Package 'dataexcept'

November 29, 2026

Title Structured, Classed Failures that Cross Language Boundaries

Version 0.4.2

Description Classed error conditions for data and modelling workflows that carry structured fields and machine-readable recovery metadata (whether a failure is transient or permanent, and whether a retry can succeed), and that serialise to the language-neutral failure envelope published by the 'DataExcept' 'Python' package, so failures written by 'R' and 'Python' code share one format. Envelopes from either language can be read back into 'R' conditions. Credentials in URLs are redacted before they reach a condition or an envelope. A companion layer gives classes to the warnings base 'R' and 'stats' signal with only a message, recognising them through the translated message catalogue so the match holds in any session language.

License MIT + file LICENSE

Encoding UTF-8

Language en-GB

RoxygenNote 7.3.3

Depends R (>= 4.1)

Imports jsonlite, stats, utils

Suggests knitr, lme4, otel (>= 0.2.0), otelsdk, rlang, rmarkdown, spelling, survival, testthat (>= 3.1.7)

Config/testthat/edition 3

VignetteBuilder knitr

URL https://github.com/DiogoRibeiro7/r-dataexcept, https://diogoribeiro7.github.io/r-dataexcept/

BugReports https://github.com/DiogoRibeiro7/r-dataexcept/issues

NeedsCompilation no

Author Diogo Ribeiro [aut, cre, cph] (ORCID: <https://orcid.org/0009-0001-2022-7072>)

Maintainer Diogo Ribeiro <dfr@esmad.ipp.pt>

Repository CRAN

Date/Publication 2026-11-29 11:00:08 UTC

Contents

condition_failure . . . 2
condition_group . . . 3
condition_to_envelope . . . 5
condition_to_event . . . 7
condition_to_otel_attributes . . . 8
condition_to_pino . . . 10
dataexcept_classes . . . 12
data_engineering_errors . . . 12
data_frame_errors . . . 14
data_quality_errors . . . 15
envelope_schema . . . 17
envelope_to_condition . . . 18
failure_metadata . . . 19
file_errors . . . 20
job_errors . . . 21
modelling_errors . . . 23
model_quality_errors . . . 24
new_dataexcept_error . . . 26
operation_context . . . 28
operation_context_from_otel . . . 29
pino_schema . . . 30
pipeline_errors . . . 31
record_otel_exception . . . 33
redact_url . . . 35
service_errors . . . 36
validate_envelope . . . 37
validation_error . . . 38
with_classed_warnings . . . 39
wrap_errors . . . 40

Index **42**

`condition_failure` *Read the failure metadata of a condition*

Description

`condition_failure()` returns the recovery metadata carried by a dataexcept error, or by a condition read back from an envelope that had a `failure` record. Conditions dataexcept did not classify – a base R `simpleError`, an rlang error, or a third-party exception in an envelope – return `NULL`.

Usage

```
condition_failure(cnd)

is_retryable(cnd)
```

Arguments

cnd	A condition object.

Details

`is_retryable()` is the common question: `TRUE` or `FALSE` when the failure is classified, `NA` when it is not.

Value

`condition_failure()` returns a `dataexcept_failure_metadata` object or `NULL`. `is_retryable()` returns `TRUE`, `FALSE` or `NA`.

Examples

```
condition_failure(validation_error("age", -1))
is_retryable(validation_error("age", -1))
is_retryable(simpleError("boom"))
```

---

condition_group *Report several failures as one*

Description

`condition_group()` combines several conditions into one error, so that a check which finds many problems can report all of them instead of stopping at the first. It is written to an envelope as an exception group – the `exceptions` array the schema defines, which Python's `ExceptionGroup` also produces – with every member rendered in full, chains included.

Usage

```
condition_group(
  conditions,
  message = NULL,
  ...,
  class = character(),
  parent = NULL,
  call = NULL,
  failure = NULL
)
```

```
group_members(cnd)

collect_errors(...)
```

Arguments

conditions	A non-empty list of conditions.
message	A short description of what failed, or NULL. The count of failures is appended: "orders failed validation (3 failures)".
...	Named fields to store on the group, as in new_dataexcept_error().
class	Additional R classes, most specific first.
parent	The condition that caused the whole group, or NULL.
call	The call to report, or NULL.
failure	Failure metadata from failure_metadata(), or NULL to derive it from the members.
cnd	A condition.

Details

collect_errors() is the way to gather the members: it evaluates each of its arguments in turn, catches any error one of them signals, and carries on with the next. group_members() returns the members of a group, whether it was built here or read from an envelope.

A group is caught by a dataexcept_condition_group handler, and by dataexcept_error like every other dataexcept error. A group read from an envelope – including a Python ExceptionGroup – has the dataexcept_condition_group class too.

The group's failure metadata, unless given, is its members' when they all agree, and unclassified otherwise: a batch of permanent validation failures is permanent, but a mix of transient and permanent failures has no single answer to whether a retry would help. When the members agree and some give a retry delay, the longest one is used.

Value

condition_group() returns a condition of class dataexcept_condition_group. collect_errors() returns a list of the errors caught, named after the arguments that signalled them when those are named, and empty when there were none. group_members() returns a list of conditions, empty for a condition that is not a group.

Examples

```
orders <- data.frame(id = 1:3, amount = c("10", "20", "30"))

require_column <- function(df, column) {
  if (!column %in% names(df)) stop(missing_column_error(column, dataframe = "orders"))
}
require_numeric <- function(df, column) {
  if (!is.numeric(df[[column]])) {
    stop(dtype_mismatch_error(column, "numeric", class(df[[column]])[1]))
```

```
  }
}

errors <- collect_errors(
  require_column(orders, "customer_id"),
  require_column(orders, "amount"),
  require_numeric(orders, "amount")
)
length(errors)

err <- tryCatch(
  if (length(errors) > 0) stop(condition_group(errors, "orders failed validation")),
  dataexcept_condition_group = identity
)
conditionMessage(err)
vapply(group_members(err), function(e) class(e)[1], character(1))
cat(condition_to_json(err, pretty = TRUE))
```

`condition_to_envelope` *Write a condition as a DataExcept envelope*

Description

`condition_to_envelope()` renders a condition, and the chain of conditions behind it, as a DataExcept envelope: the language-neutral failure record the Python DataExcept package defines in its published JSON Schema (`envelope_schema()`). `condition_to_json()` encodes it as strict JSON.

Usage

```
condition_to_envelope(cnd, include_attributes = TRUE, max_depth = 8L)

condition_to_json(
  cnd,
  include_attributes = TRUE,
  max_depth = 8L,
  pretty = FALSE
)

condition_type(cnd)
```

Arguments

`cnd`	A condition object.
`include_attributes`	Include the `attributes` field?
`max_depth`	How many levels of chained conditions to render. A cause beyond this depth is replaced by the truncation marker `{"truncated": true}`.
`pretty`	Indent the JSON for reading?

Details

Any condition can be written, not only dataexcept ones:

- `type` is the dataexcept type, or else the condition's most specific R class (`simpleError`, `rlang_error`, ...).
- `module` is `"dataexcept"` for dataexcept conditions, `"base"` for base R's own condition classes, the package for a class named with its package prefix (`rlang_error` belongs to rlang), and `"unknown"` where the package cannot be established.
- `message` is the condition message.
- `failure` appears on dataexcept errors only. A condition dataexcept did not classify has no `failure` field, and that absence is information: it means unclassified, not "not retryable".
- `attributes` holds the condition's fields, in a JSON-safe form. The message, call, parent and backtrace are not attributes, nor is any field whose name begins with `.` or `_`.
- `cause` is the condition in the `parent` field – the field dataexcept and rlang both use for a chained cause – rendered the same way.

Text is redacted on the way out: every URL in the message, in attribute values and in attribute names loses its credentials and its path.

A vector of length one is written as a scalar. Wrap a field in `I()` when it is a list by meaning, so that it is written as an array even with a single element – the convention jsonlite uses. The constructors do this for the fields the Python package always writes as lists, such as the `expected` types of `dtype_mismatch_error()`.

