

# Package ‘httptest’

June 12, 2023

**Type** Package

**Title** A Test Environment for HTTP Requests

**Description** Testing and documenting code that communicates with remote servers can be painful. Dealing with authentication, server state, and other complications can make testing seem too costly to bother with. But it doesn't need to be that hard. This package enables one to test all of the logic on the R sides of the API in your package without requiring access to the remote service. Importantly, it provides three contexts that mock the network connection in different ways, as well as testing functions to assert that HTTP requests were---or were not---made. It also allows one to safely record real API responses to use as test fixtures. The ability to save responses and load them offline also enables one to write vignettes and other dynamic documents that can be distributed without access to a live server.

**Version** 4.2.1

**URL** <https://enpiar.com/r/httptest/>,  
<https://github.com/nealrichardson/httptest>

**BugReports** <https://github.com/nealrichardson/httptest/issues>

**License** MIT + file LICENSE

**Depends** R (>= 3.5.0), testthat

**Imports** curl, digest, httr, jsonlite, stats, utils

**Suggests** knitr, pkgload, rmarkdown, spelling, xml2

**Language** en-US

**RoxygenNote** 7.2.3

**Encoding** UTF-8

**VignetteBuilder** knitr

**Config/testthat/edition** 3

**Config/Needs/coverage** covr

**NeedsCompilation** no

**Author** Neal Richardson [aut, cre] (<<https://orcid.org/0009-0002-7992-3520>>),  
Jonathan Keane [ctb],  
Maëlle Salmon [ctb] (<<https://orcid.org/0000-0002-2815-0399>>)

**Maintainer** Neal Richardson <[neal.p.richardson@gmail.com](mailto:neal.p.richardson@gmail.com)>

**Repository** CRAN

**Date/Publication** 2023-06-12 12:40:02 UTC

## R topics documented:

.mockPaths	2
block_requests	3
build_mock_url	4
capture_requests	5
change_state	7
expect_header	7
expect_json_equivalent	8
expect_verb	9
fake_response	10
gsub_response	11
httptest	12
public	13
redact_cookies	14
set_redactor	15
set_requester	16
skip_if_disconnected	16
start_vignette	17
stop_mocking	18
use_httptest	18
use_mock_api	19
without_internet	20
with_fake_http	21
with_mock_api	22
with_mock_dir	22
<b>Index</b>	<b>24</b>

---

.mockPaths

*Set an alternate directory for mock API fixtures*

---

### Description

By default, `with_mock_api` will look for mocks relative to the current working directory (the test directory). If you want to look in other places, you can call `.mockPaths` to add directories to the search path.

**Usage**

```
.mockPaths(new)
```

**Arguments**

`new` Either a character vector of path(s) to add, or NULL to reset to the default.

**Details**

It works like `base::.libPaths()`: any directories you specify will be added to the list and searched first. The default directory will be searched last. Only unique values are kept: if you provide a path that is already found in `.mockPaths`, the result effectively moves that path to the first position.

For finer-grained control, or to completely override the default behavior of searching in the current working directory, you can set the option "httptest.mock.paths" directly.

**Value**

If `new` is omitted, the function returns the current search paths, a character vector. If `new` is provided, the updated value will be returned invisibly.

**Examples**

```
identical(.mockPaths(), ".")
.mockPaths("/var/somewhere/else")
identical(.mockPaths(), c("/var/somewhere/else", "."))
.mockPaths(NULL)
identical(.mockPaths(), ".")
```

---

block_requests	<i>Block HTTP requests</i>
----------------	----------------------------

---

**Description**

This function intercepts HTTP requests made through `httr` and raises an informative error instead. It is what `without_internet()` does, minus the automatic disabling of mocking when the context finishes.

**Usage**

```
block_requests()
```

**Details**

Note that you in order to resume normal request behavior, you will need to call `stop_mocking()` yourself—this function does not clean up after itself as 'without\_internet' does.

**Value**

Nothing; called for its side effects.

**See Also**

[without\\_internet\(\)](#) [stop\\_mocking\(\)](#) [use\\_mock\\_api\(\)](#)

---

build_mock_url	<i>Convert a request to a mock file path</i>
----------------	--

---

**Description**

Requests are translated to mock file paths according to several rules that incorporate the request method, URL, query parameters, and body.

