

# Package ‘lgrExtra’

May 8, 2026

**Type** Package

**Title** Extra Appenders for 'lgr'

**Version** 0.2.2

**Description** Additional appenders for the logging package 'lgr' that support logging to 'Elasticsearch', 'Dynatrace', 'AWSCloudWatchLog', databases, 'syslog', email- and push notifications, and more.

**License** MIT + file LICENSE

**URL** <https://s-fleck.github.io/lgrExtra/>

**Imports** data.table, lgr (>= 0.5.0), R6

**Suggests** covr, DBI, elastic, gmailr, httr2, jsonlite, knitr, paws.management (>= 0.4.0), pool, RMariaDB, rmarkdown, RPostgres, RPushbullet, RSQLite, rsyslog, sendmailR, testthat, utils

**Encoding** UTF-8

**RoxygenNote** 7.3.2

**NeedsCompilation** no

**Author** Stefan Fleck [aut, cre] (ORCID: <https://orcid.org/0000-0003-3344-9851>),  
Jimmy Briggs [ctb, rev] (ORCID: <https://orcid.org/0000-0002-7489-8787>)

**Maintainer** Stefan Fleck <stefan.b.fleck@gmail.com>

**Repository** CRAN

**Date/Publication** 2025-10-31 14:30:02 UTC

## Contents

AppenderAWSCloudWatchLog . . . . .	2
AppenderDbi . . . . .	5
AppenderDigest . . . . .	7
AppenderDynatrace . . . . .	8
AppenderElasticSearch . . . . .	10

AppenderGmail . . . . .	12
AppenderMail . . . . .	13
AppenderPool . . . . .	15
AppenderPushbullet . . . . .	17
AppenderSendmail . . . . .	20
AppenderSyslog . . . . .	22
as_unix_epoch_ms . . . . .	24
LayoutDbi . . . . .	24
LayoutElasticSearch . . . . .	27
select_dbi_layout . . . . .	28
Serializer . . . . .	28
transform_event_dynatrace . . . . .	30
transform_event_elasticsearch . . . . .	30
unpack_json_cols . . . . .	31
<b>Index</b>	<b>33</b>

---

AppenderAWSCloudWatchLog  
*Log to AWS CloudWatch Logs*

---

## Description

Log to [AWS CloudWatch Logs](#).

## Value

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

## Buffered Logging

By default, `AppenderAWSCloudWatchLog` writes each `LogEvent` which can be relatively slow. To improve performance it is possible to tell `AppenderAWSCloudWatchLog` to buffer db writes by setting `buffer_size` to something greater than 0. This buffer is written to AWS CloudWatch 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 `AppenderAWSCloudWatchLog` is linked to an AWS Account using the [paws sdk package](#). If the log group 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.

## Super classes

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

**Active bindings**

client a [paws.management cloudwatchlogs client](#)

log\_group\_name The name of the AWS CloudWatch log group.

log\_stream\_name The name of the log stream within the log\_group\_name.

**Methods****Public methods:**

- [AppenderAWSCloudWatchLog\\$new\(\)](#)
- [AppenderAWSCloudWatchLog\\$set\\_client\(\)](#)
- [AppenderAWSCloudWatchLog\\$set\\_log\\_group\\_name\(\)](#)
- [AppenderAWSCloudWatchLog\\$set\\_log\\_stream\\_name\(\)](#)
- [AppenderAWSCloudWatchLog\\$set\\_log\\_group\\_retention\\_days\(\)](#)
- [AppenderAWSCloudWatchLog\\$flush\(\)](#)

**Method new():**

*Usage:*

```
AppenderAWSCloudWatchLog$new(
  log_group_name,
  log_stream_name = paste(log_group_name, Sys.Date(), sep = "/"),
  log_group_retention_days = NULL,
  paws_config = list(),
  threshold = NA_integer_,
  layout = LayoutFormat$new(fmt = "%L: %m", colors = list()),
  buffer_size = 0,
  flush_threshold = "error",
  flush_on_exit = TRUE,
  flush_on_rotate = TRUE,
  should_flush = NULL,
  filters = NULL
)
```

*Arguments:*

log\_group\_name The name of the AWS CloudWatch log group.