Values that JSON cannot represent degrade to a description of themselves rather than disappearing: a data frame becomes `"<data.frame: 150 rows x 5 columns>"`, a function `"<function>"`, a vector longer than 100 elements a description of its type and length, and `NaN` or `Inf` the strings `"nan"` or `"inf"`, as the Python package writes them. `NA` becomes `null`.

Value

`condition_to_envelope()` returns the envelope as a list (JSON objects as named lists, arrays as unnamed lists, `null` as `NULL`). `condition_to_json()` returns a single string.

`condition_type()` returns the `type` a condition is written with: the dataexcept type when there is one – including the type of a condition read from an envelope – and otherwise the condition's most specific R class.

See Also

`envelope_to_condition()` for the reverse direction.

Examples

```
cnd <- tryCatch(
  stop(missing_column_error("customer_id", dataframe = "orders")),
  error = identity
)
cat(condition_to_json(cnd, pretty = TRUE))
```

```
# A chain: the base R error becomes the cause.
cnd <- tryCatch(
  tryCatch(
    read.csv(tempfile()),
    warning = function(w) stop(file_read_error("orders.csv", parent = w))
  ),
  error = identity
)
str(condition_to_envelope(cnd), max.level = 2)
```

`condition_to_event` *A failure event for logs and observability tools*

Description

`condition_to_event()` pairs the envelope of a condition with the operation context it happened in, in the event format the Python package's `exception_to_observability_event()` produces:

Usage

```
condition_to_event(
  cnd,
  operation_context = NULL,
  include_attributes = TRUE,
  max_depth = 8L
)

condition_to_event_json(
  cnd,
  operation_context = NULL,
  include_attributes = TRUE,
  max_depth = 8L,
  pretty = FALSE
)
```

Arguments

`cnd`	A condition object.
`operation_context`	An operation context from `operation_context()`, or `NULL`.
`include_attributes`	Include the `attributes` field?
`max_depth`	How many levels of chained conditions to render. A cause beyond this depth is replaced by the truncation marker `{"truncated": true}`.
`pretty`	Indent the JSON for reading?

Details

```
{"event": "exception", "exception": {...envelope...}, "operation": {...}}
```

The envelope stays exactly as `condition_to_envelope()` writes it, redacted and with its chain of causes. The operation context sits under its own key instead of being flattened into the failure, so a log pipeline can index the label fields and keep the run identifiers for correlation. With no context, or an empty one, there is no `operation` key: nothing is invented.

`condition_to_event_json()` writes the event as one line of strict JSON, ready for a JSON-lines log.

Value

`condition_to_event()` returns the event as a list; `condition_to_event_json()` returns it as a single JSON string.

Examples

```
context <- operation_context(
  system = "batch", operation = "load_orders", job_id = "job-42"
)
err <- tryCatch(
  stop(missing_column_error("customer_id", dataframe = "orders")),
  error = identity
)
cat(condition_to_event_json(err, operation_context = context, pretty = TRUE))

# Log every failure of a job as one JSON line.
log_file <- tempfile(fileext = ".jsonl")
tryCatch(
  stop(validation_error("age", -1)),
  dataexcept_error = function(e) {
    cat(condition_to_event_json(e, operation_context = context), "\n",
      file = log_file, append = TRUE, sep = ""
    )
  }
)
readLines(log_file)
```

---

condition_to_otel_attributes

OpenTelemetry attributes for a condition

Description

`condition_to_otel_attributes()` describes a condition as OpenTelemetry exception attributes, with the names the Python package's `exception_to_otel_attributes()` uses, so spans from R and Python report failures the same way:

Usage

```
condition_to_otel_attributes(
  cnd,
  operation_context = NULL,
  include_attributes = TRUE,
  max_depth = 8L,
  include_stacktrace = TRUE,
  include_envelope = FALSE
)
```

Arguments

cnd	A condition object.
operation_context	An operation context from operation_context(), or NULL.
include_attributes	Include the attributes field?
max_depth	How many levels of chained conditions to render. A cause beyond this depth is replaced by the truncation marker {"truncated": true}.
include_stacktrace	Include exception.stacktrace when the condition has one?
include_envelope	Include the whole envelope as a JSON attribute?

Details

- exception.type, exception.message and, when available, exception.stacktrace: the standard OpenTelemetry exception attributes.
- dataexcept.failure.kind, dataexcept.failure.retryable and dataexcept.failure.retry_after_seconds: the failure metadata, flat and typed, present only for what the condition's failure record carries.
- dataexcept.operation.*: the operation context's labels and run identifiers. trace_id and span_id are left out, because a span already carries them natively.
- With include_envelope = TRUE, dataexcept.envelope (the envelope as compact JSON with sorted keys) and dataexcept.envelope.schema (its schema id).

exception.type is the envelope's module and type, such as "dataexcept.MissingColumnError" or "base.simpleError"; for a failure read from a Python envelope it is the Python class's qualified name. The message is the envelope's, redacted. The stack trace is an rlang backtrace when the condition has one, and otherwise the condition's call, which is the only stack information a base R condition carries; either is redacted. A condition with neither, such as one read from an envelope, has no stack trace attribute.

This is a pure conversion: invalid input is an error. record_otel_exception() is the fail-open version for use while handling a failure.

Value

A named list of attributes, each a single string, logical or number, ready for span$record_exception() or span$add_event().

See Also

`record_otel_exception()` to record a condition on a span.

Examples

```
context <- operation_context(system = "batch", operation = "settle_invoices")
err <- with_failure_metadata(
  api_error("https://user:pw@api.example.com/v1/invoices", status_code = 503L),
  failure_metadata("transient", retryable = TRUE, retry_after_seconds = 30)
)
str(condition_to_otel_attributes(err, operation_context = context))
```

---

condition_to_pino *Write a condition in the shape Pino logs errors*

Description

A Node.js service that logs with Pino expects an error under its `err` key with `type`, `message` and, optionally, `stack` – the fields Pino itself writes for an error, and the ones pino-pretty, transports and error trackers read. `condition_to_pino()` writes a condition in that shape: the Pino profile the Python DataExcept package publishes (`pino-1.0.0.json`, see `pino_schema()`). `envelope_to_pino()` does the same for an envelope already in hand, such as one read from a queue.

Usage

```
condition_to_pino(
  cnd,
  include_attributes = TRUE,
  max_depth = 8L,
  include_stack = FALSE
)

condition_to_pino_json(
  cnd,
  include_attributes = TRUE,
  max_depth = 8L,
  include_stack = FALSE,
  pretty = FALSE
)

envelope_to_pino(envelope, stack = NULL)
```

Arguments

`cnd`	A condition object.
`include_attributes`	Include the `attributes` field?

max_depth	How many levels of chained conditions to render. A cause beyond this depth is replaced by the truncation marker {"truncated": true}.
include_stack	Write the condition's rlang backtrace, when it has one, as stack?
pretty	Indent the JSON for reading?
envelope	An envelope: a list from condition_to_envelope() or jsonlite::parse_json(), or a JSON string.
stack	A stack received with the envelope, as a single string, or NULL. It is redacted.

Details

The profile is a projection of the envelope, not a replacement for it, and it differs from the envelope in three places only:

- an exception group's exceptions are errors, the name JavaScript's AggregateError uses;
- stack is written only when there is a real stack to write, and never made up;
- attributes stay nested, so that an attribute called type or stack cannot overwrite the fields a consumer reads.

Everything else – type, module, message, failure, the causes, contexts and members, the cycle and truncation markers, and the redaction already applied to the envelope – is carried through unchanged.

In R, the real stack is an rlang backtrace: the one rlang::abort() records, or that rlang::global_entrace() adds to base R errors. With include_stack = TRUE it is written, redacted like every other exported string, and without one stack is left out. A call is not a stack, and is not written as one.

Value

condition_to_pino() and envelope_to_pino() return the error as a list (JSON objects as named lists, arrays as unnamed lists, null as NULL). condition_to_pino_json() returns a single string of JSON.