**Usage**

```
build_mock_url(req, method = "GET")
```

**Arguments**

req	A request object, or a character "URL" to convert
method	character HTTP method. If req is a 'request' object, its request method will override this argument

**Details**

First, the request protocol, such as "https://", is removed from the URL. Second, if the request URL contains a query string, it will be popped off, hashed by `digest::digest()`, and the first six characters appended to the file being read. Third, request bodies are similarly hashed and appended. Finally, if a request method other than GET is used it will be appended to the end of the end of the file name.

Mock file paths also have a file extension appended, based on the Content-Type of the response, though this function, which is only concerned with the request, does not add the extension. In an HTTP API, a "directory" itself is a resource, so the extension allows distinguishing directories and files in the file system. That is, a mocked GET("http://example.com/api/") may read a "example.com/api.json" file, while GET("http://example.com/api/object1/") reads "example.com/api/object1.json".

Other examples:

- GET("http://example.com/api/object1/?a=1") may read "example.com/api/object1-b64371.xml".
- POST("http://example.com/api/object1/?a=1") may read "example.com/api/object1-b64371-POST.json".

This function is exported so that other packages can construct similar mock behaviors or override specific requests at a higher level than `with_mock_api` mocks.

Note that if you are trying to guess the mock file paths corresponding to a test for which you intend to create a mock file manually, instead of trying to build the URL, you should run the test with `with_mock_api` as the error message will contain the mock file path.

### Value

A file path and name, without an extension. The file, or a file with some extension appended, may or may not exist: existence is not a concern of this function.

### See Also

[with\\_mock\\_api\(\)](#) [capture\\_requests\(\)](#)

---

capture_requests	<i>Record API responses as mock files</i>
------------------	---

---

### Description

`capture_requests` is a context that collects the responses from requests you make and stores them as mock files. This enables you to perform a series of requests against a live server once and then build your test suite using those mocks, running your tests in [with\\_mock\\_api\(\)](#).

### Usage

```
capture_requests(expr, path, ...)
```

```
start_capturing(path = NULL, simplify = TRUE)
```

```
stop_capturing()
```

### Arguments

<code>expr</code>	Code to run inside the context
<code>path</code>	Where to save the mock files. Default is the first directory in <code>.mockPaths()</code> , which if not otherwise specified is the current working directory. It is generally better to call <code>.mockPaths()</code> directly if you want to write to a different path, rather than using the <code>path</code> argument.
<code>...</code>	Arguments passed through <code>capture_requests</code> to <code>start_capturing</code>
<code>simplify</code>	logical: if TRUE (default), JSON responses with status 200 will be written as just the text of the response body. In all other cases, and when <code>simplify</code> is FALSE, the "response" object will be written out to a .R file using <code>base::dput()</code> .

## Details

`start_capturing` and `stop_capturing` allow you to turn on/off request recording for more convenient use in an interactive session.

Recorded responses are written out as plain-text files. By storing fixtures as plain-text files, you can more easily confirm that your mocks look correct, and you can more easily maintain them without having to re-record them. If the API changes subtly, such as when adding an additional attribute to an object, you can just touch up the mocks.

If the response has status 200 OK and the Content-Type maps to a supported file extension—currently `.json`, `.html`, `.xml`, `.txt`, `.csv`, and `.tsv`—just the response body will be written out, using the appropriate extension. 204 No Content status responses will be stored as an empty file with extension `.204`. Otherwise, the response will be written as a `.R` file containing syntax that, when executed, recreates the `httr` "response" object.

If you have trouble when recording responses, or are unsure where the files are being written, set `options(httptest.verbose=TRUE)` to print a message for every file that is written containing the absolute path of the file.

## Value

`capture_requests` returns the result of `expr`. `start_capturing` invisibly returns the path it is given. `stop_capturing` returns nothing; it is called for its side effects.

## See Also

[build\\_mock\\_url\(\)](#) for how requests are translated to file paths. And see `vignette("redacting")` for details on how to prune sensitive content from responses when recording.

