Package ‘vcr’

March 10, 2025

Title Record 'HTTP' Calls to Disk

Description Record test suite 'HTTP' requests and replays them during future runs. A port of the Ruby gem of the same name (<https://github.com/vcr/vcr/>). Works by hooking into the 'webmockr' R package for matching 'HTTP' requests by various rules ('HTTP' method, 'URL', query parameters, headers, body, etc.), and then caching real 'HTTP' responses on disk in 'cassettes'. Subsequent 'HTTP' requests matching any previous requests in the same 'cassette' use a cached 'HTTP' response.

Version 1.7.0

URL https://github.com/ropensci/vcr/, https://books.ropensci.org/http-testing/, https://docs.ropensci.org/vcr/

BugReports https://github.com/ropensci/vcr/issues

License MIT + file LICENSE

Encoding UTF-8

Language en-US

LazyData true

VignetteBuilder knitr

Imports crul (>= 0.8.4), httr, httr2, webmockr (>= 0.8.0), urltools, yaml, R6, base64enc, rprojroot

Suggests roxygen2 (>= 7.2.1), jsonlite, testthat, knitr, rmarkdown, desc, crayon, cli, curl, withr, webfakes

X-schema.org-applicationCategory Web

X-schema.org-keywords http, https, API, web-services, curl, mock, mocking, http-mocking, testing, testing-tools, tdd

X-schema.org-isPartOf https://ropensci.org

RoxygenNote 7.3.2

NeedsCompilation no

Author Scott Chamberlain [aut, cre] (<https://orcid.org/0000-0003-1444-9135>),
Aaron Wolen [aut] (<https://orcid.org/0000-0003-2542-2202>),
Maëlle Salmon [aut] (<https://orcid.org/0000-0002-2815-0399>),
Daniel Possenriede [aut] (<https://orcid.org/0000-0002-6738-9845>),
rOpenSci [fnd] (019jywm96)

Maintainer Scott Chamberlain <myrmecocystus@gmail.com>

Repository CRAN

Date/Publication 2025-03-10 10:10:02 UTC

Contents

`as.cassette` *Coerce names, etc. to cassettes*

Description

Coerce names, etc. to cassettes

Coerce to a cassette path

Usage

```
as.cassette(x, ...)

as.cassettepath(x)
```

Arguments

`x`	Input, a cassette name (character), or something that can be coerced to a cassette
`...`	further arguments passed on to `cassettes()` or [read_cassette_meta()

Value

a cassette of class `Cassette`

Examples

```
## Not run:
vcr_configure(dir = tempfile())
insert_cassette("foobar")
cassettes(on_disk = FALSE)
cassettes(on_disk = TRUE)
as.cassette("foobar", on_disk = FALSE)
eject_cassette() # eject the current cassette

# cleanup
unlink(file.path(tempfile(), "foobar.yml"))

## End(Not run)
```

`cassettes` *List cassettes, get current cassette, etc.*

Description

List cassettes, get current cassette, etc.

Usage

```
cassettes(on_disk = TRUE, verb = FALSE)

current_cassette()

cassette_path()
```

Arguments

`on_disk`	(logical) Check for cassettes on disk + cassettes in session (`TRUE`), or check for only cassettes in session (`FALSE`). Default: `TRUE`
`verb`	(logical) verbose messages

Details

- `cassettes()`: returns cassettes found in your R session, you can toggle whether we pull from those on disk or not
- `current_cassette()`: returns an empty list when no cassettes are in use, while it returns the current cassette (a `Cassette` object) when one is in use
- `cassette_path()`: just gives you the current directory path where cassettes will be stored

Examples

```
vcr_configure(dir = tempdir())

# list all cassettes
cassettes()
cassettes(on_disk = FALSE)

# list the currently active cassette
insert_cassette("stuffthings")
current_cassette()
eject_cassette()

cassettes()
cassettes(on_disk = FALSE)

# list the path to cassettes
cassette_path()
vcr_configure(dir = file.path(tempdir(), "foo"))
cassette_path()

vcr_configure_reset()
```

`check_cassette_names` *Check cassette names*

Description

Check cassette names

Usage

```
check_cassette_names(
  pattern = "test-",
  behavior = "stop",
  allowed_duplicates = NULL
)
```

Arguments

pattern	(character) regex pattern for file paths to check. this is done inside of `tests/testthat/`. default: "test-"
behavior	(character) "stop" (default) or "warning". if "warning", we use `immediate.=TRUE` so the warning happens at the top of your tests rather than you seeing it after tests have run (as would happen by default)
allowed_duplicates	(character) cassette names that can be duplicated

Details

Cassette names:

- Should be meaningful so that it is obvious to you what test/function they relate to. Meaningful names are important so that you can quickly determine to what test file or test block a cassette belongs. Note that vcr cannot check that your cassette names are meaningful.
- Should not be duplicated. Duplicated cassette names would lead to a test using the wrong cassette.
- Should not have spaces. Spaces can lead to problems in using file paths.
- Should not include a file extension. vcr handles file extensions for the user.
- Should not have illegal characters that can lead to problems in using file paths: `/`, `?`, `<`, `>`, `\`, `:`, `*`, `|`, and `\"`
- Should not have control characters, e.g., `\n`
- Should not have just dots, e.g., `.` or `..`
- Should not have Windows reserved words, e.g., `com1`
- Should not have trailing dots
- Should not be longer than 255 characters

`vcr::check_cassette_names()` is meant to be run during your tests, from a `helper-*.R` file inside the `tests/testthat` directory. It only checks that cassette names are not duplicated. Note that if you do need to have duplicated cassette names you can do so by using the `allowed_duplicates` parameter in `check_cassette_names()`. A helper function `check_cassette_names()` runs inside `insert_cassette()` that checks that cassettes do not have: spaces, file extensions, unaccepted characters (slashes).

`crul_request` *An HTTP request as prepared by the* **crul** *package*

Description

The object is a list, and is the object that is passed on to **webmockr** and **vcr** instead of routing through **crul** as normal. Used in examples/tests.

Format

A list

eject_cassette *Eject a cassette*

Description

Eject a cassette

Usage

```
eject_cassette(
  cassette = NULL,
  options = list(),
  skip_no_unused_interactions_assertion = NULL
)
```

Arguments

`cassette`	(character) a single cassette names to eject; see `name` parameter definition in `insert_cassette()` for cassette name rules
`options`	(list) a list of options to apply to the eject process
`skip_no_unused_interactions_assertion`	(logical) If `TRUE`, this will skip the "no unused HTTP interactions" assertion enabled by the `allow_unused_http_interactions = FALSE` cassette option. This is intended for use when your test has had an error, but your test framework has already handled it - IGNORED FOR NOW

Value

The ejected cassette if there was one

See Also

`use_cassette()`, `insert_cassette()`

Examples

```
vcr_configure(dir = tempdir())
insert_cassette("hello")
(x <- current_cassette())

# by default does current cassette
x <- eject_cassette()
x
# can also select by cassette name
# eject_cassette(cassette = "hello")
```

`HTTPInteraction` *HTTPInteraction class*

Description

object holds request and response objects

Details

Methods

`to_hash()` Create a hash from the HTTPInteraction object

`from_hash(hash)` Create a HTTPInteraction object from a hash

Public fields

`request` A `Request` class object

`response` A `VcrResponse` class object

`recorded_at` (character) Time http interaction recorded at

Methods

Public methods:

- `HTTPInteraction$new()`
- `HTTPInteraction$to_hash()`
- `HTTPInteraction$from_hash()`
- `HTTPInteraction$clone()`

Method `new()`**:** Create a new `HTTPInteraction` object

Usage:

```
HTTPInteraction$new(request, response, recorded_at)
```

Arguments:

`request` A `Request` class object

`response` A `VcrResponse` class object

`recorded_at` (character) Time http interaction recorded at

Returns: A new `HTTPInteraction` object

Method `to_hash()`**:** Create a hash from the HTTPInteraction object

Usage:

```
HTTPInteraction$to_hash()
```

Returns: a named list

Method `from_hash()`**:** Create a HTTPInteraction object from a hash

Usage:

```
HTTPInteraction$from_hash(hash)
```

Arguments:

`hash` a named list

Returns: a new `HttpInteraction` object

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Not run:
# make the request
library(vcr)
url <- "https://hb.opencpu.org/post"
body <- list(foo = "bar")
cli <- crul::HttpClient$new(url = url)
res <- cli$post(body = body)

# build a Request object
(request <- Request$new("POST", uri = url,
  body = body, headers = res$response_headers))
# build a VcrResponse object
(response <- VcrResponse$new(
   res$status_http(),
   res$response_headers,
   res$parse("UTF-8"),
   res$response_headers$status))

# make HTTPInteraction object
(x <- HTTPInteraction$new(request = request, response = response))
x$recorded_at
x$to_hash()

# make an HTTPInteraction from a hash with the object already made
x$from_hash(x$to_hash())

# Make an HTTPInteraction from a hash alone
my_hash <- x$to_hash()
HTTPInteraction$new()$from_hash(my_hash)

## End(Not run)
```

`HTTPInteractionList` *HTTPInteractionList class*

Description

keeps track of all HTTPInteraction objects

Details

Private Methods

`has_unused_interactions()` Are there any unused interactions? returns boolean

`matching_interaction_index_for()` asdfadf

`matching_used_interaction_for(request)` asdfadfs

`interaction_matches_request(request, interaction)` Check if a request matches an interaction (logical)

`from_hash()` Get a hash back.

`request_summary(z)` Get a request summary (character)

`response_summary(z)` Get a response summary (character)

Public fields

`interactions` (list) list of interaction class objects

`request_matchers` (character) vector of request matchers

`allow_playback_repeats` whether to allow playback repeats

`parent_list` A list for empty objects, see `NullList`

`used_interactions` (list) Interactions that have been used

Methods

Public methods:

- `HTTPInteractionList$new()`
- `HTTPInteractionList$response_for()`
- `HTTPInteractionList$has_interaction_matching()`
- `HTTPInteractionList$has_used_interaction_matching()`
- `HTTPInteractionList$remaining_unused_interaction_count()`
- `HTTPInteractionList$assert_no_unused_interactions()`
- `HTTPInteractionList$clone()`

Method `new()`**:** Create a new `HTTPInteractionList` object

Usage:

```
HTTPInteractionList$new(
  interactions,
  request_matchers,
  allow_playback_repeats = FALSE,
  parent_list = NullList$new(),
  used_interactions = list()
)
```

Arguments:

`interactions` (list) list of interaction class objects

`request_matchers` (character) vector of request matchers

`allow_playback_repeats` whether to allow playback repeats or not

`parent_list` A list for empty objects, see `NullList`

`used_interactions` (list) Interactions that have been used. That is, interactions that are on disk in the current cassette, and a request has been made that matches that interaction

Returns: A new `HTTPInteractionList` object

Method `response_for()`**:** Check if there's a matching interaction, returns a response object

Usage:

`HTTPInteractionList$response_for(request)`

Arguments:

`request` The request from an object of class `HTTPInteraction`

Method `has_interaction_matching()`**:** Check if has a matching interaction

Usage:

`HTTPInteractionList$has_interaction_matching(request)`

Arguments:

`request` The request from an object of class `HTTPInteraction`

Returns: logical

Method `has_used_interaction_matching()`**:** check if has used interactions matching a given request

Usage:

`HTTPInteractionList$has_used_interaction_matching(request)`

Arguments:

`request` The request from an object of class `HTTPInteraction`

Returns: logical

Method `remaining_unused_interaction_count()`**:** Number of unused interactions

Usage:

`HTTPInteractionList$remaining_unused_interaction_count()`

Returns: integer

Method `assert_no_unused_interactions()`**:** Checks if there are no unused interactions left.

Usage:

```
HTTPInteractionList$assert_no_unused_interactions()
```

Returns: various

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Not run:
vcr_configure(
 dir = tempdir(),
 record = "once"
)

# make interactions
## make the request
### turn off mocking
crul::mock(FALSE)
url <- "https://hb.opencpu.org/post"
cli <- crul::HttpClient$new(url = url)
res <- cli$post(body = list(a = 5))

## request
(request <- Request$new("POST", url, list(a = 5), res$headers))
## response
(response <- VcrResponse$new(
   res$status_http(),
   res$response_headers,
   res$parse("UTF-8"),
   res$response_headers$status))
## make an interaction
(inter <- HTTPInteraction$new(request = request, response = response))

# make an interactionlist
(x <- HTTPInteractionList$new(
   interactions = list(inter),
   request_matchers = vcr_configuration()$match_requests_on
))
x$interactions
x$request_matchers
x$parent_list
x$parent_list$response_for()
x$parent_list$has_interaction_matching()
x$parent_list$has_used_interaction_matching()
x$parent_list$remaining_unused_interaction_count()
x$used_interactions
x$allow_playback_repeats
```

```
x$interactions
x$response_for(request)

## End(Not run)
```

`http_interactions` *Get the http interactions of the current cassette*

Description

Get the http interactions of the current cassette

Usage

```
http_interactions()
```

Value

object of class `HTTPInteractionList` if there is a current cassette in use, or `NullList` if no cassette in use

Examples

```
## Not run:
vcr_configure(dir = tempdir())
insert_cassette("foo_bar")
webmockr::webmockr_allow_net_connect()
library(crul)
cli <- crul::HttpClient$new("https://hb.opencpu.org/get")
one <- cli$get(query = list(a = 5))
z <- http_interactions()
z
z$interactions
z$used_interactions
# on eject, request written to the cassette
eject_cassette("foo_bar")

# insert cassette again
insert_cassette("foo_bar")
# now interactions will be present
z <- http_interactions()
z$interactions
z$used_interactions
invisible(cli$get(query = list(a = 5)))
z$used_interactions

# cleanup
unlink(file.path(tempdir(), "foo_bar.yml"))

## End(Not run)
```

insert_cassette *Insert a cassette to record HTTP requests*

Description

Insert a cassette to record HTTP requests

Usage

```
insert_cassette(
  name,
  record = NULL,
  match_requests_on = NULL,
  update_content_length_header = FALSE,
  allow_playback_repeats = FALSE,
  serialize_with = NULL,
  persist_with = NULL,
  preserve_exact_body_bytes = NULL,
  re_record_interval = NULL,
  clean_outdated_http_interactions = NULL
)
```

Arguments

`name`	The name of the cassette. vcr will check this to ensure it is a valid file name. Not allowed: spaces, file extensions, control characters (e.g., `\n`), illegal characters ('/', '?', '<', '>', '\', ':', '*', '|', and '\"'), dots alone (e.g., '.', '..'), Windows reserved words (e.g., 'com1'), trailing dots (can cause problems on Windows), names longer than 255 characters. See section "Cassette names"
`record`	The record mode (default: `"once"`). See recording for a complete list of the different recording modes.
`match_requests_on`	List of request matchers to use to determine what recorded HTTP interaction to replay. Defaults to `["method", "uri"]`. The built-in matchers are "method", "uri", "host", "path", "headers", "body" and "query"
`update_content_length_header`	(logical) Whether or not to overwrite the `Content-Length` header of the responses to match the length of the response body. Default: `FALSE`
`allow_playback_repeats`	(logical) Whether or not to allow a single HTTP interaction to be played back multiple times. Default: `FALSE`.
`serialize_with`	(character) Which serializer to use. Valid values are "yaml" (default) and "json". Note that you can have multiple cassettes with the same name as long as they use different serializers; so if you only want one cassette for a given cassette name, make sure to not switch serializers, or clean up files you no longer need.

