Package 'httptest'

August 5, 2017

Type Package

Title A Test Environment for HTTP Requests

Description Testing code and packages that communicate with remote servers can be painful. Dealing with authentication, bootstrapping server state, cleaning up objects that may get created during the test run, network flakiness, 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 test contexts that mock the network connection in different ways, and it offers additional expectations to assert that HTTP requests were--or were not--made. Using these tools, one can test that code is making the intended requests and that it handles the expected responses correctly, all without depending on a connection to a remote API.

Version 2.1.2

URL https://github.com/nealrichardson/httptest

BugReports https://github.com/nealrichardson/httptest/issues

License MIT + file LICENSE

Depends R (>= 3.0.0), testthat

Imports digest, httr, jsonlite

Suggests knitr

RoxygenNote 6.0.0

VignetteBuilder knitr

NeedsCompilation no

Author Neal Richardson [aut, cre]

Maintainer Neal Richardson <neal.p.richardson@gmail.com>

Repository CRAN

Date/Publication 2017-08-04 23:06:41 UTC

R topics documented:

.mockPaths . . . 2
buildMockURL . . . 3
capture_requests . . . 4
expect-verb . . . 5
expect_header . . . 6
expect_json_equivalent . . . 7
fakeResponse . . . 8
httptest . . . 9
public . . . 9
skip_if_disconnected . . . 10
without_internet . . . 11
with_fake_HTTP . . . 12
with_mock_API . . . 13
with_trace . . . 13

Index **15**

`.mockPaths` *Set an alternate directory or directories 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. 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.

Usage

```
.mockPaths(new)
```

Arguments

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

Details

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 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(), ".")
```

`buildMockURL` *Convert a request to a mock file path*

Description

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

Usage

```
buildMockURL(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 URL is modified in two ways in order to allow it to map to a local file system. All mock files have the request protocol such as "http://" removed from the URL, and they also have a file extension appended. 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".

The extension also gives information on content type. Two extensions are currently supported: (1) .json and (2) .R. JSON mocks can be stored in .json files, and when they are loaded by `with_mock_API()`, relevant request metadata (headers, status code, etc.) are inferred. If your API doesn't return JSON, or if you want to simulate requests with other behavior (201 Location response, or 400 Bad Request, for example), you can store full `response` objects in .R files that `with_mock_API` will `source` to load. Any request can be stored as a .R mock, but the .json mocks offer a simplified, more readable alternative. (`capture_requests()` will record simplified .json files where appropriate and .R mocks otherwise by default.)

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. For example, `GET("api/object1/?a=1")` reads "api/object1-b64371.json". 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. For example, `POST("api/object1/?a=1")` reads "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.

Value

A file path and name, with .json extension. The file may or may not exist: existence is not a concern of this function.

See Also

`with_mock_API()` `capture_requests()`

`capture_requests` *Collect API Responses as Mock Files*

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

Usage

```
capture_requests(expr, path = .mockPaths()[1], simplify = TRUE,
  verbose = FALSE)

start_capturing(path = .mockPaths()[1], simplify = TRUE, verbose = FALSE)

stop_capturing()
```

Arguments

`expr`	Code to run inside the context
`path`	Where to save the mock files. Default is the first directory in `.mockPaths()`, which if not otherwise specified is the current working directory.
`simplify`	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 `simplify` is `FALSE`, the "response" object will be written out to a .R file using `base::dput()`.
`verbose`	logical: if `TRUE`, a `message` is printed for every file that is written when capturing requests containing the absolute path of the file. Useful for debugging if you're capturing but don't see the fixture files being written in the expected location. Default is `FALSE`.

Details

Mocks stored by this context are written out as plain-text files, either with extension `.json` if the request returned JSON content or with extension `.R` otherwise. The `.R` files contain syntax that when executed recreates the `httr` "response" object. 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.

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.