## Examples

```
## Not run:
capture_requests({
  GET("http://httpbin.org/get")
  GET("http://httpbin.org")
  GET("http://httpbin.org/response-headers",
      query = list(`Content-Type` = "application/json")
  )
})
# Or:
start_capturing()
GET("http://httpbin.org/get")
GET("http://httpbin.org")
GET("http://httpbin.org/response-headers",
    query = list(`Content-Type` = "application/json")
)
stop_capturing()

## End(Not run)
```

---

change_state	<i>Handle a change of server state</i>
--------------	--

---

**Description**

In a vignette, put a call to `change_state()` before any code block that makes a change on the server, or rather, before any code block that might repeat the same request previously done and expect a different result.

**Usage**

```
change_state()
```

**Details**

`change_state()` works by layering a new directory on top of the existing `.mockPaths()`, so fixtures are recorded/loaded there, masking rather than overwriting previously recorded responses for the same request. In vignettes, these mock layers are subdirectories with integer names.

**Value**

Invisibly, the return of `.mockPaths()` with the new path added.

---

expect_header	<i>Test that an HTTP request is made with a header</i>
---------------	--

---

**Description**

This expectation checks that a HTTP header (and potentially header value) is present in a request. It works by inspecting the request object and raising warnings that are caught by `testthat::expect_warning()`.

**Usage**

```
expect_header(..., ignore.case = TRUE)
```

**Arguments**

<code>...</code>	Arguments passed to <code>expect_warning</code>
<code>ignore.case</code>	logical: if FALSE, the pattern matching is <i>case sensitive</i> and if TRUE, case is ignored during matching. Default is TRUE; note that this is the opposite of <code>expect_warning</code> but is appropriate here because HTTP header names are case insensitive.

**Details**

`expect_header` works both in the mock HTTP contexts and on "live" HTTP requests.

**Value**

NULL, according to expect\_warning.

**Examples**

```
library(httr)
with_fake_http({
  expect_header(
    GET("http://example.com", config = add_headers(Accept = "image/png")),
    "Accept: image/png"
  )
})
```

---

expect\_json\_equivalent

*Test that objects would generate equivalent JSON*

---

**Description**

Named lists in R are ordered, but they translate to unordered objects in JSON. This test expectation loosens the equality check of two objects to ignore the order of elements in a named list.

**Usage**

```
expect_json_equivalent(
  object,
  expected,
  info = NULL,
  label = "object",
  expected.label = "expected"
)
```

**Arguments**

object	object to test
expected	expected value
info	extra information to be included in the message
label	character name by which to refer to object in the test result. Because the tools for deparsing object names that 'testthat' uses aren't exported from that package, the default here is just "object".
expected.label	character same as label but for expected

**Value**

Invisibly, returns object for optionally passing to other expectations.



**See Also**

[testthat::expect\\_equivalent\(\)](#)

---

expect\_verb

*Expectations for mocked HTTP requests*

---

**Description**

The mock contexts in `httptest` can raise errors or messages when requests are made, and those (error) messages have three elements, separated by space: (1) the request method (e.g. "GET"); (2) the request URL; and (3) the request body, if present. These verb-expectation functions look for this message shape. `expect_PUT`, for instance, looks for a request message that starts with "PUT".

This means that `expect_verb` functions won't work outside of mock context, as no error would be raised while making a request. Thus, any `expect_verb` function should be wrapped inside a mocking function like `without_internet()`, as shown in the examples.

**Usage**

```
expect_GET(object, url = "", ...)
```

```
expect_POST(object, url = "", ...)
```

```
expect_PATCH(object, url = "", ...)
```

```
expect_PUT(object, url = "", ...)
```

```
expect_DELETE(object, url = "", ...)
```

```
expect_no_request(object, ...)
```

**Arguments**

`object` Code to execute that may cause an HTTP request

`url` character: the URL you expect a request to be made to. Default is an empty string, meaning that you can just assert that a request is made with a certain method without asserting anything further.

`...` character segments of a request payload you expect to be included in the request body, to be joined together by `paste0`. You may also pass any of the following named logical arguments, which will be passed to `base::grepl()`:

- `fixed`: Should matching take the pattern as is or treat it as a regular expression. Default: TRUE, and note that this default is the opposite of the default in `grepl`. (The rest of the arguments follow its defaults.)
- `ignore.case`: Should matching be done case insensitively? Default: FALSE, meaning matches are case sensitive.
- `perl`: Should Perl-compatible regular expressions be used? Default: FALSE
- `useBytes`: Should matching be done byte-by-byte rather than character-by-character? Default: FALSE

**Value**

A testthat 'expectation'.

