Package ‘lgrExtra’

January 6, 2023

Type  Package
Title  Extra Appenders for 'lgr'
Version 0.0.8
Maintainer Stefan Fleck <stefan.b.fleck@gmail.com>
Description Additional appenders for the logging package 'lgr' that support logging to databases, email and push notifications.
License MIT + file LICENSE
Imports data.table, lgr, R6
Suggests covr, DBI, elastic, gmailr, jsonlite, knitr, RMariaDB, rmarkdown, RPostgres, RPushbullet, RSQLite, rsyslog, sendmailR, testthat
Encoding UTF-8
RoxygenNote 7.2.1.9000
NeedsCompilation no
Author Stefan Fleck [aut, cre] (<https://orcid.org/0000-0003-3344-9851>)
Repository CRAN
Date/Publication 2023-01-06 09:10:02 UTC

R topics documented:

  AppenderDbi .................................................. 2
  AppenderDigest ............................................. 4
  AppenderDt .................................................. 5
  AppenderElasticSearch ................................. 8
  AppenderGmail ............................................ 10
  AppenderMail ............................................. 11
  AppenderPushbullet ..................................... 13
  AppenderSendmail ....................................... 15
  AppenderSyslog .......................................... 17
  LayoutDbi .................................................. 19
  LayoutElasticSearch ................................... 21
Log to databases via DBI

Description

Log to a database table with any DBI compatible backend. Please be aware that AppenderDbi does not support case sensitive / quoted column names, and you advised to only use all-lowercase names for custom fields (see ... argument of LogEvent). When appending to a database table all LogEvent values for which a column exists in the target table will be appended, all others are ignored.

NOTE: AppenderDbi works reliable for most databases, but is still considered experimental, especially because the configuration is excessively complicated. Expect breaking changes to AppenderDbi in the future.

Value

The $new() method returns an R6::R6 that inherits from lgr::Appender and can be uses as an appender by a lgr::Logger.

Buffered Logging

By default, AppenderDbi writes each LogEvent directly to the target database which can be relatively slow. To improve performance it is possible to tell AppenderDbi to buffer db writes by setting buffer_size to something greater than 0. This buffer is written to the database whenever it is full (buffer_size), whenever a LogEvent with a level of fatal or error is encountered (flush_threshold), or when the Appender is garbage collected (flush_on_exit), i.e. when you close the R session or shortly after you remove the Appender object via rm().

Creating a New Appender

An AppenderDbi is linked to a database table via its table argument. If the table does not exist it is created either when the Appender is first instantiated or (more likely) when the first LogEvent would be written to that table. Rather than to rely on this feature, it is recommended that you create the target table first using an SQL CREATE TABLE statement as this is safer and more flexible. See also LayoutDbi.

Choosing the correct DBI Layout

Layouts for relational database tables are tricky as they have very strict column types and further restrictions. On top of that implementation details vary between database backends.

To make setting up AppenderDbi as painless as possible, the helper function select_dbi_layout() tries to automatically determine sensible LayoutDbi settings based on conn and - if it exists in the
AppenderDbi

database already - table. If table does not exist in the database and you start logging, a new table will be created with the col_types from layout.

Super classes

`lgr::Filterable` -> `lgr::Appender` -> `lgr::AppenderMemory` -> `AppenderDbi`

Active bindings

`conn` a DBI connection

`close_on_exit` TRUE or FALSE. Close the Database connection when the Logger is removed?

`col_types` a named character vector providing information about the column types in the database.

How the column types are reported depends on the database driver. For example, SQLite returns human readable data types (character, double, ...) while IBM DB2 returns numeric codes representing the data type.

`table` a character scalar or a `DBI::Id` specifying the target database table

`table_name` character scalar. Like $table, but always returns a character scalar

`table_id` `DBI::Id`. Like $table, but always returns a `DBI::Id`

Methods

Public methods:

- `AppenderDbi$new()`
- `AppenderDbi$set_close_on_exit()`
- `AppenderDbi$set_conn()`
- `AppenderDbi$show()`
- `AppenderDbi$flush()`

Method `new()`:

Usage:

```r
AppenderDbi$new(
  conn,
  table,
  threshold = NA_integer_,
  layout = select_dbi_layout(conn, table),
  close_on_exit = TRUE,
  buffer_size = 0,
  flush_threshold = "error",
  flush_on_exit = TRUE,
  flush_on_rotate = TRUE,
  should_flush = NULL,
  filters = NULL
)
```

Arguments:

- `conn` see section Fields
- `table` see section Fields
- `threshold`, `flush_threshold`, `layout`, `buffer_size` see `AppenderBuffer`
Method set_close_on_exit():
Usage:
AppenderDbi$set_close_on_exit(x)