log\_stream\_name The name of the log stream within the log\_group\_name.

log\_group\_retention\_days The number of days to retain the log events in the specified log group. [AWS API Documentation](#)

paws\_config list of paws config. Please see section [https://www.paws-r-sdk.com/docs/set\\_service\\_parameter/](https://www.paws-r-sdk.com/docs/set_service_parameter/)

threshold, flush\_threshold, layout, buffer\_size see [lgr::AppenderBuffer](#)

**Method set\_client():** set paws.management cloudwatchlogs client

*Usage:*

```
AppenderAWSCloudWatchLog$set_client(client)
```

*Arguments:*

client (paws.management::cloudwatchlogs) client. [AWS CloudWatch](#)

**Method** `set_log_group_name()`: set log group name for AWS CloudWatch

*Usage:*

```
AppenderAWSCloudWatchLog$set_log_group_name(log_group_name)
```

*Arguments:*

`log_group_name` (character) name of AWS CloudWatch

**Method** `set_log_stream_name()`: set log stream name within AWS CloudWatch log group

*Usage:*

```
AppenderAWSCloudWatchLog$set_log_stream_name(log_stream_name)
```

*Arguments:*

`log_stream_name` (character) log stream name with AWS CloudWatch log group

**Method** `set_log_group_retention_days()`: set log group retention days for AWS CloudWatch Log Group.

*Usage:*

```
AppenderAWSCloudWatchLog$set_log_group_retention_days(log_group_retention_days)
```

*Arguments:*

`log_group_retention_days` The number of days to retain the log events in the specified log group. [AWS API Documentation](#)

**Method** `flush()`:

*Usage:*

```
AppenderAWSCloudWatchLog$flush()
```

## See Also

Other Appenders: [AppenderDbi](#), [AppenderDynatrace](#), [AppenderElasticSearch](#), [AppenderGmail](#), [AppenderPool](#), [AppenderPushbullet](#), [AppenderSendmail](#), [AppenderSyslog](#)

## Examples

```
## Not run:
library(lgrExtra)
app <- AppenderAWSCloudWatchLog$new(log_group_name = "lgrExtra")
lg <- lgr::get_logger("lgrExtra")$add_appender(app)$set_propagate(FALSE)
lg$info("test")
print(lg$appenders[[1]]$data)

invisible(lg$config(NULL)) # cleanup

## End(Not run)
```

## 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 are advised to only use all-lowercase names for custom fields (see `...` argument of [lgr::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 reliably 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 used 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 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:*

```
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, table see section *Fields*

threshold, flush\_threshold, layout, buffer\_size see [lgr::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: [AppenderAWSCloudWatchLog](#), [AppenderDynatrace](#), [AppenderElasticSearch](#), [AppenderGmail](#), [AppenderPool](#), [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 [lgr::Layout](#) used to format the last [lgr::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**

[lgr::LayoutFormat](#), [lgr::LayoutGlue](#)

Other abstract classes: [AppenderMail](#)

Other Digest Appenders: [AppenderMail](#), [AppenderPushbullet](#), [AppenderSendmail](#)

---

AppenderDynatrace      *Log to Dynatrace via HTTP*

---

**Description**

Log to Dynatrace via the Dynatrace log ingestion API.

**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](#) -> AppenderDynatrace

**Active bindings**

url a string url

api\_key a string api\_key. Also referred to as "Api Token"

**Methods****Public methods:**

- [AppenderDynatrace\\$new\(\)](#)
- [AppenderDynatrace\\$set\\_url\(\)](#)
- [AppenderDynatrace\\$set\\_api\\_key\(\)](#)
- [AppenderDynatrace\\$get\\_data\(\)](#)
- [AppenderDynatrace\\$show\(\)](#)
- [AppenderDynatrace\\$flush\(\)](#)

**Method new():**

*Usage:*

```
AppenderDynatrace$new(  
  url,  
  api_key,  
  threshold = NA_integer_,  
  layout = DynatraceLayout,  
  buffer_size = 0,  
  flush_threshold = "error",  
  flush_on_exit = TRUE,  
  flush_on_rotate = TRUE,  
  should_flush = NULL,  
  filters = NULL  
)
```

*Arguments:*

url see section *Fields*

threshold, flush\_threshold, layout, buffer\_size see [lgr::AppenderBuffer](#)

**Method set\_url():**

*Usage:*

