Package 'otel'

November 7, 2026

Title OpenTelemetry R API

Version 0.3.0

Description High-quality, ubiquitous, and portable telemetry to enable effective observability. OpenTelemetry is a collection of tools, APIs, and SDKs used to instrument, generate, collect, and export telemetry data (metrics, logs, and traces) for analysis in order to understand your software's performance and behavior.
This package implements the OpenTelemetry API: <https://opentelemetry.io/docs/specs/otel/>.
Use this package as a dependency if you want to instrument your R package for OpenTelemetry.

License MIT + file LICENSE

Encoding UTF-8

Depends R (>= 3.6.0)

Suggests callr, cli, glue, jsonlite, otelsdk (>= 0.2.4), processx, shiny, spelling, testthat (>= 3.0.0), utils, withr

Config/Needs/website tidyverse/tidytemplate

Config/testthat/edition 3

URL https://otel.r-lib.org, https://github.com/r-lib/otel

BugReports https://github.com/r-lib/otel/issues

Config/roxygen2/version 8.1.1

NeedsCompilation no

Author Gábor Csárdi [aut, cre]

Maintainer Gábor Csárdi <csardi.gabor@gmail.com>

Repository CRAN

Date/Publication 2026-11-07 15:20:02 UTC

Contents

as_attributes . . . 3
counter_add . . . 4
default_tracer_name . . . 5
end_span . . . 6
Environment Variables . . . 7
extract_http_context . . . 10
gauge_record . . . 11
Getting Started . . . 12
get_active_span . . . 16
get_active_span_context . . . 17
get_default_logger_provider . . . 18
get_default_meter_provider . . . 19
get_default_tracer_provider . . . 20
get_logger . . . 21
get_meter . . . 22
get_tracer . . . 23
histogram_record . . . 24
is_logging_enabled . . . 25
is_measuring_enabled . . . 26
is_tracing_enabled . . . 27
local_active_span . . . 28
log . . . 29
log_severity_levels . . . 30
meter_provider_noop . . . 31
otel_counter . . . 31
otel_gauge . . . 32
otel_histogram . . . 33
otel_logger . . . 34
otel_logger_provider . . . 37
otel_meter . . . 38
otel_meter_provider . . . 41
otel_span . . . 42
otel_span_context . . . 47
otel_tracer . . . 49
otel_tracer_provider . . . 51
otel_up_down_counter . . . 52
pack_http_context . . . 54
start_local_active_span . . . 55
start_span . . . 56
tracer_provider_noop . . . 58
tracing-constants . . . 59
up_down_counter_add . . . 60
with_active_span . . . 61
Zero Code Instrumentation . . . 62

Index **64**

`as_attributes` *R objects as OpenTelemetry attributes*

Description

Convert a list of R objects to a form that is suitable as OpenTelemetry attributes.

Usage

```
as_attributes(x)
```

Arguments

`x`	A list of R objects, to be used as OpenTelemetry attributes.

Value

A named list that can be used as the `attributes` argument to the `start_span()` method of otel_tracer, the `log()` method of otel_logger, etc.

If `x` is not named, or some names are the empty string or `NA`, then integer numbers as used for the missing or invalid names.

If some elements in `x` are not of the natively supported R types in OpenTelemetry (character, logical, double, integer), then their printed form is captured using `utils::capture.output()`.

Length one R arrays (e.g. from `as.array()`) are kept as arrays, so they are recorded as arrays. See "Array attributes" in otel_span.

Limits:

The number of attributes can be limited with the OTEL_ATTRIBUTE_COUNT_LIMIT environment variable. The default is 128.

The length of the each attribute (vector) can be limited with the OTEL_ATTRIBUTE_VALUE_LENGTH_LIMIT environment variable. The default is `Inf`. Note that this is applied to the length of each attribute as an R vector. E.g. it does *not* currently limit the number of characters in individual strings.

Examples

```
as_attributes(list(
  number = 1.0,
  vector = 1:10,
  string = "otel",
  string_vector = letters,
  object = mtcars
))
```

counter_add *Increase an OpenTelemetry counter*

Description

Increase an OpenTelemetry counter

Usage

```
counter_add(name, value = 1L, attributes = NULL, context = NULL, meter = NULL)
```

Arguments

name	Name of the counter.
value	Value to add to the counter, defaults to 1.
attributes	Additional attributes to add.
context	Span context. If missing the active context is used, if any.
meter	Meter object (otel_meter). Otherwise it is passed to `get_meter()` to get a meter.

Value

The counter object (otel_counter), invisibly.

See Also

'Environment Variables' needed to enable OpenTelemetry recording.

Other OpenTelemetry metrics instruments: `gauge_record()`, `histogram_record()`, `up_down_counter_add()`

Other OpenTelemetry metrics API: `gauge_record()`, `histogram_record()`, `is_measuring_enabled()`, `up_down_counter_add()`

Examples

```
otel::counter_add("total-session-count", 1)
```

`default_tracer_name` *Default tracer name (and meter and logger name) for an R package*

Description

Exporters, like the ones in the otelsdk package, can use this function to determine the default tracer name, if the instrumentation author hasn't specified one. If you are an instrumentation author, you probably do not need to call this function directly, but do read on to learn about choosing and setting the tracer name.

Usage

```
default_tracer_name(name = NULL)
```

Arguments

`name`	Custom tracer name. If `NULL` then otel will construct a a tracer (meter, logger) name according to the algorithm detailed below. The calling package is detected even if `name` is given, so the package name can be used in `OTEL_R_EMIT_SCOPES` and `OTEL_R_SUPPRESS_SCOPES`.

Details

About tracer names:

The name of a tracer identifies an OpenTelemetry instrumentation scope. Instrumentation scopes can be used to organize the collected telemetry data. otel can also use instrumentation scopes to suppress emitting unneeded telemetry data, see 'Environment Variables'.

For the otel R package it makes sense to create a separate instrumentation scope for each R package that emits telemetry data. otel can do this automatically, with some from the package author.

Setting the tracer name:

As a package author, you can define the `otel_tracer_name` symbol in your package and set it do the desired tracer name. For example, the callr package has this in an `.R` file:

```
otel_tracer_name <- "org.r-lib.callr"
```

See below for tips on choosing a tracer name.

If you don't like the default tracer name, you can call `get_tracer()` (or `get_logger()` or `get_meter()` manually with the desired name.

Automatic tracer name detection in otel:

This is the detailed algorithm that otel uses in `default_tracer_name`:

- Using `base::topenv()` it finds the calling package (or other top level environment), recursively.
- It ignores the otel and otelsdk packages while searching.

- If it finds the base environment or the global environment, then it checks for the `otel_tracer_name` global variable (in the global environment). If that exists, then it must be a scalar string and it is used as the tracer name. Otherwise `org.project.R` is used as the tracer name.
- Otherwise it looks for the `otel_tracer_name` symbol inside the top level environment it has found. If this symbol exists then it must be a string scalar and otel will use it as the tracer name.
- If this symbol does not exist, then otel will use `r.package.<environment-name>` as the tracer name. `<environment-name>` is usually the package name.

Choosing a tracer name:

The OpenTelemetry specification recommends using a tracer name that identifies the instrumentation scope, i.e. your package.

Some tips on choosing the tracer name:

- If your R package can be associated with a URL, you can use the "reverse" of that URL. E.g. since the callr package's online manual is at https://callr.r-lib.org, it can use `org.r-lib.callr`.
- If your R package belongs to your company, you can use the "reverse" of the company URL, possibly with an additional prefix. E.g. for the shiny R package by Posit, `co.posit.r-package.shiny` seems like a good name.
- If you don't set `otel_tracer_name`, then `default_tracer_name` will use `r.package.<package-name>` as the tracer name.

Value

A list with entries:

- `name`: The supplied or auto-detected tracer name.
- `package`: Auto-detected package name, or `"R"` if called from the global environment or base R.
- `on`: Whether tracing is enabled for this package.

Examples

```
default_tracer_name()
```

---

end_span *End an OpenTelemetry span*

Description

Spans created with `start_local_active_span()` end automatically by default. You must end every other span manually, by calling `end_span`, or using the `end_on_exit` argument of `local_active_span()` or `with_active_span()`.

Usage

```
end_span(span, status_code = NULL)
```

Arguments

span	The span to end.
status_code	Span status code to set before ending the span. Possible values: unset, ok, error. See span_status_codes. If NULL, the status is not changed.

Value

Nothing.

See Also

Other OpenTelemetry trace API: Zero Code Instrumentation, is_tracing_enabled(), local_active_span(), start_local_active_span(), start_span(), tracing-constants, with_active_span()

Examples

```
fun <- function() {
  # start span, do not activate
  spn <- otel::start_span("myfun")
  # do not leak resources
  on.exit(otel::end_span(spn), add = TRUE)
  myfun <- function() {
     # activate span for this function
     otel::local_active_span(spn)
     # create child span
     spn2 <- otel::start_local_active_span("myfun/2")
  }

  myfun2 <- function() {
    # activate span for this function
    otel::local_active_span(spn)
    # create child span
    spn3 <- otel::start_local_active_span("myfun/3")
  }
  myfun()
  myfun2()
  end_span(spn)
}
fun()
```

Environment Variables *Environment variables to configure otel*

Description

This manual page contains the environment variables you can use to configure the otel package.

See also the Environment Variables in the otelsdk package, which is charge of the data collection configuration.

You need set these environment variables when configuring the collection of telemetry data, unless noted otherwise.

Value

Not applicable.

Production or Development Environment

- `OTEL_ENV`

  By default otel runs in production mode. In production mode otel functions never error. Errors in the telemetry code will not stop the monitored application.

  This behavior is not ideal for development, where one would prefer to catch errors early. Set

  ```
  OTEL_ENV=dev
  ```

  to run otel in development mode, where otel functions fail on error, make it easier to fix errors.

Selecting Exporters

otel is responsible for selecting the providers to use for traces, logs and metrics. You can use the environment variables below to point the otel functions to the desired providers.

If none of these environment variables are set, then otel will not emit any telemetry data.

- `OTEL_TRACES_EXPORTER`

  The name of the selected tracer provider. See `get_default_tracer_provider()` for the possible values.

- `OTEL_R_TRACES_EXPORTER`

  R specific version of `OTEL_TRACES_EXPORTER`.

- `OTEL_LOGS_EXPORTER`

  The name of the selected logger provider. See `get_default_logger_provider()` for the possible values.

- `OTEL_R_LOGS_EXPORTER`

  R specific version of `OTEL_LOGS_EXPORTER`.

- `OTEL_METRICS_EXPORTER`

  The name of the selected meter provider. See `get_default_meter_provider()` for the possible values.

- `OTEL_R_METRICS_EXPORTER`

  R specific version of `OTEL_METRICS_EXPORTER`.

Suppressing Instrumentation Scopes (R Packages)

otel has two environment variables to fine tune which instrumentation scopes (i.e. R packages, typically) emit telemetry data. By default, i.e. if neither of these are set, all packages emit telemetry data.

- `OTEL_R_EMIT_SCOPES`

  Set this environment variable to a comma separated string of instrumentation scope names or R package names to restrict telemetry to these packages only. The name of the instrumentation scope is the same as the name of the tracer, logger or meter, see `default_tracer_name()`.

  You can mix package names and instrumentation scope names and you can also use wildcards (globbing). For example the value

  OTEL_R_EMIT_SCOPES="org.r-lib.*,dplyr"

  selects all packages with an instrumentation scope that starts with `org.r-lib.` and also dplyr.

- `OTEL_R_SUPPRESS_SCOPES`

  Set this environment variable to a comma separated string of instrumentation scope names or R package names to suppress telemetry data from these packages. The name of the instrumentation scope is the same as the name of the tracer, logger or meter, see `default_tracer_name()`.