Method set_conn():
Usage:
AppenderDbi$set_conn(conn)

Method show():
Usage:
AppenderDbi$show(threshold = NA_integer_, n = 20)

Method flush():
Usage:
AppenderDbi$flush()

See Also
Other Appenders: AppenderDt, AppenderElasticSearch, AppenderGmail, AppenderPushbullet, AppenderSendmail, AppenderSyslog

Examples
if (requireNamespace("RSQLite")){
  app <- AppenderDbi$new(
    conn = DBI::dbConnect(RSQLite::SQLite(), dbname = "::memory:"),
    table = "log"
  )

  lg <- lgr::get_logger("test/dbi")$add_appender(app, "db")$set_propagate(FALSE)
  lg$info("test")
  print(lg$appenders[[1]]$data)

  invisible(lg$config(NULL)) # cleanup
}

AppenderDigest  Abstract class for digests (multi-log message notifications)

Description
Digests is an abstract class for report-like output that contain several log messages and a title; e.g. an E-mail containing the last 10 log messages before an error was encountered or a push notification.

Abstract classes, only exported for package developers.
**Value**

Abstract classes cannot be instantiated with `$new()` and therefore do not return anything. They are solely for developers that want to write their own extension to lgr.

**Super classes**

`lgr::Filterable -> lgr::Appender -> lgr::AppenderMemory -> AppenderDigest`

**Active bindings**

subject_layout A Layout used to format the last LogEvent in this Appenders buffer when it is flushed. The result will be used as the subject of the digest (for example, the E-mail subject).

**Methods**

**Public methods:**

- `AppenderDigest$new()`
- `AppenderDigest$set_subject_layout()`

**Method new():**

*Usage:*

`AppenderDigest$new(...)`

**Method set_subject_layout():**

*Usage:*

`AppenderDigest$set_subject_layout(layout)`

**See Also**

- LayoutFormat, LayoutGlue
- Other abstract classes: AppenderMail
- Other Digest Appenders: AppenderMail, AppenderPushbullet, AppenderSendmail

---

**AppenderDt**

`Log to an in-memory data.table`

**Description**

An Appender that outputs to an in-memory data.table. It fulfill a similar purpose as the more flexible AppenderBuffer and is mainly included for historical reasons/backwards compatibility with older version of lgr.

**NOTE:** AppenderDt has been superseded by `lgr::AppenderBuffer` and is kept mainly for archival purposes.
Value

The $new() method returns an R6::R6 that inherits from lgr::Appender and can be used as an appender by a lgr::Logger.

Custom Fields

AppenderDt supports custom fields, but they have to be pre-allocated in the prototype argument. Custom fields that are not part of the prototype are inserted in the list-column .fields if it exists.

Creating a Data Table Appender

In addition to the usual fields, AppenderDt$new() requires that you supply a buffer_size and a prototype. These determine the structure of the data.table used to store the log this appender creates and cannot be modified anymore after the instantiation of the appender.

The lgr::Layout for this Appender is used only to format console output of its $show() method.

Comparison AppenderBuffer and AppenderDt

Both AppenderBuffer and AppenderDt do in memory buffering of events. AppenderBuffer retains a copies of the events it processes and has the ability to pass the buffered events on to other Appenders. AppenderDt converts the events to rows in a data.table and is a bit harder to configure. Used inside loops (several hundred iterations), AppenderDt has much less overhead than AppenderBuffer. For single logging calls and small loops, AppenderBuffer is more performant. This is related to how memory pre-allocation is handled by the appenders.

Super classes

lgr::Filterable -> lgr::Appender -> AppenderDt

Methods

Public methods:

- AppenderDt$new()
- AppenderDt$append()
- AppenderDt$show()
- AppenderDt$set_layout()

Method new(): Creating a new AppenderDt

Usage:

AppenderDt$new(
    threshold = NA_integer_,
    layout = LayoutFormat$new(fmt = "%L [%t] %m %f", timestamp_fmt = "%H:%M:%OS3",
                           colors = getOption("lgr.colors", list())),
    prototype = data.table::data.table(.id = NA_integer_, level = NA_integer_,
                                        timestamp = Sys.time(),
                                        logger = NA_character_,
                                        caller = NA_character_,
                                        msg = NA_character_,
                                        .fields = list(list())),
    buffer_size = 1e+05,
    filters = NULL)
Arguments:

prototype A prototype data.table. The prototype must be a data.table with the same columns and column types as the data you want to log. The actual content of the columns is irrelevant. There are a few reserved column names that have special meaning: *.id: integer (mandatory). Must always be the first column and is used internally by the Appender *.fields: list (optional). If present all custom values of the event (that are not already part of the prototype) are stored in this list column.