`persist_with`	(character) Which cassette persister to use. Default: "file_system". You can also register and use a custom persister.
`preserve_exact_body_bytes`	(logical) Whether or not to base64 encode the bytes of the requests and responses for this cassette when serializing it. See also `preserve_exact_body_bytes` in `vcr_configure()`. Default: `FALSE`
`re_record_interval`	(integer) How frequently (in seconds) the cassette should be re-recorded. default: `NULL` (not re-recorded)
`clean_outdated_http_interactions`	(logical) Should outdated interactions be recorded back to file? default: `FALSE`

Details

Cassette names:

- Should be meaningful so that it is obvious to you what test/function they relate to. Meaningful names are important so that you can quickly determine to what test file or test block a cassette belongs. Note that vcr cannot check that your cassette names are meaningful.
- Should not be duplicated. Duplicated cassette names would lead to a test using the wrong cassette.
- Should not have spaces. Spaces can lead to problems in using file paths.
- Should not include a file extension. vcr handles file extensions for the user.
- Should not have illegal characters that can lead to problems in using file paths: `/`, ?, <, >, `\`, `:`, `*`, `|`, and `\"`
- Should not have control characters, e.g., `\n`
- Should not have just dots, e.g., `.` or `..`
- Should not have Windows reserved words, e.g., `com1`
- Should not have trailing dots
- Should not be longer than 255 characters

`vcr::check_cassette_names()` is meant to be run during your tests, from a `helper-*.R` file inside the `tests/testthat` directory. It only checks that cassette names are not duplicated. Note that if you do need to have duplicated cassette names you can do so by using the `allowed_duplicates` parameter in `check_cassette_names()`. A helper function `check_cassette_names()` runs inside `insert_cassette()` that checks that cassettes do not have: spaces, file extensions, unaccepted characters (slashes).

Value

an object of class `Cassette`

Cassette options

Default values for arguments controlling cassette behavior are inherited from vcr's global configuration. See `vcr_configure()` for a complete list of options and their default settings. You can override these options for a specific cassette by changing an argument's value to something other than `NULL` when calling either `insert_cassette()` or `use_cassette()`.

See Also

use_cassette(), eject_cassette()

Examples

```
## Not run:
library(vcr)
library(crul)
vcr_configure(dir = tempdir())
webmockr::webmockr_allow_net_connect()

(x <- insert_cassette(name = "leo5"))
current_cassette()
x$new_recorded_interactions
x$previously_recorded_interactions()
cli <- crul::HttpClient$new(url = "https://hb.opencpu.org")
cli$get("get")
x$new_recorded_interactions # 1 interaction
x$previously_recorded_interactions() # empty
webmockr::stub_registry() # not empty
# very important when using inject_cassette: eject when done
x$eject() # same as eject_cassette("leo5")
x$new_recorded_interactions # same, 1 interaction
x$previously_recorded_interactions() # now not empty
## stub_registry now empty, eject() calls webmockr::disable(), which
## calls the disable method for each of crul and httr adadapters,
## which calls webmockr's remove_stubs() method for each adapter
webmockr::stub_registry()

# cleanup
unlink(file.path(tempdir(), "leo5.yml"))

## End(Not run)
```

---

lightswitch   *Turn vcr on and off, check on/off status, and turn off for a given http call*

Description

Turn vcr on and off, check on/off status, and turn off for a given http call

Usage

```
turned_off(..., ignore_cassettes = FALSE)

turn_on()

turned_on()
```

```
turn_off(ignore_cassettes = FALSE)
```

Arguments

`...`	Any block of code to run, presumably an http request
`ignore_cassettes`	(logical) Controls what happens when a cassette is inserted while vcr is turned off. If `TRUE` is passed, the cassette insertion will be ignored; otherwise an error will be raised. Default: `FALSE`

Details

Sometimes you may need to turn off `vcr`, either for individual function calls, individual test blocks, whole test files, or for the entire package. The following attempts to break down all the options.

`vcr` has the following four exported functions:

- `turned_off()` - Turns vcr off for the duration of a code block
- `turn_off()` - Turns vcr off completely, so that it no longer handles every HTTP request
- `turn_on()` - turns vcr on; the opposite of `turn_off()`
- `turned_on()` - Asks if vcr is turned on, returns a boolean

Instead of using the above four functions, you could use environment variables to achieve the same thing. This way you could enable/disable `vcr` in non-interactive environments such as continuous integration, Docker containers, or running R non-interactively from the command line. The full set of environment variables `vcr` uses, all of which accept only `TRUE` or `FALSE`:

- `VCR_TURN_OFF`: turn off vcr altogether; set to `TRUE` to skip any vcr usage; default: `FALSE`
- `VCR_TURNED_OFF`: set the `turned_off` internal package setting; this does not turn off vcr completely as does `VCR_TURN_OFF` does, but rather is looked at together with `VCR_IGNORE_CASSETTES`
- `VCR_IGNORE_CASSETTES`: set the `ignore_cassettes` internal package setting; this is looked at together with `VCR_TURNED_OFF`

turned_off:

`turned_off()` lets you temporarily make a real HTTP request without completely turning `vcr` off, unloading it, etc.

What happens internally is we turn off `vcr`, run your code block, then on exit turn `vcr` back on - such that `vcr` is only turned off for the duration of your code block. Even if your code block errors, `vcr` will be turned back on due to use of `on.exit(turn_on())`

```
library(vcr)
library(crul)
turned_off({
  con <- HttpClient$new(url = "https://httpbin.org/get")
  con$get()
})
```

```
#> <crul response>
#>   url: https://httpbin.org/get
#>   request_headers:
#>     User-Agent: libcurl/7.54.0 r-curl/4.3 crul/0.9.0
#>     Accept-Encoding: gzip, deflate
#>     Accept: application/json, text/xml, application/xml, */*
#>   response_headers:
#>     status: HTTP/1.1 200 OK
#>     date: Fri, 14 Feb 2020 19:44:46 GMT
#>     content-type: application/json
#>     content-length: 365
#>     connection: keep-alive
#>     server: gunicorn/19.9.0
#>     access-control-allow-origin: *
#>     access-control-allow-credentials: true
#>   status: 200
```

turn_off/turn_on:

`turn_off()` is different from `turned_off()` in that `turn_off()` is not aimed at a single call block, but rather it turns `vcr` off for the entire package. `turn_off()` does check first before turning `vcr` off that there is not currently a cassette in use. `turn_off()` is meant to make R ignore `vcr::insert_cassette()` and `vcr::use_cassette()` blocks in your test suite - letting the code in the block run as if they were not wrapped in `vcr` code - so that all you have to do to run your tests with cached requests/responses AND with real HTTP requests is toggle a single R function or environment variable.

```
library(vcr)
vcr_configure(dir = tempdir())
# real HTTP request works - vcr is not engaged here
crul::HttpClient$new(url = "https://eu.httpbin.org/get")$get()
# wrap HTTP request in use_cassette() - vcr is engaged here
use_cassette("foo_bar", {
  crul::HttpClient$new(url = "https://eu.httpbin.org/get")$get()
})
# turn off & ignore cassettes - use_cassette is ignored, real HTTP request made
turn_off(ignore_cassettes = TRUE)
use_cassette("foo_bar", {
  crul::HttpClient$new(url = "https://eu.httpbin.org/get")$get()
})
# if you turn off and don't ignore cassettes, error thrown
turn_off(ignore_cassettes = FALSE)
use_cassette("foo_bar", {
  res2=crul::HttpClient$new(url = "https://eu.httpbin.org/get")$get()
})
# vcr back on - now use_cassette behaves as before
turn_on()
use_cassette("foo_bar3", {
  res2=crul::HttpClient$new(url = "https://eu.httpbin.org/get")$get()
})
```

turned_on:

`turned_on()` does what it says on the tin - it tells you if `vcr` is turned on or not.

```
library(vcr)
turn_on()
turned_on()
```

```
## [1] TRUE
```

```
turn_off()
```

```
## vcr turned off; see ?turn_on to turn vcr back on
```

```
turned_on()
```

```
## [1] FALSE
```

Environment variables:

The `VCR_TURN_OFF` environment variable can be used within R or on the command line to turn off `vcr`. For example, you can run tests for a package that uses `vcr`, but ignore any `use_cassette`/`insert_cassette` usage, by running this on the command line in the root of your package:

```
VCR_TURN_OFF=true Rscript -e "devtools::test()"
```

Or, similarly within R:

```
Sys.setenv(VCR_TURN_OFF = TRUE)
devtools::test()
```

The `VCR_TURNED_OFF` and `VCR_IGNORE_CASSETTES` environment variables can be used in combination to achieve the same thing as `VCR_TURN_OFF`:

```
VCR_TURNED_OFF=true VCR_IGNORE_CASSETTES=true Rscript -e "devtools::test()"
```

Examples

```
## Not run:
vcr_configure(dir = tempdir())

turn_on()
turned_on()
turn_off()

# turn off for duration of a block
library(crul)
turned_off({
 res <- HttpClient$new(url = "https://hb.opencpu.org/get")$get()
})
res

# turn completely off
turn_off()
library(webmockr)
crul::mock()
```

```
# HttpClient$new(url = "https://hb.opencpu.org/get")$get(verbose = TRUE)
turn_on()

## End(Not run)
```

---

real_http_connections_allowed *Are real http connections allowed?*

Description

Are real http connections allowed?

Usage

```
real_http_connections_allowed()
```

Value

boolean, `TRUE` if real HTTP requests allowed; `FALSE` if not

Examples

```
real_http_connections_allowed()
```

---

recording *vcr recording options*

Description

vcr recording options

Details

Record modes dictate under what circumstances http requests/responses are recorded to cassettes (disk). Set the recording mode with the parameter `record` in the `use_cassette()` and `insert_cassette()` functions.

once:

The once record mode will:

- Replay previously recorded interactions.
- Record new interactions if there is no cassette file.
- Cause an error to be raised for new requests if there is a cassette file.

It is similar to the `new_episodes` record mode, but will prevent new, unexpected requests from being made (i.e. because the request URI changed or whatever).

`once` is the default record mode, used when you do not set one.

none:

The `none` record mode will:

- Replay previously recorded interactions.
- Cause an error to be raised for any new requests.

This is useful when your code makes potentially dangerous HTTP requests. The `none` record mode guarantees that no new HTTP requests will be made.

new_episodes:

The `new_episodes` record mode will:

- Record new interactions.
- Replay previously recorded interactions.

It is similar to the `once` record mode, but will **always** record new interactions, even if you have an existing recorded one that is similar (but not identical, based on the `match_request_on` option).

all:

The `all` record mode will:

- Record new interactions.
- Never replay previously recorded interactions.

This can be temporarily used to force vcr to re-record a cassette (i.e. to ensure the responses are not out of date) or can be used when you simply want to log all HTTP requests.

`request-matching` *vcr request matching*

Description

There are a number of options, some of which are on by default, some of which can be used together, and some alone.

Details

To match previously recorded requests, `vcr` has to try to match new HTTP requests to a previously recorded one. By default, we match on HTTP method (e.g., `GET`) and URI (e.g., `http://foo.com`), following Ruby's VCR gem.

You can customize how we match requests with one or more of the following options, some of which are on by default, some of which can be used together, and some alone.

- `method`: Use the **method** request matcher to match requests on the HTTP method (i.e. GET, POST, PUT, DELETE, etc). You will generally want to use this matcher. The **method** matcher is used (along with the **uri** matcher) by default if you do not specify how requests should match.
- `uri`: Use the **uri** request matcher to match requests on the request URI. The **uri** matcher is used (along with the **method** matcher) by default if you do not specify how requests should match.

- `host`: Use the **host** request matcher to match requests on the request host. You can use this (alone, or in combination with **path**) as an alternative to **uri** so that non-deterministic portions of the URI are not considered as part of the request matching.
- `path`: Use the **path** request matcher to match requests on the path portion of the request URI. You can use this (alone, or in combination with **host**) as an alternative to **uri** so that non-deterministic portions of the URI
- `query`: Use the **query** request matcher to match requests on the query string portion of the request URI. You can use this (alone, or in combination with others) as an alternative to **uri** so that non-deterministic portions of the URI are not considered as part of the request matching.
- `body`: Use the **body** request matcher to match requests on the request body.
- `headers`: Use the **headers** request matcher to match requests on the request headers.

You can set your own options by tweaking the `match_requests_on` parameter in `use_cassette()`:

```
library(vcr)


use_cassette(name = "foo_bar", {
    cli$post("post", body = list(a = 5))
  },
  match_requests_on = c('method', 'headers', 'body')
)
```

Matching:

headers:

```
library(crul)
library(vcr)
cli <- crul::HttpClient$new("https://httpbin.org/get",
  headers = list(foo = "bar"))
use_cassette(name = "nothing_new", {
    one <- cli$get()
  },
  match_requests_on = 'headers'
)
cli$headers$foo <- "stuff"
use_cassette(name = "nothing_new", {
    two <- cli$get()
  },
  match_requests_on = 'headers'
)
one$request_headers
two$request_headers
```

`RequestHandler` *RequestHandler*

Description

Base handler for http requests, deciding whether a request is stubbed, to be ignored, recordable, or unhandled

Details

Private Methods

`request_type(request)` Get the request type

`externally_stubbed()` just returns FALSE

`should_ignore()` should we ignore the request, depends on request ignorer infrastructure that's not working yet

`has_response_stub()` Check if there is a matching response stub in the http interaction list

`get_stubbed_response()` Check for a response and get it

`request_summary(request)` get a request summary

`on_externally_stubbed_request(request)` on externally stubbed request do nothing

`on_ignored_request(request)` on ignored request, do something

`on_recordable_request(request)` on recordable request, record the request

`on_unhandled_request(request)` on unhandled request, run UnhandledHTTPRequestError