See Also

condition_to_envelope() for the envelope itself, which stays the canonical record.

Examples

```
cnd <- tryCatch(
  stop(condition_group(list(
    missing_column_error("customer_id", dataframe = "orders"),
    validation_error("amount", -5)
  ), "orders failed validation")),
  error = identity
)
cat(condition_to_pino_json(cnd, pretty = TRUE))

# A Pino log line, for a log stream a Node.js service reads.
line <- sprintf(
  '{"level":50,"time":%.0f,"msg":"ingest failed","err":%s}',
```

```
  as.numeric(Sys.time()) * 1000,
  condition_to_pino_json(cnd)
)

# An envelope received as JSON.
json <- condition_to_json(api_error("https://api.example.com/v1", status_code = 503L))
str(envelope_to_pino(json))
```

`dataexcept_classes` *The dataexcept condition classes*

Description

Lists every condition type dataexcept defines: its envelope `type`, the R class a handler catches it by, its parent, whether it is an error or a warning, its default failure classification, and whether the same type exists in the Python package.

Usage

```
dataexcept_classes()
```

Details

A handler can catch a whole family through a parent class: a `tryCatch(dataexcept_data_frame_error = ...)` handler catches a missing column, a type mismatch and a merge-key failure alike, and `dataexcept_error` catches every error the package defines.

Value

A data frame with one row per type.

Examples

```
dataexcept_classes()[, c("type", "class", "parent")]
```

`data_engineering_errors`
Data engineering failures

Description

Errors raised by batch data processing: a transformation step that failed, an ETL job that did not complete, and a batch that could not be processed. All inherit from `dataexcept_data_engineering_error`.

Usage

```
data_transformation_error(step, details = NULL, parent = NULL, call = NULL)

etl_job_error(job_name, message = NULL, parent = NULL, call = NULL)

batch_processing_error(batch_id, parent = NULL, call = NULL)
```

Arguments

step	Name of the transformation step.
details	Optional detail, appended to the message.
parent	The condition that caused this one, or NULL.
call	The call to report, or NULL.
job_name	Name of the ETL job.
message	A message to use instead of the default.
batch_id	Identifier of the batch.

Value

A condition inheriting from dataexcept_data_engineering_error.

See Also

Other dataexcept errors: data_frame_errors, data_quality_errors, file_errors, job_errors, model_quality_errors, modelling_errors, pipeline_errors, service_errors, validation_error()

Examples

```
process_batch <- function(id, rows) {
  tryCatch(
    sum(rows),
    error = function(e) stop(batch_processing_error(id, parent = e))
  )
}
err <- tryCatch(process_batch("2026-09-27", list("a", 1)), error = identity)
conditionMessage(err)

data_transformation_error("normalise_amounts", details = "12 rows had no currency")
```

`data_frame_errors` *Data frame failures*

Description

Errors about the shape of a data frame: a required column that is absent, a column of the wrong type, and a join whose keys do not line up. All three inherit from `dataexcept_data_frame_error`, so one handler can catch the family.

Usage

```
missing_column_error(column, dataframe = NULL, parent = NULL, call = NULL)

dtype_mismatch_error(column, expected, found, parent = NULL, call = NULL)

merge_key_error(left_keys, right_keys, parent = NULL, call = NULL)
```

Arguments

`column`	Name of the column.
`dataframe`	Optional name of the data frame, for the message.
`parent`	The condition that caused this one, or `NULL`.
`call`	The call to report, or `NULL`.
`expected`	Character vector of acceptable types, such as `c("numeric", "integer")`.
`found`	The type actually found, typically `class(x)[1]`.
`left_keys, right_keys`	Character vectors of join keys.

Value

A condition inheriting from `dataexcept_data_frame_error`.

See Also

Other dataexcept errors: `data_engineering_errors`, `data_quality_errors`, `file_errors`, `job_errors`, `model_quality_errors`, `modelling_errors`, `pipeline_errors`, `service_errors`, `validation_error()`

Examples

```
orders <- data.frame(id = 1:3, amount = c("10", "20", "30"))

require_numeric <- function(df, column) {
  if (!column %in% names(df)) {
    stop(missing_column_error(column, dataframe = "orders"))
  }
  if (!is.numeric(df[[column]])) {
```

```
    stop(dtype_mismatch_error(column, "numeric", class(df[[column]])[1]))
  }
  invisible(df)
}

tryCatch(
  require_numeric(orders, "amount"),
  dataexcept_data_frame_error = function(e) conditionMessage(e)
)
```

`data_quality_errors` *Data quality failures*

Description

Errors about the data itself rather than the code handling it: a file in the wrong format, a schema that does not match, drift from a reference distribution, leakage between training and test data, classes too imbalanced to learn from, and outlier detection that could not run. All inherit from `dataexcept_data_science_error`.

Usage

```
data_format_error(expected_formats, found_format, parent = NULL, call = NULL)

schema_mismatch_error(expected, found, parent = NULL, call = NULL)

data_drift_error(
  feature,
  drift_score,
  message = NULL,
  parent = NULL,
  call = NULL
)

data_leakage_error(feature, stage, message = NULL, parent = NULL, call = NULL)

data_imbalance_error(
  ratio,
  threshold,
  message = NULL,
  parent = NULL,
  call = NULL
)

outlier_detection_error(method, details = NULL, parent = NULL, call = NULL)
```

Arguments

expected_formats	Character vector of the formats accepted, such as c("csv", "parquet").
found_format	The format found.
parent	The condition that caused this one, or NULL.
call	The call to report, or NULL.
expected, found	Descriptions of the schema expected and the schema found, each a single string.
feature	Name of the feature concerned.
drift_score	The drift statistic, written to four decimal places in the default message.
message	A message to use instead of the default.
stage	The stage at which the leak was found, such as "cross-validation".
ratio	The ratio of the smallest class to the largest.
threshold	The smallest ratio accepted.
method	The name of the detection method.
details	Optional detail, appended to the message.

Details

Their failure metadata is "unknown" by default: the same data would fail again, but whether the data will change is for the caller to say.

Value

A condition inheriting from dataexcept_data_science_error.

See Also

Other dataexcept errors: data_engineering_errors, data_frame_errors, file_errors, job_errors, model_quality_errors, modelling_errors, pipeline_errors, service_errors, validation_error()

Examples

```
check_balance <- function(y, threshold = 0.2) {
  counts <- table(y)
  ratio <- min(counts) / max(counts)
  if (ratio < threshold) stop(data_imbalance_error(ratio, threshold))
  invisible(y)
}
tryCatch(
  check_balance(c(rep("no", 95), rep("yes", 5))),
  dataexcept_data_imbalance_error = function(e) conditionMessage(e)
)

err <- data_drift_error("income", drift_score = 0.4172)
err$drift_score
```

```
schema_mismatch_error(
  expected = "id: integer, amount: numeric",
  found = "id: integer, amount: character"
)
```

`envelope_schema` *The envelope JSON Schema*

Description

dataexcept writes and reads envelopes that follow the JSON Schema published by the Python DataExcept package, `envelope-1.0.0.json` (JSON Schema draft 2020-12). The schema is versioned independently of either package: it describes the payload, and within 1.x a later version may add fields but does not change what an established field means. Consumers must ignore fields they do not recognise.

Usage

```
envelope_schema()

envelope_schema_version()

envelope_schema_id()
```

Details

A copy of the schema ships with this package, together with the reference fixtures the Python package generates with its own writer; the test suite reads every fixture and writes it back unchanged.

Value

`envelope_schema()` returns the schema as a list. `envelope_schema_version()` returns the schema version, and `envelope_schema_id()` its `$id`, each a single string.

Examples

```
envelope_schema_version()
envelope_schema_id()
names(envelope_schema()$`$defs`)

# The file itself, for a JSON Schema validator:
system.file("schema", "envelope-1.0.0.json", package = "dataexcept")
```

envelope_to_condition *Read a DataExcept envelope back into an R condition*

Description

envelope_to_condition() turns an envelope – written by this package or by the Python DataExcept package, or by any producer that follows the published schema – into an R condition that can be inspected, handled or re-signalled.