buffer_size integer scalar. Number of rows of the in-memory data.table

Method append():
Usage:
AppenderDt$append(event)

Method show():
Usage:
AppenderDt$show(threshold = NA_integer_, n = 20L)

Method set_layout():
Usage:
AppenderDt$set_layout(layout)

See Also

data.table::data.table

Other Appenders: AppenderDbi, AppenderElasticSearch, AppenderGmail, AppenderPushbullet, AppenderSendmail, Appender Syslog

Examples

lg <- lgr::get_logger("test")
lg$config(list(
  appenders = list(memory = AppenderDt$new()),
  threshold = NA,
  propagate = FALSE  # to prevent routing to root logger for this example
))
lg$debug("test")
lg$error("test")

# Displaying the log
lg$appenders$memory$data
lg$appenders$memory$show()
lgr::show_log(target = lg$appenders$memory)

# If you pass a Logger to show_log(), it looks for the first AppenderDt
# that it can find.
lgr::show_log(target = lg)

# Custom fields are stored in the list column .fields by default
AppenderElasticSearch

Description

Log to ElasticSearch via HTTP

Details

NOTE: Experimental; not yet fully documented and and details are subject to change

Value

The \$new() method returns an \texttt{R6::R6} that inherits from \texttt{lgr::Appender} and can be uses as an appender by a \texttt{lgr::Logger}.

Super classes

\texttt{lgr::Filterable -> lgr::Appender -> lgr::AppenderMemory -> AppenderElasticSearch}

Active bindings

\texttt{conn} a ElasticSearch connection

\texttt{close_on_exit} TRUE or FALSE. Close the ElasticSearch connection when the Logger is removed?

\texttt{index} a character scalar or a \texttt{DBI::Id} specifying the target ElasticSearch index

Methods

\textbf{Public methods:}

\begin{itemize}
  \item \texttt{AppenderElasticSearch\$new()}
  \item \texttt{AppenderElasticSearch\$set_close_on_exit()}
  \item \texttt{AppenderElasticSearch\$set_conn()}
  \item \texttt{AppenderElasticSearch\$get_data()}
  \item \texttt{AppenderElasticSearch\$show()}
  \item \texttt{AppenderElasticSearch\$flush()}
\end{itemize}

\textbf{Method} \texttt{new()}: 

\textit{Usage:}
AppenderElasticSearch$new(
  conn,
  index,
  threshold = NA_integer_,
  layout = LayoutElasticSearch$new(),
  close_on_exit = TRUE,
  buffer_size = 0,
  flush_threshold = "error",
  flush_on_exit = TRUE,
  flush_on_rotate = TRUE,
  should_flush = NULL,
  filters = NULL
)

Arguments:
conn, index  see section Fields
threshold, flush_threshold, layout, buffer_size  see AppenderBuffer A data data.frame.
content of index

Method set_close_on_exit():
Usage:
AppenderElasticSearch$set_close_on_exit(x)

Method set_conn():
Usage:
AppenderElasticSearch$set_conn(conn)

Method get_data():
Usage:
AppenderElasticSearch$get_data(
  n = 20L,
  threshold = NA,
  result_type = "data.frame"
)

Arguments:
n  integer scalar. Retrieve only the last n log entries that match threshold
threshold  character or integer scalar. The minimum log level that should be displayed
result_type  character scalar. Any of:
  • data.frame
  • data.table (shortcut: dt)
  • list (unprocessed list with ElasticSearch metadata)
  • json (raw ElasticSearch JSON)

Returns:  see result_type

Method show():
Usage:
AppenderElasticSearch$show(threshold = NA_integer_, n = 20)

**Method** flush()

*Usage:*
AppenderElasticSearch$flush()

**See Also**

Other Appenders: AppenderDbi, AppenderDt, AppenderGmail, AppenderPushbullet, AppenderSendmail, AppenderSyslog

---

**AppenderGmail**

Send emails via the Gmail REST API

**Description**

Send mails via gmailr::gm_send_message(). This Appender keeps an in-memory buffer like AppenderBuffer. If the buffer is flushed, usually because an event of specified magnitude is encountered, all buffered events are concatenated to a single message. The default behavior is to push the last 30 log events in case a fatal event is encountered.

**NOTE:** This Appender requires that you set up google API authorization, please refer to the documentation of gmailr for details.

**Value**

The $new() method returns an R6::R6 that inherits from lgr::Appender and can be used as an appender by a lgr::Logger.

