RSyslog Documentation

rsyslog 8.2104.0 documentation

Templates

Description

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 those hardcoded templates. Search for “template_” in rsconf.c and you will find the hardcoded ones.

Templates are specified by template() statements. They can also be specified via $template legacy statements.

Note: Note: key elements of templates are rsyslog properties. See the rsyslog properties reference for a list of which are available.

Template processing

Due to lack of standardization regarding logs formats, when a template is specified it’s supposed to include HEADER, as defined in RFC5424

It’s very important to have this in mind, and also to understand how rsyslog parsing works.

For example, if the MSG field is set to “this:is a message” and neither HOSTNAME nor TAG are specified, the outgoing parser will split the message as:

```
TAG:this:
MSG:is a message
```

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 template 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:

- list
- subtree
- string
- plugin

The various types are described below.

List

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 to be used 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 need 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 (for instance to change its case to upper or lower case).

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 built 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 recognized inside the constant text:

- \ - single backslash
- \n - LF
- \ooo - (three octal digits) - represents a character with this octal numerical value (e.g. \101 equals “A”). Note that three octal digits must be given (in contrast to some languages, 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 a character with this hexadecimal numerical value (e.g. \x41 equals “A”). Note that two hexadecimal digits must be given (in contrast to some languages where either 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")
}
```

To generate a constant json field, the *format* parameter can be used, as in this example

```
template(name="outfmt" type="list" option.jsonf="on") {
          property(outname="message" name="msg" format="jsonf")
          constant(outname="@version" value="1" format="jsonf")
}
```

The constant statement in this case will generate *“@version”:”1”*. Note that to do this, both the *value* and the *format* parameters must be given.

The “constant” statement supports the following parameters:

- value - the constant value to use
- outname - the output field name (for structured outputs)
- format - can be either empty or *jsonf*

Property statement

This statement is used to include property values. 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** - the output field name (for structured outputs)
- **dateformat** - the date format to use (only for date-related properties). Here you can find a list of all property options. **TODO:** right now, the property replacer documentation contains property format options for string templates, only. The formats for non-string templates differ. For example, date format options in string templates start with “date-” whereas those in property statements do not (e.g. “date-year” vs. just “year”). The technical reason behind this is that inside string templates, the option must include what it applies to whereas with the explicit format that is part of the parameter name.

  To create a customised format you can use multiple property options together. The following example would result in **YYYY-MM-DD**:

```
property(name="timereported" dateformat="year")
constant(value="-")
property(name="timereported" dateformat="month")
constant(value="-")
property(name="timereported" dateformat="day")
```

- **date.inUTC** - date shall be shown in UTC (please note that this requires a bit more performance due to the necessary conversions) Available since 8.18.0.
- **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** - specify 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)
- **fixedwidth** - changes behaviour of position.to so that it pads the source string with spaces up to the value of position.to if the source string is shorter. "on" or "off" (default) (available since rsyslog v8.13.0)
- **compressspace** - compresses multiple spaces (US-ASCII SP character) inside the string to a single one. This compression happens at a very late stage in processing. Most importantly, it happens after substring extraction, so the **position.from** and **position.to** positions are **NOT** affected by this option. (available since v8.18.0).
- **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
- **datatype** - for "jsonf" format ONLY; permits to set a datatype Log messages as string data types natively. Thus every property inside rsyslog is string based. However, in some end systems you need different data types like numbers of boolean. This setting, in jsonf mode, permits to configure a desired data type. Supported data types are:
  - **number - value is treated as a JSON number and not enclosed in quotes.**
    If the property is empty, the value 0 is generated.
  - string - value is a string and enclosed in quotes
  - **auto - value is treated as number if numeric and as string otherwise.**
    The current implementation treats only integers as numeric to avoid confusion.
  - bool - the value is treated as boolean. If it is empty or 0, it will generate "false", else "true".

  If not specified, 'string' datatype is assumed. This is a feature of rsyslog 8.1905.0 or later.
- **onEmpty** - for "jsonf" format ONLY; specifies how empty values shall be handled. Possible values are:
  - keep - emit the empty element
  - skip - completely ignore the element, do not emit anything
  - null - emit a JSON 'null' value

  If not specified, 'keep' is assumed. This is a feature of rsyslog 8.1905.0 or later.

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 structure 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.

This is a sample for a string-based template:

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

The text between percent signs ('%') is interpreted by the rsyslog property replacer. In a nutshell, it contains the property to use as well as options for formatting and further processing. This is very similar to what the `property` object in list templates does (it actually is just a different language to express most of the same things).

Everything outside of the percent signs is constant text. In the above case, we have mostly spaces between the property values. At the end of the string, an escape sequence is used.

Escape sequences permit to specify nonprintable characters. They work very similar to escape sequences in C and many other languages. They are initiated by the backslash characters and followed by one or more characters that specify the actual character. For example \7 is the US-ASCII BEL character and \n is a newline. The set is similar to what C and perl support, but a bit more limited.

Plugin

In this case, the template is generated by a plugin (which is then called a "strgen" or "string generator"). The format is fixed 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 parameter **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 a whole and is a 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.

option.jsonf - format the string as JSON object. This means a leading and trailing curly brace “{” will be added as well as a comma between all non-terminal properties and constants.

option.casesensitive - treat property name references as case sensitive. The default is “off”, where all property name references are first converted to lowercase during template definition. With this option turned “on”, property names are looked up as defined in the template. Use this option if you have JSON (`$!*`), local (`!.*`), or global (`$!\\*`) properties which contain uppercase letters. The normal Rsyslog properties are case-insensitive, so this option is not needed for properly referencing those properties.

Use of the options **option.sql**, **option.stdsql**, and **option.json** are mutually exclusive. Using more than one at the same time can cause unpredictable behaviour.

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 accidentally 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.