Usage

```
envelope_to_condition(x, max_depth = 32L)
```

Arguments

x	An envelope: a JSON string, or a list as returned by condition_to_envelope() or jsonlite::parse_json().
max_depth	The deepest chain of nested records to accept.

Details

The condition always has class dataexcept_remote_condition, plus error (or warning, for a warning written by this package) and condition. When the envelope names a dataexcept type, the condition also gets that type's R classes, whichever language wrote it: a Python MissingColumnError becomes a dataexcept_missing_column_error, so an R handler written for local failures catches remote ones too. A type from a DataExcept module that R has no class for, such as Python's ServiceTimeoutError, is still a dataexcept_error, since every DataExcept exception derives from DataExceptError; condition_type() gives its type.

The envelope's attributes become fields (cnd$column), except any that would collide with message, call or parent. Its cause becomes parent, so the chain can be walked and printed as usual.

Written back with condition_to_envelope(), a condition read here reproduces the envelope it came from, including its attributes, failure record, context, group members and cycle and truncation markers.

Envelopes may come from outside the process, so reading is bounded: a chain of causes, contexts or group members nested more than max_depth levels deep is rejected before it is walked, and JSON text nested deeper than the envelope could legitimately need is rejected before it is parsed. Envelopes written with either package's default depth (8) are far inside the limit.

Value

A condition object.

See Also

validate_envelope(), which this function applies first.

Examples

```
# An envelope written by the Python package, one of the reference
# fixtures shipped with dataexcept.
path <- system.file("schema", "fixtures", "ordinary-exception.json", package = "dataexcept")
json <- paste(readLines(path), collapse = "\n")
cat(json)

cnd <- envelope_to_condition(json)
class(cnd)
cnd$field
is_retryable(cnd)

tryCatch(stop(cnd), dataexcept_validation_error = function(e) "caught")
```

`failure_metadata` *Describe whether a failure can be recovered from*

Description

Failure metadata is the machine-readable part of a dataexcept error: whether the underlying condition is known to be transient, permanent for the same operation and payload, or unclassified, and whether retrying the same operation can succeed. It describes the failure; it does not prescribe a retry policy, which stays with the calling code.

Usage

```
failure_metadata(
  kind = "unknown",
  retryable = NULL,
  retry_after_seconds = NULL
)

with_failure_metadata(cnd, metadata)
```

Arguments

`kind`	One of `"unknown"`, `"transient"` or `"permanent"`.
`retryable`	`TRUE`, `FALSE`, or `NULL` when no classification is warranted. `NULL` is deliberately distinct from `FALSE`.
`retry_after_seconds`	How long to wait before retrying, when the backend said: a finite, non-negative number, or `NULL`.
`cnd`	A condition object.
`metadata`	A `dataexcept_failure_metadata` object from `failure_metadata()`.

Details

Every dataexcept error carries metadata. The class default is deliberately conservative (`"unknown"`, retryable `NULL`), and a class receives a stronger default only where it can be defended without knowing the backend: a `validation_error()` is permanent and not retryable for the same unchanged payload. Code that does know the backend's answer attaches it with `with_failure_metadata()`.

Value

`failure_metadata()` returns an object of class `dataexcept_failure_metadata`: a list with elements `kind`, `retryable` and `retry_after_seconds`.

`with_failure_metadata()` returns `cnd` with the metadata attached.

See Also

`condition_failure()` to read the metadata back from a condition.

Examples

```
failure_metadata("transient", retryable = TRUE, retry_after_seconds = 2)

err <- api_error("https://api.example.com/v1/orders", status_code = 503L)
err <- with_failure_metadata(err, failure_metadata("transient", TRUE, 30))
is_retryable(err)
```

---

file_errors *File failures*

Description

Errors reading or writing a file. Both inherit from `dataexcept_file_error`. The underlying condition, typically the error or warning from the base R function that failed, goes in `parent`, and its message is appended to this one.

Usage

```
file_read_error(path, parent = NULL, call = NULL)

file_write_error(path, parent = NULL, call = NULL)
```

Arguments

`path`	The file path. A URL is redacted.
`parent`	The condition that caused this one, or `NULL`.
`call`	The call to report, or `NULL`.

Value

A condition inheriting from `dataexcept_file_error`.

See Also

Other dataexcept errors: data_engineering_errors, data_frame_errors, data_quality_errors, job_errors, model_quality_errors, modelling_errors, pipeline_errors, service_errors, validation_error()

Examples

```
read_orders <- function(path) {
  tryCatch(
    read.csv(path),
    error = function(e) stop(file_read_error(path, parent = e)),
    warning = function(w) stop(file_read_error(path, parent = w))
  )
}
tryCatch(
  read_orders(tempfile(fileext = ".csv")),
  dataexcept_file_error = function(e) conditionMessage(e)
)
```

---

job_errors *Job failures*

Description

Errors that stop a job before or while it does its work: credentials that are refused, a permission that is missing, a configuration option that is invalid, a resource that does not exist, and an operation that ran out of time. All inherit from `dataexcept_job_error`, as they inherit from `JobError` in the Python package.

Usage

```
authentication_error(user, message = NULL, parent = NULL, call = NULL)

authorization_error(user, permission, parent = NULL, call = NULL)

configuration_error(option, message = NULL, parent = NULL, call = NULL)

resource_not_found_error(resource_type, identifier, parent = NULL, call = NULL)

operation_timeout_error(operation, timeout, parent = NULL, call = NULL)
```

Arguments

`user`	The user whose credentials or permissions were refused.
`message`	A message to use instead of the default.
`parent`	The condition that caused this one, or `NULL`.
`call`	The call to report, or `NULL`.

`permission`	The permission the user lacks.
`option`	The configuration option that is invalid.
`resource_type`	The kind of resource, such as `"Dataset"`.
`identifier`	The identifier that was looked up.
`operation`	The operation that timed out.
`timeout`	The time limit that elapsed, in seconds.

Details

`authentication_error()` and `authorization_error()` are permanent and not retryable: the same credentials will be refused again. The others are `"unknown"`.

In the Python package `ValidationError` is a `JobError` too. In R, `validation_error()` stays a direct child of `dataexcept_error`, as it has been since the first release, so a `dataexcept_job_error` handler does not catch it.

Value

A condition inheriting from `dataexcept_job_error`.

See Also

Other dataexcept errors: `data_engineering_errors`, `data_frame_errors`, `data_quality_errors`, `file_errors`, `model_quality_errors`, `modelling_errors`, `pipeline_errors`, `service_errors`, `validation_error()`

Examples

```
get_setting <- function(settings, option) {
  value <- settings[[option]]
  if (is.null(value)) stop(configuration_error(option))
  value
}
tryCatch(
  get_setting(list(retries = 3), "timeout"),
  dataexcept_job_error = function(e) conditionMessage(e)
)

err <- resource_not_found_error("Dataset", "sales-2026-q3")
conditionMessage(err)

is_retryable(authorization_error("analyst", "write:reports"))
```

`modelling_errors` *Data and modelling failures*

Description

Errors raised while loading data and fitting or using a model. All inherit from `dataexcept_data_science_error`; `convergence_error()` also inherits from `dataexcept_model_training_error`, so a handler for training failures catches non-convergence too.

Usage

```
data_loading_error(source, parent = NULL, call = NULL)

missing_data_error(feature, message = NULL, parent = NULL, call = NULL)

model_training_error(
  model_type,
  epoch = NULL,
  message = NULL,
  parent = NULL,
  call = NULL
)

convergence_error(
  model_type,
  iterations,
  message = NULL,
  parent = NULL,
  call = NULL
)

prediction_error(
  model_type,
  inputs = NULL,
  message = NULL,
  parent = NULL,
  call = NULL
)
```

Arguments

`source`	The file path or URL the data came from. A URL is redacted.
`parent`	The condition that caused this one, or `NULL`.
`call`	The call to report, or `NULL`.
`feature`	Name of the missing feature.
`message`	A message to use instead of the default.