**Super classes**

lgr::Filterable -> lgr::Appender -> lgr::AppenderMemory -> lgrExtra::AppenderDigest
-> lgrExtra::AppenderMail -> AppenderGmail

**Methods**

**Public methods:**

- AppenderGmail$new()
- AppenderGmail$flush()

**Method** new(): see AppenderMail for details

*Usage:*
AppenderGmail$new(
to,
threshold = NA_integer_,
flush_threshold = "fatal",
layout = LayoutFormat$new(fmt = "%L [%t] %m %f", timestamp_fmt = "%H:%M:%S"),
...)
AppenderMail

subject_layout = LayoutFormat$new(fmt = "[LGR] %L: %m"),
buffer_size = 30,
from = get_user(),
cc = NULL,
bcc = NULL,
html = FALSE,
filters = NULL
)

Method flush():

Usage:
AppenderGmail$flush()

See Also

LayoutFormat, LayoutGlue

Other Appenders: AppenderDbi, AppenderDt, AppenderElasticSearch, AppenderPushbullet,
AppenderSendmail, AppenderSyslog

---

AppenderMail

Abstract class for email Appenders

Description

Abstract classes, only exported for package developers.

Value

Abstract classes cannot be instantiated with $new() and therefore do not return anything. They are
solely for developers that want to write their own extension to lgr.

Super classes

lgr::Filterable -> lgr::Appender -> lgr::AppenderMemory -> lgrExtra::AppenderDigest
-> AppenderMail

Active bindings

to character vector. The email addresses of the recipient
from character vector. The email address of the sender
cc character vector. The email addresses of the cc-recipients (carbon copy)
bcc character vector. The email addresses of bcc-recipients (blind carbon copy)
html logical scalar. Send a html email message? This does currently only format the log contents
as monospace verbatim text.
Methods

Public methods:

• `AppenderMail$new()`
• `AppenderMail$set_to()`
• `AppenderMail$set_from()`
• `AppenderMail$set_cc()`
• `AppenderMail$set_bcc()`
• `AppenderMail$set_html()`
• `AppenderMail$format()`

Method `new()`:

Usage:
`AppenderMail$new(...)`

Method `set_to()`:

Usage:
`AppenderMail$set_to(x)`

Method `set_from()`:

Usage:
`AppenderMail$set_from(x)`

Method `set_cc()`:

Usage:
`AppenderMail$set_cc(x)`

Method `set_bcc()`:

Usage:
`AppenderMail$set_bcc(x)`

Method `set_html()`:

Usage:
`AppenderMail$set_html(x)`

Method `format()`:

Usage:
`AppenderMail$format(color = FALSE, ...)`

See Also

Other abstract classes: `AppenderDigest`

Other Digest Appenders: `AppenderDigest, AppenderPushbullet, AppenderSendmail`
**AppenderPushbullet**

_Send push-notifications via RPushbullet_

---

**Description**

Send push notifications via Pushbullet. This Appender keeps an in-memory buffer like AppenderBuffer. If the buffer is flushed, usually because an event of specified magnitude is encountered, all buffered events are concatenated to a single message that is sent to `RPushbullet::pbPost()`. The default behavior is to push the last 7 log events in case a fatal event is encountered.

**Value**

The `$new()` method returns an `R6::R6` that inherits from `lgr::Appender` and can be used as an appender by a `lgr::Logger`.

**Super classes**

`lgr::Filterable` -> `lgr::Appender` -> `lgr::AppenderMemory` -> `lgrExtra::AppenderDigest` -> `AppenderPushbullet`

**Active bindings**

- apikey see `RPushbullet::pbPost()`
- recipients see `RPushbullet::pbPost()`
- email see `RPushbullet::pbPost()`
- channel see `RPushbullet::pbPost()`
- devices see `RPushbullet::pbPost()`

**Methods**

**Public methods:**

- `AppenderPushbullet$new()`
- `AppenderPushbullet$flush()`
- `AppenderPushbullet$set_apikey()`
- `AppenderPushbullet$set_recipients()`
- `AppenderPushbullet$set_email()`
- `AppenderPushbullet$set_channel()`
- `AppenderPushbullet$set_devices()`

**Method** `new()`:

_Usage:_
AppenderPushbullet$new(
  threshold = NA_integer_,
  flush_threshold = "fatal",
  layout = LayoutFormat$new(fmt = "%K %t> %m %f", timestamp_fmt = "%H:%M:%S"),
  subject_layout = LayoutFormat$new(fmt = "[LGR] %L: %m"),
  buffer_size = 6,
  recipients = NULL,
  email = NULL,
  channel = NULL,
  devices = NULL,
  apikey = getOption("rpushbullet.key"),
  filters = NULL
)

