Package: fnware (via r-universe)

November 11, 2026

Title Ready-Made Function Decorators

Version 0.1.0

Description Ready-made function decorators for R, built on the 'adorn' framework. Covers resilience (retrying, timeouts, rate limiting, circuit breaking and fallbacks), caching, call logging, context managers and argument validation, with asynchronous versions for functions that return promises.

License MIT + file LICENSE

URL https://dyfanjones.github.io/fnware/, https://github.com/DyfanJones/fnware

BugReports https://github.com/DyfanJones/fnware/issues

Depends R (>= 4.4.0)

Imports adorn, S7 (>= 0.2.2.9000), rlang, cachem, stats, withr

Remotes DyfanJones/adorn, RConsortium/S7

Suggests callr, later, litedown, logger, mirai, promises, qs2, testthat (>= 3.0.0)

Config/testthat/edition 3

Config/roxygen2/version 8.1.1

Encoding UTF-8

Roxygen list(markdown = TRUE)

Collate 'async_utils.R' 'cache_utils.R' 'async_cache.R' 'circuit_breaker_utils.R' 'async_circuit_breaker.R' 'fallback.R' 'async_fallback.R' 'log_logger.R' 'log_call.R' 'async_log_call.R' 'once.R' 'async_once.R' 'rate_limit.R' 'async_rate_limit.R' 'retry_utils.R' 'retry_after.R' 'retry_before.R' 'retry_retry.R' 'retry_wait.R' 'retry_stop.R' 'async_retry.R' 'timeout_method.R' 'timeout_utils.R' 'async_timeout.R' 'validate.R' 'async_validate.R' 'cache.R' 'circuit_breaker.R' 'contextmanager.R' 'log_extend.R' 'reexport-S7.R' 'reexport-adorn.R' 'retry.R' 'retry_extend.R' 'timeout.R' 'utils.R' 'zzz.R'

VignetteBuilder litedown

Repository https://dyfanjones.r-universe.dev

Date/Publication 2026-11-11 17:02:32 UTC

RemoteUrl https://github.com/DyfanJones/fnware

RemoteRef HEAD

RemoteSha 4e5ee11f56d3c838d0b8a2bf2e2ab22d51755a05

Contents

after_sleep_base . . . 3
after_sleep_none . . . 5
as_decorator . . . 5
async_circuit_breaker . . . 6
async_fallback . . . 7
async_log_call . . . 8
async_lru_cache . . . 9
async_once . . . 10
async_rate_limit . . . 11
async_retry . . . 12
async_timeout . . . 13
async_validate_return . . . 15
before_sleep_none . . . 16
circuit_breaker . . . 16
contextmanager . . . 17
extending_log . . . 19
extending_timeout . . . 20
fallback . . . 21
log_call . . . 22
log_levels . . . 23
logger_base . . . 23
lru_cache . . . 25
once . . . 27
rate_limit . . . 27
retry . . . 28
retry_all . . . 29
stop_never . . . 30
timeout_r . . . 32
validate_args . . . 33
wait_fixed . . . 34
with_context . . . 36

Index **37**

`after_sleep_base` *Write your own stop, wait, retry and hook strategies*

Description

`retry()` is built from five kinds of strategy. Each has an abstract base class and a generic function, both exported, so you can add your own the same way the built-in ones are written with S7:

Usage

```
after_sleep_base()

after_sleep_fn(class, state, ...)

before_sleep_base()

before_sleep_fn(class, state, ...)

retry_base()

retry_fn(class, outcome, ...)

wait_base()

wait_fn(class, attempt, prev_wait = NULL, ...)

stop_base()

stop_fn(class, state, ...)
```

Arguments

`class`	A strategy object. Methods are dispatched on this argument.
`state`, `outcome`, `attempt`, `prev_wait`	Described above.
`...`	Passed to methods.

Details

`retry()` argument	Base class	Generic	The method returns
`stop`	`stop_base`	`stop_fn(class, state)`	`TRUE` to give up
`wait`	`wait_base`	`wait_fn(class, attempt, prev_wait)`	seconds to sleep
`retry`	`retry_base`	`retry_fn(class, outcome)`	`TRUE` to try again
`before_sleep`	`before_sleep_base`	`before_sleep_fn(class, state)`	nothing
`after_sleep`	`after_sleep_base`	`after_sleep_fn(class, state)`	nothing

To add a strategy, make a class that inherits from the base class, then register a method for the generic with `S7::method<-()`. Create the object with `new_object()` on an instance of the base class, as the built-in strategies do. Load S7 with `library(S7)` to get `new_class()`, `new_object()` and `method<-`.

Strategies of the same kind compose with the built-in ones. A new wait works in `wait_chain()` and with `+`, and a new stop works in `stop_any()` and `stop_all()`.

If you register methods from another package, call `S7::methods_register()` in its `.onLoad()`.

Value

`stop_fn()` and `retry_fn()` return a single `TRUE` or `FALSE`, `wait_fn()` returns a number of seconds, and the hook generics return nothing useful.

Arguments passed to methods

- `state`, for `stop_fn()`, `before_sleep_fn()` and `after_sleep_fn()`, is a list describing the attempt that just failed: `attempt` (1-based), `wait` (seconds `retry()` is about to sleep), `elapsed` (seconds since the first attempt), `error` (the condition caught, or `NULL` if `retry_if_result()` rejected a returned value) and `value` (the value returned, or `NULL` if the call errored).
- `attempt` and `prev_wait`, for `wait_fn()`, are the 1-based attempt number and the previous sleep in seconds (`NULL` on the first retry).
- `outcome`, for `retry_fn()`, is `list(value, error)` for the attempt that just finished. `error` is `NULL` when the call returned normally, and `value` is `NULL` when it errored.

Examples

```
library(S7)

# Wait `step` seconds longer after each failure
wait_linear := new_class(
  parent = wait_base,
  properties = list(step = new_property(class_numeric, default = 1))
)

method(wait_fn, wait_linear) <- function(class, attempt, prev_wait = NULL, ...) {
  class@step * attempt
}

f <- retry(
  stop = stop_after_attempt(3),
  wait = wait_linear(0.01) + wait_fixed(0.01)
) %@% function() stop("flaky")
try(f(), silent = TRUE)
```

after_sleep_none *Run a hook after* retry() *sleeps*

Description

- after_sleep_none() does nothing.
- after_sleep_log() passes a message describing the attempt to logger.

Usage

```
after_sleep_none()

after_sleep_log(logger = message)
```

Arguments

logger	Function taking a single message string. Defaults to message().