Public fields

`request_original` original, before any modification

`request` the request, after any modification

`vcr_response` holds VcrResponse object

`stubbed_response` the stubbed response

`cassette` the cassette holder

Methods

Public methods:

- `RequestHandler$new()`
- `RequestHandler$handle()`
- `RequestHandler$clone()`

Method `new()`**:** Create a new `RequestHandler` object

Usage:

```
RequestHandler$new(request)
```

Arguments:

`request` The request from an object of class `HttpInteraction`

Returns: A new `RequestHandler` object

Method `handle()`**:** Handle the request (`request` given in `$initialize()`)

Usage:

`RequestHandler$handle()`

Returns: handles a request, outcomes vary

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

Usage:

`RequestHandler$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Not run:
# record mode: once
vcr_configure(
 dir = tempdir(),
 record = "once"
)

data(crul_request)
crul_request$url$handle <- curl::new_handle()
crul_request
x <- RequestHandler$new(crul_request)
# x$handle()

# record mode: none
vcr_configure(
 dir = tempdir(),
 record = "none"
)
data(crul_request)
crul_request$url$handle <- curl::new_handle()
crul_request
insert_cassette("testing_record_mode_none", record = "none")
#file.path(vcr_c$dir, "testing_record_mode_none.yml")
x <- RequestHandlerCrul$new(crul_request)
# x$handle()
crul_request$url$url <- "https://api.crossref.org/works/10.1039/c8sm90002g/"
crul_request$url$handle <- curl::new_handle()
z <- RequestHandlerCrul$new(crul_request)
# z$handle()
eject_cassette("testing_record_mode_none")

## End(Not run)
```

`RequestHandlerCrul` *RequestHandlerCrul*

Description

Methods for the crul package, building on RequestHandler

Super class

`vcr::RequestHandler` -> `RequestHandlerCrul`

Methods

Public methods:

- `RequestHandlerCrul$clone()`

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

Usage:

`RequestHandlerCrul$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Not run:
vcr_configure(
 dir = tempdir(),
 record = "once"
)

data(crul_request)
crul_request$url$handle <- curl::new_handle()
crul_request
x <- RequestHandlerCrul$new(crul_request)
# x$handle()

# body matching
library(vcr)
library(crul)
vcr_configure(dir = tempdir(), log = TRUE,
 log_opts = list(file = file.path(tempdir(), "vcr.log")))
cli <- HttpClient$new(url = "https://hb.opencpu.org")

## testing, same uri and method, changed body in 2nd block
use_cassette(name = "apple7", {
  resp <- cli$post("post", body = list(foo = "bar"))
}, match_requests_on = c("method", "uri", "body"))
## should error, b/c record="once"
```

```
if (interactive()) {
  use_cassette(name = "apple7", {
    resp <- cli$post("post", body = list(foo = "bar"))
    resp2 <- cli$post("post", body = list(hello = "world"))
  }, match_requests_on = c("method", "uri", "body"))
}
cas <- insert_cassette(name = "apple7",
  match_requests_on = c("method", "uri", "body"))
resp2 <- cli$post("post", body = list(foo = "bar"))
eject_cassette("apple7")

## testing, same body, changed method in 2nd block
if (interactive()) {
use_cassette(name = "apple8", {
  x <- cli$post("post", body = list(hello = "world"))
}, match_requests_on = c("method", "body"))
use_cassette(name = "apple8", {
  x <- cli$get("post", body = list(hello = "world"))
}, match_requests_on = c("method", "body"))
}

## testing, same body, changed uri in 2nd block
# use_cassette(name = "apple9", {
#   x <- cli$post("post", body = list(hello = "world"))
#   w <- cli$post("get", body = list(hello = "world"))
# }, match_requests_on = c("method", "body"))
# use_cassette(name = "apple9", {
#   NOTHING HERE
# }, match_requests_on = c("method", "body"))
# unlink(file.path(vcr_configuration()$dir, "apple9.yml"))

## End(Not run)
```

`RequestHandlerHttr` *RequestHandlerHttr*

Description

Methods for the httr package, building on RequestHandler

Super class

`vcr::RequestHandler` -> `RequestHandlerHttr`

Methods

Public methods:

- `RequestHandlerHttr$new()`
- `RequestHandlerHttr$clone()`

Method `new()`**:** Create a new `RequestHandlerHttr` object

Usage:

```
RequestHandlerHttr$new(request)
```

Arguments:

`request` The request from an object of class `HttpInteraction`

Returns: A new `RequestHandlerHttr` object

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Not run:
vcr_configure(
 dir = tempdir(),
 record = "once"
)

# GET request
library(httr)
load("~/httr_req.rda")
req
x <- RequestHandlerHttr$new(req)
# x$handle()

# POST request
library(httr)
webmockr::httr_mock()
mydir <- file.path(tempdir(), "testing_httr")
invisible(vcr_configure(dir = mydir))
use_cassette(name = "testing2", {
  res <- POST("https://hb.opencpu.org/post", body = list(foo = "bar"))
}, match_requests_on = c("method", "uri", "body"))

load("~/httr_req_post.rda")
insert_cassette("testing3")
httr_req_post
x <- RequestHandlerHttr$new(httr_req_post)
x
# x$handle()
self=x


## End(Not run)
```

`RequestHandlerHttr2` *RequestHandlerHttr2*

Description

Methods for the httr2 package, building on RequestHandler

Super class

`vcr::RequestHandler` -> `RequestHandlerHttr2`

Methods

Public methods:

- `RequestHandlerHttr2$new()`
- `RequestHandlerHttr2$clone()`

Method `new()`**:** Create a new `RequestHandlerHttr2` object

Usage:

`RequestHandlerHttr2$new(request)`

Arguments:

`request` The request from an object of class `HttpInteraction`

Returns: A new `RequestHandlerHttr2` object

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

Usage:

`RequestHandlerHttr2$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Not run:
# GET request
library(httr2)
req <- request("https://hb.opencpu.org/post") %>%
   req_body_json(list(foo = "bar"))
x <- RequestHandlerHttr2$new(req)
# x$handle()

# POST request
library(httr2)
mydir <- file.path(tempdir(), "testing_httr2")
invisible(vcr_configure(dir = mydir))
req <- request("https://hb.opencpu.org/post") %>%
  req_body_json(list(foo = "bar"))
```

```
use_cassette(name = "testing3", {
  response <- req_perform(req)
}, match_requests_on = c("method", "uri", "body"))
use_cassette(name = "testing3", {
  response2 <- req_perform(req)
}, match_requests_on = c("method", "uri", "body"))

## End(Not run)
```

`RequestMatcherRegistry` *RequestMatcherRegistry*

Description

handles request matchers

Public fields

`registry` initialze registry list with a request, or leave empty

`default_matchers` request matchers to use. default: method, uri

Methods

Public methods:

- `RequestMatcherRegistry$new()`
- `RequestMatcherRegistry$register()`
- `RequestMatcherRegistry$register_built_ins()`
- `RequestMatcherRegistry$try_to_register_body_as_json()`
- `RequestMatcherRegistry$clone()`

Method `new()`**:** Create a new RequestMatcherRegistry object

Usage:

```
RequestMatcherRegistry$new(
  registry = list(),
  default_matchers = list("method", "uri")
)
```

Arguments:

`registry` initialze registry list with a request, or leave empty

`default_matchers` request matchers to use. default: method, uri

Returns: A new `RequestMatcherRegistry` object

Method `register()`**:** Register a custom matcher

Usage:

```
RequestMatcherRegistry$register(name, func)
```

Arguments:

`name` matcher name

`func` function that describes a matcher, should return a single boolean

Returns: no return; registers the matcher

Method `register_built_ins()`**:** Register all built in matchers

Usage:

```
RequestMatcherRegistry$register_built_ins()
```

Returns: no return; registers all built in matchers

Method `try_to_register_body_as_json()`**:** Try to register body as JSON

Usage:

```
RequestMatcherRegistry$try_to_register_body_as_json(r1, r2)
```

Arguments:

`r1, r2` Request class objects

Returns: no return; registers the matcher

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Note

r1=from new request; r2=from recorded interaction

Examples

```
## Not run:
(x <- RequestMatcherRegistry$new())
x$default_matchers
x$registry

## End(Not run)
```

`skip_if_vcr_off` *Skip tests if vcr is off*

Description

Custom testthat skipper to skip tests if vcr is turned off via the environment variable `VCR_TURN_OFF`.