Arguments:
threshold, flush_threshold, layout, buffer_size see AppenderBuffer
subject_layout A lgr::LayoutFormat object.
recipients, email, channel, devices, apikey see RPushbullet::pbPost

Method flush():
Usage:
AppenderPushbullet$flush()

Method set_apikey():
Usage:
AppenderPushbullet$set_apikey(x)

Method set_recipients():
Usage:
AppenderPushbullet$set_recipients(x)

Method set_email():
Usage:
AppenderPushbullet$set_email(x)

Method set_channel():
Usage:
AppenderPushbullet$set_channel(x)

Method set_devices():
Usage:
AppenderPushbullet$set_devices(x)

See Also
LayoutFormat, LayoutGlue

Other Appenders: AppenderDbi, AppenderDt, AppenderElasticSearch, AppenderGmail, AppenderSendmail, AppenderSyslog

Other Digest Appenders: AppenderDigest, AppenderMail, AppenderSendmail
Examples

```r
if (requireNamespace("RPushbullet") && !is.null(getOption("rpushbullet.key"))) {
    app <- AppenderPushbullet$new()

    lg <- lgr::get_logger("test/dbi")
    add_appender(app, "pb")
    set_propagate(FALSE)

    lg$fatal("info")
    lg$fatal("test")

    invisible(lg$config(NULL))
}
```

---

**Description**

Send mails via `sendmailR::sendmail()`, which requires that you have access to an SMTP server that does not require authentication. This Appender keeps an in-memory buffer like `AppenderBuffer`. If the buffer is flushed, usually because an event of specified magnitude is encountered, all buffered events are concatenated to a single message. The default behavior is to push the last 30 log events in case a fatal event is encountered.

**Value**

The `$new()` method returns an `R6::R6` that inherits from `lgr::Appender` and can be used as an appender by a `lgr::Logger`.

**Super classes**

`lgr::Filterable` -> `lgr::Appender` -> `lgr::AppenderMemory` -> `lgrExtra::AppenderDigest` -> `lgrExtra::AppenderMail` -> `AppenderSendmail`

**Active bindings**

- `control` see `sendmailR::sendmail()`
- `headers` see `sendmailR::sendmail()`

**Methods**

- **Public methods:**
  - `AppenderSendmail$new()`  
  - `AppenderSendmail$flush()`  
  - `AppenderSendmail$set_control()`  
  - `AppenderSendmail$set_headers()`
Method new(): see AppenderMail for details

Usage:
AppenderSendmail$new(
to,
control,
threshold = NA_integer_,
flush_threshold = "fatal",
layout = LayoutFormat$new(fmt = " %L [%t] %m %f", timestamp_fmt = "%H:%M:%S"),
subject_layout = LayoutFormat$new(fmt = "[LGR] %L: %m"),
buffer_size = 29,
from = get_user(),
cc = NULL,
bcc = NULL,
html = FALSE,
headers = NULL,
filters = NULL
)

Method flush():

Usage:
AppenderSendmail$flush()

Method set_control():

Usage:
AppenderSendmail$set_control(x)

Method set_headers():

Usage:
AppenderSendmail$set_headers(x)

Note
The default Layout’s fmt indents each log entry with 3 blanks. This is a workaround so that Microsoft Outlook does not mess up the line breaks.

See Also

LayoutFormat, LayoutGlue

Other Appenders: AppenderDbi, AppenderDt, AppenderElasticSearch, AppenderGmail, AppenderPushbullet, AppenderSyslog

Other Digest Appenders: AppenderDigest, AppenderMail, AppenderPushbullet

Examples

## Not run:
lgr::AppenderSendmail$new(
to = "user@ecorp.com",
control = list(smtpServer = "mail.ecorp.com"),

---
AppenderSyslog

from = "lgr_user@yourmail.com"
)

## End(Not run)

if (requireNamespace("sendmailR")){
# requires that you have access to an SMTP server

lg <- lgr::get_logger("lgrExtra/test/mail")$set_propagate(FALSE)$add_appender(AppenderSendmail$new(
  from = "ceo@ecorp.com",
  to = "some.guy@ecorp.com",
  control = list(smtpServer = "mail.somesmptserver.com")
))

# cleanup
invisible(lg$config(NULL))
}

---

### AppenderSyslog

**Log to the POSIX system log**

**Description**

An Appender that writes to the syslog on supported POSIX platforms. Requires the **rsyslog** package.

**Value**

The `$new()` method returns an R6::R6 that inherits from lgr::Appender and can be used as an appender by a lgr::Logger.