```
AppenderDynatrace$set_url(url)
```

**Method set\_api\_key():**

*Usage:*

```
AppenderDynatrace$set_api_key(api_key)
```

**Method get\_data():** Get log as data.frame: Not supported for Dynatrace

*Usage:*

```
AppenderDynatrace$get_data(n = 20L, threshold = NA, result_type = "data.frame")
```

**Method show():** Show log in console: Not supported for Dynatrace

*Usage:*

```
AppenderDynatrace$show(threshold = NA_integer_, n = 20)
```

**Method** flush():

*Usage:*

```
AppenderDynatrace$flush()
```

### See Also

<https://docs.dynatrace.com/docs/analyze-explore-automate/logs/lma-log-ingestion/lma-log-ingestion-via-api>

Other Appenders: [AppenderAWSCloudWatchLog](#), [AppenderDbi](#), [AppenderElasticSearch](#), [AppenderGmail](#), [AppenderPool](#), [AppenderPushbullet](#), [AppenderSendmail](#), [AppenderSyslog](#)

---

AppenderElasticSearch *Log to ElasticSearch*

---

### Description

**NOTE: Maturing;** not yet fully documented but well tested in a production scenario

### 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` -> `AppenderElasticSearch`

### Active bindings

`conn` a [ElasticSearch connection](#)

`index` target ElasticSearch index. May either be:

- a character scalar, or
- a function returning a character scalar

`index_create_body` • character scalar json string (or NULL).

- a function returning a character scalar json string (or NULL) Optional settings, mappings, aliases, etc... in case the target index has to be created by the logger. See <https://www.elastic.co/docs/api/doc/elasticsearch/operation/operation-indices-create>

**Methods****Public methods:**

- [AppenderElasticSearch\\$new\(\)](#)
- [AppenderElasticSearch\\$set\\_conn\(\)](#)
- [AppenderElasticSearch\\$get\\_data\(\)](#)
- [AppenderElasticSearch\\$show\(\)](#)
- [AppenderElasticSearch\\$flush\(\)](#)

**Method new():***Usage:*

```
AppenderElasticSearch$new(
  conn,
  index,
  threshold = NA_integer_,
  layout = ElasticSearchLayout,
  index_create_body = NULL,
  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 [lgr::AppenderBuffer](#) A data data.frame.  
content of index**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: [AppenderAWSCloudWatchLog](#), [AppenderDbi](#), [AppenderDynatrace](#), [AppenderGmail](#), [AppenderPool](#), [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 [lgr::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 uses 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"),
  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**

[lgr::LayoutFormat](#), [lgr::LayoutGlue](#)

Other Appenders: [AppenderAWSCloudWatchLog](#), [AppenderDbi](#), [AppenderDynatrace](#), [AppenderElasticSearch](#), [AppenderPool](#), [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](#)

---

AppenderPool

*Log to databases via pool*

---

**Description**

Log to a database table using a connection pool from the **pool** package. This provides better performance and connection management compared to direct DBI connections, especially for applications with concurrent users. Like AppenderDbi, it does *not* support case sensitive / quoted column names, and you are advised to only use all-lowercase names for custom fields (see . . . argument of [lgr::LogEvent](#)).

**Value**

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

**Benefits of Pooled Connections**

Using connection pools instead of direct DBI connections provides several advantages:

- Connections are reused rather than created for each query
- Connection management is automated (creation, validation, destruction)
- Better handles concurrent requests in multi-user applications
- Improves overall performance by reducing connection overhead

**Buffered Logging**

Like AppenderDbi, AppenderPool supports buffered logging 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`).

**Creating a New Appender**

An AppenderPool 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 when the first LogEvent would be written to that table. It is recommended to create the target table first using an SQL `CREATE TABLE` statement for more control and safety.

**Super classes**

[lgr::Filterable](#) -> [lgr::Appender](#) -> [lgr::AppenderMemory](#) -> AppenderPool

**Active bindings**

pool a [pool connection](#)

close\_on\_exit TRUE or FALSE. Close the pool connection when the Logger is removed? Usually not necessary as pools manage their own lifecycle.

col\_types a named character vector providing information about the column types in the database.

