Package 'reqres'

May 9, 2026

Type Package

Title Powerful Classes for HTTP Requests and Responses

Version 1.2.0

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 (>= 3.5)

Imports R6, stringi, urltools, tools, brotli, jsonlite, xml2, webutils, utils, cli, rlang, lifecycle, base64enc, sodium, promises, mirai, otel

RoxygenNote 7.3.3

Suggests fiery, testthat (>= 3.0.0), covr, keyring, shiny

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

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

Config/testthat/edition 3

Config/build/compilation-database true

Collate 'aaa.R' 'formatters.R' 'import-standalone-obj-type.R' 'import-standalone-types-check.R' 'mock_request.R' 'otel.R' 'parsers.R' 'problems.R' 'reqres-package.R' 'request.R' 'response.R' 'zzz.R'

NeedsCompilation yes

Author Thomas Lin Pedersen [cre, aut] (ORCID: <https://orcid.org/0000-0002-5147-4711>)

Maintainer Thomas Lin Pedersen <thomasp85@gmail.com>

Repository CRAN

Date/Publication 2025-12-11 06:10:13 UTC

Contents

`abort_http_problem` *Abort request processing with an HTTP problem response*

Description

This set of functions throws a classed error indicating that the request should be responded to with an HTTP problem according to the spec defined in RFC 9457 or a bare response code. These conditions should be caught and handled by the `handle_problem()` function.

Usage

```
abort_http_problem(
  code,
  detail,
  title = NULL,
  type = NULL,
  instance = NULL,
  ...,
  message = detail,
  call = caller_env()
)

abort_status(code, message = status_phrase(code), ..., call = caller_env())

abort_bad_request(
  detail,
  instance = NULL,
  ...,
  message = detail,
  call = caller_env()
)
```

```
abort_unauthorized(
  detail,
  instance = NULL,
  ...,
  message = detail,
  call = caller_env()
)

abort_forbidden(
  detail,
  instance = NULL,
  ...,
  message = detail,
  call = caller_env()
)

abort_not_found(
  detail,
  instance = NULL,
  ...,
  message = detail,
  call = caller_env()
)

abort_method_not_allowed(
  detail,
  instance = NULL,
  ...,
  message = detail,
  call = caller_env()
)

abort_not_acceptable(
  detail,
  instance = NULL,
  ...,
  message = detail,
  call = caller_env()
)

abort_conflict(
  detail,
  instance = NULL,
  ...,
  message = detail,
  call = caller_env()
)
```

```
abort_gone(detail, instance = NULL, ..., message = detail, call = caller_env())

abort_internal_error(
  detail,
  instance = NULL,
  ...,
  message = detail,
  call = caller_env()
)

handle_problem(response, cnd)

is_reqres_problem(cnd)
```

Arguments

code	The HTTP status code to use
detail	A string detailing the problem. Make sure the information given does not pose a security risk
title	A human-readable title of the issue. Should not vary from instance to instance of the specific issue. If NULL then the status code title is used
type	A URI that uniquely identifies this type of problem. The URI must resolve to an HTTP document describing the problem in human readable text. If NULL, the most recent link to the given status code definition is used
instance	A unique identifier of the specific instance of this problem that can be used for further debugging. Can be omitted.
...	Arguments passed on to rlang::error_cnd class The condition subclass. use_cli_format Whether to use the cli package to format message. See local_use_cli(). trace A trace object created by trace_back(). parent A parent condition object.
message	A default message to inform the user about the condition when it is signalled.
call	A function call to be included in the error message. If an execution environment of a running function, the corresponding function call is retrieved.
response	The Response object associated with the request that created the condition
cnd	The thrown condition

---

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(root_name = "document", 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 'rowmajor' 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 (default is 2). Use a negative number for tabs instead of spaces.
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.
root_name	The name of the root element of the created xml
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()
```

`mock_request` *Create a mock request to use in testing*

Description

This function creates a new `Request` for a specific resource defined by a URL.

Usage

