIRED
             label         CDATA                        #IMPLIED
             col-start     %int.type;                   #REQUIRED
             col-end       %int.type;                   #REQUIRED
             col-width     %int.type;                   #IMPLIED
             max-chars     %int.type;                   #IMPLIED
             avg-chars     %int.type;                   #IMPLIED >
```

`group-summary` element

The `group-summary` contains one `value` element for all the columns that contains numerical data. These elements will contains the statistics computed over all the DLF records which were processed by the group or the subreport.

`group-summary`'s attribute

nrecords

The number of DLF records that were processed by this group or subreport.

missing-cases

This attribute contains the number of `LIRE_NOTAVAIL` values found when computing the statistic. This number represents the number of records which didn't have the required information to group the records appropriately. If ommited or equals to 0, it means that all records had all the required information.

row-idx

Specify the row index in the table at which the group's summary `value` should be displayed. If this is attribute is omitted, the summary values won't be displayed.

```
<!ELEMENT %LIRE.group-summary; (%LIRE.value;)*                      >
<!ATTLIST %LIRE.group-summary;
             nrecords         %int.type;                   #REQUIRED
             missing-cases    %int.type;                    #IMPLIED
     row-idx       %int.type;     #IMPLIED >
```

`group` element

The `group` element can be used to subdivide logically a report. It's used for aggregate reports like message per user per domain.

It contains a `group-summary` element which contains the group's values for the whole group followed by the entries that makes the group.

Groups can be nested more than once, but too much nesting augments information clutter and isn't useful for the user.

`group`'s attributes

id

A unique identifier that can be used to link to this element.

show

the number of entry to display. By default all entries should be displayed.

```
<!ELEMENT %LIRE.group;        (%LIRE.group-summary;, (%LIRE.entry;)*)>
<!ATTLIST %LIRE.group;
     id   ID     #IMPLIED
```

```
            show         %int.type;                          #IMPLIED >
```

`entry` element

The `entry` contains the data from the report. It is similar to a row in a table altough one entry may represents several rows when it includes nested groups.

The `name` elements contain categorical items of data like user name, email, browser type, url. Note that numeric ranges (like time period for example) are also considered categorical data items.

The `value` elements contain numericical data which are the result of a descriptive statistical operation: message count, bytes transferred, average delay, etc.

`entry`'s attribute

id

A unique identifier that can be used to link to this element.

row-idx

Specify the row index in the table at which this entry's `name` and `value` elements should be rendered. If this is attribute is omitted, the entry won't be displayed.

```
<!--
                                                                    -->
<!ELEMENT %LIRE.entry;        (%LIRE.name;,
(%LIRE.name;|%LIRE.value;|%LIRE.group;)+)>
<!ATTLIST %LIRE.entry;
     id   ID     #IMPLIED
     row-idx   %int.type;     #IMPLIED >
```

`name` element

The `name` elements contains categorical data column value. Its also used for numerical values that represents a class of values (like produced by the `rangegroup` or timegroup operations for example.)

`name`'s attributes

id

A unique identifier that can be used to link to this element.

col

The column's name. It should be the same than the one in the corresponding `column-info` element.

value

When the displayed format is different from the DLF representation, this attribute contains the DLF representation.

range

In some cases (like in report generated by the `timegroup`, `timeslot` or rangegroup specification), this attribute will contains the range's length from the starting value which is in the 'value' attribute.

```
<!ELEMENT %LIRE.name;         (#PCDATA)                              >
<!ATTLIST %LIRE.name;
     id       ID     #IMPLIED
     col       NMTOKEN     #REQUIRED
             value            CDATA                          #IMPLIED
             range            %number.type;                  #IMPLIED >
```

`value` element

The value element contains numerical column value..

`value`'s attributes

id

A unique identifier that can be used to link to this element.

col

The column's name. It should be the same than the one in the corresponding `column-info` element.

value

contains the value in numeric format. This is used when the value was scaled (1k, 5M, etc.)

total

for average value, this contains the total used to compute the average.

n

for average value, this contains the n value that was used to compute the average.

missing-cases

This attribute contains the number of `LIRE_NOTAVAIL` values found when computing the statistic. When omitted, its assume to have a value of 0, i.e. that the value was defined in each DLF record.

```
<!ELEMENT %LIRE.value;        (#PCDATA)                              >
<!ATTLIST %LIRE.value;
     id          ID     #IMPLIED
     col  NMTOKEN          #REQUIRED
     missing-cases %int.type;     #IMPLIED
             value       %number.type;                  #IMPLIED
             total       %number.type;                  #IMPLIED
```

```
n           %number.type;                    #IMPLIED >
```

`chart-configs` element

This element contains one or more chart configurations that should be generated from the table's. These chart configurations are specified using the Lire Report Configuration Markup Language.

This element has no attribute.

