Templates

Templates are a key feature of rsyslog. They allow to specify any format a user might want. They are also used for dynamic file name generation. Every output in rsyslog uses templates - this holds true for files, user messages and so on. The database writer expects its template to be a proper SQL statement - so this is highly customizable too. You might ask how does all of this work when no templates at all are specified. Good question ;) The answer is simple, though. Templates compatible with the stock syslogd formats are hardcoded into rsyslogd. So if no template is specified, we use one of these hardcoded templates. Search for "template_" in syslogd.c and you will find the hardcoded ones.

Templates are specified by template() statements. They can also be specified via $Template legacy statements. Note that these are scheduled for removal in later versions of rsyslog, so it is probably a good idea to avoid them for new uses.

The template() statement

The template() statement is used to define templates. Note that it is a **static** statement, that means all templates are defined when rsyslog reads the config file. As such, templates are not affected by if-statements or config nesting.

The basic structure of the template statement is as follows:

```
template(parameters)
```

In addition to this simpler syntax, list templates (to be described below) support an extended syntax:

```
template(parameters) { list-descriptions }
```

Each template has a parameter **name**, which specifies the templates name, and a parameter **type**, which specifies the template type. The name parameter must be unique, and behaviour is unpredictable if it is not. The **type** parameter specifies different template types. Different types simply enable different ways to specify the template content. The template type **does not** affect what an (output) plugin can do with it. So use the type that best fits your needs (from a config writing point of view!). The following types are available:

- subtree
- string
- plugin

The various types are described below.
In this case, the template is generated by a list of constant and variable statements. These follow the template spec in curly braces. This type is also primarily meant for use with structure-aware outputs, like ommongodb. However, it also works perfectly with text-based outputs. We recommend to use this mode if more complex property substitutions needs to be done. In that case, the list-based template syntax is much clearer than the simple string-based one.

The list template contains the template header (with **type="list"**) and is followed by **constant** and **property** statements, given in curly braces to signify the template statement they belong to. As the name says, **constant** statements describe constant text and **property** describes property access. There are many options to **property**, described further below. Most of these options are used to extract only partial property contents or to modify the text obtained (like to change its case to upper or lower case, only).

To grasp the idea, an actual sample is:

```
template(name="tpl1" type="list") {
        constant(value="Syslog MSG is: '")
        property(name="msg")
        constant(value="', ")
        property(name="timereported" dateFormat="rfc3339" caseConversion="lower")
        constant(value="\n")
        }
```

This sample is probably primarily targeted at the usual file-based output.

constant statement

This provides a way to specify constant text. The text is used literally. It is primarily intended for text-based output, so that some constant text can be included. For example, if a complex template is build for file output, one usually needs to finish it by a newline, which can be introduced by a constant statement. Here is an actual sample of that use case from the rsylsog testbench:

```
template(name="outfmt" type="list") {
        property(name="$!usr!msgnum")
        constant(value="\n")
}
```

The following escape sequences are recogniced inside the constant text:

- \\ - single backslash
- \n - LF
- \ooo - (three octal digits) - represents character with this numerical value (e.g. \101 equals "A"). Note that three octal digits must be given (in contrast to some languagues, where between one and three are valid). While we support octal notation, we recommend to use hex notation as this is better known.
- \xhh - (where h is a hex digit) - represents character with this numerical value (e.g. \x41 equals "A"). Note that two hexadecimal digits must be given (in contrast to some languagues where one or two are valid).
- ... some others ... list needs to be extended

Note: if an unsupported character follows a backslash, this is treated as an error. Behaviour is unpredictable in this case.

To aid usage of the same template both for text-based outputs and structured ones, constant text without an "outname" parameter will be ignored when creating the name/value tree for structured outputs. So if you want to supply some constant text e.g. to mongodb, you must include an outname, as can be seen here:

```
template(name="outfmt" type="list") {
        property(name="$!usr!msgnum")
        constant(value="\n" outname="IWantThisInMyDB")
}
```

The "constant" statement supports the following parameters:

- value - the constant value to use
- outname - output field name (for structured outputs)

property statement

This statement is used to include property text. It can access all properties. Also, options permit to specify picking only part of a property or modifying it. It supports the following parameters:

- name - the name of the property to access
- outname - output field name (for structured outputs)
- dateformat - date format to use (only for date-related properties)
- caseconversion - permits to convert case of the text. supported values are "lower" and "upper"
- controlcharacters - specifies how to handle control characters. Supported values are "escape", which escapes them, "space", which replaces them by a single space, and "drop", which simply removes them from the string.
- securepath - used for creating pathnames suitable for use in dynafile templates
- format - specifiy format on a field basis. Supported values are:
  - "csv" for use when csv-data is generated
  - "json" which formats proper json content (but without a field header)
  - "jsonf" which formats as a complete json field
  - "jsonr" which avoids double escaping the value but makes it safe for a json field
  - "jsonfr" which is the combination of "jsonf" and "jsonr".