**Examples**

```
library(httr)
# without_internet provides required mock context for expectations
without_internet({
  expect_GET(
    GET("http://httpbin.org/get"),
    "http://httpbin.org/get"
  )
  expect_GET(GET("http://httpbin.org/get"),
    "http://httpbin.org/[a-z]+",
    fixed = FALSE
  ) # For regular expression matching
  expect_PUT(
    PUT("http://httpbin.org/put", body = '{"a":1}'),
    "http://httpbin.org/put",
    '{"a":1}'
  )
  expect_PUT(PUT("http://httpbin.org/put", body = '{"a":1}'))
  expect_no_request(rnorm(5))
})
```

---

fake\_response

*Return something that looks like a 'response'*

---

**Description**

These functions allow mocking of HTTP requests without requiring an internet connection or server to run against. Their return shape is a 'httr' "response" class object that should behave like a real response generated by a real request.

**Usage**

```
fake_response(
  request,
  verb = "GET",
  status_code = 200,
  headers = list(),
  content = NULL
)
```

**Arguments**

`request` An 'httr' request-class object. A character URL is also accepted, for which a fake request object will be created, using the `verb` argument as well.

verb	Character name for the HTTP verb, if request is a URL. Default is "GET".
status_code	Integer HTTP response status
headers	Optional list of additional response headers to return
content	If supplied, a JSON-serializable list that will be returned as response content with Content-Type: application/json. If no content is provided, and if the status_code is not 204 No Content, the url will be set as the response content with Content-Type: text/plain.

**Value**

An 'httr' response class object.

---

gsub_response	<i>Find and replace within a 'response' or 'request'</i>
---------------	--

---

**Description**

These functions pass their arguments to `base::gsub()` in order to find and replace string patterns (regular expressions) within request or response objects. `gsub_request()` replaces in the request URL and any request body fields; `gsub_response()` replaces in the response URL, the response body, and it calls `gsub_request()` on the request object found within the response.

**Usage**

```
gsub_response(response, pattern, replacement, ...)
```

```
gsub_request(request, pattern, replacement, ...)
```

**Arguments**

response	An 'httr' response object to sanitize.
pattern	From <code>base::gsub()</code> : "character string containing a regular expression (or character string for <code>fixed = TRUE</code> ) to be matched in the given character vector." Passed to <code>gsub()</code> . See the docs for <code>gsub()</code> for further details.
replacement	A replacement for the matched pattern, possibly including regular expression backreferences. Passed to <code>gsub()</code> . See the docs for <code>gsub()</code> for further details.
...	Additional logical arguments passed to <code>gsub()</code> : <code>ignore.case</code> , <code>perl</code> , <code>fixed</code> , and <code>useBytes</code> are the possible options.
request	An 'httr' request object to sanitize.

**Details**

Note that, unlike `gsub()`, the first argument of the function is request or response, not pattern, while the equivalent argument in `gsub()`, "x", is placed third. This difference is to maintain consistency with the other redactor functions in `httpptest`, which all take response as the first argument.

**Value**

A request or response object, same as was passed in, with the pattern replaced in the URLs and bodies.

---

 httptest

 httptest: A Test Environment for HTTP Requests
 

---

**Description**

If **httr** makes HTTP easy and **testthat** makes testing fun, **httptest** makes testing your code that uses HTTP a simple pleasure.

**Details**

The httptest package lets you test R code that wraps an API without requiring access to the remote service. It provides three test **contexts** that mock the network connection in different ways. `with_mock_api()` lets you provide custom fixtures as responses to requests, stored as plain-text files in your test directory. `without_internet()` converts HTTP requests into errors that print the request method, URL, and body payload, if provided, allowing you to assert that a function call would make a correctly-formed HTTP request or assert that a function does not make a request (because if it did, it would raise an error in this context). `with_fake_http()` raises a "message" instead of an "error", and HTTP requests return a "response"-class object. Like `without_internet`, it allows you to assert that the correct requests were (or were not) made, but it doesn't cause the code to exit with an error.

httptest offers additional **expectations** to assert that HTTP requests were—or were not—made. `expect_GET()`, `expect_PUT()`, `expect_PATCH()`, `expect_POST()`, and `expect_DELETE()` assert that the specified HTTP request is made within one of the test contexts. They catch the error or message raised by the mocked HTTP service and check that the request URL and optional body match the expectation. `expect_no_request()` is the inverse of those: it asserts that no error or message from a mocked HTTP service is raised. `expect_header()` asserts that an HTTP request, mocked or not, contains a request header. `expect_json_equivalent()` checks that two R objects would generate equivalent JSON, taking into account how JSON objects are unordered whereas R named lists are ordered.