  You can mix package names and instrumentation scope names and you can also use wildcards (globbing). For example the value

  OTEL_R_SUPPRESS_SCOPES="org.r-lib.*,dplyr"

  excludes packages with an instrumentation scope that starts with `org.r-lib.` and also dplyr.

Zero Code Instrumentation

otel can instrument R packages for OpenTelemetry data collection without changing their source code. This relies on changing the code of the R functions manually using `base::trace()` and can be configured using environment variables.

- `OTEL_R_INSTRUMENT_PKGS`

  Set `OTEL_R_INSTRUMENT_PKGS` to a comma separated list of packages to instrument. The automatic instrumentation happens when the otel package is loaded, so in general it is best to set this environment variable before loading R.

- `OTEL_R_INSTRUMENT_PKGS_<pkg>_INCLUDE`

  For an automatically instrumented package, set this environment variable to only instrument a subset of its functions. It is parsed as a comma separated string of function names, which may also include `?` and `*` wildcards (globbing).

- `OTEL_R_INSTRUMENT_PKGS_<pkg>_EXCLUDE`

  For an automatically instrumented package, set this environment variable to exclude some functions from instrumentation. It has the same syntax as its `*_INCLUDE` pair. If both are set, then inclusion is applied and the exclusion.

Attribute Limits

- `OTEL_ATTRIBUTE_COUNT_LIMIT`

  Set this environment variable to limit the number of attributes for a single span, log record, metric measurement, etc. If unset, the default limit is 128 attributes. Note that only attributes specified with `as_attributes()` are subject to this environment variable.

- `OTEL_ATTRIBUTE_VALUE_LENGTH_LIMIT`

  Set this environment variable to limit the length of vectors in attributes for a single span, log record, metric measurement, etc. If unset, there is no limit on the lengths of vectors in attributes. Note that only attributes specified with `as_attributes()` are subject to this environment variable.

See Also

Environment Variables in otelsdk

Examples

```
# To start an R session using the OTLP exporter:
# OTEL_TRACES_EXPORTER=http R -q -f script.R
```

`extract_http_context` *Extract a span context from HTTP headers received from a client*

Description

The return value can be used as the `parent` option when starting a span.

Usage

```
extract_http_context(headers, tracer = NULL)
```

Arguments

`headers`	A named list with one or two strings: `traceparent` is mandatory, and `tracestate` is optional.
`tracer`	Tracer object (otel_tracer) or tracer name to use. If `NULL`, then otel uses an internal tracer. You usually do not need to set this, the active span does not depend on the tracer. Passing a tracer might give the wrong result. If it is a no-op tracer, then the result is an invalid span (context), or no HTTP headers, even if there is an active span. This happens if the tracer's scope is turned off, e.g. via the `OTEL_R_SUPPRESS_SCOPES` environment variable, or if the tracer was created before tracing was turned on. The default `NULL` does not have this problem.

Value

And otel_span_context object.

See Also

`pack_http_context()`

Examples

```
hdr <- otel::pack_http_context()
ctx <- otel::extract_http_context()
ctx$is_valid()
```

gauge_record *Record a value of an OpenTelemetry gauge*

Description

Record a value of an OpenTelemetry gauge

Usage

```
gauge_record(name, value, attributes = NULL, context = NULL, meter = NULL)
```

Arguments

name	Name of the gauge
value	Value to record.
attributes	Additional attributes to add.
context	Span context. If missing the active context is used, if any.
meter	Meter object (otel_meter). Otherwise it is passed to `get_meter()` to get a meter.

Value

The gauge object (otel_gauge), invisibly.

See Also

'Environment Variables' needed to enable OpenTelemetry recording.

Other OpenTelemetry metrics instruments: `counter_add()`, `histogram_record()`, `up_down_counter_add()`

Other OpenTelemetry metrics API: `counter_add()`, `histogram_record()`, `is_measuring_enabled()`, `up_down_counter_add()`

Examples

```
otel::gauge_record("temperature", 27)
```

Getting Started *Getting Started*

Description

This page is about instrumenting you R package or project for OpenTelemetry. If you want to start collecting OpenTelemetry data for instrumented packages, see Collecting Telemetry Data in the otelsdk package.

About OpenTelemetry

OpenTelemetry is an observability framework. OpenTelemetry is a collection of tools, APIs, and SDKs used to instrument, generate, collect, and export telemetry data such as metrics, logs, and traces, for analysis in order to understand your software's performance and behavior.

For an introduction to OpenTelemetry, see the OpenTelemetry website docs.

The otel and otelsdk R packages

Use the otel package as a dependency if you want to instrument your R package or project for OpenTelemetry.

Use the otelsdk package to produce OpenTelemetry output from an R package or project that was instrumented with the otel package.

Complete Example

To instrument your package with otel, you need to do a couple of steps. In this section we show how to instrument the callr package.

Add the otel package as a dependency:

The first step is to add the otel package as a dependency. otel is a very lightweight package, so may want to add it as a hard dependency. This has the advantage that you don't need to check if otel is installed every time you call an otel function. Add otel to the `Imports` section in `DESCRIPTION`:

```
Imports:
    otel
```

Alternatively, you may add otel as a soft dependency. Add otel to the `Suggests` section in `DESCRIPTION`:

```
Suggests:
    otel
```

If you add otel in `Suggests`, then it makes sense to create a helper function that checks if otel is installed and also that tracing is enabled for the caller. You can put this function in any R file, e.g. `R/utils.R` is a nice place for it:

```
is_otel_tracing <- function() {
  requireNamespace("otel", quietly = TRUE) && otel::is_tracing_enabled()
}
```

Choose a tracer name:

Every package should have its own tracer with a name that is unique for the package. See `default_tracer_name()` for tips on choosing a good tracer name. Set the `otel_tracer_name` variable to the tracer name. No need to export this symbol. In callr, we'll add

```
otel_tracer_name <- "org.r-lib.callr"
```

to the `R/callr-package.R` file.

Create spans for selected functions:

Select the functions you want to add tracing to. It is overkill to add tracing to small functions that are called lots of times. It makes sense to add spans to the main functions of the package.

The callr package has various ways of starting another R process and then running R code in it. We'll add tracing to the

- `callr::r()`
- `callr::rcmd()`
- `callr::rscript()`

functions first.

We add to `callr::r()` in `eval.R`:

```
if (is_otel_tracing()) {
  otel::start_local_active_span(
    "callr::r",
    attributes = otel::as_attributes(options)
  )
}
```

- We use the `is_otel_tracing()` helper function, defined above.
- `start_local_active_span()` starts a span and also activates it. It also sets up an exit handler that ends the span when the caller function (`callr::r()`) exits.
- `options` contain a long list of user-provided and other option, we add these to the span as attributes.

We add essentially the same code to `callr::rcmd()`:

```
if (is_otel_tracing()) {
  otel::start_local_active_span(
    "callr::rcmd",
    attributes = otel::as_attributes(options)
  )
}
```

And to `callr::rscript()`:

```
if (is_otel_tracing()) {
  otel::start_local_active_span(
    "callr::rscript",
    attributes = otel::as_attributes(options)
  )
}
```

Concurrency:

An instance of the `callr::r_session` R6 class represents persistent R background processes. We want to collect all spans from an R process into the same trace. Since the R processes are running concurrently, their (sub)spans will not form the correct hierarchy if we use the default, timing-based otel mechanism to organize spans into trees. We need to manage the lifetime and activation of the spans that represent the R processes manually.

A generic strategy for handling concurrency in otel is:

1. Create a new long lasting span with `start_span()`. (I.e. *not* `start_local_active_span()`!)
2. Assign the returned span into the corresponding object of the concurrent and/or asynchronous computation. Every span has a finalizer that closes the span.
3. When running code that belongs to the concurrent computation represented by the span, activate it for a specific R scope by calling `with_active_span()` or `local_active_span()`.
4. When the concurrent computation ends, close the span manually with its `$end()` method or `end_span()`. (Otherwise it would be only closed at the next garbage collection, assuming there are no references to it.)

This code goes into the constructor of the `r_session` object:

```
if (is_otel_tracing()) {
  private$options$otel_session <- otel::start_span(
    "callr::r_session",
    attributes = otel::as_attributes(options)
  )
}
```

The `finalize()` method (the finalizer) gets a call to close the span:

```
if (is_otel_tracing()) {
  private$options$otel_session$end()
}
```

We also add (sub)spans to other operations, e.g. the `read()` method gets

```
if (is_otel_tracing()) {
  otel::local_session(private$options$otel_session)
  spn <- otel::start_local_active_span("callr::r_session$read")
}
```

Testing:

To test your instrumentation, you need to install the otelsdk package and you also need a local or remote OpenTelemetry collector.

I suggest you use `otel-tui`, a terminal OpenTelemetry viewer. To configure it, use the `http` exporter, see 'Environment Variables':

```
OTEL_TRACES_EXPORTER=http R -q
```

Development mode:

By default otel functions never error, to avoid taking down a production app. For development this is not ideal, we want to catch errors early. I suggest you always turn on development mode when instrumenting a package:

```
OTEL_ENV=dev
```

Context propagation:

OpenTelemetry supports distributed tracing. A span (context) can be serialized, copied to another process, and there it can be used to create child spans.

For applications communicating via HTTP the serialized span context is transmitted in HTTP headers. For our callr example we can copy the context to the R subprocess in environment variables.

For example in the `callr:r()` code we may write:

```
if (is_otel_tracing()) {
  otel::start_local_active_span(
    "callr::r",
    attributes = otel::as_attributes(options)
  )
  hdrs <- otel::pack_http_context()
  names(hdrs) <- toupper(names(hdrs))
  options$env[names(hdrs)] <- hdrs
}
```

`options$env` contains the environment variables callr will set in the newly started R process. This is where we need to add the output of `pack_http_context()`, which contains the serialized representation of the active span, if there is any.

Additionally, the subprocess needs to pick up the span context from the environment variables. The `callr:::common_hook()` internal function contains the code that the subprocess runs at startup. Here we need to add:

```
has_otel <- nzchar(Sys.getenv("TRACEPARENT")) &&
  requireNamespace("otel", quietly = TRUE)
assign(envir = env$`__callr_data__`, "has_otel", has_otel)
if (has_otel) {
  hdrs <- as.list(c(
    traceparent = Sys.getenv("TRACEPARENT"),
    tracestate = Sys.getenv("TRACESTATE"),
    baggage = Sys.getenv("BAGGAGE")
  ))
  prtctx <- otel::extract_http_context(hdrs)
  reg.finalizer(
    env$`__callr_data__`,
    function(e) e$otel_span$end(),
    onexit = TRUE
  )
  assign(
    envir = env$`__callr_data__`,
    "otel_span",
    otel::start_span(
      "callr subprocess",
      options = list(parent = prtctx)
    )
```

```
  )
}
```

First we check if the TRACEPARENT environment variable is set. This contains the serialization of the parent span. If it exists and the otel package is also available, then we extract the span context from the environment variables, and start a new span that is a child span or the remote span obtained from the environment variables. We also set up a finalizer that closes this span when the R process terminates.

Examples

```
# See above
```

`get_active_span` *Returns the active span, if any*

Description

This is sometimes useful, to add additional attributes or links to the currently active span.

Usage

```
get_active_span(tracer = NULL)
```

Arguments

`tracer`	Tracer object (otel_tracer) or tracer name to use. If NULL, then otel uses an internal tracer. You usually do not need to set this, the active span does not depend on the tracer. Passing a tracer might give the wrong result. If it is a no-op tracer, then the result is an invalid span (context), or no HTTP headers, even if there is an active span. This happens if the tracer's scope is turned off, e.g. via the OTEL_R_SUPPRESS_SCOPES environment variable, or if the tracer was created before tracing was turned on. The default NULL does not have this problem.

Value

The active span, an otel_span object, if any, or an invalid span if there is no active span.