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"))
    utils::download.file("http://httpbin.org/gzip", tempfile())
})
# Or:
start_capturing()
GET("http://httpbin.org/get")
GET("http://httpbin.org")
GET("http://httpbin.org/response-headers",
    query=list(`Content-Type`="application/json"))
utils::download.file("http://httpbin.org/gzip", tempfile())
stop_capturing()

## End(Not run)
```

`expect-verb` *Expecations 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".

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`

Value

A `testthat` 'expectation'.

Examples

```
library(httr)
without_internet({
    expect_GET(GET("http://httpbin.org/get"),
        "http://httpbin.org/get")
    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))
})
```

---

expect_header *Test that an HTTP request is made with a header*

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(...)
```

Arguments

...	Arguments passed to expect_warning

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()`

`fakeResponse` *Return something that looks enough like an httr '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

```
fakeResponse(url = "", verb = "GET", status_code = 200,
  headers = list(), content = NULL)

fakeDownload(url, destfile, ...)
```

Arguments

`url`	A character URL for the request. For `fakeDownload`, this should be a path to a file that exists.
`verb`	Character name for the HTTP verb. 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.
`destfile`	For `fakeDownload`, character file path to "download" to. `fakeDownload` will copy the file at `url` to this path.
`...`	Additional arguments for `fakeDownload`

Value

The fake verbs return a 'httr' response class object. `fakeDownload` returns 0, the success code returned by `utils::download.file()`.

`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.

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`.

Using these tools, you can test that code is making the intended requests and that it handles the expected responses correctly, all without depending on a connection to a remote API during the test run.

`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).

`skip_if_disconnected` *Skip tests that need an internet connection if you don't have one*

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()`

`without_internet` *Make all HTTP requests raise an error*

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`, or `utils::download.file()`, will raise an error. The error message raised has a well-defined shape, made of three elements, separated by space: (1) the request method (e.g. "GET", or for downloading, "DOWNLOAD"); (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

```
without_internet(expr)
```

Arguments

`expr`	Code to run inside the mock context

Value

The result of `expr`

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` *Make all HTTP requests return a fake 'response' object*

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 for downloading, "DOWNLOAD"); (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 *Serve a mock API from files*

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 `buildMockURL()` 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

`buildMockURL()` `.mockPaths()`

with_trace *Wrapper around 'trace' to untrace when finished*

Description

Wrapper around 'trace' to untrace when finished

Usage

```
with_trace(x, where = topenv(parent.frame()), print = FALSE, ..., expr)
```

Arguments

x	Name of function to trace. See base::trace().
where	where to look for the function to be traced.
print	Logical: print a message every time the traced function is hit? Default is FALSE; note that in trace, the default is TRUE.
...	Additional arguments to pass to trace. At minimum, must include either tracer and at, or exit.
expr	Code to run inside the context

Value

The result of expr

Index

.mockPaths, 2
.mockPaths(), *4*, *13*

base::.libPaths(), *2*
base::dput(), *4*
base::trace(), *14*
buildMockURL, 3
buildMockURL(), *13*

capture_requests, 4
capture_requests(), *3*, *4*, *9*

digest::digest(), *3*

expect-verb, 5
expect_DELETE *(*expect-verb*)*, 5
expect_DELETE(), *9*
expect_GET *(*expect-verb*)*, 5
expect_GET(), *9*, *11*
expect_header, 6
expect_header(), *9*
expect_json_equivalent, 7
expect_json_equivalent(), *9*
expect_no_request *(*expect-verb*)*, 5
expect_no_request(), *9*
expect_PATCH *(*expect-verb*)*, 5
expect_PATCH(), *9*
expect_POST *(*expect-verb*)*, 5
expect_POST(), *9*, *11*
expect_PUT *(*expect-verb*)*, 5
expect_PUT(), *9*

fakeDownload *(*fakeResponse*)*, 8
fakeResponse, 8

httptest, 9
httptest-package *(*httptest*)*, 9

public, 9

skip_if_disconnected, 10
start_capturing *(*capture_requests*)*, 4
stop_capturing *(*capture_requests*)*, 4

testthat::expect_equivalent(), *8*
testthat::expect_warning(), *6*
testthat::skip(), *10*, *11*

utils::download.file(), *8*, *11*

with_fake_HTTP, 12
with_fake_HTTP(), *9*, *11*
with_mock_API, 13
with_mock_API(), *3*, *4*, *9*
with_trace, 13
without_internet, 11
without_internet(), *9*, *11–13*