For an overview of testing with httptest, see `vignette("httptest")`.

The package also includes `capture_requests()`, a context that collects the responses from requests you make and stores them as mock files. This enables you to perform a series of requests against a live server once and then build your test suite using those mocks, running your tests in `with_mock_api`.

When recording requests, by default httptest looks for and redacts the standard ways that auth credentials are passed in requests. This prevents you from accidentally publishing your personal tokens. The redacting behavior is fully customizable, either by providing a function (response) `{...}` to `set_redactor()`, or by placing a function in your package's `inst/httptest/redact.R` that will be used automatically any time you record requests with your package loaded. See `vignette("redacting")` for details.

httptest also enables you to write package vignettes and other R Markdown documents that communicate with a remote API. By adding as little as `start_vignette()` to the beginning of your vignette, you can safely record API responses from a live session, using your secret credentials. These API responses are scrubbed of sensitive personal information and stored in a subfolder in your vignettes directory. Subsequent vignette builds, including on continuous-integration services, CRAN, and your package users' computers, use these recorded responses, allowing the document to regenerate without a network connection or API credentials. To record fresh API responses, delete the subfolder of cached responses and re-run. See `vignette("vignettes")` for more discussion and links to examples.

---

public

*Test that functions are exported*

---

## Description

It's easy to forget to document and export a new function. Using `testthat` for your test suite makes it even easier to forget because it evaluates your test code inside the package's namespace, so internal, non-exported functions can be accessed. So you might write a new function, get passing tests, and then tell your package users about the function, but when they try to run it, they get `Error: object 'coolNewFunction' not found`.

## Usage

```
public(...)
```

## Arguments

```
...           Code to evaluate
```

## Details

Wrap `public()` around test blocks to assert that the functions they call are exported (and thus fail if you haven't documented them with `@export` or otherwise added them to your package `NAMESPACE` file).

An alternative way to test that your functions are exported from the package namespace is with examples in the documentation, which `R CMD check` runs in the global namespace and would thus fail if the functions aren't exported. However, code that calls remote APIs, potentially requiring specific server state and authentication, may not be viable to run in examples in `R CMD check`. `public()` provides a solution that works for these cases because you can test your namespace exports in the same place where you are testing the code with API mocks or other safe testing contexts.

## Value

The result of `...` evaluated in the global environment (and not the package environment).

---

redact_cookies	<i>Remove sensitive content from HTTP responses</i>
----------------	---

---

### Description

When recording requests for use as test fixtures, you don't want to include secrets like authentication tokens and personal ids. These functions provide a means for redacting this kind of content, or anything you want, from responses that [capture\\_requests\(\)](#) saves.

### Usage

```
redact_cookies(response)

redact_headers(response, headers = c())

within_body_text(response, FUN)

redact_auth(response)
```

### Arguments

response	An 'httr' response object to sanitize.
headers	For <code>redact_headers()</code> , a character vector of header names to sanitize. Note that <code>redact_headers()</code> itself does not do redacting but returns a function that when called does the redacting.
FUN	For <code>within_body_text()</code> , a function that takes as its argument a character vector and returns a modified version of that. This function will be applied to the text of the response's "content".

### Details

`redact_cookies()` removes cookies from 'httr' response objects. `redact_headers()` lets you target selected request and response headers for redaction. `redact_auth()` is a convenience wrapper around them for a useful default redactor in `capture_requests()`.

`within_body_text()` lets you manipulate the text of the response body and manages the parsing of the raw (binary) data in the 'response' object.

### Value

All redacting functions return a well-formed 'httr' response object.

### See Also

`vignette("redacting", package="httptest")` for a detailed discussion of what these functions do and how to customize them. [gsub\\_response\(\)](#) is another redactor.

---

set_redactor	<i>Set a response redactor</i>
--------------	--------------------------------

---

## Description

A redactor is a function that alters the response content being written out in the `capture_requests()` context, allowing you to remove sensitive values, such as authentication tokens, as well as any other modification or truncation of the response body. By default, the `redact_auth()` function will be used to purge standard auth methods, but `set_redactor()` allows you to provide a different one.

