Package: reqres (via r-universe)

August 12, 2024

Type Package

Title Powerful Classes for HTTP Requests and Responses

Version 0.2.5.9000

Maintainer Thomas Lin Pedersen <thomasp85@gmail.com>

Description In order to facilitate parsing of http requests and
creating appropriate responses this package provides two
classes to handle a lot of the housekeeping involved in working
with http exchanges. The infrastructure builds upon the 'rook'
specification and is thus well suited to be combined with
'httpuv' based web servers.

License MIT + file LICENSE

Encoding UTF-8

Depends R (>= 2.10)

Imports R6, assertthat, stringi, urltools, tools, brotli, jsonlite,
xml2, webutils, utils

RoxygenNote 7.2.1

Roxygen list(markdown = TRUE)

Suggests fiery, testthat, covr

URL https://reqres.data-imaginist.com,
https://github.com/thomasp85/reqres#reqres

BugReports https://github.com/thomasp85/reqres/issues

Repository https://thomasp85.r-universe.dev

RemoteUrl https://github.com/thomasp85/reqres

RemoteRef HEAD

RemoteSha ad524e65e6e73d93d3cc805749789eb7a6df708b

Contents

---

default_formatters   *A list of default formatter mappings*

Description

This list matches the most normal mime types with their respective formatters using default arguments. For a no-frills request parsing this can be supplied directly to Response$format(). To add or modify to this list simply supply the additional parsers as second, third, etc, argument and they will overwrite or add depending on whether it specifies a mime type already present.

Usage

```
default_formatters
```

See Also

formatters for an overview of the build in formatters in reqres

Examples

```
## Not run:
res$format(default_formatters, 'text/plain' = format_plain(sep = ' '))

## End(Not run)
```

---

default_parsers *A list of default parser mappings*

Description

This list matches the most normal mime types with their respective parsers using default arguments. For a no-frills request parsing this can be supplied directly to `Request$parse()`. To add or modify to this list simply supply the additional parsers as second, third, etc, argument and they will overwrite or add depending on whether it specifies a mime type already present.

Usage

```
default_parsers
```

See Also

[parsers](#) for an overview of the build in parsers in `reqres`

Examples

```
## Not run:
req$parse(default_parsers, 'application/json' = parse_json(flatten = TRUE))

## End(Not run)
```

---

formatters *Pre-supplied formatting generators*

Description

This set of functions can be used to construct formatting functions adhering to the Response$format() requirements.

Usage

```
format_json(
  dataframe = "rows",
  matrix = "rowmajor",
  Date = "ISO8601",
  POSIXt = "string",
  factor = "string",
  complex = "string",
  raw = "base64",
  null = "list",
  na = "null",
```

```
    auto_unbox = FALSE,
    digits = 4,
    pretty = FALSE,
    force = FALSE
)

format_plain(sep = "\n")

format_xml(encoding = "UTF-8", options = "as_xml")

format_html(encoding = "UTF-8", options = "as_html")

format_table(...)
```

Arguments

dataframe   how to encode data.frame objects: must be one of 'rows', 'columns' or 'values'

matrix   how to encode matrices and higher dimensional arrays: must be one of 'rowma-
              jor' or 'columnmajor'.

Date   how to encode Date objects: must be one of 'ISO8601' or 'epoch'

POSIXt   how to encode POSIXt (datetime) objects: must be one of 'string', 'ISO8601',
              'epoch' or 'mongo'

factor   how to encode factor objects: must be one of 'string' or 'integer'

complex   how to encode complex numbers: must be one of 'string' or 'list'

raw   how to encode raw objects: must be one of 'base64', 'hex' or 'mongo'

null   how to encode NULL values within a list: must be one of 'null' or 'list'

na   how to print NA values: must be one of 'null' or 'string'. Defaults are class
              specific

auto_unbox   automatically [unbox()](#) all atomic vectors of length 1. It is usually safer to avoid
              this and instead use the [unbox()](#) function to unbox individual elements. An
              exception is that objects of class AsIs (i.e. wrapped in [I()](#)) are not automatically
              unboxed. This is a way to mark single values as length-1 arrays.

digits   max number of decimal digits to print for numeric values. Use [I()](#) to specify
              significant digits. Use NA for max precision.