```
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")
    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: 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")
    property(name="msg" spifno1stsp="on" )
    property(name="msg")
    }
```

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: The template string itself must be on a single line.

Standard Template for writing to the MySQL database

```
template(name="StdSQLformat" type="list" option.sql="on") {
        constant(value="insert into SystemEvents (Message, Facility, FromHost, Priority, DeviceReportedTime, ReceivedAt, Infol
        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,
        )
```

Note: The template string itself must be on a single line.

Generating JSON

This is especially useful for RESTful APIs, like for example ElasticSearch provides.

This template

```
template(name="outfmt" type="list" option.jsonf="on") {
         property(outname="@timestamp" name="timereported" dateFormat="rfc3339" format="jsonf")
         property(outname="host" name="hostname" format="jsonf")
         property(outname="severity" name="syslogseverity" caseConversion="upper" format="jsonf" datatype="number")
```

```
        property(outname="facility" name="syslogfacility" format="jsonf" datatype="number")
        property(outname="syslog-tag" name="syslogtag" format="jsonf")
        property(outname="source" name="app-name" format="jsonf" onEmpty="null")
        property(outname="message" name="msg" format="jsonf")

 }
```

Generates output similar to this

```
{"@timestamp":"2018-03-01T01:00:00+00:00", "host":"172.20.245.8", "severity":7, "facility":20, "syslog-tag":"tag", "source":"t
```

Pretty-printed this looks like

```
{
  "@timestamp": "2018-03-01T01:00:00+00:00",
  "host": "172.20.245.8",
  "severity": 7,
  "facility": 20,
  "syslog-tag": "tag",
  "source": "tag",
  "message": " msgnum:00000000:"
}
```

Note: The output is **not** pretty-printed as this is just waste of resources when used in RESTful APIs.

If the “app-name” property is empty, a JSON null value is generated as the *onEmpty="null"* parameter is used

```
{"@timestamp":"2018-03-01T01:00:00+00:00", "host":"172.20.245.8", "severity":7, "facility":20, "syslog-tag":"tag", "source":nu
```

Creating Dynamic File Names for omfile

Templates can be used to generate actions 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")
```

Reserved Template Names

Template names beginning with “RSYSLOG_” are reserved for rsyslog use. Do NOT use them, otherwise you may cause conflicts 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 them:

RSYSLOG_TraditionalFileFormat - The “old style” default log file format with low-precision timestamps.

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

RSYSLOG_FileFormat - A modern-style logfile format similar to TraditionalFileFormat, both with high-precision timestamps and timezone information.

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

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.

```
template(name="RSYSLOG_TraditionalForwardFormat" type="string"
     string="<%PRI%>%TIMESTAMP% %HOSTNAME% %syslogtag:1:32%%msg:::sp-if-no-1st-sp%%msg%")
```

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.

```
template(name="RSYSLOG_SysklogdFileFormat" type="string"
     string="%TIMESTAMP% %HOSTNAME% %syslogtag%%msg:::sp-if-no-1st-sp%%msg%\n")
```

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.

```
template(name="RSYSLOG_ForwardFormat" type="string"
     string="<%PRI%>%TIMESTAMP:::date-rfc3339% %HOSTNAME% %syslogtag:1:32%%msg:::sp-if-no-1st-sp%%msg%")
```

RSYSLOG_SyslogProtocol23Format - the format specified in IETF's internet-draft ietf-syslog-protocol-23, which is very close to the actual syslog standard RFC5424 (we couldn't update this template as things were in production for quite some time when RFC5424 was finally approved). This format includes several improvements. You may use this format with all relatively recent versions of rsyslog or syslogd.

```
template(name="RSYSLOG_SyslogProtocol23Format" type="string"
     string="<%PRI%>1 %TIMESTAMP:::date-rfc3339% %HOSTNAME% %APP-NAME% %PROCID% %MSGID% %STRUCTURED-DATA% %msg%\n")
```