```
mock_request(
  url,
  method = "get",
  content = "",
  headers = list(),
  app_location = "",
  remote_address = "123.123.123.123"
)
```

Arguments

`url`	A complete url for the resource the request should ask for, including querystring if needed
`method`	The request type (get, post, put, etc). Defaults to `"get"`
`content`	The content of the request, either a raw vector or a string
`headers`	A list of name-value pairs that defines the request headers
`app_location`	A string giving the first part of the url path that should be stripped from the path
`remote_address`	The IP address of the presumed sender

Value

A `Request` object

Examples

```
req <- mock_request(
    'http://www.my-fake-website.com/path/to/a/query/?key=value&key2=value2',
    content = 'Some important content'
)

# Get the main address of the URL
req$host

# Get the query string
req$query

# ... etc.

# Cleaning up connections
rm(req)
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(delim = NULL)

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 substitution

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.
delim	The delimiter to use for parsing arrays in non-exploded form. Either NULL (no delimiter) or one of "," , "|", or " "
...	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()
```

---

query_parser *Parse a query string*

Description

This function facilitates the parsing of querystrings, either from the URL or a POST or PUT body with Content-Type set to application/x-www-form-urlencoded.

Usage

```
query_parser(query = NULL, delim = NULL)
```

Arguments

query	The query as a single string
delim	Optional delimiter of array values. If omitted it is expected that arrays are provided in exploded form (e.g. arg1=3&arg1=7)

Value

A named list giving the keys and values of the query. Values fron the same key are combined if given multiple times

Examples

```
# Using delimiter to provide array
query_parser("?name=Thomas+Lin+Pedersen&numbers=1%202%203", delim = " ")

# No delimiter (exploded form)
query_parser("?name=Thomas%20Lin%20Pedersen&numbers=1&numbers=2&numbers=3")
```

---

random_key *Generate a random key compatible with encryption and decryption in requests and responses*

Description

The encryption/decryption used in reqres is based on the sodium package and requires a 32-bit encryption key encoded as hexadecimal values. While you can craft your own, this function will take care of creating a compliant key using a cryptographically secure pseudorandom number generator from sodium::helpers().

Usage

```
random_key()
```

Details

Keep your encryption keys safe! Anyone with the key will be able to eavesdrop on your communication and tamper with the information stored in encrypted cookies through man-in-the-middle attacks. The best approach is to use the keyring package to manage your keys, but as an alternative you can store it as environment variables.

NEVER STORE THE KEY IN PLAIN TEXT.

NEVER PUT THE KEY SOMEWHERE WHERE IT CAN ACCIDENTALLY BE COMMITTED TO GIT OR OTHER VERSION CONTROL SOFTWARE

Value

A 32-bit key as a hex-encoded string

Examples

```
# Store a key with keyring and use it
keyring::key_set_with_value("reqres_key", random_key())

rook <- fiery::fake_request("http://example.com")

Request$new(rook, key = keyring::key_get("reqres_key"))
```

`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)
```

Active bindings

`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*

`body_raw` The raw content of the request body as a raw vector. No unpacking or parsing has been performed on this, even if the request has been parsed.

`session` The content of the session cookie. If session cookies has not been activated it will be an empty write-protected list. If session cookies are activated but the request did not contain one it will be an empty list. The content of this field will be send encrypted as part of the response according to the cookie settings in `$session_cookie_settings`. This field is reflected in the `Response$session` field and using either produces the same result

`has_session_cookie` Query whether the request came with a session cookie *Immutable*

`session_cookie_settings` Get the settings for the session cookie as they were provided during initialisation cookie *Immutable*

`has_key` Query whether the request was initialised with an encryption key *Immutable*

`compression_limit` Query the compression limit the request was initialized with *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. *Immutable*

`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*

`query_delim` The delimiter used for specifying multiple values in a query. If `NULL` then queries are expected to contain multiple key-value pairs for the same key in order to provide an array, e.g. `?arg1=3&arg1=7`. If setting it to `","`, `""`, `"|"`, or `" "` then an array can be provided in a single key-value pair, e.g. `?arg1=3|7`

`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 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. Changing this will force the request to reparse itself.