**Super classes**

lgr::Filterable -> lgr::Appender -> AppenderSyslog

**Public fields**

- `syslog_levels`. Either a named character vector or a function mapping lgr log levels to rsyslog log levels. See `$set_syslog_levels()`.

**Active bindings**

- `identifier` character scalar. A string identifying the process; if NULL defaults to the logger name.
- `syslog_levels`. Either a named character vector or a function mapping lgr log levels to rsyslog log levels. See `$set_syslog_levels()`.
Methods

Public methods:

- `AppenderSyslog$new()`
- `AppenderSyslog$append()`
- `AppenderSyslog$set_syslog_levels()`
- `AppenderSyslog$set_identifier()`

**Method new():**

*Usage:*

```
AppenderSyslog$new(
    identifier = NULL,
    threshold = NA_integer_,
    layout = LayoutFormat$new("%m"),
    filters = NULL,
    syslog_levels = c(CRITICAL = "fatal", ERR = "error", WARNING = "warn", INFO = "info",
                     DEBUG = "debug", DEBUG = "trace")
)
```

**Method append():**

*Usage:*

```
AppenderSyslog$append(event)
```

**Method set_syslog_levels():** Define conversion between lgr and syslog log levels

*Usage:*

```
AppenderSyslog$set_syslog_levels(x)
```

*Arguments:*

- `x` • a named character vector mapping whose names are log levels as understood by `rsyslog::syslog()` and whose values are lgr log levels (either character or numeric)
- a function that takes a vector of lgr log levels as input and returns a character vector of log levels for `rsyslog::syslog()`.

**Method set_identifier():** Set a string to identify the process.

*Usage:*

```
AppenderSyslog$set_identifier(x)
```

**See Also**

- `LayoutFormat`, `LayoutJson`
- Other Appenders: `AppenderDbi`, `AppenderDt`, `AppenderElasticSearch`, `AppenderGmail`, `AppenderPushbullet`, `AppenderSendmail`
Examples

```r
if (requireNamespace("rsyslog", quietly = TRUE) && Sys.info()["sysname"] == "Linux") {
  lg <- lgr::get_logger("rsyslog/test")
  lg$add_appender(AppenderSyslog$new(), "syslog")
  lg$info("A test message")
  print(system("journalctl -t 'rsyslog/test'"))
	nonvisible(lg$config(NULL))  # cleanup
}
```

**Description**

`LayoutDbi` can contain `col_types` that `AppenderDbi` can use to create new database tables; however, it is safer and more flexible to set up the log table up manually with an SQL `CREATE TABLE` statement instead.

**Details**

The `LayoutDbi` parameters `fmt`, `timestamp_fmt`, `colors` and `pad_levels` are only applied for for console output via the `$show()` method and do not influence database inserts in any way. The inserts are pre-processed by the methods `$format_data()`, `$format_colnames` and `$format_tablenames`.

It does not format LogEvents directly, but their `data.table` representations (see `as.data.table.LogEvent`), as well as column- and table names.

**Value**

The `$new()` method returns an `R6::R6` that inherits from `lgr::Layout` and can used as a Layout by an `lgr::Appender`.

**Database Specific Layouts**

Different databases have different data types and features. Currently the following `LayoutDbi` sub-classes exist that deal with specific databases, but this list is expected to grow as `lgrExtra` matures:

- `LayoutSqlite`: For SQLite databases
- `LayoutPostgres`: For Postgres databases
- `LayoutMySql`: For MySQL databases
- `LayoutDb2`: For DB2 databases

The utility function `select_dbi_layout()` tries returns the appropriate `Layout` for a DBI connection, but this does not work for odbc and JDBC connections where you have to specify the layout manually.

For creating custom DB-specific layouts it should usually be enough to create an `R6::R6` class that inherits from `LayoutDbi` and choosing different defaults for `$format_table_name`, `$format_colnames` and `$format_data`. 
Super classes

`lgr::Layout` -> `lgr::LayoutFormat` -> `LayoutDbi`

Public fields

`format_table_name` a function to format the table name before inserting to the database. The function will be applied to the `$table_name` before inserting into the database. For example, some databases prefer all lowercase names, some uppercase. SQL updates should be case-agnostic, but sadly in practice not all DBI backends behave consistently in this regard.

`format_colnames` a function to format the column names before inserting to the database. The function will be applied to the column names of the data frame to be inserted into the database.

`format_data` a function to format the data before inserting into the database. The function will be applied to the whole data frame.

`names` of the columns that contain data that has been serialized to JSON strings

Active bindings

`col_types` a named character vector of column types supported by the target database. If not NULL this is used by `AppenderDbi` or similar Appenders to create a new database table on instantiation of the Appender. If the target database table already exists, `col_types` is not used.