Value

A hook to pass to retry().

Examples

```
after_sleep_log(function(msg) cat(msg, "\n"))
```

as_decorator *Use a context manager as a decorator*

Description

Decorator form of a contextmanager(): the decorated function runs inside the context, and cleanup runs when it returns or errors. As in Python's ContextDecorator, the context's value is not passed to the function.

Usage

```
as_decorator(ctx, ...)
```

Arguments

ctx	A context manager.
...	Arguments for ctx.

Value

A decorator.

Examples

```
local_tempdir := contextmanager %@% function() {
  path <- tempfile()
  dir.create(path)
  teardown(unlink(path, recursive = TRUE))
  path
}
g := as_decorator(local_tempdir) %@% function() "ran inside context"
g()
```

---

async_circuit_breaker *Stop calling an asynchronous function that keeps failing*

Description

Asynchronous version of `circuit_breaker()`, for functions that return a promise. It has the same states and arguments. A call fails when its promise rejects, or the function throws. Once the circuit is open, calls return a promise already rejected with `fnware_circuit_open`, without running the function.

Usage

```
async_circuit_breaker(f, failures = 5, reset_after = 30)
```

Arguments

`f`	The function to decorate. Usually supplied by `%@%` rather than directly.
`failures`	Number of consecutive errors that open the circuit. A whole number of at least 1. Defaults to 5.
`reset_after`	Seconds the circuit stays open before a trial call. A number greater than 0. Defaults to 30.

Details

Calls overlap, so a failure is counted when its promise rejects, and calls made before earlier ones settle all go through. Two more rules:

- While half-open, only one trial call runs and others are rejected. A trial not settled after `reset_after` seconds is abandoned, so a promise that never settles cannot hold the circuit shut. Put `async_timeout()` inside to bound it sooner.
- A call that started before the circuit opened cannot close it, or be counted again, when it settles.

Stack it outside `async_retry()`. Needs the promises package and an event loop.

Value

A decorator. The decorated function returns a promise.

Examples

```
## Not run:
down <- function() promises::promise_reject(simpleError("service down"))
f := async_circuit_breaker(failures = 2, reset_after = 60) %@% down

# Failures are counted when a promise settles, so let each call settle
try_once <- function() {
  promises::then(f(), onRejected = function(err) cat(class(err)[[1]], "\n"))
  while (!later::loop_empty()) later::run_now(0.1)
}
try_once() # simpleError
try_once() # simpleError, and the circuit opens
try_once() # fnware_circuit_open, down() was not called

fetch := async_circuit_breaker(failures = 3, reset_after = 30) %@%
  async_retry(stop = stop_after_attempt(3)) %@%
  function(url) down()

## End(Not run)
```

`async_fallback` *Return a default when an asynchronous function fails*

Description

Asynchronous version of `fallback()`. The decorated function returns a promise that resolves to `value` (or the result of `handler`) if the wrapped function rejects or throws with a condition in `class`. Other rejections carry on unchanged. Needs the promises package.

Usage

```
async_fallback(f, value = NULL, handler = NULL, class = "error")
```

Arguments

`f`	The function to decorate. Usually supplied by `%@%` rather than directly.
`value`	Value to return on failure. Ignored if `handler` is given.
`handler`	Optional function taking the condition and returning the result.
`class`	Condition classes to catch. Defaults to `"error"`.

Value

A decorator. The decorated function returns a promise.

Examples

```
## Not run:
f := async_fallback(value = "default") %@% function() stop("boom")
promises::then(f(), print)
while (!later::loop_empty()) later::run_now(0.05)

## End(Not run)
```

---

async_log_call *Log the calls to an asynchronous function*

Description

Asynchronous version of log_call(), for functions that return a promise. It logs the start when the function is called, and the finish or the failure when the promise settles, with the time taken up to that point. A rejection is logged then passed on unchanged. Needs the promises package.

Usage

```
async_log_call(f, logger = logger_base(), level = log_levels(), elapsed = TRUE)
```

Arguments

f	The function to decorate. Usually supplied by %@% rather than directly.
logger	A logger, such as logger_message(). See extending_log to use a logging package.
level	A log_levels() giving the level of the start, finish and error events.
elapsed	Include the time taken in the finish message?

Details

It logs no trace event, as the calls that led to a rejection are no longer on the stack by the time it settles.

Value

A decorator. The decorated function returns a promise.

Examples

```
## Not run:
fn := async_log_call(logger = logger_message()) %@%
  function(x) promises::promise_resolve(x)
promises::then(fn(1), print)
while (!later::loop_empty()) later::run_now(0.05)

## End(Not run)
```

async_lru_cache *Cache the results of asynchronous functions*

Description

Asynchronous versions of `lru_cache()`, `cache()` and `disk_cache()`, for functions that return a promise. They take the same arguments and keys, and the decorated function always returns a promise. The wrapped function may return a promise or a plain value.

Usage

```
async_lru_cache(f, maxsize = 1024 * 1024^2, TTL = Inf)

async_cache(f)

async_disk_cache(f, dir, maxsize = 1024 * 1024^2, TTL = Inf, format = c("rds", "qs2"))
```

Arguments

`f`	The function to decorate. Usually supplied by `%@%` rather than directly.
`maxsize`	Maximum cache size in bytes. Defaults to 1 GiB.
`TTL`	Time to live in seconds. Entries older than this are treated as missing and recomputed on the next call. Defaults to `Inf`, so entries never expire. Not available for `cache()`, which never evicts.
`dir`	Directory for `disk_cache()` to keep its files in. It is created if needed. There is no default, so persistence is always a choice.
`format`	File format for `disk_cache()`: `"rds"` (the default) or `"qs2"`, which needs the qs2 package.

Details

- The resolved value is cached, not the promise, so a rejection is not cached and the next call tries again.
- Identical calls made while one is running share its promise, so a burst of the same request makes one call.
- Because only values are stored, `async_disk_cache()` works, and it shares files with `disk_cache()` for the same function.

Needs the promises package and an event loop.

Value

A decorator. The decorated function returns a promise.

Examples

```
## Not run:
calls <- 0
request := async_lru_cache() %@% function(id) {
  calls <<- calls + 1
  promises::promise(function(resolve, reject) {
    later::later(function() resolve(id * 2), 0.05)
  })
}
# Three identical calls at once make one request
for (i in 1:3) promises::then(request(21), function(value) cat(value, "\n"))
while (!later::loop_empty()) later::run_now(0.05)
calls

# Results survive the session on disk
f := async_disk_cache(tempfile()) %@% function(x) x^2

## End(Not run)
```

`async_once` *Run an asynchronous function only once*

Description

Asynchronous version of `once()`, for functions that return a promise. The first call's resolved value is kept and every later call returns a promise of it, ignoring its arguments. Calls made while the first is still running share its promise. A rejection stores nothing, so the next call tries again. Needs the promises package.