- position.from - obtain substring starting from this position (1 is the first position)
- position.to - obtain substring up to this position
- position.relativeToEnd - the from and to position is relative to the end of the string instead of the usual start of string. (available since rsyslog v7.3.10)
- field.number - obtain this field match

- field.delimiter - decimal value of delimiter character for field extraction
- regex.expression - expression to use
- regex.type - either ERE or BRE
- regex.nomatchmode - what to do if we have no match
- regex.match - match to use
- regex.submatch - submatch to use
- droplastlf - drop a trailing LF, if it is present
- mandatory - signifies a field as mandatory. If set to "on", this field will always be present in data passed to structured outputs, even if it is empty. If "off" (the default) empty fields will not be passed to structured outputs. This is especially useful for outputs that support dynamic schemas (like ommongodb).
- spifno1stsp - expert options for RFC3164 template processing

subtree

Available since rsyslog 7.1.4

In this case, the template is generated based on a complete (CEE) subtree. This type of template is most useful for outputs that know how to process hierarchical structure, like ommongodb. With that type, the parameter **subtree** must be specified, which tells which subtree to use. For example template(name="tpl1" type="subtree" subtree="$!") includes all CEE data, while template(name="tpl2" type="subtree" subtree="$!usr!tpl2") includes only the subtree starting at $!usr!tpl2. The core idea when using this type of template is that the actual data is prefabricated via set and unset script statements, and the resulting strucuture is then used inside the template. This method MUST be used if a complete subtree needs to be placed *directly* into the object's root. With all other template types, only subcontainers can be generated. Note that subtree type can also be used with text-based outputs, like omfile. HOWEVER, you do not have any capability to specify constant text, and as such cannot include line breaks. As a consequence, using this template type for text outputs is usually only useful for debugging or very special cases (e.g. where the text is interpreted by a JSON parser later on).

Use case

A typical use case is to first create a custom subtree and then include it into the template, like in this small example:

```
set $!usr!tpl2!msg = $msg;
set $!usr!tpl2!dataflow = field($msg, 58, 2);
template(name="tpl2" type="subtree" subtree="$!usr!tpl2")
```

Here, we assume that $msg contains various fields, and the data from a field is to be extracted and stored - together with the message - as field content.

string

This closely resembles the legacy template statement. It has a mandatory parameter **string**, which holds the template string to be applied. A template string is a mix of constant text and replacement variables (see property replacer). These variables are taken from message or other dynamic content when the final string to be passed to a plugin is generated. String-based templates are a great way to specify textual content, especially if no complex manipulation to properties is necessary. Full details on how to specify template text can be found below.
Config example:

```
template(name="tpl3" type="string" string="%TIMESTAMP:::date-rfc3339% %HOSTNAME% %syslogtag%%msg:::sp-if-no-1st-sp%%msg:::drop-last-lf%\n")
```

plugin

In this case, the template is generated by a plugin (which is then called a "strgen" or "string generator"). The format is fix as it is coded. While this is inflexible, it provides superior performance, and is often used for that reason (not that "regular" templates are slow - but in very demanding environments that "last bit" can make a difference). Refer to the plugin's documentation for further details. For this type, the paramter **plugin** must be specified and must contain the name of the plugin as it identifies itself. Note that the plugin must be loaded prior to being used inside a template.
Config example:

```
template(name="tpl4" type="plugin" plugin="mystrgen")
```

options

The <options> part is optional. It carries options influencing the template as whole and is part of the template parameters. See details below. Be sure NOT to mistake template options with property options - the latter ones are processed by the property replacer and apply to a SINGLE property, only (and not the whole template).