`model_type`	The model, as a name such as `"glm"`.
`epoch`	Optional iteration or epoch at which training failed.
`iterations`	Number of iterations run before giving up.
`inputs`	Optional snapshot of the inputs a prediction failed on. It is stored as given; the envelope renders a data frame or matrix as a short description rather than its contents.

Value

A condition inheriting from `dataexcept_data_science_error`.

See Also

Other dataexcept errors: `data_engineering_errors`, `data_frame_errors`, `data_quality_errors`, `file_errors`, `job_errors`, `model_quality_errors`, `pipeline_errors`, `service_errors`, `validation_error()`

Examples

```
fit_or_fail <- function(formula, data, maxit = 25) {
  fit <- suppressWarnings(glm(formula,
    family = binomial, data = data,
    control = glm.control(maxit = maxit)
  ))
  if (!fit$converged) {
    stop(convergence_error("glm", iterations = fit$iter))
  }
  fit
}

separated <- data.frame(y = c(0, 0, 1, 1), x = 1:4)
tryCatch(
  fit_or_fail(y ~ x, separated, maxit = 5),
  dataexcept_model_training_error = function(e) e$iterations
)
```

`model_quality_errors` *Model evaluation and training failures*

Description

Errors about a model's quality and the process that produces it: a metric that could not be computed, cross-validation that failed, an invalid hyperparameter, training that ran out of time, a model that fits its training data too closely or not closely enough, a model that could not be saved, and a computation that exceeded its resources. All inherit from `dataexcept_data_science_error`; `training_timeout_error()` also inherits from `dataexcept_model_training_error`, so a handler for training failures catches it.

Usage

```
model_evaluation_error(
  metric,
  value,
  message = NULL,
  parent = NULL,
  call = NULL
)

cross_validation_error(folds, details = NULL, parent = NULL, call = NULL)

hyperparameter_error(param, value, message = NULL, parent = NULL, call = NULL)

training_timeout_error(model_type, timeout, parent = NULL, call = NULL)

model_serialization_error(path, parent = NULL, call = NULL)

overfitting_error(train_metric, val_metric, parent = NULL, call = NULL)

underfitting_error(train_metric, threshold, parent = NULL, call = NULL)

resource_limit_error(resource, limit, parent = NULL, call = NULL)
```

Arguments

`metric`	Name of the metric.
`value`	For `model_evaluation_error()`, the value computed, often `NaN`. For `hyperparameter_error()`, the value rejected, stored as given.
`message`	A message to use instead of the default.
`parent`	The condition that caused this one, or `NULL`.
`call`	The call to report, or `NULL`.
`folds`	Number of folds.
`details`	Optional detail, appended to the message.
`param`	Name of the hyperparameter.
`model_type`	The model, as a name such as `"xgboost"`.
`timeout`	The time limit that was exceeded, in seconds.
`path`	The file the model was being written to. A URL is redacted.
`train_metric`	The metric on the training data.
`val_metric`	The metric on the validation data.
`threshold`	The smallest training metric accepted.
`resource`	The resource that ran out, such as `"memory"`.
`limit`	The limit that was exceeded, stored as given.

Value

A condition inheriting from `dataexcept_data_science_error`.

See Also

Other dataexcept errors: `data_engineering_errors`, `data_frame_errors`, `data_quality_errors`, `file_errors`, `job_errors`, `modelling_errors`, `pipeline_errors`, `service_errors`, `validation_error()`

Examples

```
check_fit <- function(train_auc, val_auc, tolerance = 0.1) {
  if (train_auc - val_auc > tolerance) stop(overfitting_error(train_auc, val_auc))
  invisible(TRUE)
}
tryCatch(check_fit(0.99, 0.71), dataexcept_data_science_error = conditionMessage)

# A metric that could not be computed is usually NaN.
err <- model_evaluation_error("auc", NaN)
conditionMessage(err)

# The cause of a failed save is kept, and its message appended.
saved <- tryCatch(
  saveRDS(lm(dist ~ speed, cars), file.path(tempfile(), "model.rds")),
  warning = function(w) model_serialization_error("model.rds", parent = w)
)
conditionMessage(saved)
```

`new_dataexcept_error` *Create a dataexcept error*

Description

`new_dataexcept_error()` is the constructor every dataexcept error is built with, and the way to define your own. It returns a condition object without signalling it; signal it with `stop()` (or `rlang::abort()`), exactly as you would a condition from `errorCondition()`.

Usage

```
new_dataexcept_error(
  message,
  ...,
  type = "DataExceptError",
  class = character(),
  parent = NULL,
  call = NULL,
  failure = NULL
)
```

Arguments

message	The condition message, a single string.
...	Named fields to store on the condition.
type	The envelope type. A registered type (see dataexcept_classes()) supplies the R classes and the default failure metadata. An unregistered type is allowed for your own errors: pass the R class in class.
class	Additional R classes, most specific first. They are placed in front of the classes of type.
parent	The condition that caused this one, or NULL.
call	The call to report, or NULL. Pass sys.call() from inside your function to report its call.
failure	Failure metadata from failure_metadata(); defaults to the default of type.

Details

A dataexcept error is an ordinary R condition. Its fields are list elements, so a handler reads them directly (cnd$column); its classes are what tryCatch() matches; and its cause is stored in parent, the field rlang uses, so rlang prints the chain as "Caused by error".

Credentials never reach a dataexcept condition. URLs in the message and in character fields are redacted at construction: userinfo and credential query parameters are removed, while scheme, host and port stay, because they are what makes an error actionable.

Value

A condition object of class c(class, <classes of type>, "error", "condition").

See Also

The constructors for specific failures, such as missing_column_error() and validation_error().

Examples

```
# Your own error, still caught as a dataexcept error and serialisable to
# the envelope.
quota_error <- function(used, limit) {
  new_dataexcept_error(
    sprintf("Quota exceeded: %d of %d requests used", used, limit),
    used = used,
    limit = limit,
    type = "QuotaExceededError",
    class = "myapp_quota_exceeded_error",
    failure = failure_metadata("transient", retryable = TRUE)
  )
}

tryCatch(
  stop(quota_error(1000L, 1000L)),
  dataexcept_error = function(e) condition_to_json(e)
)
```

`operation_context` *Describe where an operation is running*

Description

An operation context records where a failure happened, separately from what failed. It is the R counterpart of the Python package's `OperationContext`, with the same fields and the same rules, so a failure event from an R job and one from a Python service can be filtered and correlated together.

Usage

```
operation_context(
  system = NULL,
  component = NULL,
  operation = NULL,
  request_id = NULL,
  job_id = NULL,
  correlation_id = NULL,
  trace_id = NULL,
  span_id = NULL
)

operation_index_fields(context)
```

Arguments

`system`, `component`, `operation`
: Low-cardinality labels: the system, the component within it, and the operation being performed.

`request_id`, `job_id`, `correlation_id`
: Identifiers of this run.

`trace_id`, `span_id`
: Trace identifiers. To fill them from the active OpenTelemetry span, use `operation_context_from_otel(`

`context`
: An operation context.

Details

The fields fall into two groups:

- `system`, `component` and `operation` are labels. They should stay low-cardinality – a route template such as `"POST /users/{id}"`, a job name, a workflow step – so they are safe to filter and group by. `operation_index_fields()` returns only these.
- `request_id`, `job_id`, `correlation_id`, `trace_id` and `span_id` identify one run. They are for correlating a specific execution and should not become metric dimensions or indexed tags.

Every field is optional: a missing identifier stays missing rather than being invented. URLs in any field lose their credentials and their path.

Value

`operation_context()` returns an object of class `dataexcept_operation_context`: a named list holding only the fields that were given, in the order above. `operation_index_fields()` returns the same with only the label fields.

See Also

`condition_to_event()`, which pairs a context with a condition.