Usage

```
async_once(f)
```

Arguments

`f`	The function to decorate. Usually supplied by `%@%` rather than directly.

Value

A decorator. The decorated function returns a promise.

Examples

```
## Not run:
setup := async_once %@% function() {
  cat("setting up\n")
  promises::promise_resolve(42)
}
promises::then(setup(), print)
```

```
promises::then(setup(), print) # no second "setting up"
while (!later::loop_empty()) later::run_now(0.05)

## End(Not run)
```

`async_rate_limit` *Limit how often an asynchronous function can be called*

Description

Asynchronous version of `rate_limit()`. It allows at most `calls` calls to start in any window of `period` seconds, but a call over the limit is scheduled with `later::later()` instead of sleeping, so the session carries on. The decorated function returns a promise that follows the wrapped function's result, a plain value or a promise.

Usage

```
async_rate_limit(f, calls, period = 1)
```

Arguments

`f`	The function to decorate. Usually supplied by `%@%` rather than directly.
`calls`	Maximum number of calls in one window. A whole number of at least 1.
`period`	Length of the window in seconds. Defaults to 1.

Details

Each call reserves its slot when it is made, so calls start in order and queued calls count toward the limit. As with `rate_limit()`, an error still uses its slot. It limits when calls start, not how many run, so slow calls can overlap. Stack it inside `async_retry()` to throttle every attempt.

Needs the promises and later packages, and an event loop.

Value

A decorator. The decorated function returns a promise.

Examples

```
## Not run:
f := async_rate_limit(calls = 2, period = 0.2) %@% function(i) i
start <- Sys.time()
for (i in 1:4) {
  local({
    i <- i
    promises::then(f(i), function(value) {
      cat("call", value, "at", round(as.numeric(Sys.time() - start), 1), "s\n")
    })
  })
```

```
}
while (!later::loop_empty()) later::run_now(0.05)

## End(Not run)
```

`async_retry` *Retry a function without blocking*

Description

Asynchronous version of `retry()`, for functions that return a promise. It uses the same strategies and hooks, but schedules the next attempt with `later::later()` instead of sleeping, so the session is free between attempts.

Usage

```
async_retry(
  f,
  stop = stop_never(),
  wait = wait_none(),
  retry = retry_all(),
  before_sleep = before_sleep_none(),
  after_sleep = after_sleep_none(),
  reraise = FALSE,
  retry_error_callback = NULL
)
```

Arguments

`f`	The function to decorate. Usually supplied by `%@%` rather than directly.
`stop`	A stop condition, e.g. `stop_after_attempt()`. Defaults to never stopping.
`wait`	A wait strategy. Defaults to no waiting.
`retry`	A retry condition. Defaults to retrying every error.
`before_sleep, after_sleep`	Hooks run before and after each sleep, see `before_sleep_log()` and `after_sleep_log()`.
`reraise`	If `TRUE`, re-throw the last error when `stop` fires instead of throwing a `fnware_retry_error`.
`retry_error_callback`	Optional function called with the final state when `stop` fires. Its return value is returned instead of throwing. The state is a list with `attempt`, `wait`, `elapsed`, `error` and `value`.

Details

The decorated function returns a promise. It resolves with the first acceptable result, and rejects with the last error, or a `fnware_retry_error` once `stop` fires. The wrapped function may return a promise or a plain value, and a rejection or a thrown error counts as a failure.

Use it as the outermost decorator. `timeout()` cannot bound a promise, so cap each attempt with `async_timeout()`. The hooks and `retry_error_callback` run in the event loop, so keep them quick. Needs the promises and later packages, and an event loop for the promise to settle.

Value

A decorator. The decorated function returns a promise.

Examples

```
## Not run:
n <- 0
flaky <- function() {
  n <<- n + 1
  if (n < 3) stop("not yet")
  n
}
f := async_retry(stop = stop_after_attempt(5), wait = wait_fixed(0.05)) %@%
  flaky
promises::then(f(), function(value) cat("worked on attempt", value, "\n"))

# Shiny and plumber run the event loop for you. In a script, run it yourself.
while (!later::loop_empty()) later::run_now(0.1)

## End(Not run)
```

`async_timeout` *Bound how long an asynchronous call can take*

Description

Asynchronous version of `timeout()`, for functions that return a promise. If the promise has not settled after `seconds`, the promise returned by the decorated function rejects with a `fnware_timeout_error`, so `async_retry()` retries it.

Usage

```
async_timeout(f, seconds, method = async_timeout_later())

async_timeout_base()

async_timeout_fn(class, call, seconds, ...)

async_timeout_later()
```

Arguments

f	The function to decorate. Usually supplied by %@% rather than directly.
seconds	Time limit for one call, in seconds. A number greater than 0.
method	How to enforce the limit, e.g. async_timeout_later().
class	An async timeout method object, which methods dispatch on.
call	A function of no arguments that starts the work and returns a promise.
...	Passed to methods.

Details

The wrapped function may return a promise or a plain value. A thrown error rejects the promise. Limits:

- It does not stop the work. The caller is told, but the underlying request or task keeps running. A custom method can cancel it.
- It cannot bound a function that blocks the session before returning its promise, as the timer needs the event loop. Use timeout() for that.
- Nested async_timeout()s each run their own timer, so the soonest rejects first. There is no shared deadline like timeout_remaining().

Needs the promises and later packages, and an event loop.

Value

async_timeout() returns a decorator whose decorated function returns a promise. async_timeout_later() returns a method, and async_timeout_fn() returns a promise.

Timeout methods

method decides how the limit is enforced. async_timeout_later(), the only built-in, races the promise against a later::later() timer. To write your own, make an S7 class inheriting from async_timeout_base and register a method for async_timeout_fn(class, call, seconds). call() starts the work and returns a promise. The method must return a promise that settles with its result, or rejects with a fnware_timeout_error once seconds pass. async_timeout() adds the function name and limit, and signals a fnware_timeout_method_error if the method does not return a promise.