Examples

```
fun <- function() {
  otel::start_local_active_span("fun")
  spn <- otel::get_active_span()
  spn$set_attribute("key", "attribute-value")
}
fun()
```

`get_active_span_context` *Returns the active span context*

Description

This is sometimes useful for logs or metrics, to associate logging and metrics reporting with traces.

Usage

```
get_active_span_context(tracer = NULL)
```

Arguments

`tracer`	Tracer object (otel_tracer) or tracer name to use. If `NULL`, then otel uses an internal tracer. You usually do not need to set this, the active span does not depend on the tracer. Passing a tracer might give the wrong result. If it is a no-op tracer, then the result is an invalid span (context), or no HTTP headers, even if there is an active span. This happens if the tracer's scope is turned off, e.g. via the `OTEL_R_SUPPRESS_SCOPES` environment variable, or if the tracer was created before tracing was turned on. The default `NULL` does not have this problem.

Details

Note that logs and metrics instruments automatically use the current span context, so often you don't need to call this function explicitly.

Value

The active span context, an otel_span_context object. If there is no active span context, then an invalid span context is returned, i.e. `spc$is_valid()` will be `FALSE` for the returned `spc`.

Examples

```
fun <- function() {
  otel::start_local_active_span("fun")
  fun2()
}
fun2 <- function() {
  otel::log("Log message", span_context = otel::get_active_span_context())
}
fun()
```

`get_default_logger_provider` *Get the default logger provider*

Description

The logger provider defines how logs are exported when collecting telemetry data. It is unlikely that you need to call this function directly, but read on to learn how to configure which exporter to use.

Usage

```
get_default_logger_provider()
```

Details

If there is no default set currently, then it creates and sets a default.

The default logger provider is created based on the OTEL_R_LOGS_EXPORTER environment variable. This environment variable is specifically for R applications with OpenTelemetry support.

If this is not set, then the generic OTEL_LOGS_EXPORTER environment variable is used. This applies to all applications that support OpenTelemetry and use the OpenTelemetry SDK.

The following values are allowed:

- `none`: no traces are exported.
- `stdout` or `console`: uses otelsdk::logger_provider_stdstream, to write traces to the standard output.
- `stderr`: uses otelsdk::logger_provider_stdstream, to write traces to the standard error.
- `http` or `otlp`: uses otelsdk::logger_provider_http, to send traces through HTTP, using the OpenTelemetry Protocol (OTLP).
- `otlp/file` uses otelsdk::logger_provider_file to write logs to a JSONL file.
- `otlp/stdout` uses otelsdk::logger_provider_file to write logs to the standard output, in OTLP JSON format. This needs otelsdk 0.2.4.9001 or later.
- `<package>::<provider>`: will select the `<provider>` object from the `<package>` package to use as a logger provider. It calls `<package>::<provider>$new()` to create the new logger provider. If this fails for some reason, e.g. the package is not installed, then it throws an error.

Value

The default logger provider, an otel_logger_provider object.

See Also

Other low level logs API: `get_logger()`, `logger_provider_noop`, `otel_logger`, `otel_logger_provider`

Examples

```
get_default_logger_provider()
```

`get_default_meter_provider` *Get the default meter provider*

Description

The meter provider defines how metrics are exported when collecting telemetry data. It is unlikely that you need to call this function directly, but read on to learn how to configure which exporter to use.

Usage

```
get_default_meter_provider()
```

Details

If there is no default set currently, then it creates and sets a default.

The default meter provider is created based on the OTEL_R_METRICS_EXPORTER environment variable. This environment variable is specifically for R applications with OpenTelemetry support.

If this is not set, then the generic OTEL_METRICS_EXPORTER environment variable is used. This applies to all applications that support OpenTelemetry and use the OpenTelemetry SDK.

The following values are allowed:

- `none`: no metrics are exported.
- `stdout` or `console`: uses otelsdk::meter_provider_stdstream, to write metrics to the standard output.
- `stderr`: uses otelsdk::meter_provider_stdstream, to write metrics to the standard error.
- `http` or `otlp`: uses otelsdk::meter_provider_http, to send metrics through HTTP, using the OpenTelemetry Protocol (OTLP).
- `otlp/file` uses otelsdk::meter_provider_file to write metrics to a JSONL file.
- `otlp/stdout` uses otelsdk::meter_provider_file to write metrics to the standard output, in OTLP JSON format. This needs otelsdk 0.2.4.9001 or later.
- `<package>::<provider>`: will select the `<provider>` object from the `<package>` package to use as a meter provider. It calls `<package>::<provider>$new()` to create the new meter provider. If this fails for some reason, e.g. the package is not installed, then it throws an error.

Value

The default meter provider, an otel_meter_provider object.

See Also

Other low level metrics API: `get_meter()`, `meter_provider_noop`, `otel_counter`, `otel_gauge`, `otel_histogram`, `otel_meter`, `otel_meter_provider`, `otel_up_down_counter`

Examples

```
get_default_meter_provider()
```

`get_default_tracer_provider` *Get the default tracer provider*

Description

The tracer provider defines how traces are exported when collecting telemetry data. It is unlikely that you need to call this function directly, but read on to learn how to configure which exporter to use.

Usage

```
get_default_tracer_provider()
```

Details

If there is no default set currently, then it creates and sets a default.

The default tracer provider is created based on the OTEL_R_TRACES_EXPORTER environment variable. This environment variable is specifically for R applications with OpenTelemetry support.

If this is not set, then the generic OTEL_TRACES_EXPORTER environment variable is used. This applies to all applications that support OpenTelemetry and use the OpenTelemetry SDK.

The following values are allowed:

- `none`: no traces are exported.
- `stdout` or `console`: uses otelsdk::tracer_provider_stdstream, to write traces to the standard output.
- `stderr`: uses otelsdk::tracer_provider_stdstream, to write traces to the standard error.
- `http` or `otlp`: uses otelsdk::tracer_provider_http, to send traces through HTTP, using the OpenTelemetry Protocol (OTLP).
- `otlp/file` uses otelsdk::tracer_provider_file to write traces to a JSONL file.
- `otlp/stdout` uses otelsdk::tracer_provider_file to write traces to the standard output, in OTLP JSON format. This needs otelsdk 0.2.4.9001 or later.
- `<package>::<provider>`: will select the `<provider>` object from the `<package>` package to use as a tracer provider. It calls `<package>::<provider>$new()` to create the new tracer provider. If this fails for some reason, e.g. the package is not installed, then it throws an error.

Value

The default tracer provider, an otel_tracer_provider object. See otel_tracer_provider for its methods.

See Also

Other low level trace API: get_tracer(), otel_span, otel_span_context, otel_tracer, otel_tracer_provider, tracer_provider_noop

Examples

```
get_default_tracer_provider()
```

`get_logger` *Get a logger from the default logger provider*

Description

Get a logger from the default logger provider

Usage

```
get_logger(
  name = NULL,
  minimum_severity = NULL,
  version = NULL,
  schema_url = NULL,
  attributes = NULL,
  ...,
  provider = NULL
)
```

Arguments

`name`	Name of the new tracer. If missing, then deduced automatically.
`minimum_severity`	A log level, the minimum severity log messages to log. See log_severity_levels.
`version`	Optional. Specifies the version of the instrumentation scope if the scope has a version (e.g. R package version). Example value: `"1.0.0"`.
`schema_url`	Optional. Specifies the Schema URL that should be recorded in the emitted telemetry.
`attributes`	Optional. Specifies the instrumentation scope attributes to associate with emitted telemetry.
`...`	Additional arguments are passed to the `get_logger()` method of the provider.
`provider`	Tracer provider to use. If `NULL`, then it uses get_default_tracer_provider() to get a tracer provider.

Value

An otel_logger object.

See Also

Other low level logs API: `get_default_logger_provider()`, `logger_provider_noop`, `otel_logger`, `otel_logger_provider`

Examples

```
myfun <- function() {
  lgr <- otel::get_logger()
  otel::log("Log message", logger = lgr)
}
myfun()
```

---

get_meter *Get a meter from the default meter provider*

Description

Get a meter from the default meter provider

Usage

```
get_meter(
  name = NULL,
  version = NULL,
  schema_url = NULL,
  attributes = NULL,
  ...,
  provider = NULL
)
```

Arguments

`name`	Name of the new tracer. If missing, then deduced automatically.
`version`	Optional. Specifies the version of the instrumentation scope if the scope has a version (e.g. R package version). Example value: `"1.0.0"`.
`schema_url`	Optional. Specifies the Schema URL that should be recorded in the emitted telemetry.
`attributes`	Optional. Specifies the instrumentation scope attributes to associate with emitted telemetry.
`...`	Additional arguments are passed to the `get_meter()` method of the provider.
`provider`	Meter provider to use. If `NULL`, then it uses `get_default_meter_provider()` to get a tracer provider.

Value

An otel_meter object.

See Also

Other low level metrics API: get_default_meter_provider(), meter_provider_noop, otel_counter, otel_gauge, otel_histogram, otel_meter, otel_meter_provider, otel_up_down_counter

Examples

```
myfun <- function() {
  mtr <- otel::get_meter()
  ctr <- mtr$create_counter("session-count")
  ctr$add(1)
}
myfun()
```

---

get_tracer *Get a tracer from the default tracer provider*

Description

Calls get_default_tracer_provider() to get the default tracer provider. Then calls its $get_tracer() method to create a new tracer.

Usage

```
get_tracer(
  name = NULL,
  version = NULL,
  schema_url = NULL,
  attributes = NULL,
  ...,
  provider = NULL
)
```

Arguments

name	Name of the new tracer. If missing, then deduced automatically using default_tracer_name(). Make sure you read the manual page of default_tracer_name() before using this argument.
version	Optional. Specifies the version of the instrumentation scope if the scope has a version (e.g. R package version). Example value: "1.0.0".
schema_url	Optional. Specifies the Schema URL that should be recorded in the emitted telemetry.
attributes	Optional. Specifies the instrumentation scope attributes to associate with emitted telemetry.
...	Additional arguments are passed to the get_tracer() method of the provider.
provider	Tracer provider to use. If NULL, then it uses get_default_tracer_provider() to get a tracer provider.

Details

Usually you do not need to call this function directly, because `start_local_active_span()` calls it for you.

Calling `get_tracer()` multiple times with the same `name` (or same auto-deduced name) will return the same (internal) tracer object. (Even if the R external pointer objects representing them are different.)

A tracer is only deleted if its tracer provider is deleted and garbage collected.

Value

An OpenTelemetry tracer, an otel_tracer object.

See Also

Other low level trace API: `get_default_tracer_provider()`, `otel_span`, `otel_span_context`, `otel_tracer`, `otel_tracer_provider`, `tracer_provider_noop`

Examples

```
myfun <- function() {
  trc <- otel::get_tracer()
  spn <- trc$start_span()
  on.exit(otel::end_span(spn), add = TRUE)
  otel::local_active_span(spn, end_on_exit = TRUE)
}
myfun()
```

---

histogram_record *Record a value of an OpenTelemetry histogram*

Description

Record a value of an OpenTelemetry histogram

Usage

```
histogram_record(name, value, attributes = NULL, context = NULL, meter = NULL)
```

Arguments

`name`	Name of the histogram.
`value`	Value to record.
`attributes`	Additional attributes to add.
`context`	Span context. If missing the active context is used, if any.
`meter`	Meter object (otel_meter). Otherwise it is passed to `get_meter()` to get a meter.

Value

The histogram object (otel_histogram), invisibly.

See Also

'Environment Variables' needed to enable OpenTelemetry recording.

Other OpenTelemetry metrics instruments: `counter_add()`, `gauge_record()`, `up_down_counter_add()`

Other OpenTelemetry metrics API: `counter_add()`, `gauge_record()`, `is_measuring_enabled()`, `up_down_counter_add()`

Examples

```
otel::histogram_record("response-time", 0.2)
```

---

is_logging_enabled *Check whether OpenTelemetry logging is active*

Description

This is useful for avoiding computation when logging is inactive.