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:**

- [AppenderPool\\$new\(\)](#)
- [AppenderPool\\$set\\_close\\_on\\_exit\(\)](#)
- [AppenderPool\\$set\\_pool\(\)](#)
- [AppenderPool\\$show\(\)](#)
- [AppenderPool\\$flush\(\)](#)

**Method new():**

*Usage:*

```
AppenderPool$new(
  pool,
  table,
  threshold = NA_integer_,
  layout = select_dbi_layout(pool::poolCheckout(pool), table),
  close_on_exit = FALSE,
  buffer_size = 0,
  flush_threshold = "error",
  flush_on_exit = TRUE,
  flush_on_rotate = TRUE,
  should_flush = NULL,
  filters = NULL
)
```

*Arguments:*

pool, table see section *Fields*

threshold, flush\_threshold, layout, buffer\_size see [lgr::AppenderBuffer](#)

**Method set\_close\_on\_exit():**

*Usage:*

```
AppenderPool$set_close_on_exit(x)
```

**Method set\_pool():**

*Usage:*

```
AppenderPool$set_pool(pool)
```

**Method** show():*Usage:*

AppenderPool\$show(threshold = NA\_integer\_, n = 20)

**Method** flush():*Usage:*

AppenderPool\$flush()

**See Also**

Other Appenders: [AppenderAWSCloudWatchLog](#), [AppenderDbi](#), [AppenderDynatrace](#), [AppenderElasticSearch](#), [AppenderGmail](#), [AppenderPushbullet](#), [AppenderSendmail](#), [AppenderSyslog](#)

**Examples**

```
if (requireNamespace("RSQLite") && requireNamespace("pool")){
  pool <- pool::dbPool(
    drv = RSQLite::SQLite(),
    dbname = ":memory:"
  )

  app <- AppenderPool$new(
    pool = pool,
    table = "log"
  )

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

  invisible(lg$config(NULL)) # cleanup
  pool::poolClose(pool)
}
```

---

AppenderPushbullet      *Send push-notifications via RPushbullet*

---

**Description**

Send push notifications via **Pushbullet**. This Appender keeps an in-memory buffer like [lgr::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 [lgr::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**

[lgr::LayoutFormat](#), [lgr::LayoutGlue](#)

Other Appenders: [AppenderAWSCloudWatchLog](#), [AppenderDbi](#), [AppenderDynatrace](#), [AppenderElasticSearch](#), [AppenderGmail](#), [AppenderPool](#), [AppenderSendmail](#), [AppenderSyslog](#)

Other Digest Appenders: [AppenderDigest](#), [AppenderMail](#), [AppenderSendmail](#)

**Examples**

```
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))
}
```

---

AppenderSendmail      *Send emails via sendmailR*

---

### 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 `lgr::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**

[lgr::LayoutFormat](#), [lgr::LayoutGlue](#)

Other Appenders: [AppenderAWSCloudWatchLog](#), [AppenderDbi](#), [AppenderDynatrace](#), [AppenderElasticSearch](#), [AppenderGmail](#), [AppenderPool](#), [AppenderPushbullet](#), [AppenderSyslog](#)

Other Digest Appenders: [AppenderDigest](#), [AppenderMail](#), [AppenderPushbullet](#)

**Examples**

```

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

[lgr::LayoutFormat](#), [lgr::LayoutGlue](#)

Other Appenders: [AppenderAWSCloudWatchLog](#), [AppenderDbi](#), [AppenderDynatrace](#), [AppenderElasticSearch](#), [AppenderGmail](#), [AppenderPool](#), [AppenderPushbullet](#), [AppenderSendmail](#)

**Examples**

```

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'"))

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

```

---

as_unix_epoch_ms	<i>Convert POSIXct to Unix Epoch milliseconds</i>
------------------	---

---

**Description**

Convert POSIXct to Unix Epoch milliseconds

**Usage**

```
as_unix_epoch_ms(x)
```

**Arguments**

x	a POSIXct vector
---	------------------

---

LayoutDbi	<i>Format log events for output to databases</i>
-----------	--

---

**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 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 [lgr::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 be used as a Layout by an [lgr::Appender](#).

**Database Specific Layouts**

Different databases have different data types and features. Currently the following LayoutDbi subclasses 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(lobj$col_types)`)

### Methods

#### Public methods:

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

#### Method `new()`:

*Usage:*