`names` of the columns that contain data that has been serialized to JSON strings

`col_names` column names of the target table (the same as `names(lo$col_types)`) 

Methods

Public methods:

- `LayoutDbi$new()`
- `LayoutDbi$set_col_types()`
- `LayoutDbi$set_serialized_cols()`
- `LayoutDbi$sql_create_table()`
- `LayoutDbi$toString()`
- `LayoutDbi$clone()`

**Method** `new()`:

*Usage:*

```r
LayoutDbi$new(
  col_types = c(level = "integer", timestamp = "timestamp", logger = "varchar(256)",
                caller = "varchar(256)", msg = "varchar(2048)"),
  serialized_cols = NULL,
  fmt = "%L [%t] %m %f",
  timestamp_fmt = "%Y-%m-%d %H:%M:%S",
  colors =getOption("lgr.colors", list()),
  pad_levels = "right",
  format_table_name = identity,
  format_colnames = identity,
)```
Method set_col_types():
Usage:
LayoutDbi$set_col_types(x)

Method set_serialized_cols():
Usage:
LayoutDbi$set_serialized_cols(x)

Method sql_create_table():
Usage:
LayoutDbi$sql_create_table(table)

Method toString():
Usage:
LayoutDbi$toString()

Method clone(): The objects of this class are cloneable with this method.
Usage:
LayoutDbi$clone(deep = FALSE)
Arguments:
dep Whether to make a deep clone.

See Also

select_dbj_layout(), DBI::DBI,
Other Layout: LayoutElasticSearch

Description

Similar to lgr::LayoutJson, but with some modifications to prepare data for ElasticSearch.

Value

The $new() method returns an R6::R6 that inherits from lgr::Layout and can used as a Layout by an lgr::Appender.

Super class

lgr::Layout -> LayoutElasticSearch
Active bindings

- toJSON_args: a list of values passed on to `jsonlite::toJSON()
- transform_event: a function with a single argument event that takes a `lgr::LogEvent` and returns a list.

Methods

Public methods:

- `LayoutElasticSearch$new()`
- `LayoutElasticSearch$format_event()`
- `LayoutElasticSearch$set_toJSON_args()`
- `LayoutElasticSearch$set_transform_event()`
- `LayoutElasticSearch$clone()`

Method `new()`:

Usage:
```
LayoutElasticSearch$new(
  toJSON_args = list(auto_unbox = TRUE),
  transform_event = function(event) get("values", event)
)
```

Method `format_event()`:

Usage:
```
LayoutElasticSearch$format_event(event)
```

Method `set_toJSON_args()`:

Usage:
```
LayoutElasticSearch$set_toJSON_args(x)
```

Method `set_transform_event()`:

Usage:
```
LayoutElasticSearch$set_transform_event(x)
```

Method `clone()`: The objects of this class are cloneable with this method.

Usage:
```
LayoutElasticSearch$clone(deep = FALSE)
```

Arguments:
- `deep` Whether to make a deep clone.

See Also

Other Layout: `LayoutDbi`
select_db_layout

Automatically select appropriate layout for logging to a database

Description

Selects an appropriate Layout for a database table based on a DBI connection and - if it already exists in the database - the table itself.

Usage

select_db_layout(conn, table, ...)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn</td>
<td>a DBI connection</td>
</tr>
<tr>
<td>table</td>
<td>a character scalar. The name of the table to log to.</td>
</tr>
<tr>
<td>...</td>
<td>passed on to the appropriate LayoutDbi subclass constructor.</td>
</tr>
</tbody>
</table>

Serializer

Serializers

Description

Serializers are used by AppenderDbi to store multiple values in a single text column in a Database table. Usually you just want to use the default SerializerJson. Please note that AppenderDbi as well as Serializers are still experimental.

Value

a Serializer R6::R6 object for AppenderDbi.

Methods

Public methods:

- Serializer$clone()

Method clone(): The objects of this class are cloneable with this method.

Usage:
Serializer$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.

Super class

lgrExtra::Serializer -> SerializerJson
Methods

**Public methods:**
- `SerializerJson$new()`
- `SerializerJson$serialize()`
- `SerializerJson$clone()`

**Method new():**

*Usage:*

```r
SerializerJson$new(
  cols = "*",
  cols_exclude = c("level", "timestamp", "logger", "caller", "msg"),
  col_filter = is.atomic,
  max_nchar = 2048L,
  auto_unbox = TRUE
)
```

**Method serialize():**

*Usage:*