Template options are case-insensitive. Currently defined are:
option.sql - format the string suitable for a SQL statement in MySQL format. This will replace single quotes ("'") and the backslash character by their backslash-escaped counterpart ("\'" and "\\") inside each field. Please note that in MySQL configuration, the NO_BACKSLASH_ESCAPES mode must be turned off for this format to work (this is the default).

option.stdsql - format the string suitable for a SQL statement that is to be sent to a standards-compliant sql server. This will replace single quotes ("'") by two single quotes ("''") inside each field. You must use stdsql together with MySQL if in MySQL configuration the NO_BACKSLASH_ESCAPES is turned on.

option.json - format the string suitable for a json statement. This will replace single quotes ("'") by two single quotes ("''") inside each field.

At no time, multiple template option should be used. This can cause unpredictable behaviour and is against all logic.

Either the **sql** or **stdsql** option **must** be specified when a template is used for writing to a database, otherwise injection might occur. Please note that due to the unfortunate fact that several vendors have violated the sql standard and introduced their own escape methods, it is impossible to have a single option doing all the work. So you yourself must make sure you are using the right format. **If you choose the wrong one, you are still vulnerable to sql injection.**

Please note that the database writer *checks* that the sql option is present in the template. If it is not present, the write database action is disabled. This is to guard you against accidental forgetting it and then becoming vulnerable to SQL injection. The sql option can also be useful with files - especially if you want to import them into a database on another machine for performance reasons. However, do NOT use it if you do not have a real need for it - among others, it takes some toll on the processing time. Not much, but on a really busy system you might notice it ;)

The default template for the write to database action has the sql option set. As we currently support only MySQL and the sql option matches the default MySQL configuration, this is a good choice. However, if you have turned on NO_BACKSLASH_ESCAPES in your MySQL config, you need to supply a template with the stdsql option. Otherwise you will become vulnerable to SQL injection.

To escape:
% = \%
\ = \\ --> '\' is used to escape (as in C)
template (name="TraditionalFormat" type="string" string="%timegenerated% %HOSTNAME% %syslogtag%%msg%\n"

Examples

Standard Template for Writing to Files

```
template(name="FileFormat" type="list") {
        property(name="timestamp" dateFormat="rfc3339")
        constant(value=" ")
        property(name="hostname")
        constant(value=" ")
        property(name="syslogtag")
        constant(value=" ")
        property(name="msg" spifno1stsp="on" )
        property(name="msg" droplastlf="on" )
        constant(value="\n")
        }
```

The equivalent string template looks like this:

```
template(name="FileFormat" type="string"
```

```
  string= "%TIMESTAMP% %HOSTNAME% %syslogtag%%msg:::sp-if-no-1st-sp%%msg:::drop-last-lf%\n"
)
```
Note that the template string itself must be on a single line.

Standard Template for Forwarding to a Remote Host (RFC3164 mode)

```
template(name="ForwardFormat" type="list") {
        constant(value="<")
        property(name="pri")
        constant(value=">")
        property(name="timestamp" dateFormat="rfc3339")
        constant(value=" ")
        property(name="hostname")
        constant(value=" ")
        property(name="syslogtag" position.from="1" position.to="32")
        constant(value=" ")
        property(name="msg" spifno1stsp="on" )
        }
```
The equivalent string template looks like this:

```
template(name="forwardFormat" type="string"
  string="<%PRI%>%TIMESTAMP:::date-rfc3339% %HOSTNAME% %syslogtag:1:32%%msg:::sp-if-no-1st-sp%%msg%"
)
```
Note that the template string itself must be on a single line.

Standard Template for write to the MySQL database

```
template(name="StdSQLformat" type="list" option.sql="on") {
        constant(value="insert into SystemEvents (Message, Facility, FromHost, Priority, DeviceReportedTime, ReceivedAt, InfoUnitID, SysLc
        constant(value=" values ('")
        property(name="msg")
        constant(value="', ")
        property(name="syslogfacility")
        constant(value=", '")
        property(name="hostname")
        constant(value="', ")
        property(name="syslogpriority")
        constant(value=", '")
        property(name="timereported" dateFormat="mysql")
        constant(value="', '")
        property(name="timegenerated" dateFormat="mysql")
        constant(value="', ")
        property(name="iut")
        constant(value=", '")
        property(name="syslogtag")
        constant(value="')")
        }
```
The equivalent string template looks like this:

```
template(name="stdSQLformat" type="string" option.sql="on"
  string="insert into SystemEvents (Message, Facility, FromHost, Priority, DeviceReportedTime, ReceivedAt, InfoUnitID, SysLogTag) values (
)
```

Note that the template string itself must be on a single line.

legacy format

In pre v6-versions of rsyslog, you need to use the `$template` statement to configure templates. They provide the equivalent to string- and plugin-based templates. The legacy syntax continous to work in v7, however we recommend to avoid legacy format for newly written config files. Legacy and current config statements can coexist within the same config file.