Usage

```
is_logging_enabled(severity = "info", logger = NULL)
```

Arguments

`severity`	Check if logs are emitted at this severity level.
`logger`	Logger object (otel_logger), or a logger name, the instrumentation scope, to pass to `get_logger()`.

Details

It calls `get_logger()` with `name` and then it calls the logger's `$is_enabled()` method.

Value

`TRUE` is OpenTelemetry logging is active, `FALSE` otherwise.

See Also

'Environment Variables' needed to enable OpenTelemetry logging.

Other OpenTelemetry logs API: `log()`, `log_severity_levels`

Examples

```
fun <- function() {
  if (otel::is_logging_enabled()) {
    xattr <- calculate_some_extra_attributes()
    otel::log("Starting fun", attributes = xattr)
  }
  # ...
}
```

`is_measuring_enabled` *Check whether OpenTelemetry metrics collection is active*

Description

This is useful for avoiding computation when metrics collection is inactive.

Usage

```
is_measuring_enabled(meter = NULL)
```

Arguments

`meter`	Meter object (otel_meter), or a meter name, the instrumentation scope, to pass to `get_meter()`.

Details

It calls `get_meter()` with `name` and then it calls the meter's `$is_enabled()` method.

Value

`TRUE` is OpenTelemetry metrics collection is active, `FALSE` otherwise.

See Also

'Environment Variables' needed to enable OpenTelemetry recording.

Other OpenTelemetry metrics API: `counter_add()`, `gauge_record()`, `histogram_record()`, `up_down_counter_add()`

Examples

```
fun <- function() {
  if (otel::is_measuring_enabled()) {
    xattr <- calculate_some_extra_attributes()
    otel::counter_add("sessions", 1, attributes = xattr)
  }
  # ...
}
```

`is_tracing_enabled` *Check if tracing is active*

Description

Checks whether OpenTelemetry tracing is active. This can be useful to avoid unnecessary computation when tracing is inactive.

Usage

```
is_tracing_enabled(tracer = NULL)
```

Arguments

`tracer`	Tracer object (otel_tracer). It can also be a tracer name, the instrumentation scope, or `NULL` for determining the tracer name automatically. Passed to `get_tracer()` if not a tracer object.

Details

It calls `get_tracer()` with `name` and then it calls the tracer's `$is_enabled()` method.

Value

`TRUE` is OpenTelemetry tracing is active, `FALSE` otherwise.

See Also

'Environment Variables' needed to enable OpenTelemetry tracing.

Other OpenTelemetry trace API: `Zero Code Instrumentation`, `end_span()`, `local_active_span()`, `start_local_active_span()`, `start_span()`, `tracing-constants`, `with_active_span()`

Examples

```
fun <- function() {
  if (otel::is_tracing_enabled()) {
    xattr <- calculate_some_extra_attributes()
    otel::start_local_active_span("fun", attributes = xattr)
  }
  # ...
}
```

`local_active_span` *Activate an OpenTelemetry span for an R scope*

Description

Activates the span for the caller (or other) frame.

Usually you need this function for spans created with `start_span()`, which does not activate the new span. Usually you don't need it for spans created with `start_local_active_span()`, because it activates the new span automatically.

Usage

```
local_active_span(span, end_on_exit = FALSE, activation_scope = parent.frame())
```

Arguments

`span`	The OpenTelemetry span to activate.
`end_on_exit`	Whether to end the span when exiting the activation scope.
`activation_scope`	The scope to activate the span for, defaults to the caller frame.

Details

When the frame ends, the span is deactivated and the previously active span will be active again, if there was any.

It is possible to activate the same span for multiple R frames.

Value

Nothing.

See Also

Other OpenTelemetry trace API: `Zero Code Instrumentation`, `end_span()`, `is_tracing_enabled()`, `start_local_active_span()`, `start_span()`, `tracing-constants`, `with_active_span()`

Other tracing for concurrent code: `with_active_span()`

Examples

```
fun <- function() {
  # start span, do not activate
  spn <- otel::start_span("myfun")
  # do not leak resources
  on.exit(otel::end_span(spn), add = TRUE)
  myfun <- function() {
     # activate span for this function
     otel::local_active_span(spn)
```

```
    # create child span
    spn2 <- otel::start_local_active_span("myfun/2")
  }

  myfun2 <- function() {
    # activate span for this function
    otel::local_active_span(spn)
    # create child span
    spn3 <- otel::start_local_active_span("myfun/3")
  }
  myfun()
  myfun2()
  end_span(spn)
}
fun()
```

---

log *Log an OpenTelemetry log message*

Description

Log an OpenTelemetry log message

Usage

```
log(msg, ..., severity = "info", logger = NULL)

log_trace(msg, ..., logger = NULL)

log_debug(msg, ..., logger = NULL)

log_info(msg, ..., logger = NULL)

log_warn(msg, ..., logger = NULL)

log_error(msg, ..., logger = NULL)

log_fatal(msg, ..., logger = NULL)
```

Arguments

msg	Log message.
...	Additional arguments are passed to the `$log()` method of the logger.
severity	Log severity, a string, one of "trace", "trace2", "trace3", "trace4", "debug", "debug2", "debug3", "debug4", "info", "info2", "info3", "info4", "warn", "warn2", "warn3", "warn4", "error", "error2", "error3", "error4", "fatal", "fatal2", "fatal3", "fatal4".
logger	Logger to use. If not an OpenTelemetry logger object (otel_logger), then it passed to `get_logger()` to get a logger.

Details

`log_trace()` is the same as `log()` with `severity_level` "trace".

`log_debug()` is the same as `log()` with `severity_level` "debug".

`log_info()` is the same as `log()` with `severity_level` "info".

`log_warn()` is the same as `log()` with `severity_level` "warn".
`log_error)` is the same as `log()` with `severity_level` "error".

`log_fatal()` is the same as `log()` with `severity_level` "fatal".

Value

The logger, invisibly.

See Also

'Environment Variables' needed to enable OpenTelemetry logging.

Other OpenTelemetry logs API: `is_logging_enabled()`, `log_severity_levels`

Examples

```
host <- "my.db.host"
port <- 6667
otel::log("Connecting to database at {host}:{port}")
```

`log_severity_levels` *OpenTelemetry log severity levels*

Description

A named integer vector, the severity levels in numeric form. The names are the severity levels in text form. otel functions accept both forms as severity levels, but the text form is more readable.

Value

Not applicable.

See Also

'Environment Variables' needed to set OpenTelemetry logging level.

Other OpenTelemetry logs API: `is_logging_enabled()`, `log()`

Examples

```
log_severity_levels
```

meter_provider_noop *No-op Meter Provider*

Description

This is the meter provider (otel_meter_provider) otel uses when metrics collection is disabled.

Details

All methods are no-ops or return objects that are also no-ops.

Value

Not applicable.

See Also

Other low level metrics API: `get_default_meter_provider()`, `get_meter()`, `otel_counter`, `otel_gauge`, `otel_histogram`, `otel_meter`, `otel_meter_provider`, `otel_up_down_counter`

Examples

```
meter_provider_noop$new()
```

otel_counter *OpenTelemetry Counter Object*

Description

otel_meter_provider -> otel_meter -> otel_counter, otel_up_down_counter, otel_histogram, otel_gauge

Details

Usually you do not need to deal with otel_counter objects directly. `counter_add()` automatically sets up a meter and creates a counter instrument, as needed.

A counter object is created by calling the `create_counter()` method of an `otel_meter_provider()`.

You can use the `add()` method to increment the counter by a positive amount.

In R counters are represented by double values.

Value

Not applicable.

Methods

`counter$add()`:

Increment the counter by a fixed amount.

Usage:

```
counter$add(value, attributes = NULL, span_context = NULL, ...)
```

Arguments:

- `value`: Value to increment the counter with.
- `attributes`: Additional attributes to add.
- `span_context`: Span context. If missing, the active context is used, if any.

Value:
The counter object itself, invisibly.

See Also

Other low level metrics API: `get_default_meter_provider()`, `get_meter()`, `meter_provider_noop`, `otel_gauge`, `otel_histogram`, `otel_meter`, `otel_meter_provider`, `otel_up_down_counter`

Examples

```
mp <- get_default_meter_provider()
mtr <- mp$get_meter()
ctr <- mtr$create_counter("session")
ctr$add(1)
```

---

otel_gauge *OpenTelemetry Gauge Object*

Description

otel_meter_provider -> otel_meter -> otel_counter, otel_up_down_counter, otel_histogram, otel_gauge

Details

Usually you do not need to deal with otel_gauge objects directly. `gauge_record()` automatically sets up a meter and creates a gauge instrument, as needed.

A gauge object is created by calling the `create_gauge()` method of an `otel_meter_provider()`.

You can use the `record()` method to record the current value.

In R gauge values are represented by doubles.

Value

Not applicable.

Methods

`gauge$record()`:

Update the statistics with the specified amount.

Usage:

```
gauge$record(value, attributes = NULL, span_context = NULL, ...)
```

Arguments:

- `value`: A numeric value. The current absolute value.
- `attributes`: Additional attributes to add.
- `span_context`: Span context. If missing, the active context is used, if any.

Value:
The gauge object itself, invisibly.

See Also

Other low level metrics API: `get_default_meter_provider()`, `get_meter()`, `meter_provider_noop`, `otel_counter`, `otel_histogram`, `otel_meter`, `otel_meter_provider`, `otel_up_down_counter`

Examples

```
mp <- get_default_meter_provider()
mtr <- mp$get_meter()
gge <- mtr$create_gauge("response-time")
gge$record(1.123)
```

---

otel_histogram *OpenTelemetry Histogram Object*

Description

otel_meter_provider -> otel_meter -> otel_counter, otel_up_down_counter, otel_histogram, otel_gauge

Details

Usually you do not need to deal with otel_histogram objects directly. `histogram_record()` automatically sets up a meter and creates a histogram instrument, as needed.

A histogram object is created by calling the `create_histogram()` method of an `otel_meter_provider()`.

You can use the `record()` method to update the statistics with the specified amount.

In R histogram values are represented by doubles.

Value

Not applicable.

Methods

`histogram$record()`:

Update the statistics with the specified amount.

Usage:

```
histogram$record(value, attributes = NULL, span_context = NULL, ...)
```

Arguments:

- `value`: A numeric value to record.
- `attributes`: Additional attributes to add.
- `span_context`: Span context. If missing, the active context is used, if any.

Value:
The histogram object itself, invisibly.

See Also

Other low level metrics API: `get_default_meter_provider()`, `get_meter()`, `meter_provider_noop`, `otel_counter`, `otel_gauge`, `otel_meter`, `otel_meter_provider`, `otel_up_down_counter`

Examples

```
mp <- get_default_meter_provider()
mtr <- mp$get_meter()
hst <- mtr$create_histogram("response-time")
hst$record(1.123)
```

---

otel_logger *OpenTelemetry Logger Object*

Description

otel_logger_provider -> otel_logger

Details

Usually you do not need to deal with otel_logger objects directly. `log()` automatically sets up the logger for emitting the logs.

A logger object is created by calling the `get_logger()` method of an otel_logger_provider.

You can use the `log()` method of the logger object to emit logs.

Typically there is a separate logger object for each instrumented R package.

Value

Not applicable.

Methods

`logger$is_enabled()`:
Whether the logger is active and emitting logs at a certain severity level.
This is equivalent to the `is_logging_enabled()` function.

Usage:

```
logger$is_enabled(severity = "info", event_id = NULL)
```

Arguments:

- `severity`: Check if logs are emitted at this severity level.
- `event_id`: Not implemented yet.

Value:
Logical scalar.

`logger$get_minimum_severity()`:
Get the current minimum severity at which the logger is emitting logs.

Usage:

```
logger_get_minimum_severity()
```

Value:
Named integer scalar.

`logger$set_minimum_severiry()`:
Set the minimum severity for emitting logs.