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.

```
template(name="RSYSLOG_DebugFormat" type="list") {
     constant(value="Debug line with all properties:\nFROMHOST: '")
     property(name="fromhost")
     constant(value="', fromhost-ip: '")
     property(name="fromhost-ip")
     constant(value="', HOSTNAME: '")
     property(name="hostname")
     constant(value="', PRI: '")
     property(name="pri")
     constant(value="',\nsyslogtag '")
     property(name="syslogtag")
     constant(value="', programname: '")
     property(name="programname")
     constant(value="', APP-NAME: '")
     property(name="app-name")
     constant(value="', PROCID: '")
     property(name="procid")
     constant(value="', MSGID: '")
     property(name="msgid")
     constant(value="',\nTIMESTAMP: '")
     property(name="timereported")
     constant(value="', STRUCTURED-DATA: '")
     property(name="structured-data")
     constant(value="',\nmsg: '")
     property(name="msg")
     constant(value="'\nescaped msg: '")
     property(name="msg" controlcharacters="drop")
     constant(value="'\ninputname: ")
     property(name="inputname")
     constant(value=" rawmsg: '")
     property(name="rawmsg")
     constant(value="'\n$!:")
     property(name="$!")
     constant(value="\n$.:")
     property(name="$.")
     constant(value="\n$/:")
     property(name="$/")
     constant(value="\n\n")
}
```

RSYSLOG_WallFmt - Contains information about the host and the time the message was generated and at the end the syslogtag and message itself.

```
template(name="RSYSLOG_WallFmt" type="string"
     string="\r\n\7Message from syslogd@%HOSTNAME% at %timegenerated% ...\r\n%syslogtag%%msg%\n\r")
```

RSYSLOG_StdUsrMsgFmt - The syslogtag followed by the message is returned.

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

```
     string=" %syslogtag%%msg%\n\r")
```

RSYSLOG_StdDBFmt - Generates a insert command with the message properties, into table SystemEvents for a mysql database.

```
template(name="RSYSLOG_StdDBFmt" type="list") {
     constant(value="insert into SystemEvents (Message, Facility, FromHost, Priority, DeviceReportedTime, ReceivedAt, InfoUni
     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="date-mysql")
     constant(value="', '")
     property(name="timegenerated" dateFormat="date-mysql")
     constant(value="', ")
     property(name="iut")
     constant(value=", '")
     property(name="syslogtag")
     constant(value="')")
}
```

RSYSLOG_StdPgSQLFmt - Generates a insert command with the message properties, into table SystemEvents for a pgsql database.

```
template(name="RSYSLOG_StdPgSQLFmt" type="string"
string="insert into SystemEvents (Message, Facility, FromHost, Priority, DeviceReportedTime,
     ReceivedAt, InfoUnitID, SysLogTag) values ('%msg%', %syslogfacility%, '%HOSTNAME%',
     %syslogpriority%, '%timereported:::date-pgsql%', '%timegenerated:::date-pgsql%', %iut%,
     '%syslogtag%')")
```

RSYSLOG_spoofadr - Generates a message containing nothing more than the ip address of the sender.

```
template(name="RSYSLOG_spoofadr" type="string" string="%fromhost-ip%")
```

RSYSLOG_StdJSONFmt - Generates a JSON structure containing the message properties.

```
template(name="RSYSLOG_StdJSONFmt" type="string"
     string="{\"message\":\"%msg:::json%\",\"fromhost\":\"%HOSTNAME:::json%\",\"facility\":
             \"%syslogfacility-text%\",\"priority\":\"%syslogpriority-text%\",\"timereported\":
             \"%timereported:::date-rfc3339%\",\"timegenerated\":
             \"%timegenerated:::date-rfc3339%\"}")
```

See Also

- How to bind a template
- Adding the BOM to a message
- How to separate log files by host name of the sending device

See also: Help with configuring/using `Rsyslog` :
- Mailing list - best route for general questions
- GitHub: rsyslog source project - detailed questions, reporting issues that are believed to be bugs with `Rsyslog`
- Stack Exchange (View, Ask) - experimental support from rsyslog community

See also: Contributing to `Rsyslog` :
- Source project: rsyslog project README.
- Documentation: rsyslog-doc project README

About

- About Adiscon / Impressum
- Contact Us
- Privacy policy / Datenschutzrichtlinien
- Rainer's Blog

Related Products

- LogAnalyzer
- WinSyslog