Examples

```
## Not run:
# A stand-in for a slow request: resolves after `delay` seconds
slow_request <- function(delay) {
  promises::promise(function(resolve, reject) {
    later::later(function() resolve("response"), delay)
  })
}

f := async_timeout(0.1) %@% slow_request
```

```
promises::then(
  f(1),
  onFulfilled = function(value) cat("got", value, "\n"),
  onRejected = function(err) cat("failed:", conditionMessage(err), "\n")
)
while (!later::loop_empty()) later::run_now(0.1)

# Cap each attempt of an async_retry()
fetch := async_retry(stop = stop_after_attempt(3), wait = wait_exp()) %@%
  async_timeout(5) %@%
  function(url) slow_request(1)

## End(Not run)
```

`async_validate_return` *Validate the value an asynchronous function resolves to*

Description

Asynchronous version of `validate_return()`, for functions that return a promise. The validator checks the resolved value, not the promise. If it fails, the promise returned by the decorated function rejects with a `fnware_validation_error`. Rejections from the wrapped function carry on unchanged. Needs the promises package.

Usage

```
async_validate_return(validator)
```

Arguments

`validator`	For `validate_return()`, a validator for the returned value.

Details

Arguments are available at call time, so `validate_args()` works for asynchronous functions as it is.

Value

A decorator. The decorated function returns a promise.

Examples

```
## Not run:
f := async_validate_return(is.character) %@%
  function() promises::promise_resolve(1)
promises::catch(f(), function(err) cat(conditionMessage(err), "\n"))
while (!later::loop_empty()) later::run_now(0.05)

## End(Not run)
```

before_sleep_none *Run a hook before* retry() *sleeps*

Description

- before_sleep_none() does nothing.
- before_sleep_log() passes a message describing the attempt to logger.

Usage

```
before_sleep_none()

before_sleep_log(logger = message)
```

Arguments

logger	Function taking a single message string. Defaults to message().

Value

A hook to pass to retry().

Examples

```
before_sleep_log(function(msg) cat(msg, "\n"))
```

circuit_breaker *Stop calling a function that keeps failing*

Description

Decorator that stops calling a function after it fails repeatedly, so callers fail fast instead of waiting on something that is down. It has three states:

Usage

```
circuit_breaker(f, failures = 5, reset_after = 30)
```

Arguments

f	The function to decorate. Usually supplied by %@% rather than directly.
failures	Number of consecutive errors that open the circuit. A whole number of at least 1. Defaults to 5.
reset_after	Seconds the circuit stays open before a trial call. A number greater than 0. Defaults to 30.

Details

- **closed**: calls go through. Consecutive errors are counted.
- **open**: after `failures` consecutive errors, calls fail immediately with a `fnware_circuit_open` error, without running the function, for `reset_after` seconds.
- **half-open**: once the cooldown has passed, the next call runs as a trial. If it succeeds the circuit closes. If it fails the circuit opens again for another cooldown.

Only errors count as failures. A call that returns normally, whatever its value, resets the count. Each decorated function keeps its own state.

Stack it outside `retry()` so an open circuit rejects a call before any retries happen.

Value

A decorator.

Examples

```
n <- 0
flaky <- function() {
  n <<- n + 1
  stop("service down")
}
f := circuit_breaker(failures = 2, reset_after = 60) %@% flaky
try(f())
try(f())
try(f()) # rejected without calling flaky()
n # still 2

# Reject before any retries happen
fetch := circuit_breaker(failures = 3, reset_after = 30) %@%
  retry(stop = stop_after_attempt(3)) %@%
  function(url) url
```

`contextmanager` *Write setup and teardown once*

Description

Port of Python's `contextlib.contextmanager`. R has no `yield`, so instead of suspending the function the body calls `teardown()` right after acquiring each resource and returns the resource. From that one definition you get:

Usage

```
contextmanager(f)

teardown(expr)
```

Arguments

f	The function to decorate. Usually supplied by %@% rather than directly.
expr	Code that releases the resource. Evaluated in the setup function's environment when the context ends, so it can use the setup's local variables.

Details

- a local_ form: the function itself. Cleanup runs when the *calling* function exits, like withr::local_*().
- a with_ form: with_context() runs a block, then cleans up.
- a decorator: as_decorator() runs a function inside the context.

The decorated function takes ..., passed on to the setup function, and a .local_envir argument naming the frame whose exit triggers cleanup. It defaults to the caller, or, when called from another context manager's setup, to that context's .local_envir, so composed contexts last as long as the outer one.

teardown(expr) registers expr with withr::defer() on that frame. Cleanups run last in, first out, and the ones already registered still run if the rest of the setup fails. teardown() finds the frame by searching the call stack for the nearest running context manager and reading its .local_envir.

Value

contextmanager is a decorator. teardown() is called for its side effect.

See Also

with_context(), as_decorator()

Examples

```
local_tempdir := contextmanager %@% function(pattern = "tmp") {
  path <- tempfile(pattern)
  dir.create(path)
  teardown(unlink(path, recursive = TRUE))
  path
}

f <- function() {
  d <- local_tempdir()
  writeLines("hi", file.path(d, "x.txt"))
  d
}
d <- f()
dir.exists(d) # cleaned up when f() exited
```

extending_log *Use a logging package with* log_call()

Description

log_call() writes through a logger, see logger_base. To log with another package, inherit from logger_base and register a method for logger_fn(). The package then owns the thresholds, layouts and appenders.

Details

level is the label from the logger's property for the event's level, so "INFO" for an info event. Override the defaults of the trace, debug, info, warn, error and fatal properties if the package names levels differently. A package that needs a logger object can have it passed in through the logger property, available as class@logger.

The message may contain braces, so with the logger package wrap it in skip_formatter() to stop it being treated as a glue template.

Examples

```
library(S7)

# The logger package
## Not run:
library(logger)
logger_logger := new_class(parent = logger_base)

method(logger_fn, logger_logger) <- function(class, level, msg, ...) {
  log_level(level, skip_formatter(msg))
}
f := log_call(logger = logger_logger()) %@% function(x) x
f(1)

## End(Not run)

# log4r, which needs a logger object
## Not run:
library(log4r)
logger_log4r := new_class(parent = logger_base)

method(logger_fn, logger_log4r) <- function(class, level, msg, ...) {
  log_at(class@logger, level, msg)
}
f := log_call(logger = logger_log4r(logger = logger("INFO"))) %@%
  function(x) x
f(1)

## End(Not run)
```

`extending_timeout` *Write your own timeout method*

Description

`timeout()` hands the work of enforcing a limit to a *method*. `timeout_r()`, which uses `setTimeLimit()`, is the built-in. To add another, for example one that runs the call in a separate process and kills it, make an S7 class that inherits from `timeout_base` and register a method for `timeout_fn()` with `S7::method<-()`.