pretty   adds indentation whitespace to JSON output. Can be TRUE/FALSE or a number
              specifying the number of spaces to indent. See [prettify()](#)

force   unclass/skip objects of classes with no defined JSON mapping

sep   The line separator. Plain text will be split into multiple strings based on this.

encoding   The character encoding to use in the document. The default encoding is 'UTF-
              8'. Available encodings are specified at http://xmlsoft.org/html/libxml-encoding.html#xmlCharEncoding.

options   default: 'format'. Zero or more of

              format Format output
              no_declaration Drop the XML declaration

> > **no_empty_tags** Remove empty tags
> >
> > **no_xhtml** Disable XHTML1 rules
> >
> > **require_xhtml** Force XHTML rules
> >
> > **as_xml** Force XML output
> >
> > **as_html** Force HTML output
> >
> > **format_whitespace** Format with non-significant whitespace
>
> ...   parameters passed on to write.table()

Value

A function accepting an R object

See Also

parsers for converting `Request` bodies into R objects

default_formatters for a list that maps the most common mime types to their respective formatters

Examples

```
fake_rook <- fiery::fake_request(
  'http://example.com/test',
  content = '',
  headers = list(
    Content_Type = 'text/plain',
    Accept = 'application/json, text/csv'
  )
)

req <- Request$new(fake_rook)
res <- req$respond()
res$body <- mtcars
res$format(json = format_json(), csv = format_table(sep=','))
res$body

# Cleaning up connections
rm(fake_rook, req, res)
gc()
```

---

parsers   *Pre-supplied parsing generators*

Description

This set of functions can be used to construct parsing functions adhering to the Request$parse() requirements.

Usage

```
parse_json(
  simplifyVector = TRUE,
  simplifyDataFrame = simplifyVector,
  simplifyMatrix = simplifyVector,
  flatten = FALSE
)

parse_plain(sep = "\n")

parse_xml(encoding = "", options = "NOBLANKS", base_url = "")

parse_html(
  encoding = "",
  options = c("RECOVER", "NOERROR", "NOBLANKS"),
  base_url = ""
)

parse_multiform()

parse_queryform()

parse_table(...)
```

Arguments

simplifyVector	coerce JSON arrays containing only primitives into an atomic vector
simplifyDataFrame	
	coerce JSON arrays containing only records (JSON objects) into a data frame
simplifyMatrix	coerce JSON arrays containing vectors of equal mode and dimension into matrix or array
flatten	automatically flatten() nested data frames into a single non-nested data frame
sep	The line separator. Plain text will be split into multiple strings based on this.
encoding	Specify a default encoding for the document. Unless otherwise specified XML documents are assumed to be in UTF-8 or UTF-16. If the document is not UTF-8/16, and lacks an explicit encoding directive, this allows you to supply a default.
options	Set parsing options for the libxml2 parser. Zero or more of

> **RECOVER** recover on errors
>
> **NOENT** substitute entities
>
> **DTDLOAD** load the external subset
>
> **DTDATTR** default DTD attributes
>
> **DTDVALID** validate with the DTD
>
> **NOERROR** suppress error reports
>
> **NOWARNING** suppress warning reports

> **PEDANTIC** pedantic error reporting
>
> **NOBLANKS** remove blank nodes
>
> **SAX1** use the SAX1 interface internally
>
> **XINCLUDE** Implement XInclude substitition
>
> **NONET** Forbid network access
>
> **NODICT** Do not reuse the context dictionary
>
> **NSCLEAN** remove redundant namespaces declarations
>
> **NOCDATA** merge CDATA as text nodes
>
> **NOXINCNODE** do not generate XINCLUDE START/END nodes
>
> **COMPACT** compact small text nodes; no modification of the tree allowed afterwards (will possibly crash if you try to modify the tree)
>
> **OLD10** parse using XML-1.0 before update 5
>
> **NOBASEFIX** do not fixup XINCLUDE xml:base uris
>
> **HUGE** relax any hardcoded limit from the parser
>
> **OLDSAX** parse using SAX2 interface before 2.7.0
>
> **IGNORE_ENC** ignore internal document encoding hint
>
> **BIG_LINES** Store big lines numbers in text PSVI field

base_url   When loading from a connection, raw vector or literal html/xml, this allows you to specify a base url for the document. Base urls are used to turn relative urls into absolute urls.