Usage:

```
logger$set_minimum_severity(minimum_severity)
```

Arguments:

- `minimum_severity`: Log severity, a string, one of "trace", "trace2", "trace3", "trace4", "debug", "debug2", "debug3", "debug4", "info", "info2", "info3", "info4", "warn", "warn2", "warn3", "warn4", "error", "error2", "error3", "error4", "fatal", "fatal2", "fatal3", "fatal4".

Value:
Nothing.

`logger$log()`:
Log an OpenTelemetry log message.

Usage:

```
logger$log(
  msg = "",
  severity = "info",
  span_context = NULL,
  span_id = NULL,
  trace_id = NULL,
  trace_flags = NULL,
  timestamp = SYs.time(),
  observed_timestamp = NULL,
  attributes = NULL,
  .envir = parent.frame()
)
```

Arguments:

- `msg`: Log message, may contain R expressions to evaluate within braces.
- `severity`: Log severity, a string, one of "trace", "trace2", "trace3", "trace4", "debug", "debug2", "debug3", "debug4", "info", "info2", "info3", "info4", "warn", "warn2", "warn3", "warn4", "error", "error2", "error3", "error4", "fatal", "fatal2", "fatal3", "fatal4".
- `span_context`: An otel_span_context object to associate the log message with a span.
- `span_id`: Alternatively to `span_context`, you can also specify `span_id`, `trace_id` and `trace_flags` to associate a log message with a span.
- `trace_id`: Alternatively to `span_context`, you can also specify `span_id`, `trace_id` and `trace_flags` to associate a log message with a span.
- `trace_flags`: Alternatively to `span_context`, you can also specify `span_id`, `trace_id` and `trace_flags` to associate a log message with a span.
- `timestamp`: Time stamp, defaults to the current time. This is the time the logged event occurred.
- `observed_timestamp`: Observed time stamp, this is the time the event was observed.
- `attributes`: Optional attributes, see `as_attributes()` for the possible values.

Value:
The logger object, invisibly.

`logger$trace()`:
The same as `logger$log()`, with `severity = "trace"`.

`logger$debug()`:
The same as `logger$log()`, with `severity = "debug"`.

`logger$info()`:
The same as `logger$log()`, with `severity = "info"`.

`logger$warn()`:
The same as `logger$log()`, with `severity = "warn"`.

`logger$error()`:
The same as `logger$log()`, with `severity = "error"`.

`logger$fatal()`:
The same as `logger$log()`, with `severity = "fatal"`.

See Also

Other low level logs API: `get_default_logger_provider()`, `get_logger()`, `logger_provider_noop`, `otel_logger_provider`

Examples

```
lp <- get_default_logger_provider()
lgr <- lp$get_logger()
platform <- utils::sessionInfo()$platform
lgr$log("This is a log message from {platform}.", severity = "trace")
```

`otel_logger_provider` *OpenTelemetry Logger Provider Object*

Description

otel_logger_provider -> otel_logger

Details

The logger provider defines how logs are exported when collecting telemetry data. It is unlikely that you need to use logger provider objects directly.

Usually there is a single logger provider for an R app or script.

Typically the logger provider is created automatically, at the first `log()` call. otel decides which logger provider class to use based on 'Environment Variables'.

Value

Not applicable.

Implementations

Note that this list is updated manually and may be incomplete.

- logger_provider_noop: No-op logger provider, used when no logs are emitted.
- otelsdk::logger_provider_file: Save logs to a JSONL file.
- otelsdk::logger_provider_http: Send logs to a collector over HTTP/OTLP.
- otelsdk::logger_provider_stdstream: Write logs to standard output or error or to a file. Primarily for debugging.

Methods

`logger_provider$get_logger()`:

Get or create a new logger object.

Usage:

```
logger_provider$get_logger(
 name = NULL,
 version = NULL,
 schema_url = NULL,
 attributes = NULL
)
```

Arguments:

- `name` Logger name. It makes sense to reuse the tracer name as the logger name. See `get_logger()` and `default_tracer_name()`.
- `version`: Optional. Specifies the version of the instrumentation scope if the scope has a version (e.g. R package version). Example value: `"1.0.0"`.

- `schema_url`: Optional. Specifies the Schema URL that should be recorded in the emitted telemetry.
- `attributes`: Optional. Specifies the instrumentation scope attributes to associate with emitted telemetry. See `as_attributes()` for allowed values. You can also use `as_attributes()` to convert R objects to OpenTelemetry attributes.

Value:
An OpenTelemetry logger (otel_logger) object.

See also:
`get_default_logger_provider()`, `get_logger()`.

`logger_provider$flush()`:

Force any buffered logs to flush. Logger providers might not implement this method.

Usage:

```
logger_provider$flush()
```

Value:
Nothing.

`logger_provider$shutdown()`:

Stop the logger provider. Flushes any buffered logs and stops emitting logs.

Usage:

```
logger_provider$shutdown()
```

Value:
Nothing.

See Also

Other low level logs API: `get_default_logger_provider()`, `get_logger()`, `logger_provider_noop`, `otel_logger`

Examples

```
lp <- otel::get_default_logger_provider()
lgr <- lp$get_logger()
lgr$is_enabled()
```

---

otel_meter *OpenTelemetry Meter Object*

Description

otel_meter_provider -> otel_meter -> otel_counter, otel_up_down_counter, otel_histogram, otel_gauge

Details

Usually you do not need to deal with otel_meter objects directly. `counter_add()`, `up_down_counter_add()`, `histogram_record()` and `gauge_record()` automatically set up the meter and uses it to create instruments.

A meter object is created by calling the `get_meter()` method of an otel_meter_provider.

You can use the `create_counter()`, `create_up_down_counter()`, `create_histogram()`, `create_gauge()` methods of the meter object to create instruments.

Typically there is a separate meter object for each instrumented R package.

Value

Not applicable.

Methods

`meter$is_enabled()`:

Whether the meter is active and emitting measurements.

This is equivalent to the `is_measuring_enabled()` function.

Usage:

```
meter$is_enabled()
```

Value:
Logical scalar.

`meter$create_counter()`:

Create a new counter instrument.

Usage:

```
create_counter(name, description = NULL, unit = NULL)
```

Arguments:

- `name`: Name of the instrument.
- `description`: Optional description.
- `unit`: Optional measurement unit. If specified, it should use units from Unified Code for Units of Measure, according to the OpenTelemetry semantic conventions.

Value:
An OpenTelemetry counter (otel_counter) object.

`meter$create_up_down_counter()`:

Create a new up-down counter instrument.

Usage:

```
create_up_down_counter(name, description = NULL, unit = NULL)
```

Arguments:

- `name`: Name of the instrument.
- `description`: Optional description.

- `unit`: Optional measurement unit. If specified, it should use units from Unified Code for Units of Measure, according to the OpenTelemetry semantic conventions.

Value:
An OpenTelemetry counter (otel_up_down_counter) object.

`meter$create_histogram()`:

Create a new histogram.

Usage:

```
create_histogram(name, description = NULL, unit = NULL, boundaries = NULL)
```

Arguments:

- `name`: Name of the instrument.
- `description`: Optional description.
- `unit`: Optional measurement unit. If specified, it should use units from Unified Code for Units of Measure, according to the OpenTelemetry semantic conventions.
- `boundaries`: Optional numeric vector of explicit bucket boundaries, in increasing order. This is an advisory parameter, the SDK may ignore it, e.g. if a view configures different boundaries for the histogram. If `NULL`, the default boundaries of the SDK are used.

Value:
An OpenTelemetry histogram (otel_histogram) object.

`meter$create_gauge()`:

Create a new gauge.

Usage:

```
create_gauge(name, description = NULL, unit = NULL)
```

Arguments:

- `name`: Name of the instrument.
- `description`: Optional description.
- `unit`: Optional measurement unit. If specified, it should use units from Unified Code for Units of Measure, according to the OpenTelemetry semantic conventions.

Value:
An OpenTelemetry gauge (otel_gauge) object.

See Also

Other low level metrics API: `get_default_meter_provider()`, `get_meter()`, `meter_provider_noop`, `otel_counter`, `otel_gauge`, `otel_histogram`, `otel_meter_provider`, `otel_up_down_counter`

Examples

```
mp <- get_default_meter_provider()
mtr <- mp$get_meter()
ctr <- mtr$create_counter("session")
ctr$add(1)
```

`otel_meter_provider` *OpenTelemetry meter provider objects*

Description

otel_meter_provider -> otel_meter -> otel_counter, otel_up_down_counter, otel_histogram, otel_gauge

Details

The meter provider defines how metrics are exported when collecting telemetry data. It is unlikely that you need to use meter provider objects directly.

Usually there is a single meter provider for an R app or script.

Typically the meter provider is created automatically, at the first `counter_add()`, `up_down_counter_add()`, `histogram_record()`, `gauge_record()` or `get_meter()` call. otel decides which meter provider class to use based on 'Environment Variables'.

Value

Not applicable.

Implementations

Note that this list is updated manually and may be incomplete.

- meter_provider_noop: No-op meter provider, used when no metrics are emitted.
- otelsdk::meter_provider_file: Save metrics to a JSONL file.
- otelsdk::meter_provider_http: Send metrics to a collector over HTTP/OTLP.
- otelsdk::meter_provider_memory: Collect emitted metrics in memory. For testing.
- otelsdk::meter_provider_stdstream: Write metrics to standard output or error or to a file. Primarily for debugging.

Methods

`meter_provider$get_meter()`:

Get or create a new meter object.

Usage:

```
meter_provider$get_meter(
  name = NULL,
  version = NULL,
  schema_url = NULL,
  attributes = NULL
)
```

Arguments:

- `name`: Meter name, see `get_meter()`.

- `version`: Optional. Specifies the version of the instrumentation scope if the scope has a version (e.g. R package version). Example value: `"1.0.0"`.
- `schema_url`: Optional. Specifies the Schema URL that should be recorded in the emitted telemetry.
- `attributes`: Optional. Specifies the instrumentation scope attributes to associate with emitted telemetry. See `as_attributes()` for allowed values. You can also use `as_attributes()` to convert R objects to OpenTelemetry attributes.

Value:
Returns an OpenTelemetry meter (otel_meter) object.

See also:
`get_default_meter_provider()`, `get_meter()`.

`meter_provider$flush()`:

Force any buffered metrics to flush. Meter providers might not implement this method.

Usage:

```
meter_provider$flush()
```

Value:
Nothing.

`meter_provider$shutdown()`:

Stop the meter provider. Stops collecting and emitting measurements.

Usage:

```
meter_provider$shutdown()
```

Value:
Nothing

See Also

Other low level metrics API: `get_default_meter_provider()`, `get_meter()`, `meter_provider_noop`, `otel_counter`, `otel_gauge`, `otel_histogram`, `otel_meter`, `otel_up_down_counter`

Examples

```
mp <- otel::get_default_meter_provider()
mtr <- mp$get_meter()
mtr$is_enabled()
```

`otel_span` *OpenTelemetry Span Object*

Description

otel_tracer_provider -> otel_tracer -> otel_span -> otel_span_context

Details

An otel_span object represents an OpenTelemetry span.

Use `start_local_active_span()` or `start_span()` to create and start a span.

Call `end_span()` to end a span explicitly. (See `start_local_active_span()` and `local_active_span()` to end a span automatically.)

With otelsdk 0.3.0 or later, span methods do not throw errors. If a method fails, it emits a message of class `otel_error_message` and returns a default value, usually the span itself. Set the `OTEL_ENV` environment variable to `dev` to turn these errors into R errors.

Value

Not applicable.

Lifetime

The span starts when it is created in the `start_local_active_span()` or `start_span()` call.

The span ends when `end_span()` is called on it, explicitly or automatically via `start_local_active_span()` or `local_active_span()`.

Activation

After a span is created it may be active or inactively, independently of its lifetime. A live span (i.e. a span that hasn't ended yet) may be inactive. While this is less common, a span that has ended may still be active.