## Usage

```
set_redactor(FUN)
```

## Arguments

FUN	A function or expression that modifies response objects. Specifically, a valid input is one of: <ul style="list-style-type: none"><li>• A function taking a single argument, the response, and returning a valid response object.</li><li>• A formula as shorthand for an anonymous function with <code>.</code> as the "response" argument, as in the <code>purrr</code> package. That is, instead of function (response) <code>redact_headers(response, "X-Custom-Header")</code>, you can use <code>~ redact_headers(., "X-Custom-Header")</code></li><li>• A list of redacting functions/formulas, which will be executed in sequence on the response</li><li>• NULL, to override the default <code>redact_auth()</code>.</li></ul>
-----	---

## Details

Alternatively, you can put a redacting function in `inst/httpptest/redact.R` in your package, and any time your package is loaded (as in when running tests or building vignettes), the function will be used automatically.

For further details on how to redact responses, see `vignette("redacting")`.

## Value

Invisibly, the redacting function, validated and perhaps modified. Formulas and function lists are turned into proper functions. NULL as input returns the `force()` function.

## See Also

[set\\_requester\(\)](#)

---

set_requester	<i>Set a request preprocessor</i>
---------------	-----------------------------------

---

**Description**

Set a request preprocessor

**Usage**

```
set_requester(FUN)
```

**Arguments**

FUN	<p>A function or expression that modifies request objects. Specifically, a valid input is one of:</p> <ul style="list-style-type: none"> <li>• A function taking a single argument, the request, and returning a valid request object.</li> <li>• A formula as shorthand for an anonymous function with . as the "request" argument, as in the purrr package.</li> <li>• A list of functions/formulas, which will be executed in sequence on the request.</li> <li>• NULL, to override the default redact_auth().</li> </ul>
-----	--

**Value**

Invisibly, FUN, validated and perhaps modified.

**See Also**

[set\\_redactor\(\)](#)

---

skip_if_disconnected	<i>Skip tests that need an internet connection if you don't have one</i>
----------------------	--

---

**Description**

Temporary connection trouble shouldn't fail your build.

**Usage**

```
skip_if_disconnected(
  message = paste("Offline: cannot reach", url),
  url = "http://httpbin.org/"
)
```



**Arguments**

message            character message to be printed, passed to `testthat::skip()`  
 url                character URL to ping to check for a working connection

**Details**

Note that if you call this from inside one of the mock contexts, it will follow the mock's behavior. That is, inside `with_fake_http()`, the check will pass and the following tests will run, but inside `without_internet()`, the following tests will be skipped.

**Value**

If offline, a test skip; else invisibly returns TRUE.

**See Also**

`testthat::skip()`

---

start_vignette	<i>Set mocking/capturing state for a vignette</i>
----------------	---

---

**Description**

Use `start_vignette()` to either use previously recorded responses, if they exist, or capture real responses for future use.

**Usage**

```
start_vignette(path, ...)  
  
end_vignette()
```

**Arguments**

path                Root file path for the mocks for this vignette. A good idea is to use the file name of the vignette itself.  
 ...                Optional arguments passed to `start_capturing()`

**Details**

In a vignette or other R Markdown or Sweave document, place `start_vignette()` in an R code block at the beginning, before the first API request is made, and put `end_vignette()` in a R code chunk at the end. You may want to make those R code chunks have `echo=FALSE` in order to hide the fact that you're calling them.

The behavior changes based on the existence of the path directory. The first time you build the vignette, the directory won't exist yet, so it will make real requests and record them inside of path.

On subsequent runs, the mocks will be used. To record fresh responses from the server, delete the path directory, and the responses will be recorded again the next time the vignette runs.

If you have additional setup code that you'd like available across all of your package's vignettes, put it in `inst/httpptest/start-vignette.R` in your package, and it will be called in `start_vignette()` before the mock/record context is set. Similarly, teardown code can go in `inst/httpptest/end-vignette.R`, evaluated in `end_vignette()` after mocking is stopped.

### Value

Nothing; called for its side effect of starting/ending response recording or mocking.