Examples

```
context <- operation_context(
  system = "batch",
  component = "billing",
  operation = "settle_invoices",
  job_id = "job-42"
)
context
operation_index_fields(context)
```

---

operation_context_from_otel

Correlate an operation context with the active OpenTelemetry span

---

Description

`operation_context_from_otel()` fills the `trace_id` and `span_id` of an operation context from the active span of the r-lib otel package, so that failure events written to logs with `condition_to_event()` carry the same identifiers as the trace.

Usage

```
operation_context_from_otel(context = NULL, span_context = NULL)
```

Arguments

`context`	An operation context to extend, or `NULL` for a new one.
`span_context`	The span context to read. `NULL` means otel's active span context.

Details

Nothing is invented. When otel is not installed, tracing is off, or there is no valid active span, the context is returned unchanged. An identifier the context already has must agree with the span's: a conflicting `trace_id` or `span_id` is an error, because silently replacing provenance would make the correlation less trustworthy than omitting it.

Value

An operation context.

Examples

```
context <- operation_context(system = "batch", operation = "settle_invoices")

# With tracing off there is no active span, so nothing is added.
operation_context_from_otel(context)

# Inside a traced operation, the trace and span identifiers are filled in.
# otelsdk records the trace in memory here, so no collector is needed.
if (requireNamespace("otelsdk", quietly = TRUE)) {
  settle_invoices <- function() {
    otel::start_local_active_span("settle_invoices")
    operation_context_from_otel(context)
  }
  traced <- otelsdk::with_otel_record(settle_invoices(), what = "traces")
  print(traced$value)

  # The identifiers are the recorded span's.
  identical(traced$value$trace_id, traced$traces[[1]]$trace_id)
}
```

---

pino_schema *The Pino profile JSON Schema*

Description

The Pino profile is the shape `condition_to_pino()` writes: the DataExcept envelope projected onto the error Pino logs. Its JSON Schema, `pino-1.0.0.json`, is published by the Python DataExcept package and ships with this one, together with the Python package's projection of each reference envelope fixture, under `schema/fixtures/pino/`. The test suite requires `envelope_to_pino()` to reproduce every one of them.

Usage

```
pino_schema()
```

Value

The schema as a list.

See Also

`envelope_schema()` for the envelope's own schema.

Examples

```
pino_schema()$`$id`
names(pino_schema()$`$defs`)

# The file itself, for a JSON Schema validator:
system.file("schema", "pino-1.0.0.json", package = "dataexcept")
```

`pipeline_errors` *Pipeline and external service failures*

Description

Errors raised by a pipeline step: a call to another service that failed, timed out, or was refused; an operation retried until the limit; and a read or write to storage that failed. All inherit from `dataexcept_pipeline_error`, and the four service errors from `dataexcept_external_service_error` too.

Usage

```
external_service_error(
  service_name,
  status_code = NULL,
  response = NULL,
  message = NULL,
  parent = NULL,
  call = NULL
)

service_timeout_error(
  service_name,
  timeout_seconds = NULL,
  parent = NULL,
  call = NULL
)

service_authentication_error(
  service_name,
  message = NULL,
  parent = NULL,
  call = NULL
)

service_authorization_error(
  service_name,
  message = NULL,
  parent = NULL,
  call = NULL
```

```
)

retry_limit_exceeded_error(
  operation,
  retries,
  message = NULL,
  parent = NULL,
  call = NULL
)

storage_error(location, operation, message = NULL, parent = NULL, call = NULL)
```

Arguments

service_name	Name of the service called.
status_code	The HTTP status code, if a response arrived.
response	The response, or part of it, stored as given. The envelope writes it as an attribute, so store only what is safe to log.
message	A message to use instead of the default.
parent	The condition that caused this one, or NULL.
call	The call to report, or NULL.
timeout_seconds	The time limit that elapsed, in seconds, or NULL.
operation	For retry_limit_exceeded_error(), the operation retried. For storage_error(), what was being done, such as "write".
retries	Number of attempts made.
location	Where the data was stored, such as a path or a bucket URL. A URL is redacted.

Details

A refused credential or permission will be refused again, so service_authentication_error() and service_authorization_error() are permanent and not retryable. The others are "unknown": whether a retry would help depends on the service, and with_failure_metadata() records what it said.

Value

A condition inheriting from dataexcept_pipeline_error.

See Also

Other dataexcept errors: data_engineering_errors, data_frame_errors, data_quality_errors, file_errors, job_errors, model_quality_errors, modelling_errors, service_errors, validation_error()

Examples

```
call_rates_api <- function(status) {
  if (status == 401L) stop(service_authentication_error("rates-api"))
  if (status >= 500L) stop(external_service_error("rates-api", status_code = status))
  "ok"
}

err <- tryCatch(call_rates_api(401L), dataexcept_external_service_error = identity)
conditionMessage(err)
is_retryable(err)

err <- tryCatch(call_rates_api(503L), dataexcept_external_service_error = identity)
is_retryable(err)

storage_error("s3://reports/2026/q3.parquet", "write")
```

---

record_otel_exception *Record a condition on an OpenTelemetry span*

Description

`record_otel_exception()` records a condition as an exception event on a span, with the attributes of `condition_to_otel_attributes()`. With no span, it records on the active span of the r-lib otel package, when otel is installed and tracing is on.

Usage

```
record_otel_exception(
  cnd,
  span = NULL,
  operation_context = NULL,
  include_attributes = TRUE,
  max_depth = 8L,
  include_stacktrace = TRUE,
  include_envelope = FALSE,
  set_status = TRUE
)
```

Arguments

`cnd`	A condition object.
`span`	The span to record on: any object whose `record_exception()` method takes the condition and an `attributes` list, as otel's spans do. `NULL` means otel's active span.
`operation_context`	An operation context from `operation_context()`, or `NULL`.

include_attributes
: Include the `attributes` field?

max_depth
: How many levels of chained conditions to render. A cause beyond this depth is replaced by the truncation marker `{"truncated": true}`.

include_stacktrace
: Include `exception.stacktrace` when the condition has one?

include_envelope
: Include the whole envelope as a JSON attribute?

set_status
: Also set the span's status to `"error"`, when the span has a `set_status()` method?

Details

It is meant to be called while a failure is being handled, so it is fail-open: if otel is not installed, no span is recording, the span has no `record_exception()` method, or anything goes wrong converting or recording, it returns `FALSE` and the failure being handled carries on unchanged. Use `condition_to_otel_attributes()` directly when a mistake should be an error.

Recording through dataexcept also keeps credentials out of the trace. The otel SDK builds the exception message by printing the condition and falls back to the condition's call for the stack trace, both unredacted; the attributes recorded here replace those fields with redacted ones.

With `set_status = TRUE`, the default, the span's status is also set to `"error"`, with the redacted message as its description. That marks the operation as failed, and it matters for redaction: when an error escapes a span that otel ends automatically (one from `otel::start_local_active_span()`, `otel::with_active_span()` or `otel::local_active_span()`), the SDK records it itself, unredacted, unless the span's status is already set. Recording the failure here first means it appears once, redacted. Pass `set_status = FALSE` when the handler recovers from the failure and the operation goes on to succeed.

Value

`TRUE`, invisibly, when the exception was recorded, and `FALSE` otherwise.

Examples

```
# A span-like recorder, standing in for an otel span.
recorded <- NULL
span <- list(record_exception = function(error_condition, attributes = NULL, ...) {
  recorded <<- attributes
})

err <- validation_error("age", -1)
record_otel_exception(err, span = span)
str(recorded)

# With no span, the active otel span is used. With tracing off there is
# nothing to record on, and the call does nothing.
record_otel_exception(err)
```

redact_url *Remove credentials from URLs*

Description

`redact_url()` strips credentials from a single URL; `redact_urls_in_text()` finds every URL in free text and redacts each one. Scheme, host and port are always kept – they are what makes an error actionable – while userinfo and credential-bearing query and fragment parameters are replaced by `***`. Parameter names are matched token by token (`accessToken`, `api_key`, `X-Amz-Signature`), not by substring, so ordinary parameters such as `keyword` survive.