```r
SerializerJson$serialize(event)
```

**Method clone():** The objects of this class are cloneable with this method.

*Usage:*

```r
SerializerJson$clone(deep = FALSE)
```

*Arguments:*
- `deep` Whether to make a deep clone.

**Examples**

```r
# The default Serializer for 'custom fields' columns
SerializerJson$new()
```

---

**unpack_json_cols**

Unserialize data frame columns that contain JSON

**Description**

Unserialize data frame columns that contain JSON

**Usage**

```r
unpack_json_cols(x, cols)
```

```r
## S3 method for class 'data.table'
unpack_json_cols(x, cols)
```

```r
## S3 method for class 'data.frame'
unpack_json_cols(x, cols)
```
unpack_json_cols

Arguments

x: a data.frame

cols: character vector. The names of the text columns containing JSON strings that should be expanded.

Value

A data.frame with additional columns expanded from the columns containing JSON.

Examples

```r
x <- data.frame(
  name = "example data",
  fields = '{"letters":["a","b","c"], "LETTERS":["A","B","C"]}',
  stringsAsFactors = FALSE
)
res <- unpack_json_cols(x, "fields")
res
res$letters[[1]]
```
Index

* Appenders
  AppenderDbi, 2
  AppenderDt, 5
  AppenderElasticSearch, 8
  AppenderGmail, 10
  AppenderPushbullet, 13
  AppenderSendmail, 15
  AppenderSyslog, 17
* Digest Appenders
  AppenderDigest, 4
  AppenderMail, 11
  AppenderPushbullet, 13
  AppenderSendmail, 15
* Layout
  LayoutDbi, 19
  LayoutElasticSearch, 21
* abstract classes
  AppenderDigest, 4
  AppenderMail, 11
* database layouts
  LayoutDbi, 19
  AppenderBuffer, 3, 5, 6, 9, 10, 13–15
  AppenderDbi, 2, 7, 10, 11, 14, 16, 18–20, 23
  AppenderDigest, 4, 12, 14, 16
  AppenderDt, 4, 5, 6, 10, 11, 14, 16, 18
  AppenderElasticSearch, 4, 7, 8, 11, 14, 16, 18
  AppenderGmail, 4, 7, 10, 14, 16, 18
  AppenderMail, 5, 10, 11, 14, 16
  AppenderPushbullet, 4, 5, 7, 10–12, 13, 16, 18
  AppenderSendmail, 4, 5, 7, 10–12, 14, 15, 18
  AppenderSyslog, 4, 7, 10, 11, 14, 16, 17
  as.data.table.LogEvent, 19
  custom fields, 6
  data.table::data.table, 7
  DBI connection, 3, 23
  DBI::DBI, 21
  DBI::Id, 3, 8
  ElasticSearch connection, 8
  gmailr::gm_send_message(), 10
  jsonlite::toJSON(), 22
  Layout, 5
  LayoutDb2 (LayoutDbi), 19
  LayoutDbi, 2, 19, 22
  LayoutElasticSearch, 21, 21
  LayoutFormat, 5, 11, 14, 16, 18
  LayoutGlue, 5, 11, 14, 16
  LayoutJson, 18
  LayoutMySql (LayoutDbi), 19
  LayoutPostgres (LayoutDbi), 19
  LayoutRjdbc (LayoutDbi), 19
  LayoutRjdbcDb2 (LayoutDbi), 19
  LayoutSqlite (LayoutDbi), 19
  lgr log levels, 18
  lgr::Appender, 2, 3, 5, 6, 8, 10, 11, 13, 15, 17, 19, 21
  lgr::AppenderBuffer, 5
  lgr::AppenderMemory, 3, 5, 8, 10, 11, 13, 15
  lgr::Filterable, 3, 5, 6, 8, 10, 11, 13, 15, 17
  lgr::Layout, 6, 19–21
  lgr::LayoutFormat, 14, 20
  lgr::LayoutJson, 21
  lgr::LogEvent, 22
  lgr::Logger, 2, 6, 8, 10, 13, 15, 17
  lgrExtra::AppenderDigest, 10, 11, 13, 15
  lgrExtra::AppenderMail, 10, 15
  lgrExtra::Serializer, 23
  log_levels, 17
  LogEvent, 2, 5
  R6::R6, 2, 6, 8, 10, 13, 15, 17, 19, 21, 23
  RPushbullet::pbPost, 14
  RPushbullet::pbPost(), 13
INDEX

rsyslog::syslog(), 18

select_db_layout, 23
select_db_layout(), 2, 19, 21
sendmailR::sendmail(), 15
Serializer, 23
SerializerJson (Serializer), 23

unpack_json_cols, 24