```
LayoutDbi$new(  
  col_types = c(level = "integer", timestamp = "timestamp", logger = "varchar(256)",  
    caller = "varchar(256)", msg = "varchar(2048)", rawMsg = "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,  
  format_data = data.table::as.data.table  
)
```

**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:*

`deep` Whether to make a deep clone.

**See Also**

[select\\_dbi\\_layout\(\)](#), [DBI::DBI](#),

Other Layout: [LayoutElasticSearch](#)

---

LayoutElasticSearch    *Format log events for output to ElasticSearch*

---

### 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 be 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](#)

---

<code>select_dbi_layout</code>	<i>Automatically select appropriate layout for logging to a database</i>
--------------------------------	--

---

### 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_dbi_layout(conn, table, ...)`

### Arguments

<code>conn</code>	a <a href="#">DBI connection</a>
<code>table</code>	a character scalar. The name of the table to log to.
<code>...</code>	passed on to the appropriate <code>LayoutDbi</code> subclass constructor.

---

<code>Serializer</code>	<i>Serializers</i>
-------------------------	--------------------

---

### 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:*

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

**Method** `serialize()`:

*Usage:*

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

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

*Usage:*

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

*Arguments:*

`deep` Whether to make a deep clone.

**Examples**

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

---

transform\_event\_dynatrace

*A json layout for Dynatrace ingestion*

---

**Description**

Transforms a [lgr::LogEvent](#) object into a list suitable for Dynatrace ingestion.

**Usage**

```
transform_event_dynatrace(event)
```

DynatraceLayout

**Arguments**

event            A [lgr::LogEvent](#) object.

**Format**

An object of class LayoutJson (inherits from Layout, R6) of length 17.

**Value**

A list of key-value pairs ready to be serialized to JSON for Dynatrace.

**See Also**

<https://docs.dynatrace.com/docs/discover-dynatrace/references/semantic-dictionary/fields#service>

---

transform\_event\_elasticsearch

*A json layout for Elasticsearch ingestion*

---

**Description**

Transforms a [lgr::LogEvent](#) object into a list suitable for Elasticsearch ingestion.

**Usage**

```
transform_event_elasticsearch(event)
```

ElasticSearchLayout

**Arguments**

event            A `lgr::LogEvent` object.

**Format**

An object of class `LayoutJson` (inherits from `Layout`, R6) of length 17.

**Value**

A list of key-value pairs ready to be serialized to JSON for ElasticSearch

**See Also**

<https://www.elastic.co/docs/reference/ecs>

---

unpack\_json\_cols            *Unserialize data frame columns that contain JSON*

---

**Description**

Unserialize data frame columns that contain JSON

**Usage**

```
unpack_json_cols(x, cols)