Usage

```
redact_url(url, keep_path = TRUE)

redact_urls_in_text(text, keep_path = TRUE)
```

Arguments

`url`	A single URL. Anything that is not a URL with a host is returned unchanged.
`keep_path`	Keep the URL path? Use `FALSE` where the path itself is the secret, as in the incoming-webhook URLs of Slack and similar services.
`text`	A single string that may contain URLs.

Details

These are the rules of the Python DataExcept package, so a URL redacted in either language reads the same. dataexcept applies them automatically to the message and character fields of every condition it creates, and again, with `keep_path = FALSE`, to everything it writes into an envelope.

What redaction cannot do: a bare secret that is not part of a URL cannot be recognised in free text, and is not removed.

Value

The redacted string. A URL that carries no credential is returned exactly as given.

Examples

```
redact_url("postgresql://analyst:s3cret@db.internal:5432/sales")
redact_url("https://api.example.com/v1/orders?page=2&api_key=abc123")
redact_url("https://hooks.example.com/services/T000/B000/XXXX", keep_path = FALSE)

redact_urls_in_text("GET https://user:pw@example.com/data failed: timeout")
```

`service_errors` *Database, network and service failures*

Description

Errors at the boundary with another system. Their failure metadata stays `"unknown"` by default, because only the backend's response can say whether a retry would help; attach that answer with `with_failure_metadata()` when you have it.

Usage

```
database_connection_error(db_url, message = NULL, parent = NULL, call = NULL)

query_execution_error(query, parent = NULL, call = NULL)

transaction_error(
  transaction_id = NULL,
  message = NULL,
  parent = NULL,
  call = NULL
)

host_unreachable_error(host, message = NULL, parent = NULL, call = NULL)

connection_timeout_error(host, timeout, parent = NULL, call = NULL)

api_error(
  endpoint,
  status_code = NULL,
  message = NULL,
  parent = NULL,
  call = NULL
)
```

Arguments

`db_url`	The database connection URL.
`message`	A message to use instead of the default.
`parent`	The condition that caused this one, or `NULL`.
`call`	The call to report, or `NULL`.
`query`	The SQL that failed.
`transaction_id`	Optional identifier of the transaction.
`host`	The host that could not be reached.
`timeout`	The timeout that elapsed, in seconds.
`endpoint`	The API endpoint URL.
`status_code`	The HTTP status code, if a response arrived.

Details

Connection URLs routinely carry a user name and password, and API endpoints carry tokens in their query string. Both are redacted before they reach the condition.

Value

A condition inheriting from `dataexcept_database_error`, `dataexcept_network_error` or `dataexcept_pipeline_error`

See Also

pipeline_errors for the other failures of calls to external services.

Other dataexcept errors: `data_engineering_errors`, `data_frame_errors`, `data_quality_errors`, `file_errors`, `job_errors`, `model_quality_errors`, `modelling_errors`, `pipeline_errors`, `validation_error()`

Examples

```
err <- database_connection_error("postgresql://analyst:s3cret@db.internal:5432/sales")
conditionMessage(err)

err <- api_error("https://api.example.com/v1/orders?api_key=abc123", status_code = 503L)
err <- with_failure_metadata(err, failure_metadata("transient", TRUE, 30))
condition_to_json(err)
```

---

validate_envelope *Check that a payload is a valid envelope*

Description

`validate_envelope()` checks a payload against the rules of the envelope schema (`envelope_schema()`): each node is exactly one of an exception record, a cycle record or a truncation marker; identity fields are strings; a failure record carries all three of its fields with valid values; and the two markers carry nothing beyond their own fields. Fields the schema does not know are accepted, because a newer producer may add them.

Usage

```
validate_envelope(x, max_depth = 32L)

is_envelope(x, max_depth = 32L)
```

Arguments

`x`	An envelope: a JSON string, or a list as returned by `condition_to_envelope()` or `jsonlite::parse_json()`.
`max_depth`	The deepest chain of nested records to accept.

Details

The check is structural and needs no JSON Schema validator. It is the check `envelope_to_condition()` applies before reading.

Value

`validate_envelope()` returns `x` invisibly when it is valid and signals a `dataexcept_envelope_error` listing every problem otherwise. `is_envelope()` returns `TRUE` or `FALSE`.

Examples

```
is_envelope('{"type": "ValueError", "module": "builtins", "message": "bad row"}')
is_envelope('{"type": "ValueError", "message": "bad row"}')

try(validate_envelope('{"truncated": true, "type": "ValueError"}'))
```

---

validation_error *Validation failures*

Description

`validation_error()` reports a value that fails a rule: the payload is wrong, so it is classified as permanent and not retryable for the same unchanged input.

Usage

```
validation_error(field, value, message = NULL, parent = NULL, call = NULL)
```

Arguments

`field`	Name of the field that failed validation.
`value`	The offending value. Stored as given; rendered in the default message with `deparse()`.
`message`	A message to use instead of the default.
`parent`	The condition that caused this one, or `NULL`.
`call`	The call to report, or `NULL`.

Value

A condition of class `dataexcept_validation_error`.

See Also

Other dataexcept errors: `data_engineering_errors`, `data_frame_errors`, `data_quality_errors`, `file_errors`, `job_errors`, `model_quality_errors`, `modelling_errors`, `pipeline_errors`, `service_errors`

Examples

```
check_age <- function(age) {
  if (age < 0) stop(validation_error("age", age))
  age
}
tryCatch(check_age(-1), dataexcept_validation_error = function(e) e$field)
```

`with_classed_warnings` *Give base R's text-only warnings a class*

Description

`with_classed_warnings()` evaluates `expr` and re-signals the base R and stats warnings listed by `classed_warning_rules()` as classed warnings, so they can be handled by class instead of by matching their text:

Usage

```
with_classed_warnings(expr)

classify_warning(w)

classed_warning_rules()
```

Arguments

`expr`	An expression to evaluate.
`w`	A warning condition.

Details

```
withCallingHandlers(
  with_classed_warnings(glm(y ~ x, family = binomial, data = d)),
  dataexcept_separation_warning = function(w) ...
)
```

Recognition does not depend on the session language. A warning is compared with its template translated through R's own message catalogue, the one R used to write the warning, so the same code works in an English, German or Portuguese session. Templates with values are format strings, matched with their placeholders as slots, and plural messages are matched in every form of the language.

Besides base R and stats, the rules cover the convergence warnings of survival's Cox and parametric models and of lme4's mixed models.

Each classed warning keeps the original message and call, stores the original warning in `parent`, and records the matching rule in `rule`. A warning that carries values – the iterations `kmeans()` ran, the code `optim()` returned in `arima()`, the gradient lme4 measured – has them as fields too, named in the `fields` column of `classed_warning_rules()`. Warnings no rule matches pass through

untouched, as does everything else `expr` signals. Written to an envelope, a classed warning has a `type` such as `"SeparationWarning"` and the original `simpleWarning` as its `cause`.

`classify_warning()` does the recognition for a single warning object.

Value

`with_classed_warnings()` returns the value of `expr`. `classify_warning()` returns the classed warning, or `NULL` when no rule matches. `classed_warning_rules()` returns a data frame with one row per recognised message template, and in `fields` the values it carries.

Examples

```
separated <- data.frame(y = c(0, 0, 1, 1), x = 1:4)

# Handle one family of warnings by class, whatever the session language.
withCallingHandlers(
  with_classed_warnings(glm(y ~ x, family = binomial, data = separated)),
  dataexcept_separation_warning = function(w) {
    message("Perfect separation: ", conditionMessage(w))
    invokeRestart("muffleWarning")
  },
  dataexcept_convergence_warning = function(w) invokeRestart("muffleWarning")
)

# Or fail closed: turn non-convergence into an error.
try(withCallingHandlers(
  with_classed_warnings(glm(y ~ x,
    family = binomial, data = separated,
    control = glm.control(maxit = 5)
  )),
  dataexcept_convergence_warning = function(w) {
    stop(convergence_error("glm", iterations = 5L, parent = w))
  }
))

classed_warning_rules()[, c("type", "template")]
```

---

wrap_errors *Turn errors from an expression into a dataexcept error*

Description

`wrap_errors()` evaluates `expr`, and if it signals an error, signals the error `constructor` builds instead, with the original as its cause. It is the R counterpart of the Python package's `wrap()` and `wrapping()`, and replaces the handler that is otherwise written out by hand:

Usage

```
wrap_errors(expr, constructor, ..., on = "error", failure = NULL)
```

Arguments

expr	The expression to evaluate.
constructor	A function returning a condition, called with ... and parent, the condition that was caught: any dataexcept constructor, or your own.
...	Arguments for constructor.
on	The classes of condition to wrap. The default wraps errors only; c("error", "warning") also turns a warning into the wrapped error, which suits functions such as read.csv() that warn before they fail.
failure	Failure metadata for the new error, from failure_metadata(), or NULL for the constructor's default.