Usage

```
timeout_base()

timeout_fn(class, call, seconds, ...)
```

Arguments

`class`	A timeout method object. Methods are dispatched on this argument.
`call`	A function of no arguments that runs the decorated function.
`seconds`	Seconds left before the effective deadline.
`...`	Passed to methods.

Details

`timeout_fn(class, call, seconds)` gets `call`, a function of no arguments that runs the decorated function, and `seconds`, the time left before the effective deadline. It must return the value of `call()` unchanged, or signal a `fnware_timeout_error` when `seconds` run out. `timeout()` adds the function name and limit to the message, and signals a `fnware_timeout_method_error` if `call()` ran but the method returned something else or swallowed its error. Other errors should propagate.

A method that runs `call()` in another process cannot be checked for this, and `timeout_remaining()` does not see the parent's deadline there. Set the environment variable `FNWARE_TIMEOUT_DEADLINE` in the child to `as.numeric(Sys.time()) + seconds` to pass it on. The retry vignette shows a callr method that does. Packages that register methods need `S7::methods_register()` in `.onLoad()`.

Value

`timeout_fn()` returns the value of `call()`.

Examples

```
library(S7)

# Report the time allowed, then run the call without enforcing anything
timeout_log <- new_class("timeout_log", parent = timeout_base)
```

```
method(timeout_fn, timeout_log) <- function(class, call, seconds, ...) {
  message("allowing ", format(seconds), " seconds")
  call()
}

f := timeout(3, method = timeout_log()) %@% function() "ok"
f()
```

`fallback` *Return a default when a function fails*

Description

Decorator that catches a failure and returns `value`, or the result of `handler(cnd)`, instead of signalling it. Only errors are caught by default. A caught condition ends the call, as with `tryCatch()`.

Usage

```
fallback(f, value = NULL, handler = NULL, class = "error")
```

Arguments

`f`	The function to decorate. Usually supplied by `%@%` rather than directly.
`value`	Value to return on failure. Ignored if `handler` is given.
`handler`	Optional function taking the condition and returning the result.
`class`	Condition classes to catch. Defaults to `"error"`.

Details

Stack it outside `retry()` and `circuit_breaker()` so the default is used only once retries are spent or the circuit is open.

Value

A decorator.

Examples

```
f := fallback(value = NA_real_) %@% function(x) {
  if (x < 0) stop("negative") else sqrt(x)
}
f(4)
f(-1)

g := fallback(handler = function(cnd) conditionMessage(cnd)) %@%
  function() stop("boom")
g()
```

```
# Use the default only after retries and the circuit breaker give up
fetch := fallback(value = NULL) %@%
  circuit_breaker(failures = 3) %@%
  retry(stop = stop_after_attempt(3)) %@%
  function(url) stop("down")
fetch("x")
```

`log_call` *Log the calls to a function*

Description

Decorator that logs when a call starts, and when it finishes or fails. Each event is written by `logger`, see logger_base. A failure is logged then signalled again unchanged.

Usage

```
log_call(f, logger = logger_base(), level = log_levels(), elapsed = TRUE)
```

Arguments

`f`	The function to decorate. Usually supplied by `%@%` rather than directly.
`logger`	A logger, such as `logger_message()`. See extending_log to use a logging package.
`level`	A `log_levels()` giving the level of the start, finish and error events.
`elapsed`	Include the time taken in the finish message?

Details

The message holds the function's name, its arguments matched to the formals and, once finished, the time taken. Arguments are evaluated before the start is logged.

When a call fails, the calls from `f` down to the failure are logged as a separate `trace` event after the error. Whether it is shown is up to the logger: the built-in loggers hide it until `threshold = "trace"`, see logger_base.

For a function that returns a promise, use `async_log_call()`. Here the finish would be logged when the promise is returned, not when it settles.

Value

A decorator.

Examples

```
fn := log_call(logger = logger_message()) %@% function(a, b, c) a + b + c
fn(1, 2, 3)

# Log failures as warnings
g := log_call(
  logger = logger_message(),
  level = log_levels(error = "warn")
) %@% function(x) stop("boom")
try(g(1), silent = TRUE)
```

---

log_levels *The level of each event* log_call() *logs*

Description

The level of each event log_call() logs

Usage

```
log_levels(start = "info", finish = "info", error = "error")
```

Arguments

start, finish, error	The level of the event: one of "trace", "debug", "info", "warn", "error" or "fatal".

Value

An object to pass to the level argument of log_call().

Examples

```
log_levels(finish = "trace", error = "warn")
```

---

logger_base *Loggers*

Description

log_call() hands each message to a logger. A logger is an S7 object that inherits from logger_base, and logger_fn() decides how it is written.

Usage

```
logger_base(
  trace = "TRACE",
  debug = "DEBUG",
  info = "INFO",
  warn = "WARN",
  error = "ERROR",
  fatal = "FATAL",
  threshold = "info",
  logger = NULL
)

logger_fn(class, level, msg, ...)

logger_message(
  trace = "TRACE",
  debug = "DEBUG",
  info = "INFO",
  warn = "WARN",
  error = "ERROR",
  fatal = "FATAL",
  threshold = "info",
  logger = NULL
)
```

Arguments

trace, debug, info, warn, error, fatal	Labels passed to `logger_fn()` for each level. They default to `"TRACE"`, `"DEBUG"` and so on. Change them when the logging package names its levels differently, or uses objects.
threshold	The lowest level the built-in loggers write: one of `"trace"`, `"debug"`, `"info"`, `"warn"`, `"error"` or `"fatal"`.
logger	An object a logging package needs to be passed, such as a log4r logger. Available to methods as `class@logger`.
class	A logger. Methods are dispatched on this argument.
level	The level label of the event, taken from the property of `class` named by the event's level, e.g. `class@info`.
msg	The message, a string.
...	Passed to methods.

Details

- `logger_base` is the base class. The default method prints `LEVEL [timestamp]: message` to the console.
- `logger_message` writes the same `LEVEL [timestamp]: message` with `message()`, so it can be silenced with `suppressMessages()`.

- `logger_fn(class, level, msg, ...)` is the generic. Add a method to support any logging package, see extending_log.

`logger_base` and `logger_message` hide events below `threshold`. The `trace` events, such as the calls leading to an error, are below the default `"info"`, so set `threshold = "trace"` to see them. A method for another logging package is free to ignore `threshold` and use that package's own.