When otel creates a new span, it sets the parent span of the new span to the active span by default.

Automatic spans:

`start_local_active_span()` creates a new span, starts it and activates it for the caller frame. It also automatically ends the span when the caller frame exits.

Manual spans:

`start_span()` creates a new span and starts it, but it does not activate it. You must activate the span manually using `local_active_span()` or `with_active_span()`. You must also end the span manually with an `end_span()` call. (Or the `end_on_exit` argument of `local_active_span()` or `with_active_span()`.)

Parent spans

OpenTelemetry spans form a hierarchy: a span can refer to a parent span. A span without a parent span is called a root span. A trace is a set of connected spans.

When otel creates a new span, it sets the parent span of the new span to the active span by default.

Alternatively, you can set the parent span of the new span manually. You can also make the new span be a root span, by setting `parent = NA` in `options` to the `start_local_active_span()` or `start_span()` call.

Array attributes

A length one R vector is recorded as a scalar attribute, and longer vectors are recorded as arrays. To record a length one vector as an array, make it an R array, e.g. with `as.array()`:

```
span$set_attribute("gen_ai.response.finish_reasons", as.array("stop"))
```

This works for span attributes, event attributes and link attributes, and also for the attributes of log records and metric measurements.

Methods

`span$add_event()`:

Add a single event to the span.

Usage:

```
span$add_event(name, attributes = NULL, timestamp = NULL)
```

Arguments:

- `name`: Event name.
- `attributes`: Attributes to add to the event. See `as_attributes()` for supported R types. You may also use `as_attributes()` to convert an R object to an OpenTelemetry attribute value.
- `timestamp`: A base::POSIXct object. If missing, the current time is used.

Value:

The span object itself, invisibly.

`span$end()`:

End the span. Calling this method is equivalent to calling the `end_span()` function on the span.

Spans created with `start_local_active_span()` end automatically by default. You must end every other span manually, by calling `end_span`, or using the `end_on_exit` argument of `local_active_span()` or `with_active_span()`.

Calling the `span$end()` method (or `end_span()`) on a span multiple times is not an error, the first call ends the span, subsequent calls do nothing.

Usage:

```
span$end(options = NULL, status_code = NULL)
```

Arguments:

- `options`: Named list of options. Possible entry:
  - `end_steady_time`: A base::POSIXct object that will be used as a steady timer.
- `status_code`: Span status code to set before ending the span, see the `span$set_status()` method for possible values.

Value:

The span object itself, invisibly.

`span$get_context()`:

Get a span's span context. The span context is an otel_span_context object that can be serialized, copied to other processes, and it can be used to create new child spans.

Usage:

```
span$get_context()
```

Value:
An otel_span_context object.

`span$is_recording()`:

Checks whether a span is recorded. If tracing is off, or the span ended already, or the sampler decided not to record the trace the span belongs to.

Usage:

```
span$is_recording()
```

Value:
A logical scalar, `TRUE` if the span is recorded.

`span$record_exception()`:

Record an exception (error, usually) event for a span.

If the span was created with `start_local_active_span()`, or it was ended automatically with `local_active_span()` or `with_active_span()`, then otel records exceptions automatically, and you don't need to call this function manually.

You can still use it to record exceptions that are not R errors.

Usage:

```
span$record_exception(error_condition, attributes, ...)
```

Arguments:

- `error_condition`: An R error object to record.
- `attributes`: Additional attributes to add to the exception event.
- `...`: Passed to the `span$add_event()` method.

Value:
The span object itself, invisibly.

`span$set_attribute()`:

Set a single attribute. It is better to set attributes at span creation, instead of calling this method later, since samplers can only make decisions based on attributes present at span creation.

Usage:

```
span$set_attribute(name, value)
```

Arguments:

- `name`: Attribute name.
- `value`: Attribute value. See `as_attributes()` for supported R types. You may also use `as_attributes()` to convert an R object to an OpenTelemetry attribute value.

Value:
The span object itself, invisibly.

`span$set_attributes()`:

Set multiple attributes at once. It is better to set attributes at span creation, instead of calling this method later, since samplers can only make decisions based on attributes present at span creation.

Usage:

```
span$set_attributes(attributes)
```

Arguments:

- `attributes`: A named list of attributes. See `as_attributes()` for supported R types. You may also use `as_attributes()` to convert an R object to OpenTelemetry attributes.

Value:
The span object itself, invisibly.

`span$set_status()`:

Set the status of the span.

The default span status is `unset`. If the span was created with `start_local_active_span()`, or it was ended automatically with `local_active_span()` or `with_active_span()`, and an error happened in the frame the span was activated for, then otel sets the status of the span automatically to `error`. Otherwise otel leaves the status as `unset`. Set it to `ok` manually to mark a span as explicitly free of errors.

Usage:

```
span$set_status(status_code, description = NULL)
```

Arguments:

- `status_code`: Possible values: unset, ok, error.
- `description`: Optional description, a string.

Value:
The span itself, invisibly.

`span$update_name()`:

Update the span's name. Overrides the name give in `start_local_active_span()` or `start_span()`.

It is undefined whether a sampler will use the original or the new name.

Usage:

```
span$update_name(name)
```

Arguments:

- `name`: String, the new span name.

Value:
The span object itself, invisibly.

See Also

Other low level trace API: `get_default_tracer_provider()`, `get_tracer()`, `otel_span_context`, `otel_tracer`, `otel_tracer_provider`, `tracer_provider_noop`

Examples

```
fn <- function() {
  trc <- otel::get_tracer("myapp")
  spn <- trc$start_span("fn")
  # ...
  spn$set_attribute("key", "value")
```

```
  # ...
  on.exit(spn$end(status_code = "error"), add = TRUE)
  # ...
  spn$end(status_code = "ok")
}
fn()
```

`otel_span_context` *An OpenTelemetry Span Context object*

Description

otel_tracer_provider -> otel_tracer -> otel_span -> otel_span_context

Details

This is a representation of a span that can be serialized, copied to other processes, and it can be used to create new child spans.

Value

Not applicable.

Methods

`span_context$get_span_id()`:
Get the id of the span.

Usage:

```
span_context$get_span_id()
```

Value:
String scalar, a span id. For invalid spans it is invalid_span_id.

`span_context$get_trace_flags()`:
Get the trace flags of a span.
See the specification for more details on trace flags.

Usage:

```
span_context$get_trace_flags()
```

Value:
A list with entries:

- `is_sampled`: logical flag, whether the trace of the span is sampled. If `FALSE` then the caller is not recording the trace. See details in the specification.
- `is_random`: logical flag, it specifies how trace ids are generated. See details in the specification.

`span_context$get_trace_id()`:
Get the id of the trace the span belongs to.

Usage:

```
span_context$get_trace_id()
```

Value:
A string scalar, a trace id. For invalid spans it is invalid_trace_id.

`span_context$is_remote()`:
Whether the span was propagated from a remote parent.

Usage:

```
span_context$is_remote()
```

Value:
A logical scalar.

`span_context$is_sampled()`:
Whether the span is sampled. This is the same as the `is_sampled` trace flags, see `get_trace_flags()` above.

Usage:

```
span_context$is_sampled()
```

Value:
Logical scalar.

`span_context$is_valid()`:
Whether the span is valid. Sometimes otel functions return an invalid span or a span context referring to an invalid span. E.g. `get_active_span_context()` does that if there is no active span.
`is_valid()` checks if the span is valid.
An span id of an invalid span is invalid_span_id.

Usage:

```
span_context$is_valid()
```

Value:
A logical scalar.

`span_context$to_http_headers()`:
Serialize the span context into one or more HTTP headers that can be transmitted to other processes or servers, to create a distributed trace.
The other process can deserialize these headers into a span context that can be used to create new remote spans.

Usage:

```
span_context$to_http_headers()
```

Value:
A named character vector, the HTTP header representation of the span context. Usually includes a `traceparent` header. May include other headers.

See Also

Other low level trace API: `get_default_tracer_provider()`, `get_tracer()`, `otel_span`, `otel_tracer`, `otel_tracer_provider`, `tracer_provider_noop`

Examples

```
spc <- get_active_span_context()
spc$get_trace_flags()
spc$get_trace_id()
spc$get_span_id()
spc$is_remote()
spc$is_sampled()
spc$is_valid()
spc$to_http_headers()
```

---

otel_tracer *OpenTelemetry Tracer Object*

Description

otel_tracer_provider -> otel_tracer -> otel_span -> otel_span_context

Details

Usually you do not need to deal with otel_tracer objects directly. `start_local_active_span()` (and `start_span()`) automatically sets up the tracer and uses it to create spans.

A tracer object is created by calling the `get_tracer()` method of an otel_tracer_provider.

You can use the `start_span()` method of the tracer object to create a span.

Typically there is a separate tracer object for each instrumented R package.

Value

Not applicable.

Methods

`tracer$start_span()`:

Creates and starts a new span.

It does not activate the new span.

It is equivalent to the `start_span()` function.

Usage:

```
tracer_start_span(
  name = NULL,
  attributes = NULL,
  links = NULL,
  options = NULL
)
```

Arguments:

- `name`: Name of the span. If not specified it will be `"<NA>"`.

- `attributes`: Span attributes. OpenTelemetry supports the following R types as attributes: 'character, logical, double, integer. You may use `as_attributes()` to convert other R types to OpenTelemetry attributes.
- `links`: A named list of links to other spans. Every link must be an OpenTelemetry span (otel_span) object, or a list with a span object as the first element and named span attributes as the rest.
- `options`: A named list of span options. May include:
  - `start_system_time`: Start time in system time.
  - `start_steady_time`: Start time using a steady clock.
  - `parent`: A parent span or span context. If it is `NA`, then the span has no parent and it will be a root span. If it is `NULL`, then the current context is used, i.e. the active span, if any.
  - `kind`: Span kind, one of span_kinds: "internal", "server", "client", "producer", "consumer".

Value:
A new otel_span object.

`tracer$is_enabled()`:

Whether the tracer is active and recording traces.

This is equivalent to the `is_tracing_enabled()` function.

Usage:

```
tracer$is_enabled()
```

Value:
Logical scalar.

`tracer$flush()`:

Flush the tracer provider: force any buffered spans to flush. Tracer providers might not implement this method.

Usage:

```
tracer$flush()
```

Value:
Nothing.

See Also

Other low level trace API: `get_default_tracer_provider()`, `get_tracer()`, `otel_span`, `otel_span_context`, `otel_tracer_provider`, `tracer_provider_noop`

Examples

```
tp <- get_default_tracer_provider()
trc <- tp$get_tracer()
trc$is_enabled()
```

`otel_tracer_provider` *OpenTelemetry Tracer Provider Object*

Description

otel_tracer_provider -> otel_tracer -> otel_span -> otel_span_context

Details

The tracer provider defines how traces are exported when collecting telemetry data. It is unlikely that you'd need to use tracer provider objects directly.

Usually there is a single tracer provider for an R app or script.

Typically the tracer provider is created automatically, at the first `start_local_active_span()` or `start_span()` call. otel decides which tracer provider class to use based on 'Environment Variables'.

Value

Not applicable.

Implementations

Note that this list is updated manually and may be incomplete.

- tracer_provider_noop: No-op tracer provider, used when no traces are emitted.
- otelsdk::tracer_provider_file: Save traces to a JSONL file.
- otelsdk::tracer_provider_http: Send traces to a collector over HTTP/OTLP.
- otelsdk::tracer_provider_memory: Collect emitted traces in memory. For testing.
- otelsdk::tracer_provider_stdstream: Write traces to standard output or error or to a file. Primarily for debugging.

Methods

`tracer_provider$get_tracer()`:

Get or create a new tracer object.

Usage:

```
tracer_provider$get_tracer(
  name = NULL,
  version = NULL,
  schema_url = NULL,
  attributes = NULL
)
```

Arguments:

- `name`: Tracer name, see `get_tracer()`.