### See Also

[start\\_capturing\(\)](#) for how requests are recorded; [use\\_mock\\_api\(\)](#) for how previously recorded requests are loaded; [change\\_state\(\)](#) for how to handle recorded requests when the server state is changing; [vignette\("vignettes", package="httpptest"\)](#) for an overview of all

---

<code>stop_mocking</code>	<i>Turn off request mocking</i>
---------------------------	---------------------------------

---

### Description

This function "untraces" the `httr` request functions so that normal, real requesting behavior can be resumed.

### Usage

```
stop_mocking()
```

### Value

Nothing; called for its side effects

---

<code>use_httptest</code>	<i>Use 'httptest' in your tests</i>
---------------------------	-------------------------------------

---

### Description

This function adds `httptest` to `Suggests` in the package `DESCRIPTION` and loads it in `tests/testthat/setup.R`. Call it once when you're setting up a new package test suite.

### Usage

```
use_httptest(path = ".")
```

**Arguments**

path                    character path to the package

**Details**

The function is idempotent: if `httpptest` is already added to these files, no additional changes will be made.

**Value**

Nothing: called for file system side effects.

---

use_mock_api	<i>Turn on API mocking</i>
--------------	----------------------------

---

**Description**

This function intercepts HTTP requests made through `http` and serves mock file responses instead. It is what `with_mock_api()` does, minus the automatic disabling of mocking when the context finishes.

**Usage**

```
use_mock_api()
```

**Details**

Note that you in order to resume normal request behavior, you will need to call `stop_mocking()` yourself—this function does not clean up after itself as `'with_mock_api'` does.

**Value**

Nothing; called for its side effects.

**See Also**

`with_mock_api()` `stop_mocking()` `block_requests()`

---

without_internet	<i>Make all HTTP requests raise an error</i>
------------------	--

---

### Description

without\_internet simulates the situation when any network request will fail, as in when you are without an internet connection. Any HTTP request through the verb functions in `httr` will raise an error.

### Usage

```
without_internet(expr)
```

### Arguments

expr                    Code to run inside the mock context

### Details

The error message raised has a well-defined shape, made of three elements, separated by space: (1) the request method (e.g. "GET"); (2) the request URL; and (3) the request body, if present. The verb-expectation functions, such as `expect_GET()` and `expect_POST()`, look for this shape.

### Value

The result of `expr`

### See Also

`block_requests()` to enable mocking on its own (not in a context)

### Examples

```
without_internet({
  expect_error(
    httr::GET("http://httpbin.org/get"),
    "GET http://httpbin.org/get"
  )
  expect_error(httr::PUT("http://httpbin.org/put",
    body = '{"a":1}'
  ),
  'PUT http://httpbin.org/put {"a":1}',
  fixed = TRUE
})
```

---

with_fake_http	<i>Make all HTTP requests return a fake response</i>
----------------	--

---

## Description

In this context, HTTP verb functions raise a 'message' so that test code can assert that the requests are made. As in `without_internet()`, the message raised has a well-defined shape, made of three elements, separated by space: (1) the request method (e.g. "GET" or "POST"); (2) the request URL; and (3) the request body, if present. The verb-expectation functions, such as `expect_GET` and `expect_POST`, look for this shape.

## Usage

```
with_fake_http(expr)
```

## Arguments

expr	Code to run inside the fake context
------	-------------------------------------

## Details

Unlike `without_internet`, the HTTP functions do not error and halt execution, instead returning a response-class object so that code calling the HTTP functions can proceed with its response handling logic and itself be tested. The response it returns echoes back most of the request itself, similar to how some endpoints on <http://httpbin.org> do.

## Value

The result of `expr`

## Examples

```
with_fake_http({
  expect_GET(req1 <- httr::GET("http://example.com"), "http://example.com")
  req1$url
  expect_POST(
    req2 <- httr::POST("http://example.com", body = '{"a":1}'),
    "http://example.com"
  )
  httr::content(req2)
})
```

---

with_mock_api	<i>Serve a mock API from files</i>
---------------	------------------------------------

---

### Description

In this context, HTTP requests attempt to load API response fixtures from files. This allows test code to proceed evaluating code that expects HTTP requests to return meaningful responses. Requests that do not have a corresponding fixture file raise errors, like how [without\\_internet\(\)](#) does.

### Usage

```
with_mock_api(expr)
```

### Arguments

expr	Code to run inside the fake context
------	-------------------------------------

### Details

Requests are translated to mock file paths according to several rules that incorporate the request method, URL, query parameters, and body. See [build\\_mock\\_url\(\)](#) for details.