...   parameters passed on to read.table()

Value

A function accepting a raw vector and a named list of directives

See Also

formatters for converting `Response` bodies into compatible types

default_parsers for a list that maps the most common mime types to their respective parsers

Examples

```
fake_rook <- fiery::fake_request(
  'http://example.com/test',
  content = '[1, 2, 3, 4]',
  headers = list(
    Content_Type = 'application/json'
  )
)

req <- Request$new(fake_rook)
req$parse(json = parse_json())
req$body

# Cleaning up connections
rm(fake_rook, req)
```

```
gc()
```

Request   *HTTP Request Handling*

Description

This class wraps all functionality related to extracting information from a http request. Much of the functionality is inspired by the Request class in Express.js, so the documentation for this will complement this document. As reqres is build on top of the Rook specifications the Request object is initialized from a Rook-compliant object. This will often be the request object provided by the httpuv framework. While it shouldn't be needed, the original Rook object is always accessible and can be modified, though any modifications will not propagate to derived values in the Request object (e.g. changing the HTTP_HOST element of the Rook object will not change the host field of the Request object). Because of this, direct manipulation of the Rook object is generally discouraged.

Usage

```
as.Request(x, ...)

is.Request(x)
```

Arguments

x	An object coercible to a Request.
...	Parameters passed on to Request$new()

Value

A Request object (for as.Request()) or a logical indicating whether the object is a Request (for is.Request())

Initialization

A new 'Request'-object is initialized using the new() method on the generator:

Usage

```
req <- Request$new(rook, trust = FALSE)
```

Arguments

rook	The rook request that the new object should wrap
trust	Is this request trusted blindly. If TRUE X-Forwarded-* headers will be returned when querying host, ip, and protoc

Fields

The following fields are accessible in a `Request` object:

`trust` A logical indicating whether the request is trusted. *Mutable*

`method` A string indicating the request method (in lower case, e.g. 'get', 'put', etc.). *Immutable*

`body` An object holding the body of the request. This is an empty string by default and needs to be populated using the `set_body()` method (this is often done using a body parser that accesses the Rook$input stream). *Immutable*

`cookies` Access a named list of all cookies in the request. These have been URI decoded. *Immutable*

`headers` Access a named list of all headers in the request. In order to follow R variable naming standards - have been substituted with _. Use the `get_header()` method to lookup based on the correct header name. *Immutable*

`host` Return the domain of the server given by the "Host" header if `trust == FALSE`. If `trust == true` returns the `X-Forwarded-Host` instead.

`ip` Returns the remote address of the request if `trust == FALSE`. if `trust == TRUE` it will instead return the first value of the `X-Forwarded-For` header. *Immutable*

`ips` If `trust == TRUE` it will return the full list of ips in the `X-Forwarded-For` header. If `trust == FALSE` it will return an empty vector. *Immutable*

`protocol` Returns the protocol (e.g. 'http') used for the request. If `trust == TRUE` it will use the value of the `X-Forwarded-Proto` header. *Immutable*

`root` The mount point of the application receiving this request. Can be empty if the application is mounted on the server root. *Immutable*

`path` The part of the url following the root. Defines the local target of the request (independent of where it is mounted). *Immutable*

`url` The full URL of the request. *Immutable*

`query` The query string of the request (anything following "?" in the URL) parsed into a named list. The query has been url decoded and "+" has been substituted with space. Multiple queries are expected to be separated by either "&" or "|". *Immutable*

`querystring` The unparsed query string of the request, including "?". If no query string exists it will be `""` rather than `"?"`

`xhr` A logical indicating whether the `X-Requested-With` header equals `XMLHttpRequest` thus indicating that the request was performed using a JavaScript library such as jQuery. *Immutable*

`secure` A logical indicating whether the request was performed using a secure connection, i.e. `protocol == 'https'`. *Immutable*

`origin` The original object used to create the `Request` object. As reqres currently only works with rook this will always return the original rook object. *Immutable*, though the content of the rook object itself might be manipulated as it is an environment.

`response` If a `Response` object has been created for this request it is accessible through this field. *Immutable*

Methods

The following methods are available in a `Request` object:

`set_body(content)` Sets the content of the request body. This method should mainly be used in concert with a body parser that reads the `rook$input` stream

`set_cookies(cookies)` Sets the cookies of the request. The cookies are automatically parsed and populated, so this method is mainly available to facilitate cookie signing and encryption

`get_header(name)` Get the header of the specified name.