Details

```
tryCatch(
  read.csv(path),
  error = function(e) stop(file_read_error(path, parent = e))
)
```

The original condition is always recorded as the cause, so it stays in the envelope and in rlang's "Caused by" output. Conditions other than those in on pass through untouched, as do interrupts.

Value

The value of expr, when nothing is wrapped.

Examples

```
path <- tempfile(fileext = ".csv")

err <- tryCatch(
  wrap_errors(read.csv(path), file_read_error, path = path, on = c("error", "warning")),
  dataexcept_file_error = identity
)
conditionMessage(err)
class(err$parent)

# A transient failure, as far as the caller knows.
fetch <- function() stop("connection reset by peer")
err <- tryCatch(
  wrap_errors(fetch(), api_error,
    endpoint = "https://api.example.com/v1/rates",
    failure = failure_metadata("transient", retryable = TRUE)
  ),
  dataexcept_error = identity
)
is_retryable(err)
```

Index

∗ **dataexcept errors**
  data_engineering_errors, 12
  data_frame_errors, 14
  data_quality_errors, 15
  file_errors, 20
  job_errors, 21
  model_quality_errors, 24
  modelling_errors, 23
  pipeline_errors, 31
  service_errors, 36
  validation_error, 38

api_error *(*service_errors*)*, 36
authentication_error *(*job_errors*)*, 21
authorization_error *(*job_errors*)*, 21

batch_processing_error
  *(*data_engineering_errors*)*, 12

classed_warning_rules
  *(*with_classed_warnings*)*, 39
classify_warning
  *(*with_classed_warnings*)*, 39
collect_errors *(*condition_group*)*, 3
condition_failure, 2
condition_failure(), *20*
condition_group, 3
condition_to_envelope, 5
condition_to_envelope(), *8*, *11*, *18*, *37*
condition_to_event, 7
condition_to_event(), *29*
condition_to_event_json
  *(*condition_to_event*)*, 7
condition_to_json
  *(*condition_to_envelope*)*, 5
condition_to_otel_attributes, 8
condition_to_otel_attributes(), *33*, *34*
condition_to_pino, 10
condition_to_pino(), *30*
condition_to_pino_json
  *(*condition_to_pino*)*, 10
condition_type *(*condition_to_envelope*)*, 5
condition_type(), *18*
configuration_error *(*job_errors*)*, 21
connection_timeout_error
  *(*service_errors*)*, 36
convergence_error *(*modelling_errors*)*, 23
cross_validation_error
  *(*model_quality_errors*)*, 24

data_drift_error *(*data_quality_errors*)*, 15
data_engineering_errors, 12, *14*, *16*, *21*, *22*, *24*, *26*, *32*, *37*, *38*
data_format_error
  *(*data_quality_errors*)*, 15
data_frame_errors, *13*, 14, *16*, *21*, *22*, *24*, *26*, *32*, *37*, *38*
data_imbalance_error
  *(*data_quality_errors*)*, 15
data_leakage_error
  *(*data_quality_errors*)*, 15
data_loading_error *(*modelling_errors*)*, 23
data_quality_errors, *13*, *14*, 15, *21*, *22*, *24*, *26*, *32*, *37*, *38*
data_transformation_error
  *(*data_engineering_errors*)*, 12
database_connection_error
  *(*service_errors*)*, 36
dataexcept_classes, 12
dataexcept_classes(), *27*
deparse(), *38*
dtype_mismatch_error
  *(*data_frame_errors*)*, 14
dtype_mismatch_error(), *6*

envelope_schema, 17

envelope_schema(), *5*, *30*, *37*
envelope_schema_id *(*envelope_schema*)*, 17
envelope_schema_version *(*envelope_schema*)*, 17
envelope_to_condition, 18
envelope_to_condition(), *6*
envelope_to_pino *(*condition_to_pino*)*, 10
envelope_to_pino(), *30*
errorCondition(), *26*
etl_job_error *(*data_engineering_errors*)*, 12
external_service_error *(*pipeline_errors*)*, 31

failure_metadata, 19
failure_metadata(), *4*, *27*, *41*
file_errors, *13, 14*, *16*, 20, *22*, *24*, *26*, *32*, *37, 38*
file_read_error *(*file_errors*)*, 20
file_write_error *(*file_errors*)*, 20

group_members *(*condition_group*)*, 3

host_unreachable_error *(*service_errors*)*, 36
hyperparameter_error *(*model_quality_errors*)*, 24

I(), *6*
is_envelope *(*validate_envelope*)*, 37
is_retryable *(*condition_failure*)*, 2

job_errors, *13, 14*, *16*, *21*, 21, *24*, *26*, *32*, *37, 38*

merge_key_error *(*data_frame_errors*)*, 14
missing_column_error *(*data_frame_errors*)*, 14
missing_column_error(), *27*
missing_data_error *(*modelling_errors*)*, 23
model_evaluation_error *(*model_quality_errors*)*, 24
model_quality_errors, *13, 14*, *16*, *21, 22*, *24*, 24, *32*, *37, 38*
model_serialization_error *(*model_quality_errors*)*, 24
model_training_error *(*modelling_errors*)*, 23
modelling_errors, *13, 14*, *16*, *21, 22*, 23, *26*, *32*, *37, 38*

new_dataexcept_error, 26
new_dataexcept_error(), *4*

operation_context, 28
operation_context(), *7*, *9*, *33*
operation_context_from_otel, 29
operation_context_from_otel(), *28*
operation_index_fields *(*operation_context*)*, 28
operation_timeout_error *(*job_errors*)*, 21
outlier_detection_error *(*data_quality_errors*)*, 15
overfitting_error *(*model_quality_errors*)*, 24

pino_schema, 30
pino_schema(), *10*
pipeline_errors, *13, 14*, *16*, *21, 22*, *24*, *26*, 31, *37, 38*
prediction_error *(*modelling_errors*)*, 23

query_execution_error *(*service_errors*)*, 36

read.csv(), *41*
record_otel_exception, 33
record_otel_exception(), *9, 10*
redact_url, 35
redact_urls_in_text *(*redact_url*)*, 35
resource_limit_error *(*model_quality_errors*)*, 24
resource_not_found_error *(*job_errors*)*, 21
retry_limit_exceeded_error *(*pipeline_errors*)*, 31

schema_mismatch_error *(*data_quality_errors*)*, 15
service_authentication_error *(*pipeline_errors*)*, 31
service_authorization_error *(*pipeline_errors*)*, 31
service_errors, *13, 14*, *16*, *21, 22*, *24*, *26*, *32*, 36, *38*
service_timeout_error *(*pipeline_errors*)*, 31
stop(), *26*

storage_error *(*pipeline_errors*)*, 31

training_timeout_error
*(*model_quality_errors*)*, 24
transaction_error *(*service_errors*)*, 36

underfitting_error
*(*model_quality_errors*)*, 24

validate_envelope, 37
validate_envelope(), *18*
validation_error, *13, 14*, *16*, *21, 22*, *24*, *26*, *32*, *37*, 38
validation_error(), *20*, *22*, *27*

with_classed_warnings, 39
with_failure_metadata
*(*failure_metadata*)*, 19
with_failure_metadata(), *32*, *36*
wrap_errors, 40