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

`locked` Set the `locked` status on the request. This flag does not result in any different behaviour in the request but can be used by frameworks to signal that the request should not be altered in some way

`response_headers` The list of headers the response is prepopulated with *Immutable*

`otel_span` An OpenTelemetry span to use as parent for any instrumentation happening during the handling of the request. If otel is not enabled then this will be NULL. The span is populated according to the HTTP Server semantics https://opentelemetry.io/docs/specs/semconv/http/http-spans/#http-server, except for the `http.route` attribute, which must be set by the server implementation, along with a proper name for the span (`{method}_{route}`). The span is automatically closed when the response is converted to a list, unless asked not to. *Immutable*

`start_time` The time point the Request was created

`duration` The time passed since the request was created

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$has_header()`
- `Request$respond()`
- `Request$parse()`
- `Request$parse_raw()`
- `Request$as_message()`

- Request$encode_string()
- Request$decode_string()
- Request$clear()
- Request$forward()
- Request$clone()

Method `new()`**:** Create a new request from a rook object

Usage:

```
Request$new(
  rook,
  trust = FALSE,
  key = NULL,
  session_cookie = NULL,
  compression_limit = 0,
  query_delim = NULL,
  response_headers = list(),
  with_otel = TRUE
)
```

Arguments:

`rook` The rook object to base the request on

`trust` Is this request trusted blindly. If `TRUE` `X-Forwarded-*` headers will be returned when querying host, ip, and protocol

`key` A 32-bit secret key as a hex encoded string or a raw vector to use for `$encode_string()` and `$decode_string()` and by extension to encrypt a session cookie. It must be given to turn on session cookie support. A valid key can be generated using `random_key()`. NEVER STORE THE KEY IN PLAIN TEXT. Optimalle use the keyring package to store it or set it as an environment variable

`session_cookie` Settings for the session cookie created using `session_cookie()`. Will be ignored if `key` is not provided to ensure session cookies are properly encrypted

`compression_limit` The size threshold in bytes for trying to compress the response body (it is still dependant on content negotiation)

`query_delim` The delimiter to split array-type query arguments by

`response_headers` A list of headers the response should be prepopulated with. All names must be in lower case and all elements must be character vectors. This is not checked but assumed

`with_otel` A boolean to indicate if otel instrumentation should be initiated with the creation of this request. Set to `FALSE` to avoid a span being started as well as metrics being recorded for this request. If `TRUE` you should call `request$clear()` as the last act of your request handling to ensure that the span is closed and that the duration metric is correctly reported.

Method `print()`**:** Pretty printing of the object

Usage:

```
Request$print(...)
```

Arguments:

`...` ignored

Method `set_body()`**:** 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

Usage:

```
Request$set_body(content)
```

Arguments:

`content` An R object representing the body of the request

Method `set_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

Usage:

```
Request$set_cookies(cookies)
```

Arguments:

`cookies` A named list of cookie values

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

Usage:

```
Request$accepts(types)
```

Arguments:

`types` A vector of types

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

Usage:

```
Request$accepts_charsets(charsets)
```

Arguments:

`charsets` A vector of charsets

Method `accepts_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.

Usage:

```
Request$accepts_encoding(encoding)
```

Arguments:

`encoding` A vector of encoding names

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

Usage:

```
Request$accepts_language(language)
```

Arguments:

`language` A vector of languages

Method `is()`**:** 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.

Usage:

```
Request$is(type)
```

Arguments:

`type` A vector of content types to check for. Can be fully qualified MIME types, a file extension, or a mime type with wildcards

Method `get_header()`**:** Get the header of the specified name.

Usage:

```
Request$get_header(name)
```

Arguments:

`name` The name of the header to get

Method `has_header()`**:** Test for the existence of any header given by `name`

Usage:

```
Request$has_header(name)
```

Arguments:

`name` The name of the header to look for

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

Usage:

```
Request$respond()
```

Method `parse()`**:** 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 throw an appropriate abort code if failing to parse the body. `parse()` returns `TRUE` if parsing was successful and `FALSE` if not

Usage:

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

Arguments:

`...` A named set of parser functions

`autofail` Automatically populate the response if parsing fails

Method `parse_raw()`**:** 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.

Usage:

```
Request$parse_raw(autofail = TRUE)
```

Arguments:

`autofail` Automatically populate the response if parsing fails

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

Usage:

```
Request$as_message()
```

Method `encode_string()`**:** base64-encode a string. If a key has been provided during initialisation the string is first encrypted and the final result is a combination of the encrypted text and the nonce, both base64 encoded and combined with a `"_"`.