Usage

```
skip_if_vcr_off()
```

Details

This might be useful if your test will fail with real requests: when the cassette was e.g. edited (a real request produced a 200 status code but you made it a 502 status code for testing the behavior of your code when the API errors) or if the tests are very specific (e.g. testing a date was correctly parsed, but making a real request would produce a different date).

Value

Nothing, skip test.

See Also

`turn_off()`

`str_splitter` *split string every N characters*

Description

split string every N characters

Usage

```
str_splitter(str, length)
```

Arguments

`str`	(character) a string
`length`	(integer) number of characters to split by

Examples

```
## Not run:
str = "XOVEWVJIEWNIGOIWENVOIWEWVWEW"
str_splitter(str, 5)
str_splitter(str, 5L)

## End(Not run)
```

UnhandledHTTPRequestError

UnhandledHTTPRequestError

Description

Handle http request errors

Usage

```
vcr_last_error()
```

Details

How this error class is used: If `record="once"` we trigger this.

Users can use vcr in the context of both `use_cassette()` and `insert_cassette()`

For the former, all requests go through the call_block But for the latter, requests go through webmockr.

Where is one place where we can put UnhandledHTTPRequestError that will handle both use_cassette and insert_cassette?

Error situations where this is invoked

- `record=once` AND there's a new request that doesn't match the one in the cassette on disk
  - in webmockr: if no stub found and there are recorded interactions on the cassette, and record = once, then error with UnhandledHTTPRequestError
    - but if record != once, then allow it, unless record == none
- others?

Public fields

`request` a Request object

`cassette` a cassette name

Methods

Public methods:

- `UnhandledHTTPRequestError$new()`
- `UnhandledHTTPRequestError$run()`
- `UnhandledHTTPRequestError$construct_message()`
- `UnhandledHTTPRequestError$request_description()`
- `UnhandledHTTPRequestError$current_matchers()`
- `UnhandledHTTPRequestError$match_request_on_headers()`
- `UnhandledHTTPRequestError$match_request_on_body()`
- `UnhandledHTTPRequestError$formatted_headers()`
- `UnhandledHTTPRequestError$cassettes_description()`
- `UnhandledHTTPRequestError$cassettes_list()`
- `UnhandledHTTPRequestError$get_help()`
- `UnhandledHTTPRequestError$formatted_suggestions()`
- `UnhandledHTTPRequestError$format_bullet_point()`
- `UnhandledHTTPRequestError$format_foot_note()`
- `UnhandledHTTPRequestError$suggestion_for()`
- `UnhandledHTTPRequestError$suggestions()`
- `UnhandledHTTPRequestError$no_cassette_suggestions()`
- `UnhandledHTTPRequestError$record_mode_suggestion()`
- `UnhandledHTTPRequestError$has_used_interaction_matching()`
- `UnhandledHTTPRequestError$match_requests_on_suggestion()`
- `UnhandledHTTPRequestError$clone()`

Method `new()`**:** Create a new `UnhandledHTTPRequestError` object

Usage:

```
UnhandledHTTPRequestError$new(request)
```

Arguments:

`request` (Request) a Request object

Returns: A new `UnhandledHTTPRequestError` object

Method `run()`**:** Run unhandled request handling

Usage:

```
UnhandledHTTPRequestError$run()
```

Returns: various

Method `construct_message()`**:** Construct and execute stop message for why request failed

Usage:

```
UnhandledHTTPRequestError$construct_message()
```

Returns: a stop message

Method `request_description()`**:** construct request description

Usage:
```
UnhandledHTTPRequestError$request_description()
```
Returns: character

Method `current_matchers()`**:** get current request matchers

Usage:
```
UnhandledHTTPRequestError$current_matchers()
```
Returns: character

Method `match_request_on_headers()`**:** are headers included in current matchers?

Usage:
```
UnhandledHTTPRequestError$match_request_on_headers()
```
Returns: logical

Method `match_request_on_body()`**:** is body includled in current matchers?

Usage:
```
UnhandledHTTPRequestError$match_request_on_body()
```
Returns: logical

Method `formatted_headers()`**:** get request headers

Usage:
```
UnhandledHTTPRequestError$formatted_headers()
```
Returns: character

Method `cassettes_description()`**:** construct description of current or lack thereof cassettes

Usage:
```
UnhandledHTTPRequestError$cassettes_description()
```
Returns: character

Method `cassettes_list()`**:** cassette details

Usage:
```
UnhandledHTTPRequestError$cassettes_list()
```
Returns: character

Method `get_help()`**:** get help message for non-verbose error

Usage:
```
UnhandledHTTPRequestError$get_help()
```
Returns: character

Method `formatted_suggestions()`**:** make suggestions for what to do

Usage:
```
UnhandledHTTPRequestError$formatted_suggestions()
```
Returns: character

Method `format_bullet_point()`**:** add bullet point to beginning of a line

Usage:

```
UnhandledHTTPRequestError$format_bullet_point(lines, index)
```

Arguments:

`lines` (character) vector of strings

`index` (integer) a number

Returns: character

Method `format_foot_note()`**:** make a foot note

Usage:

```
UnhandledHTTPRequestError$format_foot_note(url, index)
```

Arguments:

`url` (character) a url

`index` (integer) a number

Returns: character

Method `suggestion_for()`**:** get a suggestion by key

Usage:

```
UnhandledHTTPRequestError$suggestion_for(key)
```

Arguments:

`key` (character) a character string

Returns: character

Method `suggestions()`**:** get all suggestions

Usage:

```
UnhandledHTTPRequestError$suggestions()
```

Returns: list

Method `no_cassette_suggestions()`**:** get all no cassette suggestions

Usage:

```
UnhandledHTTPRequestError$no_cassette_suggestions()
```

Returns: list

Method `record_mode_suggestion()`**:** get the appropriate record mode suggestion

Usage:

```
UnhandledHTTPRequestError$record_mode_suggestion()
```

Returns: character

Method `has_used_interaction_matching()`**:** are there any used interactions

Usage:

```
UnhandledHTTPRequestError$has_used_interaction_matching()
```

Returns: logical

Method `match_requests_on_suggestion()`**:** match requests on suggestion

Usage:

`UnhandledHTTPRequestError$match_requests_on_suggestion()`

Returns: list

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

Usage:

`UnhandledHTTPRequestError$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Not run:
vcr_configure(dir = tempdir())
cassettes()
insert_cassette("turtle")
request <- Request$new("post", 'https://hb.opencpu.org/post?a=5',
  "", list(foo = "bar"))

err <- UnhandledHTTPRequestError$new(request)
err$request_description()
err$current_matchers()
err$match_request_on_headers()
err$match_request_on_body()
err$formatted_headers()
cat(err$formatted_headers(), "\n")
cat(err$cassettes_description(), "\n")
cat(err$cassettes_list(), "\n")
err$formatted_suggestions()
cat(err$format_bullet_point('foo bar', 1), "\n")
err$suggestion_for("use_new_episodes")
err$suggestions()
err$no_cassette_suggestions()
err$record_mode_suggestion()
err$has_used_interaction_matching()
err$match_requests_on_suggestion()

# err$construct_message()

# cleanup
eject_cassette("turtle")
unlink(tempdir())

## End(Not run)
## Not run:
# vcr_last_error()

## End(Not run)
```

`use_cassette` *Use a cassette to record HTTP requests*

Description

Use a cassette to record HTTP requests

Usage

```
use_cassette(
  name,
  ...,
  record = NULL,
  match_requests_on = NULL,
  update_content_length_header = FALSE,
  allow_playback_repeats = FALSE,
  serialize_with = NULL,
  persist_with = NULL,
  preserve_exact_body_bytes = NULL,
  re_record_interval = NULL,
  clean_outdated_http_interactions = NULL
)
```

Arguments

`name`	The name of the cassette. vcr will check this to ensure it is a valid file name. Not allowed: spaces, file extensions, control characters (e.g., `\n`), illegal characters ('/', '?', '<', '>', '\', ':', '*', '|', and '\"'), dots alone (e.g., '.', '..'), Windows reserved words (e.g., 'com1'), trailing dots (can cause problems on Windows), names longer than 255 characters. See section "Cassette names"
`...`	a block of code containing one or more requests (required). Use curly braces to encapsulate multi-line code blocks. If you can't pass a code block use `insert_cassette()` instead.
`record`	The record mode (default: `"once"`). See recording for a complete list of the different recording modes.
`match_requests_on`	List of request matchers to use to determine what recorded HTTP interaction to replay. Defaults to `["method", "uri"]`. The built-in matchers are "method", "uri", "host", "path", "headers", "body" and "query"
`update_content_length_header`	(logical) Whether or not to overwrite the `Content-Length` header of the responses to match the length of the response body. Default: `FALSE`
`allow_playback_repeats`	(logical) Whether or not to allow a single HTTP interaction to be played back multiple times. Default: `FALSE`.

`serialize_with`	(character) Which serializer to use. Valid values are "yaml" (default) and "json". Note that you can have multiple cassettes with the same name as long as they use different serializers; so if you only want one cassette for a given cassette name, make sure to not switch serializers, or clean up files you no longer need.
`persist_with`	(character) Which cassette persister to use. Default: "file_system". You can also register and use a custom persister.
`preserve_exact_body_bytes`	(logical) Whether or not to base64 encode the bytes of the requests and responses for this cassette when serializing it. See also `preserve_exact_body_bytes` in `vcr_configure()`. Default: `FALSE`
`re_record_interval`	(integer) How frequently (in seconds) the cassette should be re-recorded. default: `NULL` (not re-recorded)
`clean_outdated_http_interactions`	(logical) Should outdated interactions be recorded back to file? default: `FALSE`

Details

A run down of the family of top level **vcr** functions

- `use_cassette` Initializes a cassette. Returns the inserted cassette.
- `insert_cassette` Internally used within `use_cassette`
- `eject_cassette` ejects the current cassette. The cassette will no longer be used. In addition, any newly recorded HTTP interactions will be written to disk.

Value

an object of class `Cassette`

Cassette options

Default values for arguments controlling cassette behavior are inherited from vcr's global configuration. See `vcr_configure()` for a complete list of options and their default settings. You can override these options for a specific cassette by changing an argument's value to something other than `NULL` when calling either `insert_cassette()` or `use_cassette()`.

Behavior

This function handles a few different scenarios:

- when everything runs smoothly, and we return a `Cassette` class object so you can inspect the cassette, and the cassette is ejected
- when there is an invalid parameter input on cassette creation, we fail with a useful message, we don't return a cassette, and the cassette is ejected
- when there is an error in calling your passed in code block, we return with a useful message, and since we use `on.exit()` the cassette is still ejected even though there was an error, but you don't get an object back

- whenever an empty cassette (a yml/json file) is found, we delete it before returning from the `use_cassette()` function call. we achieve this via use of `on.exit()` so an empty cassette is deleted even if there was an error in the code block you passed in

Cassettes on disk

Note that *"eject"* only means that the R session cassette is no longer in use. If any interactions were recorded to disk, then there is a file on disk with those interactions.

Using with tests (specifically testthat)

There's a few ways to get correct line numbers for failed tests and one way to not get correct line numbers:

Correct: Either wrap your `test_that()` block inside your `use_cassette()` block, OR if you put your `use_cassette()` block inside your `test_that()` block put your `testthat` expectations outside of the `use_cassette()` block.

Incorrect: By wrapping the `use_cassette()` block inside your `test_that()` block with your **testthat** expectations inside the `use_cassette()` block, you'll only get the line number that the `use_cassette()` block starts on.

See Also

`insert_cassette()`, `eject_cassette()`

Examples

```
## Not run:
library(vcr)
library(crul)
vcr_configure(dir = tempdir())

use_cassette(name = "apple7", {
  cli <- HttpClient$new(url = "https://hb.opencpu.org")
  resp <- cli$get("get")
})
readLines(file.path(tempdir(), "apple7.yml"))

# preserve exact body bytes - records in base64 encoding
use_cassette("things4", {
  cli <- crul::HttpClient$new(url = "https://hb.opencpu.org")
  bbb <- cli$get("get")
}, preserve_exact_body_bytes = TRUE)
## see the body string value in the output here
readLines(file.path(tempdir(), "things4.yml"))

# cleanup
unlink(file.path(tempdir(), c("things4.yml", "apple7.yml")))


# with httr
library(vcr)
library(httr)
vcr_configure(dir = tempdir(), log = TRUE, log_opts = list(file = file.path(tempdir(), "vcr.log")))

use_cassette(name = "stuff350", {
  res <- GET("https://hb.opencpu.org/get")
})
readLines(file.path(tempdir(), "stuff350.yml"))

use_cassette(name = "catfact456", {
  res <- GET("https://catfact.ninja/fact")
})

# record mode: none
library(crul)
vcr_configure(dir = tempdir())

## make a connection first
conn <- crul::HttpClient$new("https://eu.httpbin.org")
## this errors because 'none' disallows any new requests
# use_cassette("none_eg", (res2 <- conn$get("get")), record = "none")
## first use record mode 'once' to record to a cassette
one <- use_cassette("none_eg", (res <- conn$get("get")), record = "once")
one; res
## then use record mode 'none' to see it's behavior
two <- use_cassette("none_eg", (res2 <- conn$get("get")), record = "none")
two; res2

## End(Not run)
```

use_vcr — *Setup vcr for a package*

Description

Setup vcr for a package

Usage

```
use_vcr(dir = ".", verbose = TRUE)
```

Arguments

`dir`	(character) path to package root. default's to current directory
`verbose`	(logical) print progress messages. default: `TRUE`

Details

Sets a mimimum vcr version, which is usually the latest (stable) version on CRAN. You can of course easily remove or change the version requirement yourself after running this function.

Value

only messages about progress, returns invisible()

`vcr_configure` *Global Configuration Options*

Description

Configurable options that define vcr's default behavior.

Usage