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

## S3 method for class 'data.frame'
unpack_json_cols(x, 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**

```
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**
  - AppenderAWSCloudWatchLog, [2](#)
  - AppenderDbi, [5](#)
  - AppenderDynatrace, [8](#)
  - AppenderElasticSearch, [10](#)
  - AppenderGmail, [12](#)
  - AppenderPool, [15](#)
  - AppenderPushbullet, [17](#)
  - AppenderSendmail, [20](#)
  - AppenderSyslog, [22](#)
- \* **Digest Appenders**
  - AppenderDigest, [7](#)
  - AppenderMail, [13](#)
  - AppenderPushbullet, [17](#)
  - AppenderSendmail, [20](#)
- \* **Layout**
  - LayoutDbi, [24](#)
  - LayoutElasticSearch, [27](#)
- \* **abstract classes**
  - AppenderDigest, [7](#)
  - AppenderMail, [13](#)
- \* **database layouts**
  - LayoutDbi, [24](#)
- \* **datasets**
  - transform\_event\_dynatrace, [30](#)
  - transform\_event\_elasticsearch, [30](#)
- AppenderAWSCloudWatchLog, [2, 7, 10, 12, 13, 17, 19, 21, 23](#)
- AppenderDbi, [4, 5, 10, 12, 13, 17, 19, 21, 23–25, 28](#)
- AppenderDigest, [7, 15, 19, 21](#)
- AppenderDynatrace, [4, 7, 8, 12, 13, 17, 19, 21, 23](#)
- AppenderElasticSearch, [4, 7, 10, 10, 13, 17, 19, 21, 23](#)
- AppenderGmail, [4, 7, 10, 12, 12, 17, 19, 21, 23](#)
- AppenderMail, [8, 13, 13, 19–21](#)
- AppenderPool, [4, 7, 10, 12, 13, 15, 19, 21, 23](#)
- AppenderPushbullet, [4, 7, 8, 10, 12, 13, 15, 17, 17, 17, 21, 23](#)
- AppenderSendmail, [4, 7, 8, 10, 12, 13, 15, 17, 19, 20, 23](#)
- AppenderSyslog, [4, 7, 10, 12, 13, 17, 19, 21, 22](#)
- as\_unix\_epoch\_ms, [24](#)
- DBI connection, [6, 28](#)
- DBI::DBI, [26](#)
- DBI::Id, [6, 16](#)
- DynatraceLayout
  - (transform\_event\_dynatrace), [30](#)
- ElasticSearch connection, [10](#)
- ElasticSearchLayout
  - (transform\_event\_elasticsearch), [30](#)
- gmailr::gm\_send\_message(), [12](#)
- jsonlite::toJSON(), [27](#)
- LayoutDb2 (LayoutDbi), [24](#)
- LayoutDbi, [5, 24, 28](#)
- LayoutElasticSearch, [26, 27](#)
- LayoutMySQL (LayoutDbi), [24](#)
- LayoutPostgres (LayoutDbi), [24](#)
- LayoutRjdbc (LayoutDbi), [24](#)
- LayoutRjdbcDb2 (LayoutDbi), [24](#)
- LayoutSqlite (LayoutDbi), [24](#)
- lgr log levels, [23](#)
- lgr::Appender, [2, 5, 8, 10, 12, 13, 15, 17, 18, 20, 22, 24, 27](#)
- lgr::AppenderBuffer, [3, 6, 9, 11, 12, 16–18, 20](#)
- lgr::AppenderMemory, [2, 5, 8, 10, 12, 13, 15, 18, 20](#)
- lgr::as.data.table.LogEvent, [24](#)
- lgr::Filterable, [2, 5, 8, 10, 12, 13, 15, 18, 20, 22](#)

`lgr::Layout`, [8](#), [24](#), [25](#), [27](#)  
`lgr::LayoutFormat`, [8](#), [13](#), [18](#), [19](#), [21](#), [23](#), [25](#)  
`lgr::LayoutGlue`, [8](#), [13](#), [19](#), [21](#), [23](#)  
`lgr::LayoutJson`, [27](#)  
`lgr::log_levels`, [22](#)  
`lgr::LogEvent`, [5](#), [8](#), [15](#), [27](#), [30](#), [31](#)  
`lgr::Logger`, [2](#), [5](#), [8](#), [10](#), [12](#), [15](#), [17](#), [20](#), [22](#)  
`lgrExtra::AppenderDigest`, [12](#), [13](#), [18](#), [20](#)  
`lgrExtra::AppenderMail`, [12](#), [20](#)  
`lgrExtra::Serializer`, [29](#)

`paws.management.cloudwatchlogs.client`,  
[3](#)

`pool.connection`, [16](#)

`R6::R6`, [2](#), [5](#), [8](#), [10](#), [12](#), [15](#), [17](#), [20](#), [22](#), [24](#), [25](#),  
[27](#), [28](#)

`RPushbullet::pbPost`, [18](#)  
`RPushbullet::pbPost()`, [17](#), [18](#)  
`rsyslog::syslog()`, [23](#)

`select_dbi_layout`, [28](#)  
`select_dbi_layout()`, [5](#), [25](#), [26](#)  
`sendmailR::sendmail()`, [20](#)  
`Serializer`, [28](#)  
`SerializerJson(Serializer)`, [28](#)

`transform_event_dynatrace`, [30](#)  
`transform_event_elasticsearch`, [30](#)

`unpack_json_cols`, [31](#)