```
<!ELEMENT %LIRE.chart-configs; (%LRCML.param;)+                    >
```

IV. Lire Developers’ Conventions

Chapter 13. Contributing Code to Lire

The LogReport team invites you to contribute code to Lire. We're very happy with any code contributions which work for you: it'll very likely will make life easier for other people too! We ask you to consider some points, when writing code to get distributed with Lire.

When adding new scripts, or extending and improving current Lire code, make sure you're working with the current Lire code. (When working with old code, the bug you're working on might be fixed already by somebody else.) You can get the current code by fetching our CVS from SourceForge, using the anonymously accessible pserver:

```
cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/logreport login
```

When prompted for a password for anonymous, simply press the Enter key.

```
cvs -z3 -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/logreport co service
```

See also the instructions on the SourceForge website (http://sourceforge.net/cvs/?group_id=5049). Alternatively, you can peek at the Lire CVS (http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/logreport/) using your webbrowser.

When you'd like to change e.g. `/usr/local/bin/lr_log2report`, you'll have to hack on `cvs/sourceforge/logreport/service/all/script/lr_log2report.in`. This file will get converted to `lr_log2report` by running `./configure`. Of course, when adding scripts or extending scripts, be sure to update the scripts' manpage too.

If you'd like the LogReport team to distribute your contribution, be sure to offer it to the team under a suitable software license. Refer to the Licensing section in the *Lire FAQ (http://logreport.org/lire/faq.php)* for details.

Once you've tested your script, you can send it too the LogReport development list on development@logreport.org. The LogReport team will be happy to ship your contribution with the next Lire release.

Chapter 14. Developers' Toolbox

Required Tools To Build From CVS

In order to be able to build the program from the CVS tree and make a tarball distribution the following tools are needed:

- DocBook XML 4.1.2 (http://www.oasis-open.org/docbook/)
- DocBook DSSSL stylesheets (http://docbook.sourceforge.net/projects/dsssl/)
- autotools
- Jade (http://www.jclark.com/jade/) or OpenJade
- lynx (http://lynx.isc.org/)
- GNU make
- Perl's XML::Parser module
- dia
- epsffit
- epstopdf
- xsltproc
- xmllint

For Debian woody the packages are: docbook-utils (http://packages.debian.org/testing/text/docbook-utils.html), docbook-xml-stylesheets, autoconf (http://packages.debian.org/testing/devel/autoconf.html), automake1.4 (http://packages.debian.org/testing/devel/automake.html), autotools-dev (http://packages.debian.org/testing/devel/autotools-dev.html), jade (http://packages.debian.org/testing/text/jade.html), lynx (http://packages.debian.org/testing/web/lynx.html), make (http://packages.debian.org/testing/devel/make.html) and libxml-parser-perl.

You need automake version 1.4. Building using automake 1.7 will very likely not work.

Accessing Lire's CVS

Make sure you've got an account on *SourceForge* (http://www.sourceforge.net). Get yourself added to the logreport project. (Joost van Baal joostvb@logreport.org can do this for you.) Make sure your ssh public key is on the sourceforge server.

A full backup of the complete LogReport CVS as hosted on SourceForge is made weekly and written to `hibou:/data/backup/cvs/`.

CVS primer

If you have a Unix like system, make sure you have this

```
CVSROOT=:ext:cvs.sourceforge.net:/cvsroot/logreport
```

```
CVS_RSH=ssh
```

in your shell environment.

Of course, you could do something like

```
$ eval ‘ssh-agent‘
$ ssh-add
```

to get a nice ssh-agent running.

Now do something like

```
$ cd ~/cvs-sourceforge/logreport
$ cvs co service
```

There are also repositories called ’docs’ and ’package’. In the former the webpages are located and in the latter the package files for Debian GNU/Linux and other distributions are kept.

Files can then be edited and commited:

```
$ vi somefile
$ cvs commit somefile
```

and get flamed ;)

Subscribe yourself to the commit list (commit-request@logreport.org), to get all commit messages, along with unified diffs.

SourceForge

Mailing Lists

Chapter 15. Coding Standards

Indentation should be four spaces. No tabs please.

See also Message-Id: <1028238571.1085.185.camel@Arendt.Contre.COM> on the development mailing list for some rationale on coding standards.

Shell Coding Standards

Shell scripts should run -e. Shell script should be portable. Refer to
http://doc.mdcc.cx/doc/autobook/html/autobook_208.html
(http://doc.mdcc.cx/doc/autobook/html/autobook_208.html).

Perl Coding Standards

Perl scripts should use strict, and run -w. Documentation should come in .pod format, documentation about script internals should be in perl comments.

No & in function call unless necessary.

Split long lines using hard return; try to respect the 72th column margin (this is kind of a soft limit).

Refer to the Lire::Program manpage for more details.

Chapter 16. Making Lire “Test-infected”

Soon after the release of Lire 1.2.1, unit tests were introduced in the source tree. Unit tests help development in several ways; the most important one being that you can make changes to code and run the unit tests to make sure that nothing was broken by that changes.

You can find helpful resources on Unit testing on the PerlUnit home page (http://perlunit.sourceforge.net/) as well as on the JUnit home page (http://perlunit.sourceforge.net/) from which it was inspired.

Unit Tests in Lire

PerlUnit

Unit tests are written using the PerlUnit framework. You need to install version 0.24 or later of the Test::Unit to run the unit tests.

Writing Tests

General information on using the PerlUnit framework can be found in the Test::Unit man page. Information on writing individual test cases can be found in the Test::Unit::TestCase man page.

Tests for individual modules should be defined in tests::*module*Test package. You can omit the Lire:: prefix and you can inline intermediary package names. For example, the unit tests of the Lire::ExtendedDlfSchema module are in the tests::ExtendedDlfSchemaTest package and the tests of the Lire::Timegroup module are in the tests::TimegroupTest package.

The Lire::Tests namespace is reserved for extensions to the PerlUnit framework that will be used to provide “fixtures” and “assertions” that are of general use for common Lire extensions.

> **Note:** This section will be expanded as common patterns for writing unit test for DLF converters, analyzers and other common Lire extension are developped.

Running Tests

To run tests, you use the **TestRunner.pl** script included with the PerlUnit distribution. You’ll need to add the directory containing the Lire libraries to perl library path. For example, if you have **TestRunner.pl** in your `~/bin` directory, you can run a test case from the top level source directory like this:

```
$ perl -Iall/lib ~/bin/TestRunner.pl tests::ExtendedDlfSchemaTest
```

tests::ExtendedDlfSchemaTest can be replaced by your TestCase module.

Some “Best Practices” on Unit Testing

This section lists some tips on how to make effective use of Unit tests in common development situations on Lire.

Changing interface/implementation. Before changing a module interface or implementation, make sure that this module has test cases and that it passes its tests before changing the implementation. This way you can know that your changes didn’t break anything.

Debugging. A good opportunity for writing tests is when bugs are reported. Before trying to chase the bug using the debugger or adding `print` statements, write a test case that will fail as long as the bug isn’t fixed. This achieves two purpose: first, you’ll know when the bug is fixed as soon as the test pass; secondly, we now have a test case that will warn us if we regress and the bug reappears.

Chapter 17. Commit Policy

Make sure your changes run on your own platform before committing. Try not to break things for other platforms though. Currently, Lire supported platforms are GNU/Linux (Debian GNU/Linux, Red Hat Linux, Mandrake Linux), FreeBSD, OpenBSD and Solaris.

Documentation should be updated ASAP, in case it's obsolete or incomplete by new commits.

CVS Branches

When doing major architectural changes to Lire, branches in CVS are created to make it possible to continue to fix bugs and to add small enhancements to the stable version while development continues on the unstable version. This applies mainly to the service repository. The doc and package repositories generally don't need branching.

BTW: A nice CVS tutorial is available in the Debian cvsbook package.

Hands-on example

A branching gets announced. Be sure to have all your pending changes commited before the branching occurs. After a branch has been made, one can do this:

```
$ cd ~/cvs-sourceforge/logreport
$ mv service service-HEAD
$ cvs co -r lire-20010924 service
$ mv service service-lire-20010924
```

or (with the same result)

```
$ mv service service-HEAD
$ cvs co -r lire-20010924 -d service-lire-20010924 service
```

Now, when working on stuff which should be shipped in the coming release, one should work in service-lire-20010924. When working on stuff which is rather fancy and experimental, and which needs a lot of work to get stabilized, one should work in service-HEAD.

Naming, what it looks like

Here is what branches schematically look like:

```
release-20010629_1 ---> lire-unstable-20010703 ---> HEAD
      \
       \
   lire-20010630 ---> lire-stable-20010701
```

In this diagram a branch named `lire-20010630` was created from the `release-20010629_1` tag. `lire-unstable-20010703` is another tag on the *trunk* (the *trunk* is the main branch). `HEAD` isn't a real tag, it always points to latest version on the trunk.

Creating a Branch

To create a branch, one runs the command **`cvs rtag -b -r release-tag branch-name module`**. Note that this command doesn't need a checkout version of the repository. For example, to create the `release-20010629_1-bugfixes` branch in the service repository, e.g. to backport bugfixes to version 20010629_1, one would use **`cvs rtag -b -r release-20010629_1 release-20010629_1-bugfixes service`**. When ready for release, this could get tagged as `release-20010629_2`.

The *`release-tag`* should exist before creating the branch. In case you want to branch from HEAD, use **`-r HEAD`**. E.g. **`cvs rtag -b -r HEAD release_1_1-branch service`**. Once Lire 1.1 gets released, tag it as `release_1_1`.

Accessing a Branch

To start working on a particular branch, you do **`cvs update -r branch-name`**. For example, to work on the `release_1_1-branch` branch, you do in your checked out version, **`cvs update -r release_1_1-branch`**. This will update your copy to the version `release_1_1-branch` and will commit all future changes on that branch.

Alternatively, you can also specify a branch when checking out a module using **`cvs co -r branch-name module`**. For example, you could checkout the stable version of Lire by using **`cvs co -r release_1_1-branch service`**.

To see if you are working on a particular branch, you can use the **`cvs status file`** command. For example, running **`cvs status NEWS`** could show:

```
===================================================================
File: NEWS              Status: Up-to-date

   Working revision: 1.74
   Repository revision: 1.74 /cvsroot/logreport/service/NEWS,v
   Sticky Tag: lire-stable
   Sticky Date: (none)
   Sticky Options: (none)
```

The branch is indicated by the `Sticky Tag:` keyword. If its value is `(none)` you are working on the `HEAD` branch.

To work on the `HEAD`, you remove the sticky tag by using the command **`cvs update -A`**.

Merging Branches on the Trunk

You can bring bug fixes and small enhancements that were made on a branch into the unstable version on the trunk by doing a merge. You do a merge by using the command **`cvs update -j branch-to-merge`** in

your working directory of the trunk. Conflicts are resolved in the usual CVS way. For example, to merge the changes of the stable branch in the development branch, you would use **`cvs update -j lire-stable`**.

You should tag the branch after each successful merge so that future changes can be easily merged. For example, after merging, you do in a checked out copy of the `lire-stable` branch: **`cvs tag lire-stable-merged-20010715`**. In this way, one week later we can merge the week's changes of the stable branch into the unstable branch by doing **`cvs update -j lire-stable-merged-20010715 -j lire-stable`**.

Chapter 18. Testing

One week before release the software should be tested on all supported platforms. In between releases the system gets tested on various platforms on an ad hoc basis. When testing, use the to-be-released tarball. Run `make dist` to generate such a tarball.

Especially when changes to the Lire core have been made, the "test" superservice can be handy, for easy setting up of tests of your code. See also the section on Unit Testing in this document.

Chapter 19. Making a Release

Before making an official Lire release, it should have been tested on all supported platforms. A release shouldn't be made unless Lire builds, installs and generates an ASCII report from all supported log files on all supported platforms. If this is not the case, the release should be delayed untill this is fixed.

Making a new release of Lire involves many steps:

1. Writing the final version number in NEWS.
2. Tagging the CVS tree.
3. Building the "Standard" Lire tarball.
4. Building the Debian GNU/Linux package.
5. Building the RPM package.
6. Making sure the FreeBSD package gets updated.
7. Uploading the tarballs and making packages available.
8. Advertising the release.

Setting version in NEWS file, checking ChangeLog

Inbetween releases, the NEWS file generally reads "version in cvs". This should of course be changed to e.g. "version 20011205".

We maintain a ChangeLog file. Make sure the ChangeLog in the toplevel directory is not too big. If needed, split off a chunk and move it to doc/. The ChangeLog is autogenerated from the CVS commits, using the **cvs2cl** tool. One could e.g. run **`cvs2cl --prune --stdout -l "-d \>yesterday" -U ../CVSROOT/users`**.

Tagging the CVS

Run e.g. **`cvs tag release-20011017`**.

Building The Tarball

1. Start from a fresh copy by running the command **`make maintainer-clean-recursive`** in the directory where you checked out Lire's source code.
    a. Make sure that there are no tarballs in the `extras` subdirectory.
2. Set the version and prepare the source tree by running the command **`./bootstrap`**. (You can overwrite the pre-cooked version by doing e.g. **`echo 'date +%Y%m%d'-R-f-jvb-1 > VERSION`** . Make sure your version hasn't got too many characters. Non-GNU tar chokes if pathnames in the archive are too long.)
3. Generate Makefiles
    a. Run **`./configure`**

4. Build Lire and create the tarball by running the command **`make distcheck`**.

   This will build a tarball `lire-`*`version`*`.tar.gz` and then make sure that the content of this tarball can be built and installed. If that command fails, Lire isn't ready to be released. Fix the errors before making the release.

5. Sign Lire's tarball with your public key. To do this with GnuPG, run **`gpg --detach-sign --armor lire-`*`version`*`.tar.gz`**.

   A file `lire-`*`version`*`.tar.gz.asc` will be created. Publish this file together with the tarball. Now, people downloading the tarball can verify its integrity by downloading the .asc as well as your public key, and running **`gpg --verify lire-`*`version`*`.tar.gz.asc`** .

Building The Debian Package

This is a raw unformatted dump of what we did to build and upload the Lire .deb.