`accepts(types)` Given a vector of response content types it returns the preferred one based on the `Accept` header.

`accepts_charsets(charsets)` Given a vector of possible character encodings it returns the preferred one based on the `Accept-Charset` header.

`accepts_encoding(encoding)` Given a vector of possible content encodings (usually compression algorithms) it selects the preferred one based on the `Accept-Encoding` header. If there is no match it will return `"identity"` signaling no compression.

`accepts_language(language)` Given a vector of possible content languages it selects the best one based on the `Accept-Language` header.

`is(type)` Queries whether the body of the request is in a given format by looking at the `Content-Type` header. Used for selecting the best parsing method.

`respond()` Creates a new `Response` object from the request

`parse(..., autofail = TRUE)` Based on provided parsers it selects the appropriate one by looking at the `Content-Type` header and assigns the result to the request body. A parser is a function accepting a raw vector, and a named list of additional directives, and returns an R object of any kind (if the parser knows the input to be plain text, simply wrap it in [rawToChar()]). If the body is compressed, it will be decompressed based on the `Content-Encoding` header prior to passing it on to the parser. See parsers for a list of pre-supplied parsers. Parsers are either supplied in a named list or as named arguments to the parse method. The names should correspond to mime types or known file extensions. If `autofail = TRUE` the response will be set with the correct error code if parsing fails. `parse()` returns `TRUE` if parsing was successful and `FALSE` if not

`parse_raw(autofail = TRUE)` This is a simpler version of the `parse()` method. It will attempt to decompress the body and set the body field to the resulting raw vector. It is then up to the server to decide how to handle the payload. It returns `TRUE` if successful and `FALSE` otherwise.

`as_message()` Prints a HTTP representation of the request to the output stream.

Methods

Public methods:

- [Request$new()]
- [Request$print()]
- [Request$set_body()]
- [Request$set_cookies()]
- [Request$accepts()]
- [Request$accepts_charsets()]

- [Request$accepts_encoding()](#)
- [Request$accepts_language()](#)
- [Request$is()](#)
- [Request$get_header()](#)
- [Request$respond()](#)
- [Request$parse()](#)
- [Request$parse_raw()](#)
- [Request$as_message()](#)
- [Request$clone()](#)

Method new():

Usage:

Request$new(rook, trust = FALSE)

Method print():

Usage:

Request$print(...)

Method set_body():

Usage:

Request$set_body(content)

Method set_cookies():

Usage:

Request$set_cookies(cookies)

Method accepts():

Usage:

Request$accepts(types)

Method accepts_charsets():

Usage:

Request$accepts_charsets(charsets)

Method accepts_encoding():

Usage:

Request$accepts_encoding(encoding)

Method accepts_language():

Usage:

Request$accepts_language(language)

Method is():

Usage:

Request$is(type)

Method `get_header()`:

Usage:

`Request$get_header(name)`

Method `respond()`:

Usage:

`Request$respond()`

Method `parse()`:

Usage:

`Request$parse(..., autofail = TRUE)`

Method `parse_raw()`:

Usage:

`Request$parse_raw(autofail = TRUE)`

Method `as_message()`:

Usage:

`Request$as_message()`

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

Usage:

`Request$clone(deep = FALSE)`

Arguments:

deep Whether to make a deep clone.

See Also

[Response](#) for handling http responses

Examples

```
fake_rook <- fiery::fake_request(
  'http://example.com/test?id=34632&question=who+is+hadley',
  content = 'This is an elaborate ruse',
  headers = list(
    Accept = 'application/json; text/*',
    Content_Type = 'text/plain'
  )
)

req <- Request$new(fake_rook)

# Get full URL
req$url

# Get list of query parameters
req$query
```

```
# Test if content is text
req$is('txt')

# Perform content negotiation for the response
req$accepts(c('html', 'json', 'txt'))

# Cleaning up connections
rm(fake_rook, req)
gc()
```

Response   *HTTP Response handling*

Description

This class handles all functionality involved in crafting a http response. Much of the functionality is inspired by the Request class in Express.js, so the documentation for this will complement this document. As reqres is build on top of the Rook specifications the Response object can be converted to a compliant list object to be passed on to e.g. the httpuv handler.