Value

`logger_fn()` returns nothing useful. The others return a logger.

Examples

```
logger_message()
f := log_call(logger = logger_message()) %@% function(x) x
f(1)
```

`lru_cache` *Cache function results*

Description

Decorators that remember a function's results, keyed on the matched arguments (including defaults).

Usage

```
lru_cache(f, maxsize = 1024 * 1024^2, TTL = Inf)

cache(f)

disk_cache(f, dir, maxsize = 1024 * 1024^2, TTL = Inf, format = c("rds", "qs2"))
```

Arguments

`f`	The function to decorate. Usually supplied by `%@%` rather than directly.
`maxsize`	Maximum cache size in bytes. Defaults to 1 GiB.
`TTL`	Time to live in seconds. Entries older than this are treated as missing and recomputed on the next call. Defaults to `Inf`, so entries never expire. Not available for `cache()`, which never evicts.
`dir`	Directory for `disk_cache()` to keep its files in. It is created if needed. There is no default, so persistence is always a choice.
`format`	File format for `disk_cache()`: `"rds"` (the default) or `"qs2"`, which needs the qs2 package.

Details

- `lru_cache()` evicts the least recently used entries when the cache is full. Port of Python's `functools.lru_cache`.
- `cache()` never evicts. Port of Python's `functools.cache`.
- `disk_cache()` stores results as files in `dir`, so they survive the R session and can be shared between sessions. Like `joblib.Memory.cache`.

Entries are keyed on the function's formals and body as well as its arguments, so editing the function invalidates them. Changes to anything it calls do not. `disk_cache()` saves results with `saveRDS()`, or with the qs2 package when `format = "qs2"`, which is usually faster and smaller for large objects. Neither round-trips environments or external pointers. Files from the two formats are not shared, so changing `format` for an existing `dir` starts with an empty cache.

Set `TTL` on `lru_cache()` or `disk_cache()` to expire entries after a number of seconds, for results that go stale (for example, data fetched from an API).

Call `lru_cache(maxsize)` to build a decorator, then apply it with `%@%` or by calling it on a function. `cache` takes no parameters, so it is applied directly.

Value

A decorator.

Examples

```
slow_square := lru_cache() %@% function(x) {
  Sys.sleep(0.1)
  x^2
}
slow_square(4) # slow
slow_square(4) # cached

# Entries expire after 60 seconds
fresh_square := lru_cache(TTL = 60) %@% function(x) x^2

slow_cube := cache %@% function(x) {
  Sys.sleep(0.1)
  x^3
}
slow_cube(2) # slow
slow_cube(2) # cached

dir <- tempfile("cache")
slow_cube := disk_cache(dir) %@% function(x) {
  Sys.sleep(0.1)
  x^3
}
slow_cube(2) # slow, and saved to `dir`
slow_cube(2) # cached
```

once *Run a function only once*

Description

Decorator that runs the function on its first successful call and returns that result on every call after it, ignoring later arguments. An error stores nothing, so the next call tries again. Unlike cache, it keeps one result whatever the arguments.

Usage

```
once(f)
```

Arguments

f	The function to decorate. Usually supplied by %@% rather than directly.

Details

For a function that returns a promise, use async_once(), which also does not store a rejection.

Value

A decorator.

Examples

```
setup := once %@% function() {
  cat("setting up\n")
  42
}
setup()
setup() # no output, same result
```

rate_limit *Limit how often a function can be called*

Description

Decorator that allows at most calls calls in any window of period seconds. A call that would exceed the limit sleeps until the oldest call in the window has aged out. Calls are counted when they start, so a call that errors still uses up its slot, as it does for an API.

Usage

```
rate_limit(f, calls, period = 1)
```

Arguments

f	The function to decorate. Usually supplied by %@% rather than directly.
calls	Maximum number of calls in one window. A whole number of at least 1.
period	Length of the window in seconds. Defaults to 1.

Details

Call rate_limit(calls, period) to build a decorator, then apply it with %@% or by calling it on a function. Each decorated function keeps its own record of calls, so two functions decorated separately have separate limits.

Stack it inside retry() so that every attempt is throttled, including retries. Stacking it outside would only throttle whole calls, and retries could still exceed the limit.

Value

A decorator.

Examples

```
f := rate_limit(calls = 3, period = 1) %@% function(x) x
vapply(1:3, f, numeric(1)) # immediate
system.time(f(4)) # waits for the window to clear

# Throttle every attempt, retries included
fetch := retry(stop = stop_after_attempt(3)) %@%
  rate_limit(calls = 5, period = 1) %@%
  function(url) url
```

---

retry *Retry a function when it fails*

Description

Decorator that calls a function again until it succeeds or a stop condition is met, sleeping between attempts. Port of Python's tenacity.

Usage

```
retry(
  f,
  stop = stop_never(),
  wait = wait_none(),
  retry = retry_all(),
  before_sleep = before_sleep_none(),
  after_sleep = after_sleep_none(),
  reraise = FALSE,
  retry_error_callback = NULL
)
```

Arguments

f	The function to decorate. Usually supplied by %@% rather than directly.
stop	A stop condition, e.g. stop_after_attempt(). Defaults to never stopping.
wait	A wait strategy. Defaults to no waiting.
retry	A retry condition. Defaults to retrying every error.
before_sleep, after_sleep	Hooks run before and after each sleep, see before_sleep_log() and after_sleep_log().
reraise	If TRUE, re-throw the last error when stop fires instead of throwing a fnware_retry_error.
retry_error_callback	Optional function called with the final state when stop fires. Its return value is returned instead of throwing. The state is a list with attempt, wait, elapsed, error and value.

Details

Call retry(stop, wait, retry, before_sleep, after_sleep, reraise, retry_error_callback) to build a decorator, then apply it with %@% or by calling it on a function.

Value

A decorator. The decorated function returns the first acceptable result, or signals a fnware_retry_error condition (with $attempt, $error and $value) once stop fires.

Examples

```
n <- 0
flaky <- function() {
  n <<- n + 1
  if (n < 3) stop("not yet")
  n
}
retry(stop = stop_after_attempt(5))(flaky)()

# Back off exponentially between attempts
f := retry(stop = stop_after_attempt(3), wait = wait_exp(min = 0.1)) %@% function() 1
f()
```

---

retry_all *Retry conditions*

Description

Decide whether retry() tries again after an attempt.

Usage

```
retry_all()

retry_never()

retry_if_error(classes = "error")

retry_if_error_message(message = NULL, match = NULL)

retry_if_result(predicate = function() NULL)
```

Arguments

`classes`	Condition classes to retry on.
`message`	Error message to match exactly.
`match`	Regular expression to match against the error message.
`predicate`	Function taking the returned value and returning `TRUE` to retry.