```
              $ cd ~/cvs-sourceforge/logreport/package/debian
              $ vi changelog

:r !date --rfc

              $ cd /usr/local/src/debian/lire/debian/20010219
```

Run something like 'DIB_V=20020214 DIB_P=lire DIB_TARDIR=../archive/ ./debian-install-build'. This does:

```
              $ cd /usr/local/src/debian/lire/debian/20010219
              $ cp \
  ~/cvs-sourceforge/logreport/service/lire-20010219.tar.gz .

              $ tar zxf lire-20010219.tar.gz
              $ cd lire/20010418
              $ mv lire-20010418 lire-20010418.orig
              $ tar zxf lire-20010418.tar.gz
              $ cd lire-20010418
              $ mkdir debian
              $ cp \
   ~/cvs-sourceforge/logreport/package/debian/[^C]* debian/
```

Export the shell environment variable EMAIL, it should hold your email address, as it is to appear in the maintainers field of the package. (One could use 'dh_make --copyright gpl -s' on first time debianizing.) Build the .deb by running:

```
$ debuild 2>&1 | tee /tmp/build
```

Check the .deb:

```
$ debc | less
```

You might also want to test wether the Debianized sources build fine on other machines: copy diff.gz, orig.tar.gz and .dsc. Then do

```
$ dpkg-source -x lire_*.dsc
$ cd lire-version
$ dpkg-buildpackage -rfakeroot
```

After having *really* tested it (dpkg -i, purge, etc.), optionally install it on any local apt-able websites you might have (Joost has one on http://mdcc.cx/debian/) and upload it to hibou's apt-able archive:

```
$ scp lire_20010418-1_all.deb \
 hibou.logreport.org:/var/www/logreport.org/pub/debian/dists/local/contrib/binary-all/admin/

$ scp lire_20010418*.gz \
 hibou.logreport.org:/var/www/logreport.org/pub/debian/dists/local/contrib/source/admin/
$ scp lire_20010418*.*s* \
 hibou.logreport.org:/var/www/logreport.org/pub/debian/dists/local/contrib/source/admin/
```

Move the old debian stuff on hibou to hibou:/pub/archive/debian/ . Update the Packages file by running

```
$ cd /var/www/logreport.org/pub/debian
$ make
```

To upload it to the official debian mirrors:

```
vanbaal@gelfand:/usr...src/debian/lire/20010418% date; \
  dupload lire_20010418-1_i386.changes
Thu Apr 19 14:27:38 CEST 2001
Uploading (ftp) to ftp.uk.debian.org:debian/UploadQueue/
[ job lire_20010418-1_i386 from lire_20010418-1_i386.changes New dpkg-dev, announcement will NOT
 lire_20010418.orig.tar.gz, md5sum ok
 lire_20010418-1.diff.gz, md5sum ok
 lire_20010418-1_all.deb, md5sum ok
 lire_20010418-1.dsc, md5sum ok
 lire_20010418-1_i386.changes ok ]
Uploading (ftp) to uk (ftp.uk.debian.org)
 lire_20010418.orig.tar.gz 163.1 kB , ok (12 s, 13.59 kB/s)
 lire_20010418-1.diff.gz 32.6 kB , ok (3 s, 10.88 kB/s)
 lire_20010418-1_all.deb 222.4 kB , ok (16 s, 13.90 kB/s)
 lire_20010418-1.dsc 0.6 kB , ok (0 s, 0.60 kB/s)
 lire_20010418-1_i386.changes 1.2 kB , ok (1 s, 1.22 kB/s) ]
```

check ftp://ftp.uk.debian.org/debian/UploadQueue/

Building The RPM Package

Making sure the FreeBSD port gets updated

Since August 21, 2002, Lire is in the FreeBSD ports collection. Edwin Groothuis has build a FreeBSD port. Ask him if he's available for updating his port. Alternatively, Cédric Gross might be able to help. If not, the LogReport team should take care of it, and submit a Problem Report to the FreeBSD system, asking for inclusion of the updated port.

Uploading The Release

To release a new distribution, publish the tarball on various places and send an announcement to the <announcement@logreport.org> mailinglist, stating the most interesting new features. Furthermore, add a newsitem to the news list of the website. We'll describe how to upload the tarball to various places.

The LogReport Webserver

Upload the tarball to the pub area on the LogReport server. The area is mirrored automagically by the download.logreport.org servers; updates are done every 6 hours. Upload like this:

```
$ scp lire-20001211.tar.gz hibou.logreport.org:/var/www/logreport.org/pub/
```

On hibou, do:

```
               $ cd /var/www/logreport.org/pub
$ chown .www lire-20010525.tar.gz
$ chmod g+w lire-20010525.tar.gz

$ tar zxf lire-20001211.tar.gz
$ rm current && ln -s lire-20001211 current
$ rm current.tar.gz && ln -s lire-20001211.tar.gz current.tar.gz
$ rm -rf lire-20001205
$ mv lire-20001205.tar.gz archive
```

Update the `README.txt` file: Run

```
$ cd /var/www/logreport.org/pub
$ ( echo \
  'current is the latest official release'; echo; ls -lF c* ) > README.txt
```

Check the symlink to the documentation stuff in the tarball.

Check if the stuff in http://logreport.org/pub/docs is still up to date.

Advertising The Release

SourceForge

In order to release a distribution on SourceForge (SF), you login with your SF account on the SF website. Once logged in you go to the project webpage (https://sourceforge.net/projects/logreport/) and choose *Admin*. Down at the bottom of that page is a a *[Edit/Add File Releases]* link (click it (https://sourceforge.net/project/admin/editpackages.php?group_id=5049)).

You are able to edit packages, like the Lire package in the LogReport project. To add a new release, choose *[Add Release]*. As a release name uses the date, like 20010407, assign it to the Lire package and then use the *Create This Release* button to makes it effective.

The next page shows 4 steps of which only one (step 2) is not straightforward. In that step you assign files to a release (.tar.gz, .deb, .rpm). These files should be uploaded to SF's Upload anonymous FTP site at ftp://upload.sourceforge.net/incoming/. Make sure the file is placed in the `/incoming` directory. Click *Refresh View* in Step 2 to add the files you uploaded to the FTP site. Check the files belonging to the release and Click *Add Files*. In step 3, set Processor to any. Set file type to .deb and source.gz. Click update/refresh. Step 4: send notice. Done.

Freshmeat.net

On Freshmeat.net, releases are not released, but get announced only. These announcements attract a lot of attention. The webpage for the Lire package can be found at http://freshmeat.net/projects/lire/.

To announce a new release go to Lire - development branch (http://freshmeat.net/branches/14593/) webpage. Choose *Add Release* from the Project pull down menu in the light blue area. The rest is very straightforward.

Chapter 20. Website Maintenance

We give hints on how to upgrade the website: installing stuff from current CVS on http://logreport.org (http://logreport.org/).

Commits to the CVS tree of the website are automatically propagated to hibou. For more information on the markup language of the website, see the WJML documentation (http://logreport.org/doc/wjml/).

Documentation on the LogReport Website

Be sure the links to stuff under `/pub/current` are still alive. E.g. the files `TODO`, `dev-manual.html` and `user-manual.html` are linked to.

Publishing the DTD's

The DTD's are published as HTML on the website by using `hibou:/usr/local/src/dtdparse/dtdparse-2.0b2-LogReportPatched.tar.gz`, which is a patched version of Norman Walsh's dtdparse utility. Before the utility is run, make sure that the DocBook DTD is not included in the parsing process, because the DocBook DTD should not be published. This is done by changing the line:

```
<!ENTITY % load.docbookx     "INCLUDE"                              >
```

into:

```
<!ENTITY % load.docbookx     "IGNORE"                               >
```

The webpages are then generated with:

```
perl ~/dtdparse-2.0b2-patched/dtdparse.pl --title "XML Lire Report Markup Language" --output lire
perl ~/dtdparse-2.0b2-patched/dtdformat.pl --html lire.xml
```

The resulting `lire` directory can be tar-ed, gziped and unpacked again on hibou in the directory `/var/www/logreport.org/pub/docs/dtd/`.

The other two DTD's are HTML-ized similarly, but remember to change the title when running **dtdparse.pl**.

Chapter 21. Writing Documentation

Documentation which comes with the Lire tarball is maintained in four formats: plain text, Perl POD, DocBook XML and UML diagrams. We'll talk about all four of these here.

Plain Text

Small files like `README`, `NEWS`, `AUTHORS`, `doc/BUGS`, and `doc/TODO` are traditionally maintained in plain text format. We adhere to this common practice.

Perl's Plain Old Documentation: maintaining manpages

We use Perl's pod (plain old documentation) for manpages. Every file installed with Lire in `/usr/bin/` must have a manpage. Every file installed in `/usr/share/perl5/Lire/` and `/usr/lib/lire/` should have a manpage. It would be nice if the files in `/etc/lire/` were documented in manpages too. And perhaps for some files in `/usr/share/lire/xml/`, `/usr/share/lire/reports/`, `/usr/share/lire/filters/` and `/usr/share/lire/schemas/` manpages could be useful.

Since the files in `/usr/bin/` are commands, ran by Lire users, the manpages describing these should focus on the user perspective. Describing the inner workings and implementations of the commands is less important than describing why someone would want to run the specific command. If there's need to make some remarks on the internals of these scripts, a section called DEVELOPERS could be added to the manpage. The perl modules installed in `/usr/share/perl5/Lire/` and the commands in `/usr/lib/lire/` are not intended as interfaces for the user. Only people wanting to change or study the operation of Lire itself will interact with these files; therefore, the manpages should explain the inner workings and implementations of these files. The configuration files in `/etc/lire/` might be changed by users. These should be properly documented: in manpages or in the *Lire User's Manual*.

Docbook XML: Reference Books and Extensive User Manuals

The main documentation of the Lire project is done in DocBook XML 4.1.2. E.g. this document is maintained in DocBook XML, as is the *Lire User's Manual*. The *Lire User's Manual* has more information about DocBook.

After editing the *Lire Developer's Manual* or the *Lire User's Manual*, you should run **`make check-xml`** to make sure the document is still a valid DocBook document. You should fix any errors before committing your changes.

If everything went right, documentation is built in txt, tex, html and pdf format by running **`make dist`**, or just **`make`** in `doc/`. We give some hints which might be helpful in case you have to build the documentation manually.

To generate PDF:

```
$ jade -t tex -d /path/to/DSSSL/docbook/print/docbook.dsl roadmap.xml
$ pdfjadetex roadmap.tex
```

The last step is actually done two or three times to resolve page numbers.

To generate HTML:

```
$ jade -t sgml -d html.dsl roadmap.xml
```

And now you can use the `html.dsl` in the `doc/source` directory. (If necessary, adjust it to reflect the location of your DSSSL stylesheets). Use lynx to generate TXT output from HTML with:

```
$ lynx -nolist -dump roadmap.html > roadmap.txt
```

V. Implementation Details

Chapter 22. Adding a New Superservice in Lire’s Distribution

Integrating a new superservice in the Lire’s several things:

1. Making new directories in CVS:

   - `/service/<superservice>/`
   - `/service/<superservice>/script/`
   - `/service/<superservice>/reports/`

2. Adding several files:

   - `/service/<superservice>/Makefile.am`
   - `/service/<superservice>/reports/Makefile.am`
   - `/service/<superservice>/script/Makefile.am`
   - `/service/<superservice>/<superservice>.cfg`
   - `/service/< superservice>/<superservice>.xml` This file specifies the DLF format of the superservice. Ideally, it should offer a place for each and every snippet of information which will ever be found in a logfile from a program which offers functionality defined by the superservice. This file should have documentation embedded; this will show up in this manual.

3. Writing service plugins (2dlf scripts):

   - `/service/<superservice>/script/<service>2dlf.in`

4. Adapting several files:

   - `/service/configure.in` (add the Makefiles and 2dlf script to AC_OUTPUT, to get them converted from <service>2dlf.in to <service>2dlf.)
   - `/service/Makefile.am` (add the superservice directory to SUBDIRS, so that make gets run there too, when called from the root source directory.)
   - `/service/all/etc/address.cf` (to make the new service known as a member of a superservice.)

5. Update Documentation:

   - User Manual: Chapter "Supported Applications".
   - Add manpages for scripts

6. Update the configuration by writing a custom config spec or extended the current one as well as by added default values to the defaults configuration files.

Chapter 23. Issues with Report Merging

In some cases, a merged report doesn't display the right information. We outline some worst case scenarios, and justify our implementation.

Suppose log file 1 ("requests" with "sizes") looks like:

request	size
A	12
B	11
C	10

while log file 2 looks like:

request	size
D	3
E	2
F	1

We report on the top 2 biggest requests, so the report from log 1 looks like:

request	size
A	12
B	11

while the report from log 2 would look like:

request	size
D	3
E	2

Now we change the superservice.cfg file to list the top-4 biggest items. A naive merge would lead to:

request	size
A	12
B	11
D	3
E	2

Of course, this should've been:

request	size
A	12
B	11
C	10

request	**size**
D	3

This effect does not occur when keeping the top-limit to the same value. However, when we're not reporting on distinct values in the log, but are summing, more horrible things might happen. Consider this: We want to report on the total size by client. Logs look like:

client	**size**
a	12
b	11
c	10

and

client	**size**
d	4
e	4
c	3

Reports from these logs would look like:

client	**size**
a	12
b	11

client	**size**
d	4
e	4

After naively merging, one would get:

client	**size**
a	12
b	11

In fact, the complete report should look like:

client	**size**
c	13
a	12

Luckily, the Lire merging algorithm is not *this* naive: in fact, the XML reports store a little more records than actually needed. This heuristic trick leads to sane merged reports in most cases. However, since this is merely a heuristic trick, it is no waterproof guarantee.

See the description of the guess_extra_entries routine in the Lire::Group manpage for more implementation details.

Chapter 24. Overview of Lire scripts

An overview of the main scripts involved. **lr_spoold** is the engine behind a Lire Online Responder. **lr_log2report** is the main Lire command line interface. The **lr_log2xml** command is a helper scripts. The **lr_xml2report** command can be used by the user to merge XML reports. The **lr_sql2report** is not yet fully integrated in the Lire system. The **lr_rawmail2mail** command manages a Lire client setup. The **lr_cron** is fired of by **cron**, in a cron-driven setup.