Usage

```
## S3 method for class 'Response'
as.list(x, ...)

is.Response(x)
```

Arguments

x	A Response object
...	Ignored

Details

A Response object is always created as a response to a Request object and contains a reference to the originating Request object. A Response is always initialized with a 404 Not Found code, an empty string as body and the Content-Type header set to text/plain. As the Content-Type header is required for httpuv to function, it will be inferred if missing when converting to a list. If the body is a raw vector it will be set to application/octet-stream and otherwise it will be set to text/plain. It is always advised to consciously set the Content-Type header though. The only exception is when attaching a standard file where the type is inferred from the file extension automatically. Unless the body is a raw vector it will automatically be converted to a character vector and collapsed to a single string with "\n" separating the individual elements before the Response object is converted to a list (that is, the body can exist as any type of object up until the moment where the Response object is converted to a list). To facilitate communication between different middleware the Response object contains a data store where information can be stored during the lifetime of the response.

Value

A rook-compliant list-response (in case of `as.list()`) or a logical indicating whether the object is a `Response` (in case of `is.Response()`)

Initialization

A new 'Response'-object is initialized using the `new()` method on the generator:

Usage

```
res <- Response$new(request)
```

But often it will be provided by the request using the `respond()` method, which will provide the response, creating one if it doesn't exist

Usage

```
res <- request$respond()
```

Arguments

request   The `Request` object that the `Response` is responding to

Fields

The following fields are accessible in a `Response` object:

status Gets or sets the status code of the response. Is initialised with `404L`

body Set or get he body of the response. If it is a character vector with a single element named `'file'` it will be interpreted as the location of a file. It is better to use the `file` field for creating a response referencing a file as it will automatically set the correct headers.

file Set or get the location of a file that should be used as the body of the response. If the body is not referencing a file (but contains something else) it will return NULL. The `Content-Type` header will automatically be inferred from the file extension, if known. If unknown it will defaults to `application/octet-stream`. If the file has no extension it will be `text/plain`. Existence of the file will be checked.

type Get or sets the `Content-Type` header of the response based on a file extension or mime-type.

request Get the original `Request` object that the object is responding to.

Methods

The following methods are available in a `Response` object:

set_header(name, value) Sets the header given by `name`. `value` will be converted to character. A header will be added for each element in `value`. Use `append_header()` for setting headers without overwriting existing ones.

get_header(name) Returns the header(s) given by `name`

remove_header(name) Removes all headers given by name

has_header(name) Test for the existence of any header given by name

append_header(name, value) Adds an additional header given by name with the value given by value. If the header does not exist yet it will be created.

set_data(key, value) Adds value to the internal data store and stores it with key

get_data(key) Retrieves the data stored under key in the internal data store.

remove_data(key) Removes the data stored under key in the internal data store.

has_data(key) Queries whether the data store has an entry given by key

attach(file, filename=basename(file), type=NULL) Sets the body to the file given by file and marks the response as a download by setting the Content-Disposition to attachment; filename=<filename>. Use the type argument to overwrite the automatic type inference from the file extension.

status_with_text(code) Sets the status to code and sets the body to the associated status code description (e.g. Bad Gateway for 502L)

set_cookie(name, value, encode = TRUE, expires = NULL, http_only = NULL, max_age = NULL, path = NULL, secure Adds the cookie given by name to the given value, optionally url encoding it, along with any additional directives. See https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie for a description of the different directives. If the cookie already exists it will be overwritten. The validity of the directives will automatically be checked. expires expects a POSIXct object, http_only and secure expect a logical, max_age expects an integer, path a string, and same_site either "Lax" or "Strict"

remove_cookie(name) Removes the cookie named name from the response.

has_cookie(name) Queries whether the response contains a cookie named name

set_links(...) Sets the Link header based on the named arguments passed to .... The names will be used for the rel directive.

format(..., autofail = TRUE, compress = TRUE) Based on the formatters passed in through ... content negotiation is performed with request and the preferred formatter is chosen. The Content-Type header is set automatically. If compress = TRUE the compress() method will be called after formatting. If an error is encountered and autofail = TRUE the response will be set to 500. If a formatter is not found and autofail = TRUE the response will be set to 406. If formatting is successful it will return TRUE, if not it will return FALSE

compress(priority = c('gzip', 'deflate', 'br', 'identity')) Based on the provided priority, an encoding is negotiated with the request and applied. The Content-Encoding header is set to the chosen compression algorithm.

content_length() Calculates the length (in bytes) of the body. This is the number that goes into the Content-Length header. Note that the Content-Length header is set automatically by httpuv so this method should only be called if the response size is needed for other reasons.

as_list() Converts the object to a list for further processing by a Rook compliant server such as httpuv. Will set Content-Type header if missing and convert a non-raw body to a single character string.