The general format is

```
$template name,param[,options]
```

where "name" is the template name and "param" is a single parameter that specifies template content. The optional "options" part is used to set template options.

string

The parameter is the same string that with the current-style format you specify in the **string** parameter, for example:

```
$template strtpl,"PRI: %pri%, MSG: %msg%\n"
```

Note that list templates are not available in legacy format, so you need to use complex property replacer constructs to do complex things.

plugin

This is equivalent to the "plugin"-type template directive. Here, the parameter is the plugin name, with an equal sign prepended. An example is:

```
$template plugintpl,=myplugin
```

Reserved Template Names

Template names beginning with "RSYSLOG_" are reserved for rsyslog use. Do NOT use them if, otherwise you may receive a conflict in the future (and quite unpredictable behaviour). There is a small set of pre-defined templates that you can use without the need to define it:

- RSYSLOG_TraditionalFileFormat - the "old style" default log file format with low-precision timestamps
- RSYSLOG_FileFormat - a modern-style logfile format similar to TraditionalFileFormat, buth with high-precision timestamps and timezone information
- RSYSLOG_TraditionalForwardFormat - the traditional forwarding format with low-precision timestamps. Most useful if you send messages to other syslogd's or rsyslogd below version 3.12.5.
- RSYSLOG_SysklogdFileFormat - sysklogd compatible log file format. If used with options: $SpaceLFOnReceive on; $EscapeControlCharactersOnReceive off; $DropTrailingLFOnReception off, the log format will conform to sysklogd log format.
- RSYSLOG_ForwardFormat - a new high-precision forwarding format very similar to the traditional one, but with high-precision timestamps and timezone information. Recommended to be used when sending messages to rsyslog 3.12.5 or above.
- RSYSLOG_SyslogProtocol23Format - the format specified in IETF's internet-draft ietf-syslog-protocol-23, which is assumed to be come the new syslog standard RFC. This format includes several improvements. The rsyslog message parser understands this format, so you can use it together with all relatively recent versions of rsyslog. Other syslogd's may get hopelessly confused if receiving that format, so check before you use it. Note that the format is unlikely to change when the final RFC comes out, but this may happen.
- RSYSLOG_DebugFormat - a special format used for troubleshooting property problems. This format is meant to be written to a log file. Do **not** use for production or remote forwarding.

The following is legacy documentation soon to be integrated.

Starting with 5.5.6, there are actually two different types of template:

- string based
- string-generator module based

String-generator module based templates have been introduced in 5.5.6. They permit a string generator, actually a C "program", the generate a format. Obviously, it is more work required to code such a generator, but the reward is speed improvement. If you do not need the ultimate

throughput, you can forget about string generators (so most people never need to know what they are). You may just be interested in learning that for the most important default formats, rsyslog already contains highly optimized string generators and these are called without any need to configure anything. But if you have written (or purchased) a string generator module, you need to know how to call it. Each such module has a name, which you need to know (look it up in the module doc or ask the developer). Let's assume that "mystrgen" is the module name. Then you can define a template for that strgen in the following way:

```
template(name="MyTemplateName" type="plugin" string="mystrgen")
```

Legacy example:

```
$template MyTemplateName,=mystrgen
```

(Of course, you must have first loaded the module via $ModLoad).
The important part is the equal sign in the legacy format: it tells the rsyslog config parser that no string follows but a strgen module name.

There are no additional parameters but the module name supported. This is because there is no way to customize anything inside such a "template" other than by modifying the code of the string generator.

So for most use cases, string-generator module based templates are **not** the route to take. Usually, we use **string based templates** instead. This is what the rest of the documentation now talks about.