```
lr_spoold
|
\_ lr_check_service
\_ lr_spool
   |
   \_ lr_processmail
      \_ lr_getbody
      |
      \_ lr_log2mail
lr_log2report
lr_log2xml
lr_xml2report
lr_rawmail2mail
\_ lr_getbody
\_ lr_deanonymize
\_ lr_xml2mail

lr_cron
```

lr_spoold monitors a Maildir spool for each responder address. lr_processmail processes an email message with a compressed log file attached. Refer to the manpages for the gory details.

Chapter 25. Source Tree Layout

Service specific scripts should reside in $CVSROOT/service/<service>/script/. Configuration data should be in <service>/etc/. Service specific documentation in <service>/doc/.

Furthermore, in each subdirectory there should be a Makefile.am.

Glossary

Definitions of particular terms used in Lire.

DLF

See: Distilled Log Format

Distilled Log Format

Example 1. DNS DLF Excerpts

```
1010912574 10.0.0.2 121.68.134.195.in-addr.arpa PTR recurs
1010912574 10.0.0.2 121.68.134.195.in-addr.arpa PTR recurs
1010912592 10.0.0.2 120.67.123.212.in-addr.arpa PTR recurs
1010912600 10.0.0.2 207.7.178.212.in-addr.arpa PTR recurs
1010912600 10.0.0.2 tr16.kennisnet.nl A recurs
1010912616 10.0.0.2 120.67.123.212.in-addr.arpa PTR recurs
1010912630 10.0.0.2 207.7.178.212.rbl.maps.vix.com ANY recurs
1010912630 10.0.0.2 NLnet.nl ANY recurs
```

This is the generic log format used by Lire to normalise the log files from different products.

Currenlty, this normalised log is a simple ASCII format where each event is represented by one line. The information about the event is represented by fields separated by spaces. All non-printable ASCII characters are replaced by `?`. Spaces in a field's value are replaced by _ (an underscore). Each line must have the same number of fields. A DLF file doesn't contain any header information. Example 1 shows an excerpt of a DNS DLF file.

See Also: Superservice, DLF Schema.

DLF Schema

Information about the order of the fields in a DLF file, their types and what they represent is specified in the DLF's schema. Schemas are defined in XML files using the Lire DLF Schema Markup Language (LDSML). Lire's offers an API (only in Perl for now) to programmatically access the information of a schema.

Log files of many different products can share a common DLF schema that makes Lire's reports easily comparable.

Report

A report is what is generated by Lire. It consists of several subreports. Those subreports can be grouped into sections. The report is computed from the DLF file (and not the native log file) based on a configuration file which describes the subreports that make up the final report along with their parameters. (Consult the *Lire User's Manual* section *Customizing Lire* for more information.)

Service

Put simply, a service is a specific application that produces log files. It is usually the case that one application will be equivalent to one service. For example, the mysql service is used to process MySQL's log files.

But more precisely, a service is a specific log format. For example, the common service can be used for all web servers that support the Common Log Format. Similarly, the welf service can be used to process firewall log files written using WebTrends Enhanced Log Format.

In order to generate a report on it, the native log will be converted to the appropriate superservice's DLF schema

Subreport

A subreport is a particular view on the DLF log's data. Subreports are defined in XML files using the Lire Report Specification Markup Language (LRSML). (Although it defines subreports, it is called a Report Specification because a report is made up out of several subreports.) Example of a subreport would be *Requests by Hours of the Day*.

Subreports are defined for a particular DLF schema.

Superservice

A superservice is a collection of services that share the same DLF schema and report. It is used to group together applications (services) that offer the same kind of functionality.

Lire currently supports eight superservices: database, dns, email, firewall, ftp, print, proxy, and www.