Methods

Public methods:

- Response$new()

- Response$print()
- Response$set_header()
- Response$get_header()
- Response$remove_header()
- Response$has_header()
- Response$append_header()
- Response$set_data()
- Response$get_data()
- Response$remove_data()
- Response$has_data()
- Response$timestamp()
- Response$attach()
- Response$status_with_text()
- Response$set_cookie()
- Response$remove_cookie()
- Response$has_cookie()
- Response$set_links()
- Response$format()
- Response$compress()
- Response$content_length()
- Response$as_list()
- Response$as_message()
- Response$clone()

Method new():

Usage:
Response$new(request)

Method print():

Usage:
Response$print(...)

Method set_header():

Usage:
Response$set_header(name, value)

Method get_header():

Usage:
Response$get_header(name)

Method remove_header():

Usage:
Response$remove_header(name)

Method `has_header()`:

Usage:

`Response$has_header(name)`

Method `append_header()`:

Usage:

`Response$append_header(name, value)`

Method `set_data()`:

Usage:

`Response$set_data(key, value)`

Method `get_data()`:

Usage:

`Response$get_data(key)`

Method `remove_data()`:

Usage:

`Response$remove_data(key)`

Method `has_data()`:

Usage:

`Response$has_data(key)`

Method `timestamp()`:

Usage:

`Response$timestamp()`

Method `attach()`:

Usage:

`Response$attach(file, filename = basename(file), type = NULL)`

Method `status_with_text()`:

Usage:

`Response$status_with_text(code)`

Method `set_cookie()`:

Usage:

```
Response$set_cookie(
  name,
  value,
  encode = TRUE,
  expires = NULL,
  http_only = NULL,
  max_age = NULL,
  path = NULL,
  secure = NULL,
  same_site = NULL
)
```

Method `remove_cookie()`:

Usage:

`Response$remove_cookie(name)`

Method `has_cookie()`:

Usage:

`Response$has_cookie(name)`

Method `set_links()`:

Usage:

`Response$set_links(...)`

Method `format()`:

Usage:

`Response$format(..., autofail = TRUE, compress = TRUE)`

Method `compress()`:

Usage:

`Response$compress(priority = c("gzip", "deflate", "br", "identity"))`

Method `content_length()`:

Usage:

`Response$content_length()`

Method `as_list()`:

Usage:

`Response$as_list()`

Method `as_message()`:

Usage:

`Response$as_message()`

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

Usage:

`Response$clone(deep = FALSE)`

Arguments:

deep Whether to make a deep clone.

See Also

[Request](#) for handling http requests

Examples

```
fake_rook <- fiery::fake_request(
  'http://example.com/test?id=34632&question=who+is+hadley',
  content = 'This is elaborate ruse',
  headers = list(
    Accept = 'application/json; text/*',
    Content_Type = 'text/plain'
  )
)

req <- Request$new(fake_rook)
res <- Response$new(req)
res

# Set the body to the associated status text
res$status_with_text(200L)
res$body

# Infer Content-Type from file extension
res$type <- 'json'
res$type

# Prepare a file for download
res$attach(system.file('DESCRIPTION', package = 'reqres'))
res$type
res$body
res$get_header('Content-Disposition')

# Cleaning up connections
rm(fake_rook, req, res)
gc()
```

---

to_http_date	*Format timestamps to match the HTTP specs*

Description

Dates/times in HTTP headers needs a specific format to be valid, and is furthermore always given in GMT time. These two functions aids in converting back and forth between the required format.

Usage

```
to_http_date(time, format = NULL)

from_http_date(time)
```

Arguments

`time`	A string or an object coercible to POSIXct
`format`	In case `time` is not a POSIXct object a specification how the string should be interpreted.

Value

`to_http_date()` returns a properly formatted string, while `from_http_date()` returns a POSIXct object

Examples

```
time <- to_http_date(Sys.time())
time
from_http_date(time)
```

Index