Details

- `retry_all()` retries on any error.
- `retry_never()` never retries, so the first error is re-thrown.
- `retry_if_error()` retries on errors inheriting from any of `classes`.
- `retry_if_error_message()` retries when the error message equals `message`, or matches the regular expression `match`. Supply exactly one.
- `retry_if_result()` retries when the call succeeds but `predicate(value)` is `TRUE`.

Value

A retry condition to pass to `retry()`.

Examples

```
retry_if_error("http_error")
retry_if_error_message(match = "timed? ?out")
retry_if_result(is.null)
```

`stop_never` *Stop conditions*

Description

Decide when `retry()` gives up. A condition is checked after each failed attempt.

Usage

```
stop_never()

stop_after_attempt(max_attempt_number = 1)

stop_after_delay(max_delay = 1)

stop_before_delay(max_delay = 1)

stop_any(...)

stop_all(...)
```

Arguments

`max_attempt_number`	Number of attempts to allow.
`max_delay`	Seconds since the first attempt.
`...`	Stop conditions to combine.

Details

- `stop_never()` never stops.
- `stop_after_attempt()` stops after a number of attempts.
- `stop_after_delay()` stops once that many seconds have passed since the first attempt.
- `stop_before_delay()` stops if sleeping again would reach that many seconds.
- `stop_any()` stops when any of the given conditions is met.
- `stop_all()` stops when all of the given conditions are met. `stop_after_delay()` and `stop_before_delay()` only check the clock between attempts. They do not interrupt an attempt that is already running, so a call that hangs never gets back to them. Use `timeout()` to bound a single attempt.

Value

A stop condition to pass to `retry()`.

Examples

```
stop_after_attempt(3)
stop_any(stop_after_attempt(5), stop_after_delay(10))
```

`timeout_r` *Bound how long a call can run*

Description

Decorator that stops a single call after `seconds` and signals a `fnware_timeout_error`. Unlike `stop_after_delay()`, which only checks between attempts, it bounds the call itself.

Usage

```
timeout_r()

timeout(f, seconds, method = timeout_r())

timeout_remaining()
```

Arguments

`f`	The function to decorate. Usually supplied by `%@%` rather than directly.
`seconds`	Time limit for one call, in seconds. A number greater than 0.
`method`	How to enforce the limit: a timeout method such as `timeout_r()`. See `timeout_fn()` to write your own.

Details

The condition inherits from `error`, so `retry_if_error()` retries it and `circuit_breaker()` counts it as a failure. It records the limit in `$limit` and the function name in `$fn`. Stack `timeout()` inside `retry()` so each attempt is capped. Outside `retry()` the limit is already spent when the error is caught, so later attempts run unbounded.

Timeouts can nest. The soonest active deadline always applies, so an inner timeout never outlasts an outer one. `timeout_remaining()` returns the seconds left before that deadline, or `Inf` outside any `timeout()`, so you can pass the budget to a library with its own timeout. It does not see deadlines in other R processes unless a method passes one on, see `timeout_fn()`.

For functions that return a promise, use `async_timeout()`.

Value

`timeout()` returns a decorator, `timeout_r()` a timeout method and `timeout_remaining()` a number of seconds.

Limitations

`timeout_r()` uses `setTimeLimit()`, which R only checks between R expressions. It cannot interrupt compiled code that never checks for interrupts (many blocking network and database calls), or a single long `Sys.sleep()`. For those, pass `timeout_remaining()` to a native timeout or use a process-based method. Code that catches every error around the slow part will also catch the time limit error.

Timeout methods

`method` decides how the limit is enforced. `timeout_r()` is the only built-in method. To write your own, see `timeout_fn()`.

Examples

```
# A loop of short steps is interrupted between steps
slow <- function() {
  for (i in 1:100) Sys.sleep(0.05)
  "done"
}
f := timeout(0.2) %@% slow
try(f())

# Each attempt is capped, and a timeout is retried
attempts <- 0
flaky := retry(stop = stop_after_attempt(3)) %@%
  timeout(0.2) %@%
  function() {
    attempts <<- attempts + 1
    if (attempts < 3) slow() else "worked"
  }
flaky()

# Pass the remaining budget on
budget := timeout(10) %@% function() timeout_remaining()
budget()
```

---

validate_args *Validate arguments and return values*

Description

Decorators that check a function's inputs or output against validators you supply. A validator is any function (or `rlang::as_function()` formula) that takes the value, so base predicates, lambdas and the `check_*()` functions of packages like checkmate all work, with no dependency on any of them.

Usage

```
validate_args(...)

validate_return(validator)
```

Arguments

`...`	For `validate_args()`, named validators, one per argument.
`validator`	For `validate_return()`, a validator for the returned value.

Details

A validator passes if it returns exactly `TRUE`. It fails if it returns anything else:

- a string, which becomes the failure message (the `check_*()` convention),
- `FALSE` or any other value, which gives a generic message.

If a validator signals an error, that error propagates unchanged, so `assert_*()` style functions work too.

`validate_args()` checks the effective value of each named argument, so defaults are validated too. Checking forces the argument, so a validated argument is evaluated at call time rather than lazily. All failures are reported in one error of class `fnware_validation_error`. Validator names are checked against the function's formals when it is decorated, so a typo fails early.

`validate_return()` checks the value the function returns. For a function that returns a promise, use `async_validate_return()`, which checks the resolved value.

Value

A decorator.

Examples

```
f := validate_args(x = is.numeric, n = ~ .x > 0) %@% function(x, n = 1) x * n
f(2, 3)
try(f("a", -1))

# check_*() style validators return a message on failure
check_string <- function(x) if (is.character(x) && length(x) == 1) TRUE else "must be a string"
g := validate_args(name = check_string) %@% function(name) name
try(g(1))

h := validate_return(is.character) %@% function() 1
try(h())
```

`wait_fixed` *Wait strategies*

Description

Decide how many seconds `retry()` sleeps after a failed attempt.

Usage

```
wait_fixed(wait = integer(0))

wait_none()

wait_exp(multiplier = 1, min = 0, max = 4611686018427387904, exp_base = 2)
```

```
wait_exp_jitter(base = 0.1, max_delay = 20, jitter = "full")

wait_rand(min = 0, max = 1)

wait_chain(...)

wait_combine(...)
```

Arguments

`wait`	Seconds to wait.
`multiplier`, `exp_base`	Scale and base of the exponential.
`min`, `max`	Lower and upper bound in seconds.
`base`	Seconds to wait on the first attempt.
`max_delay`	Longest wait in seconds.
`jitter`	Jitter type, one of `"full"`, `"equal"`, `"decorrelated"` or `"none"`.
`...`	Wait strategies to chain or combine.