- `version`: Optional. Specifies the version of the instrumentation scope if the scope has a version (e.g. R package version). Example value: `"1.0.0"`.
- `schema_url`: Optional. Specifies the Schema URL that should be recorded in the emitted telemetry.
- `attributes`: Optional. Specifies the instrumentation scope attributes to associate with emitted telemetry. See `as_attributes()` for allowed values. You can also use `as_attributes()` to convert R objects to OpenTelemetry attributes.

Value:
Returns an OpenTelemetry tracer (otel_tracer) object.

See also:
`get_default_tracer_provider()`, `get_tracer()`.

`tracer_provider$flush()`:

Force any buffered spans to flush. Tracer providers might not implement this method.

Usage:

```
tracer_provider$flush()
```

Value:
Nothing.

`tracer_provider$shutdown()`:

Stop the tracer provider. Flushes any buffered spans and stops emitting spans.

Usage:

```
tracer_provider$shutdown()
```

Value:
Nothing.

See Also

Other low level trace API: `get_default_tracer_provider()`, `get_tracer()`, `otel_span`, `otel_span_context`, `otel_tracer`, `tracer_provider_noop`

Examples

```
tp <- otel::get_default_tracer_provider()
trc <- tp$get_tracer()
trc$is_enabled()
```

---

otel_up_down_counter *OpenTelemetry Up-Down Counter Object*

Description

otel_meter_provider -> otel_meter -> otel_counter, otel_up_down_counter, otel_histogram, otel_gauge

Details

Usually you do not need to deal with otel_up_down_counter objects directly. `up_down_counter_add()` automatically sets up a meter and creates an up-down counter instrument, as needed.

An up-down counter object is created by calling the `create_up_down_counter()` method of an `otel_meter_provider()`.

You can use the `add()` method to increment or decrement the counter.

In R up-down counters are represented by double values.

Value

Not applicable.

Methods

`up_down_counter$add()`:

Increment or decrement the up-down counter by a fixed amount.

Usage:

```
up_down_counter$add(value, attributes = NULL, span_context = NULL, ...)
```

Arguments:

- `value`: Value to increment of decrement the up-down counter with.
- `attributes`: Additional attributes to add.
- `span_context`: Span context. If missing, the active context is used, if any.

Value:
The up-down counter object itself, invisibly.

See Also

Other low level metrics API: `get_default_meter_provider()`, `get_meter()`, `meter_provider_noop`, `otel_counter`, `otel_gauge`, `otel_histogram`, `otel_meter`, `otel_meter_provider`

Examples

```
mp <- get_default_meter_provider()
mtr <- mp$get_meter()
ctr <- mtr$create_up_down_counter("session")
ctr$add(1)
```

pack_http_context *Pack the currently active span context into standard HTTP Open-Telemetry headers*

Description

The returned headers can be sent over HTTP, or set as environment variables for subprocesses.

Usage

```
pack_http_context(tracer = NULL)
```

Arguments

tracer	Tracer object (otel_tracer) or tracer name to use. If NULL, then otel uses an internal tracer. You usually do not need to set this, the active span does not depend on the tracer. Passing a tracer might give the wrong result. If it is a no-op tracer, then the result is an invalid span (context), or no HTTP headers, even if there is an active span. This happens if the tracer's scope is turned off, e.g. via the OTEL_R_SUPPRESS_SCOPES environment variable, or if the tracer was created before tracing was turned on. The default NULL does not have this problem.

Value

A named character vector, with lowercase names. It might be an empty vector, e.g. if tracing is disabled.

See Also

extract_http_context()

Examples

```
hdr <- otel::pack_http_context()
ctx <- otel::extract_http_context()
ctx$is_valid()
```

start_local_active_span *Start and activate a span*

Description

Creates, starts and activates an OpenTelemetry span.

Usually you want this functions instead of `start_span()`, which does not activate the new span.

Usage

```
start_local_active_span(
  name = NULL,
  attributes = NULL,
  links = NULL,
  options = NULL,
  ...,
  tracer = NULL,
  activation_scope = parent.frame(),
  end_on_exit = TRUE
)
```

Arguments

`name`	Name of the span. If not specified, and tracing is enabled, then otel uses the name of the calling function as the span name, and also sets the `code.function.name` attribute to the function name, unless it is already set. otel prefixes this attribute with the package name, if the function is in a package. If otel cannot find the name of the calling function, e.g. at the top level, or for anonymous functions, the span name will be `"<NA>"`.
`attributes`	Span attributes. OpenTelemetry supports the following R types as attributes: ‘character, logical, double, integer. You may use `as_attributes()` to convert other R types to OpenTelemetry attributes.
`links`	A named list of links to other spans. Every link must be an OpenTelemetry span (otel_span) object, or a list with a span object as the first element and named span attributes as the rest.
`options`	A named list of span options. May include: • `start_system_time`: Start time in system time. • `start_steady_time`: Start time using a steady clock. • `parent`: A parent span or span context. If it is `NA`, then the span has no parent and it will be a root span. If it is `NULL`, then the current context is used, i.e. the active span, if any. • `kind`: Span kind, one of span_kinds: "internal", "server", "client", "producer", "consumer".
`...`	Additional arguments are passed to the `start_span()` method of the tracer.

tracer	A tracer object or the name of the tracer to use, see get_tracer(). If NULL then default_tracer_name() is used.
activation_scope	The R scope to activate the span for. Defaults to the caller frame.
end_on_exit	Whether to also end the span when the activation scope exits.

Details

If end_on_exit is TRUE (the default), then it also ends the span when the activation scope finishes.

Value

The new OpenTelemetry span object (of class otel_span), invisibly. See otel_span for information about the returned object.

See Also

Other OpenTelemetry trace API: Zero Code Instrumentation, end_span(), is_tracing_enabled(), local_active_span(), start_span(), tracing-constants, with_active_span()

Examples

```
fn1 <- function() {
  otel::start_local_active_span("fn1")
  fn2()
}
fn2 <- function() {
  otel::start_local_active_span("fn2")
}
fn1()
```

---

start_span *Start an OpenTelemetry span.*

Description

Creates a new OpenTelemetry span and starts it, without activating it.

Usually you want start_local_active_span() instead of start_span. start_local_active_span() also activates the span for the caller frame, and ends the span when the caller frame exits.

Usage

```
start_span(
  name = NULL,
  attributes = NULL,
  links = NULL,
  options = NULL,
  ...,
  tracer = NULL
)
```

Arguments

name	Name of the span. If not specified, and tracing is enabled, then otel uses the name of the calling function as the span name, and also sets the `code.function.name` attribute to the function name, unless it is already set. otel prefixes this attribute with the package name, if the function is in a package. If otel cannot find the name of the calling function, e.g. at the top level, or for anonymous functions, the span name will be `"<NA>"`.
attributes	Span attributes. OpenTelemetry supports the following R types as attributes: 'character, logical, double, integer. You may use `as_attributes()` to convert other R types to OpenTelemetry attributes.
links	A named list of links to other spans. Every link must be an OpenTelemetry span (otel_span) object, or a list with a span object as the first element and named span attributes as the rest.
options	A named list of span options. May include: • `start_system_time`: Start time in system time. • `start_steady_time`: Start time using a steady clock. • `parent`: A parent span or span context. If it is `NA`, then the span has no parent and it will be a root span. If it is `NULL`, then the current context is used, i.e. the active span, if any. • `kind`: Span kind, one of span_kinds: "internal", "server", "client", "producer", "consumer".
...	Additional arguments are passed to the `start_span()` method of the tracer.
tracer	A tracer object or the name of the tracer to use, see `get_tracer()`. If `NULL` then `default_tracer_name()` is used.

Details

Only use `start_span()` is you need to manage the span's activation manually. Otherwise use `start_local_active_span()`.

You must end the span by calling `end_span()`. Alternatively you can also end it with `local_active_span()` or `with_active_span()` by setting `end_on_exit = TRUE`.

It is a good idea to end spans created with `start_span()` in an `base::on.exit()` call.

Value

An OpenTelemetry span (otel_span).

See Also

Other OpenTelemetry trace API: `Zero Code Instrumentation`, `end_span()`, `is_tracing_enabled()`, `local_active_span()`, `start_local_active_span()`, `tracing-constants`, `with_active_span()`

Examples

```
fun <- function() {
  # start span, do not activate
  spn <- otel::start_span("myfun")
  # do not leak resources
  on.exit(otel::end_span(spn), add = TRUE)
  myfun <- function() {
     # activate span for this function
     otel::local_active_span(spn)
     # create child span
     spn2 <- otel::start_local_active_span("myfun/2")
  }

  myfun2 <- function() {
    # activate span for this function
    otel::local_active_span(spn)
    # create child span
    spn3 <- otel::start_local_active_span("myfun/3")
  }
  myfun()
  myfun2()
  end_span(spn)
}
fun()
```

---

tracer_provider_noop *No-op tracer provider*

Description

This is the tracer provider (otel_tracer_provider) otel uses when tracing is disabled.

Details

All methods are no-ops or return objects that are also no-ops.

Value

Not applicable.

See Also

Other low level trace API: `get_default_tracer_provider()`, `get_tracer()`, `otel_span`, `otel_span_context`, `otel_tracer`, `otel_tracer_provider`

Examples

```
tracer_provider_noop$new()
```

tracing-constants *OpenTelemetry tracing constants*

Description

Various constants related OpenTelemetry tracing.

Usage

```
invalid_trace_id

invalid_span_id

span_kinds

span_status_codes
```

Details

`invalid_trace_id`**:**

`invalid_trace_id` is a string scalar, an invalid trace id. If there is no active span, then `get_active_span_context()` returns a span context that has an invalid trace id.

`invalid_span_id`**:**

`invalid_span_id` is a string scalar, an invalid span id. If there is no active span, then `get_active_span_context()` returns a span context that has an invalid span id.

`span_kinds`**:**

`span_kinds` is a character vector listing all possible span kinds. See the OpenTelemetry specification for when to use which.

`span_status_codes`**:**

`span_status_codes` is a character vector listing all possible span status codes. You can set the status code of a a span with the `set_status()` method of otel_span objects. If not set explicitly, and the span is ended automatically (by `start_local_active_span()`, `local_active_span()` or `with_active_span()`) during handling an error, then otel sets the status automatically to "error". Otherwise the status stays "unset".

Value

Not applicable.

See Also

Other OpenTelemetry trace API: `Zero Code Instrumentation`, `end_span()`, `is_tracing_enabled()`, `local_active_span()`, `start_local_active_span()`, `start_span()`, `with_active_span()`

Examples

```
invalid_trace_id
invalid_span_id
span_kinds
span_status_codes
```

`up_down_counter_add` *Increase or decrease an OpenTelemetry up-down counter*

Description

Increase or decrease an OpenTelemetry up-down counter

Usage

```
up_down_counter_add(
  name,
  value = 1L,
  attributes = NULL,
  context = NULL,
  meter = NULL
)
```

Arguments

`name`	Name of the up-down counter.
`value`	Value to add to or subtract from the counter, defaults to 1.
`attributes`	Additional attributes to add.
`context`	Span context. If missing the active context is used, if any.
`meter`	Meter object (otel_meter). Otherwise it is passed to `get_meter()` to get a meter.

Value

The up-down counter object (otel_up_down_counter), invisibly.

See Also

'Environment Variables' needed to enable OpenTelemetry recording.

Other OpenTelemetry metrics instruments: `counter_add()`, `gauge_record()`, `histogram_record()`

Other OpenTelemetry metrics API: `counter_add()`, `gauge_record()`, `histogram_record()`, `is_measuring_enabled()`

Examples

```
otel::up_down_counter_add("session-count", 1)
```

`with_active_span` *Evaluate R code with an active OpenTelemetry span*

Description

Activates the span for evaluating an R expression.