File paths for API fixture files may be relative to the 'tests/testthat' directory, i.e. relative to the .R test files themselves. This is the default location for storing and retrieving mocks, but you can put them anywhere you want as long as you set the appropriate location with [.mockPaths\(\)](#).

### Value

The result of expr

### See Also

[use\\_mock\\_api\(\)](#) to enable mocking on its own (not in a context); [build\\_mock\\_url\(\)](#); [.mockPaths\(\)](#)

---

with_mock_dir	<i>Use or create mock files depending on their existence</i>
---------------	--

---

### Description

This context will switch the [.mockPaths\(\)](#) to tests/testthat/dir (and then resets it to what it was before). If the tests/testthat/dir folder doesn't exist, [capture\\_requests\(\)](#) will be run to create mocks. If it exists, [with\\_mock\\_api\(\)](#) will be run. To re-record mock files, simply delete tests/testthat/dir and run the test.

### Usage

```
with_mock_dir(dir, expr, simplify = TRUE, replace = TRUE)
```

**Arguments**

<code>dir</code>	character string, unique folder name that will be used or created under <code>tests/testthat/</code>
<code>expr</code>	Code to run inside the fake context
<code>simplify</code>	logical: if TRUE (default), JSON responses with status 200 will be written as just the text of the response body. In all other cases, and when <code>simplify</code> is FALSE, the "response" object will be written out to a .R file using <code>base::dput()</code> .
<code>replace</code>	Logical: should the mock directory replace current mock directories? Default is TRUE.

# Index

`.mockPaths`, 2  
`.mockPaths()`, 5, 7, 22

`base::libPaths()`, 3  
`base::dput()`, 5, 23  
`base::grepl()`, 9  
`base::gsub()`, 11  
`block_requests`, 3  
`block_requests()`, 19, 20  
`build_mock_url`, 4  
`build_mock_url()`, 6, 22

`capture_requests`, 5  
`capture_requests()`, 5, 12, 14, 15, 22  
`change_state`, 7  
`change_state()`, 18

`digest::digest()`, 4

`end_vignette (start_vignette)`, 17  
`expect_DELETE (expect_verb)`, 9  
`expect_DELETE()`, 12  
`expect_GET (expect_verb)`, 9  
`expect_GET()`, 12, 20  
`expect_header`, 7  
`expect_header()`, 12  
`expect_json_equivalent`, 8  
`expect_json_equivalent()`, 12  
`expect_no_request (expect_verb)`, 9  
`expect_no_request()`, 12  
`expect_PATCH (expect_verb)`, 9  
`expect_PATCH()`, 12  
`expect_POST (expect_verb)`, 9  
`expect_POST()`, 12, 20  
`expect_PUT (expect_verb)`, 9  
`expect_PUT()`, 12  
`expect_verb`, 9

`fake_response`, 10

`gsub_request (gsub_response)`, 11

`gsub_response`, 11  
`gsub_response()`, 14

`httptest`, 12

`public`, 13

`redact (redact_cookies)`, 14  
`redact_auth (redact_cookies)`, 14  
`redact_auth()`, 15  
`redact_cookies`, 14  
`redact_headers (redact_cookies)`, 14

`set_redactor`, 15  
`set_redactor()`, 16  
`set_requester`, 16  
`set_requester()`, 15  
`skip_if_disconnected`, 16  
`start_capturing (capture_requests)`, 5  
`start_capturing()`, 18  
`start_vignette`, 17  
`start_vignette()`, 13  
`stop_capturing (capture_requests)`, 5  
`stop_mocking`, 18  
`stop_mocking()`, 3, 4, 19

`testthat::expect_equivalent()`, 9  
`testthat::expect_warning()`, 7  
`testthat::skip()`, 17

`use_httptest`, 18  
`use_mock_api`, 19  
`use_mock_api()`, 4, 18, 22

`with_fake_http`, 21  
`with_fake_http()`, 12, 17  
`with_mock_api`, 22  
`with_mock_api()`, 5, 12, 19, 22  
`with_mock_dir`, 22  
`within_body_text (redact_cookies)`, 14  
`without_internet`, 20  
`without_internet()`, 3, 4, 9, 12, 17, 21, 22