Details

- `wait_none()` does not wait.
- `wait_fixed()` waits a fixed time.
- `wait_rand()` waits a random time between `min` and `max`.
- `wait_exp()` waits `multiplier * exp_base^(attempt - 1)`, kept between `min` and `max`.
- `wait_exp_jitter()` waits `base * 2^(attempt - 1)`, capped at `max_delay`, then applies `jitter`: `"none"`, `"full"` (uniform from 0), `"equal"` (half fixed, half random) or `"decorrelated"` (random between `base` and 3 times the previous wait).
- `wait_chain()` uses the n-th strategy for the n-th attempt, then repeats the last.
- `wait_combine()` adds strategies together. `a + b` is the same as `wait_combine(a, b)`.

Value

A wait strategy to pass to `retry()`.

Examples

```
wait_fixed(2)
wait_exp(multiplier = 0.5, max = 30)
wait_chain(wait_fixed(1), wait_fixed(5))
wait_fixed(1) + wait_rand(0, 2)
```

`with_context` *Run code inside a context*

Description

The scoped form of a `contextmanager()`. Evaluates `ctx` with cleanup tied to this call, passes the value it returns to `code`, and cleans up when `code` finishes, even if it errors.

Usage

```
with_context(ctx, code)
```

Arguments

`ctx`	A call to a context manager, e.g. `local_tempdir()`. It is not evaluated in your frame: `.local_envir` is set to this function's frame.
`code`	A function taking the context's value.

Value

The value of `code`.

Examples

```
local_tempdir := contextmanager %@% function() {
  path <- tempfile()
  dir.create(path)
  teardown(unlink(path, recursive = TRUE))
  path
}
with_context(local_tempdir(), function(d) dir.exists(d))
```

Index

after_sleep_base, 3
after_sleep_fn *(*after_sleep_base*)*, 3
after_sleep_log *(*after_sleep_none*)*, 5
after_sleep_log(), *12*, *29*
after_sleep_none, 5
as_decorator, 5
as_decorator(), *18*
async_cache *(*async_lru_cache*)*, 9
async_circuit_breaker, 6
async_disk_cache *(*async_lru_cache*)*, 9
async_fallback, 7
async_log_call, 8
async_log_call(), *22*
async_lru_cache, 9
async_once, 10
async_once(), *27*
async_rate_limit, 11
async_retry, 12
async_retry(), *6*, *11*, *13*
async_timeout, 13
async_timeout(), *6*, *13*, *32*
async_timeout_base *(*async_timeout*)*, 13
async_timeout_fn *(*async_timeout*)*, 13
async_timeout_later *(*async_timeout*)*, 13
async_validate_return, 15
async_validate_return(), *34*

before_sleep_base *(*after_sleep_base*)*, 3
before_sleep_fn *(*after_sleep_base*)*, 3
before_sleep_log *(*before_sleep_none*)*, 16
before_sleep_log(), *12*, *29*
before_sleep_none, 16

cache, *27*
cache *(*lru_cache*)*, 25
cache(), *9*
circuit_breaker, 16
circuit_breaker(), *6*, *21*, *32*
contextmanager, 17
contextmanager(), *5*, *36*

disk_cache *(*lru_cache*)*, 25
disk_cache(), *9*

extending_log, *8*, 19, *22*, *25*
extending_retry *(*after_sleep_base*)*, 3
extending_timeout, 20

fallback, 21
fallback(), *7*

later::later(), *11*, *12*, *14*
log_call, 22
log_call(), *8*, *19*, *23*
log_levels, 23
log_levels(), *8*, *22*
logger, *8*, *22*
logger_base, *19*, *22*, 23
logger_fn *(*logger_base*)*, 23
logger_message *(*logger_base*)*, 23
logger_message(), *8*, *22*
lru_cache, 25
lru_cache(), *9*

message(), *5*, *16*, *24*

once, 27
once(), *10*

promise, *6–13*, *15*

rate_limit, 27
rate_limit(), *11*
retry, 28
retry condition, *12*, *29*
retry(), *3*, *5*, *12*, *16*, *17*, *21*, *28–32*, *34*, *35*
retry_all, 29
retry_base *(*after_sleep_base*)*, 3
retry_fn *(*after_sleep_base*)*, 3
retry_if_error *(*retry_all*)*, 29
retry_if_error(), *32*
retry_if_error_message *(*retry_all*)*, 29

retry_if_result *(retry_all)*, 29
retry_never *(retry_all)*, 29
rlang::as_function(), *33*

S7::methods_register(), *4*, *20*
saveRDS(), *26*
setTimeLimit(), *20*, *32*
stop condition, *28*
stop_after_attempt *(stop_never)*, 30
stop_after_attempt(), *12*, *29*
stop_after_delay *(stop_never)*, 30
stop_after_delay(), *32*
stop_all *(stop_never)*, 30
stop_all(), *4*
stop_any *(stop_never)*, 30
stop_any(), *4*
stop_base *(after_sleep_base)*, 3
stop_before_delay *(stop_never)*, 30
stop_fn *(after_sleep_base)*, 3
stop_never, 30
suppressMessages(), *24*

teardown *(contextmanager)*, 17
timeout *(timeout_r)*, 32
timeout(), *13, 14*, *20*, *31*
timeout_base *(extending_timeout)*, 20
timeout_fn *(extending_timeout)*, 20
timeout_fn(), *32, 33*
timeout_r, 32
timeout_remaining *(timeout_r)*, 32
timeout_remaining(), *14*, *20*
tryCatch(), *21*

validate_args, 33
validate_args(), *15*
validate_return *(validate_args)*, 33
validate_return(), *15*

wait strategy, *12*, *29*
wait_base *(after_sleep_base)*, 3
wait_chain *(wait_fixed)*, 34
wait_chain(), *4*
wait_combine *(wait_fixed)*, 34
wait_exp *(wait_fixed)*, 34
wait_exp_jitter *(wait_fixed)*, 34
wait_fixed, 34
wait_fn *(after_sleep_base)*, 3
wait_none *(wait_fixed)*, 34
wait_rand *(wait_fixed)*, 34
with_context, 36
with_context(), *18*
withr::defer(), *18*