Usually you need this function for spans created with `start_span()`, which does not activate the new span. Usually you don't need it for spans created with `start_local_active_span()`, because it activates the new span automatically.

Usage

```
with_active_span(span, expr, end_on_exit = FALSE)
```

Arguments

`span`	The OpenTelemetry span to activate.
`expr`	R expression to evaluate.
`end_on_exit`	Whether to end after evaluating the R expression.

Details

After `expr` is evaluated (or an error occurs), the span is deactivated and the previously active span will be active again, if there was any.

It is possible to activate the same span for multiple R frames.

Value

The return value of `expr`.

See Also

Other OpenTelemetry trace API: `Zero Code Instrumentation`, `end_span()`, `is_tracing_enabled()`, `local_active_span()`, `start_local_active_span()`, `start_span()`, `tracing-constants`

Other tracing for concurrent code: `local_active_span()`

Examples

```
fun <- function() {
  # start span, do not activate
  spn <- otel::start_span("myfun")
  # do not leak resources
  on.exit(otel::end_span(spn), add = TRUE)
  myfun <- function() {
     otel::with_active_span(spn, {
       # create child span
       spn2 <- otel::start_local_active_span("myfun/2")
```

```
    })
  }

  myfun2 <- function() {
    otel::with_active_span(spn, {
      # create child span
      spn3 <- otel::start_local_active_span("myfun/3")
    })
  }
  myfun()
  myfun2()
  end_span(spn)
}
fun()
```

`Zero Code Instrumentation`

Zero Code Instrumentation

Description

otel supports zero-code instrumentation (ZCI) via the `OTEL_INSTRUMENT_R_PKGS` environment variable. Set this to a comma separated list of package names, the packages that you want to instrument. Then otel will hook up `base::trace()` to produce OpenTelemetry output from every function of these packages.

Details

By default all functions of the listed packages are instrumented. To instrument a subset of all functions set the `OTEL_INSTRUMENT_R_PKGS_<PKG>_INCLUDE` environment variable to a list of glob expressions. `<PKG>` is the package name in all capital letters. Only functions that match to at least one glob expression will be instrumented.

To exclude functions from instrumentation, set the `OTEL_INSTRUMENT_R_PKGS_<PKG>_EXCLUDE` environment variable to a list of glob expressions. `<PKG>` is the package name in all capital letters. Functions that match to at least one glob expression will not be instrumented. Inclusion globs are applied before exclusion globs.

Caveats:

If the user calls `base::trace()` on an instrumented function, that deletes the instrumentation, since the second `base::trace()` call overwrites the first.

Value

Not applicable.

See Also

'Environment Variables' needed to enable OpenTelemetry recording.

Other OpenTelemetry trace API: `end_span()`, `is_tracing_enabled()`, `local_active_span()`, `start_local_active_span()`, `start_span()`, `tracing-constants`, `with_active_span()`

Examples

```
# To run an R script with ZCI:
# OTEL_TRACES_EXPORTER=http OTEL_INSTRUMENT_R_PKGS=dplyr,tidyr R -q -f script.R
```

Index

∗ **OpenTelemetry logs API**
  is_logging_enabled, 25
  log, 29
  log_severity_levels, 30
∗ **OpenTelemetry metrics API**
  counter_add, 4
  gauge_record, 11
  histogram_record, 24
  is_measuring_enabled, 26
  up_down_counter_add, 60
∗ **OpenTelemetry metrics instruments**
  counter_add, 4
  gauge_record, 11
  histogram_record, 24
  up_down_counter_add, 60
∗ **OpenTelemetry trace API**
  end_span, 6
  is_tracing_enabled, 27
  local_active_span, 28
  start_local_active_span, 55
  start_span, 56
  tracing-constants, 59
  with_active_span, 61
  Zero Code Instrumentation, 62
∗ **low level logs API**
  get_default_logger_provider, 18
  get_logger, 21
  otel_logger, 34
  otel_logger_provider, 37
∗ **low level metrics API**
  get_default_meter_provider, 19
  get_meter, 22
  meter_provider_noop, 31
  otel_counter, 31
  otel_gauge, 32
  otel_histogram, 33
  otel_meter, 38
  otel_meter_provider, 41
  otel_up_down_counter, 52
∗ **low level trace API**
  get_default_tracer_provider, 20
  get_tracer, 23
  otel_span, 42
  otel_span_context, 47
  otel_tracer, 49
  otel_tracer_provider, 51
  tracer_provider_noop, 58
∗ **tracing for concurrent code**
  local_active_span, 28
  with_active_span, 61

 Collecting Telemetry Data, *12*

as.array(), *3*, *44*
as_attributes, 3
as_attributes(), *9*, *36*, *38*, *42*, *44–46*, *50*, *52*, *55*, *57*

base::on.exit(), *57*
base::POSIXct, *44*
base::topenv(), *5*
base::trace(), *62*

counter_add, 4
counter_add(), *11*, *25, 26*, *31*, *39*, *41*, *60*

default_tracer_name, 5
default_tracer_name(), *9*, *13*, *23*, *37*, *56, 57*

end_span, 6
end_span(), *14*, *27, 28*, *43, 44*, *56, 57*, *59*, *61, 62*
Environment Variables, *4, 5*, *7*, 7, *10, 11*, *14*, *25–27*, *30*, *37*, *41*, *51*, *60*, *62*
extract_http_context, 10
extract_http_context(), *54*

gauge_record, 11
gauge_record(), *4*, *25, 26*, *32*, *39*, *41*, *60*
get_active_span, 16

get_active_span_context, 17
get_active_span_context(), *48*, *59*
get_default_logger_provider, 18
get_default_logger_provider(), *8*, *22*, *36*, *38*
get_default_meter_provider, 19
get_default_meter_provider(), *8*, *22, 23*, *31–34*, *40*, *42*, *53*
get_default_tracer_provider, 20
get_default_tracer_provider(), *8*, *21*, *23, 24*, *46*, *48*, *50*, *52*, *58*
get_logger, 21
get_logger(), *5*, *18*, *25*, *29*, *36–38*
get_meter, 22
get_meter(), *4, 5*, *11*, *19*, *24*, *26*, *31–34*, *40–42*, *53*, *60*
get_tracer, 23
get_tracer(), *5*, *21*, *27*, *46*, *48*, *50–52*, *56–58*
Getting Started, 12
gettingstarted *(*Getting Started*)*, 12

histogram_record, 24
histogram_record(), *4*, *11*, *26*, *33*, *39*, *41*, *60*

invalid_span_id, *47, 48*
invalid_span_id *(*tracing-constants*)*, 59
invalid_trace_id, *48*
invalid_trace_id *(*tracing-constants*)*, 59
is_logging_enabled, 25
is_logging_enabled(), *30*, *35*
is_measuring_enabled, 26
is_measuring_enabled(), *4*, *11*, *25*, *39*, *60*
is_tracing_enabled, 27
is_tracing_enabled(), *7*, *28*, *50*, *56, 57*, *59*, *61, 62*

local_active_span, 28
local_active_span(), *6, 7*, *14*, *27*, *43–46*, *56, 57*, *59*, *61, 62*
log, 29
log(), *25*, *30*, *34*, *37*
log_debug *(*log*)*, 29
log_error *(*log*)*, 29
log_fatal *(*log*)*, 29
log_info *(*log*)*, 29
log_severity_levels, *21*, *25*, *30*, 30
log_trace *(*log*)*, 29
log_warn *(*log*)*, 29
logger_provider_noop, *18*, *22*, *36–38*

meter_provider_noop, *19*, *23*, 31, *32–34*, *40–42*, *53*

OTEL_ATTRIBUTE_COUNT_LIMIT *(*as_attributes*)*, 3
OTEL_ATTRIBUTE_VALUE_LENGTH_LIMIT *(*as_attributes*)*, 3
otel_counter, *4*, *19*, *23*, *31*, 31, *32–34*, *38–42*, *52, 53*
OTEL_ENV *(*Environment Variables*)*, 7
otel_gauge, *11*, *19*, *23*, *31, 32*, 32, *33, 34*, *38*, *40–42*, *52, 53*
otel_histogram, *19*, *23*, *25*, *31–33*, 33, *38*, *40–42*, *52, 53*
otel_logger, *3*, *18*, *21, 22*, *25*, *29*, *34*, 34, *37, 38*
otel_logger_provider, *18*, *22*, *34*, *36, 37*, 37
OTEL_LOGS_EXPORTER *(*Environment Variables*)*, 7
otel_meter, *4*, *11*, *19*, *22–24*, *26*, *31–34*, *38*, 38, *41, 42*, *52, 53*, *60*
otel_meter_provider, *19*, *23*, *31–34*, *38–41*, 41, *52, 53*
otel_meter_provider(), *31–33*, *53*
OTEL_METRICS_EXPORTER *(*Environment Variables*)*, 7
OTEL_R_EMIT_SCOPES *(*Environment Variables*)*, 7
OTEL_R_INSTRUMENT_PKGS *(*Zero Code Instrumentation*)*, 62
OTEL_R_LOGS_EXPORTER *(*Environment Variables*)*, 7
OTEL_R_METRICS_EXPORTER *(*Environment Variables*)*, 7
OTEL_R_SUPPRESS_SCOPES *(*Environment Variables*)*, 7
OTEL_R_TRACES_EXPORTER *(*Environment Variables*)*, 7
otel_span, *3*, *16*, *21*, *24*, *42*, 42, *47–52*, *55–59*
otel_span_context, *10*, *17*, *21*, *24*, *36*, *42*, *44–47*, 47, *49–52*, *58*
otel_tracer, *3*, *10*, *16, 17*, *21*, *24*, *27*, *42*, *46–49*, 49, *51, 52*, *54*, *58*
otel_tracer_name *(*default_tracer_name*)*, 5
otel_tracer_provider, *20, 21*, *24*, *42*, *46–51*, 51, *58*
OTEL_TRACES_EXPORTER *(*Environment Variables*)*, 7

otel_up_down_counter, *19*, *23*, *31–34*, *38*, *40–42*, *52*, 52, *60*
otelsdk::logger_provider_file, *18*, *37*
otelsdk::logger_provider_http, *18*, *37*
otelsdk::logger_provider_stdstream, *18*, *37*
otelsdk::meter_provider_file, *19*, *41*
otelsdk::meter_provider_http, *19*, *41*
otelsdk::meter_provider_memory, *41*
otelsdk::meter_provider_stdstream, *19*, *41*
otelsdk::tracer_provider_file, *20*, *51*
otelsdk::tracer_provider_http, *20*, *51*
otelsdk::tracer_provider_memory, *51*
otelsdk::tracer_provider_stdstream, *20*, *51*

pack_http_context, 54
pack_http_context(), *10*, *15*

setup_default_logger_provider *(*get_default_logger_provider*)*, 18
setup_default_meter_provider *(*get_default_meter_provider*)*, 19
setup_default_tracer_provider *(*get_default_tracer_provider*)*, 20
span_kinds, *50*, *55*, *57*
span_kinds *(*tracing-constants*)*, 59
span_status_codes, *7*
span_status_codes *(*tracing-constants*)*, 59
start_local_active_span, 55
start_local_active_span(), *6*, *7*, *13*, *14*, *24*, *27*, *28*, *43–46*, *49*, *51*, *56*, *57*, *59*, *61*, *62*
start_span, 56
start_span(), *7*, *14*, *27*, *28*, *43*, *46*, *49*, *51*, *55*, *56*, *59*, *61*, *62*

tracer_provider_noop, *21*, *24*, *46*, *48*, *50–52*, 58
tracing-constants, 59

up_down_counter_add, 60
up_down_counter_add(), *4*, *11*, *25*, *26*, *39*, *41*, *53*
utils::capture.output(), *3*

with_active_span, 61
with_active_span(), *6*, *7*, *14*, *27*, *28*, *43–46*, *56*, *57*, *59*, *62*

Zero Code Instrumentation, *9*, 62