```
vcr_configure(...)

vcr_configure_reset()

vcr_configuration()

vcr_config_defaults()
```

Arguments

`...`	configuration settings used to override defaults. See below for a complete list of valid arguments.

Configurable settings

vcr options:

File locations:

- `dir` Cassette directory
- `write_disk_path` (character) path to write files to for any requests that write responses to disk. by default this parameter is `NULL`. For testing a package, you'll probably want this path to be in your `tests/` directory, perhaps next to your cassettes directory, e.g., where your cassettes are in `tests/fixtures`, your files from requests that write to disk are in `tests/files`. If you want to ignore these files in your installed package, add them to `.Rinstignore`. If you want these files ignored on build then add them to `.Rbuildignore` (though if you do, tests that depend on these files probably will not work because they won't be found; so you'll likely have to skip the associated tests as well).

Contexts:

- `turned_off` (logical) VCR is turned on by default. Default: `FALSE`
- `allow_unused_http_interactions` (logical) Default: `TRUE`
- `allow_http_connections_when_no_cassette` (logical) Determines how vcr treats HTTP requests that are made when no vcr cassette is in use. When `TRUE`, requests made when there is no vcr cassette in use will be allowed. When `FALSE` (default), an UnhandledHTTPRequestError error will be raised for any HTTP request made when there is no cassette in use

Filtering:

- `ignore_hosts` (character) Vector of hosts to ignore. e.g., localhost, or google.com. These hosts are ignored and real HTTP requests allowed to go through
- `ignore_localhost` (logical) Default: `FALSE`
- `ignore_request` List of requests to ignore. NOT USED RIGHT NOW, sorry
- `filter_sensitive_data` named list of values to replace. Format is:

  `list(thing_to_replace_it_with = thing_to_replace)`

  We replace all instances of `thing_to_replace` with `thing_to_replace_it_with`. Uses `gsub()` internally, with `fixed=TRUE`; so does exact matches. Before recording (writing to a cassette) we do the replacement and then when reading from the cassette we do the reverse replacement to get back to the real data. Before record replacement happens in internal function `write_interactions()`, while before playback replacement happens in internal function `YAML$deserialize()`
- `filter_sensitive_data_regex` named list of values to replace. Follows `filter_sensitive_data` format, except uses `fixed=FALSE` in the `gsub()` function call; this means that the value in `thing_to_replace` is a regex pattern.
- `filter_request_headers` (character/list) **request** headers to filter. A character vector of request headers to remove - the headers will not be recorded to disk. Alternatively, a named list similar to `filter_sensitive_data` instructing vcr with what value to replace the real value of the request header.
- `filter_response_headers` (named list) **response** headers to filter. A character vector of response headers to remove - the headers will not be recorded to disk. Alternatively, a named list similar to `filter_sensitive_data` instructing vcr with what value to replace the real value of the response header.
- `filter_query_parameters` (named list) query parameters to filter. A character vector of query parameters to remove - the query parameters will not be recorded to disk. Alternatively, a named list similar to `filter_sensitive_data` instructing vcr with what value to replace the real value of the query parameter.

Errors:

- `verbose_errors` Do you want more verbose errors or less verbose errors when cassette recording/usage fails? Default is `FALSE`, that is, less verbose errors. If `TRUE`, error messages will include more details about what went wrong and suggest possible solutions. For testing in an interactive R session, if `verbose_errors=FALSE`, you can run `vcr_last_error()` to get the full error. If in non-interactive mode, which most users will be in when running the entire test suite for a package, you can set an environment variable (`VCR_VERBOSE_ERRORS`) to toggle this setting (e.g., `Sys.setenv(VCR_VERBOSE_ERRORS=TRUE); devtools::test()`)

Internals:

- `cassettes` (list) don't use
- `linked_context` (logical) linked context
- `uri_parser` the uri parser, default: `crul::url_parse()`

Logging:

- `log` (logical) should we log important vcr things? Default: `FALSE`
- `log_opts` (list) Additional logging options:
  - 'file' either `"console"` or a file path to log to

  - 'log_prefix' default: "Cassette". We insert the cassette name after that prefix, then the rest of the message.
  - More to come...

Cassette Options:

These settings can be configured globally, using `vcr_configure()`, or locally, using either `use_cassette()` or `insert_cassette()`. Global settings are applied to *all* cassettes but are overridden by settings defined locally for individual cassettes.

- `record` (character) One of 'all', 'none', 'new_episodes', or 'once'. See recording
- `match_requests_on` vector of matchers. Default: (`method`, `uri`) See request-matching for details.
- `serialize_with`: (character) "yaml" or "json". Note that you can have multiple cassettes with the same name as long as they use different serializers; so if you only want one cassette for a given cassette name, make sure to not switch serializers, or clean up files you no longer need.
- `json_pretty`: (logical) want JSON to be newline separated to be easier to read? Or remove newlines to save disk space? default: FALSE
- `persist_with` (character) only option is "FileSystem"
- `preserve_exact_body_bytes` (logical) preserve exact body bytes for
- `re_record_interval` (numeric) When given, the cassette will be re-recorded at the given interval, in seconds.
- `clean_outdated_http_interactions` (logical) Should outdated interactions be recorded back to file. Default: `FALSE`
- `quiet` (logical) Suppress any messages from both vcr and webmockr. Default: `TRUE`
- `warn_on_empty_cassette` (logical) Should a warning be thrown when an empty cassette is detected? Empty cassettes are cleaned up (deleted) either way. This option only determines whether a warning is thrown or not. Default: `FALSE`

Examples

```
vcr_configure(dir = tempdir())
vcr_configure(dir = tempdir(), record = "all")
vcr_configuration()
vcr_config_defaults()
vcr_configure(dir = tempdir(), ignore_hosts = "google.com")
vcr_configure(dir = tempdir(), ignore_localhost = TRUE)


# logging
vcr_configure(dir = tempdir(), log = TRUE,
  log_opts = list(file = file.path(tempdir(), "vcr.log")))
vcr_configure(dir = tempdir(), log = TRUE, log_opts = list(file = "console"))
vcr_configure(dir = tempdir(), log = TRUE,
 log_opts = list(
   file = file.path(tempdir(), "vcr.log"),
   log_prefix = "foobar"
))
vcr_configure(dir = tempdir(), log = FALSE)
```

```
# filter sensitive data
vcr_configure(dir = tempdir(),
  filter_sensitive_data = list(foo = "<bar>")
)
vcr_configure(dir = tempdir(),
  filter_sensitive_data = list(foo = "<bar>", hello = "<world>")
)
```

`vcr_test_path` *Locate file in tests directory*

Description

This function, similar to `testthat::test_path()`, is designed to work both interactively and during tests, locating files in the `tests/` directory.

Usage

```
vcr_test_path(...)
```

Arguments

`...`	Character vectors giving path component. each character string gets added on to the path, e.g., `vcr_test_path("a", "b")` becomes `tests/a/b` relative to the root of the package.

Value

A character vector giving the path

Note

`vcr_test_path()` assumes you are using testthat for your unit tests.

Examples

```
if (interactive()) {
vcr_test_path("fixtures")
}
```

Index