A template consists of a template directive, a name, the actual template text and optional options. A sample is:

```
template(name="MyTemplateName" type="string" string="Example: Text %property% some more text\n" options)
```

Legacy example:

```
$template MyTemplateName,"\7Text %property% some more text\n",<options>
```

The "template" (legacy: $template) is the template directive. It tells rsyslog that this line contains a template. "MyTemplateName" is the template name. All other config lines refer to this name. The text within "string" is the actual template text. The backslash is an escape character, much as it is in C. It does all these "cool" things. For example, \7 rings the bell (this is an ASCII value), \n is a new line. C programmers and perl coders have the advantage of knowing this, but the set in rsyslog is a bit restricted currently.

All text in the template is used literally, except for things within percent signs. These are properties and allow you access to the contents of the syslog message. Properties are accessed via the property replacer[1] (nice name, huh) and it can do cool things, too. For example, it can pick a substring or do date-specific formatting. More on this is below, on some lines of the property replacer.

Properties can be accessed by the property replacer[2] (see there for details).

Templates can be used in the form of a **list** as well. This has been introduced with **6.5.0** The list consists of two parts which are either a **constant** or a **property**. The constants are taking the part of "text" that you usually enter in string-based templates. The properties stay variable, as they are a substitute for different values of a certain type. This type of template is extremely useful for complicated cases, as it helps you to easily keep an overview over the template. Though, it has the disadvantage of needing more effort to create it.

Config example:

```
template(name="MyTemplate" type="list" option.json="off") {
constant(value="Test: ")
property(name="msg" outname="mymessage")
constant(value=" --!!!-- ")
property(name="timereported" dateFormat="rfc3339" caseConversion="lower")
constant(value="\n")
}
```

First, the general template option will be defined. The values of the template itself get defined in the curly brackets. As it can be seen, we have constants and properties in exchange. Whereas constants will be filled with a value and probably some options, properties do direct to a property and the options that could be needed additional format definitions.

We suggest to use separate lines for all constants and properties. This helps to keep a good overview over the different parts of the template. Though, writing it in a single line will work, it is much harder to debug if anything goes wrong with the template.

Please note that templates can also be used to generate selector lines with dynamic file names. For example, if you would like to split syslog messages from different hosts to different files (one per host), you can define the following template:

```
template (name="DynFile" type="string" string="/var/log/system-%HOSTNAME%.log")
```

Legacy example:

```
$template DynFile,"/var/log/system-%HOSTNAME%.log"
```

This template can then be used when defining an output selector line. It will result in something like "/var/log/system-localhost.log"

Legacy String-based Template Samples

This section provides some default templates in legacy format, as used in rsyslog previous to version 6. Note that this format is still supported, so there is no hard need to upgrade existing configurations. However, it is strongly recommended that the legacy constructs are not used when crafting new templates. Note that each $Template statement is on a **single** line, but probably broken accross several lines for display purposes by your browsers. Lines are separated by empty lines. Keep in mind, that line breaks are important in legacy format.

```
$template FileFormat,"%TIMESTAMP:::date-rfc3339% %HOSTNAME% %syslogtag%%msg:::sp-if-no-1st-sp%%msg:::drop-last-lf%\n"

$template TraditionalFileFormat,"%TIMESTAMP% %HOSTNAME% %syslogtag%%msg:::sp-if-no-1st-sp%%msg:::drop-last-lf%\n"

$template ForwardFormat,"<%PRI%>%TIMESTAMP:::date-rfc3339% %HOSTNAME% %syslogtag:1:32%%msg:::sp-if-no-1st-sp%%msg%"

$template TraditionalForwardFormat,"<%PRI%>%TIMESTAMP% %HOSTNAME% %syslogtag:1:32%%msg:::sp-if-no-1st-sp%%msg%"

$template StdSQLFormat,"insert into SystemEvents (Message, Facility, FromHost, Priority, DeviceReportedTime, ReceivedAt, InfoUnitID, SysLogTag)
values ('%msg%', %syslogfacility%, '%HOSTNAME%', %syslogpriority%, '%timereported:::date-mysql%', '%timegenerated:::date-mysql%', %iut%,
'%syslogtag%')",SQL
```

See Also

[manual index[3]] [rsyslog.conf[4]] [rsyslog site[5]]

1. http://www.rsyslog.com/doc/property_replacer.html
2. http://www.rsyslog.com/doc/property_replacer.html
3. http://www.rsyslog.com/doc/manual.html
4. http://www.rsyslog.com/doc/rsyslog_conf.html
5. http://www.rsyslog.com/
6. http://www.rsyslog.com/
7. http://www.gerhards.net/rainer
8. http://www.adiscon.com/