Usage:

```
Request$encode_string(val)
```

Arguments:

`val` A single string to encrypt

Method `decode_string()`**:** base64-decodes a string. If a key has been provided during initialisation the input is first split by `"_"` and then the two parts are base64 decoded and decrypted. Otherwise the input is base64-decoded as-is. It will always hold that `val == decode_string(encode_string(val))`.

Usage:

```
Request$decode_string(val)
```

Arguments:

`val` A single string to encrypt

Method `clear()`**:** Clears the content of the request and, if created, the related response. This method exists only to allow reuse of the request and response object to save a few milliseconds in latency. Use with caution and see e.g. how fiery maintains a poll of request objects

Usage:

```
Request$clear()
```

Method `forward()`**:** Forward a request to a new url, optionally setting different headers, queries, etc. Uses curl and mirai under the hood and returns a promise

Usage:

```
Request$forward(
  url,
  query = NULL,
  method = NULL,
  headers = NULL,
  body = NULL,
  return = NULL,
  ...
)
```

Arguments:

`url` The url to forward to

`query` Optional querystring to append to `url`. If `NULL` the query string of the current request will be used

`method` The HTTP method to use. If `NULL` the method of the current request will be used

`headers` A list of headers to add to the headers of the current request. You can remove a header from the current request by setting it to `NULL` here

`body` The body to send with the forward. If `NULL` the body of the current request will be used

`return` A function that takes in the fulfilled response object and whose return value is returned by the promise

`...` ignored

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. 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.

Usage

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

is.Response(x)
```

Arguments

`x`	A `Response` object
`...`	Ignored

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.

Active bindings

`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.

`formatter` Get the registered formatter for the response body.

`is_formatted` Has the body been formatted

`data_store` Access the environment that holds the response data store

`session` The content of the session cookie. If session cookies has not been activated it will be an empty write-protected list. If session cookies are activated but the request did not contain one it will be an empty list. The content of this field will be send encrypted as part of the response according to the cookie settings in `$session_cookie_settings`. This field is reflected in the `Request$session` field and using either produces the same result

`session_cookie_settings` Get the settings for the session cookie as they were provided during initialisation of the request cookie *Immutable*

`has_key` Query whether the request was initialised with an encryption key *Immutable*

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$attach()`
- `Response$as_download()`
- `Response$status_with_text()`
- `Response$problem()`
- `Response$set_cookie()`
- `Response$remove_cookie()`
- `Response$clear_cookie()`
- `Response$has_cookie()`
- `Response$set_links()`
- `Response$format()`
- `Response$set_formatter()`
- `Response$compress()`
- `Response$content_length()`
- `Response$as_list()`
- `Response$as_message()`
- `Response$encode_string()`
- `Response$decode_string()`
- `Response$reset()`
- `Response$clone()`

Method `new()`**:** Create a new response from a Request object

Usage:

```
Response$new(request)
```

Arguments:

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

Method `print()`**:** Pretty printing of the object

Usage:

```
Response$print(...)
```

Arguments:

`...` ignored

Method `set_header()`**:** 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.

Usage:

```
Response$set_header(name, value)
```

Arguments:

`name` The name of the header to set

`value` The value to assign to the header

Method `get_header()`**:** Returns the header(s) given by `name`

Usage:

```
Response$get_header(name)
```

Arguments:

`name` The name of the header to retrieve the value for

Method `remove_header()`**:** Removes all headers given by `name`

Usage:

```
Response$remove_header(name)
```

Arguments:

`name` The name of the header to remove

Method `has_header()`**:** Test for the existence of any header given by `name`

Usage:

```
Response$has_header(name)
```

Arguments:

`name` The name of the header to look for

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

Usage:

```
Response$append_header(name, value)
```

Arguments:

`name` The name of the header to append to

`value` The value to assign to the header

Method `set_data()`**:** Adds `value` to the internal data store and stores it with `key`

Usage:

```
Response$set_data(key, value)
```

Arguments:

`key` The identifier of the data you set

`value` An R object

Method `get_data()`**:** Retrieves the data stored under `key` in the internal data store.

Usage:

```
Response$get_data(key)
```

Arguments:

`key` The identifier of the data you wish to retrieve

Method `remove_data()`**:** Removes the data stored under `key` in the internal data store.

Usage:

```
Response$remove_data(key)
```

Arguments:

`key` The identifier of the data you wish to remove

Method `has_data()`**:** Queries whether the data store has an entry given by `key`

Usage:

```
Response$has_data(key)
```

Arguments:

`key` The identifier of the data you wish to look for

Method `timestamp()`**:** Set the `Date` header to the current time

Usage:

```
Response$timestamp()
```

Method `attach()`**:** 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.

Usage:

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

Arguments:

`file` The path to a file

`filename` The name of the file as it will appear to the client

`type` The file type. If not given it will be inferred

Method `as_download()`**:** Marks the response as a downloadable file, rather than data to be shown in the browser

Usage:

```
Response$as_download(filename = NULL)
```

Arguments:

`filename` Optional filename as hint for the client

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

Usage:

```
Response$status_with_text(code, clear_headers = FALSE)
```

Arguments:

`code` The status code to set

`clear_headers` Should all currently set headers be cleared (useful for converting a response to an error halfway through processing)

Method `problem()`**:** Signals an API problem using the HTTP Problems spec RFC 9457. This should only be used in cases where returning a bare response code is insufficient to describe the issue.

Usage:

```
Response$problem(
  code,
  detail,
  title = NULL,
  type = NULL,
  instance = NULL,
  clear_headers = TRUE
)
```

Arguments:

`code` The HTTP status code to use

`detail` A string detailing the problem. Make sure the information given does not pose a security risk

`title` A human-readable title of the issue. Should not vary from instance to instance of the specific issue. If `NULL` then the status code title is used

`type` A URI that uniquely identifies this type of problem. The URI must resolve to an HTTP document describing the problem in human readable text. If `NULL`, the most recent link to the given status code definition is used

`instance` A unique identifier of the specific instance of this problem that can be used for further debugging. Can be omitted.

`clear_headers` Should all currently set headers be cleared

Method `set_cookie()`**:** Sets a cookie on the response. See `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie` for a longer description

Usage:

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

Arguments:

`name` The name of the cookie

`value` The value of the cookie

`encode` Should `value` be url encoded

`expires` A POSIXct object given the expiration time of the cookie

`http_only` Should the cookie only be readable by the browser

`max_age` The number of seconds to elapse before the cookie expires

`path` The URL path this cookie is related to

`secure` Should the cookie only be send over https

`same_site` Either `"Lax"`, `"Strict"`, or `"None"` indicating how the cookie can be send during cross-site requests. If this is set to `"None"` then `secure` *must* also be set to `TRUE`

Method `remove_cookie()`**:** Removes the cookie named `name` from the response.

Usage:

```
Response$remove_cookie(name)
```

Arguments:

`name` The name of the cookie to remove

Method `clear_cookie()`**:** Request the client to delete the given cookie

Usage:

```
Response$clear_cookie(name)
```

Arguments:

`name` The name of the cookie to delete

Method `has_cookie()`**:** Queries whether the response contains a cookie named `name`

Usage:

```
Response$has_cookie(name)
```

Arguments:

`name` The name of the cookie to look for

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

Usage:

```
Response$set_links(...)
```

Arguments:

`...` key-value pairs for the links

Method `format()`**:** Based on the formatters passed in through `...` content negotiation is performed with the request and the preferred formatter is chosen and applied. 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`

Usage:

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

Arguments:

`...` A range of formatters

`autofail` Automatically populate the response if formatting fails

`compress` Should `$compress()` be run in the end

`default` The name of the default formatter, which will be used if none match. Setting this will avoid autofailing with 406 as a formatter is always selected

Method `set_formatter()`**:** Based on the formatters passed in through `...` content negotiation is performed with the request and the preferred formatter is chosen. The `Content-Type` header is set automatically. If a formatter is not found and `autofail = TRUE` the response will be set to `406`. The found formatter is registered with the response and will be applied just before handing off the response to httpuv, unless the response has been manually formatted.

Usage:

```
Response$set_formatter(..., autofail = TRUE, default = NULL)
```

Arguments:

`...` A range of formatters

`autofail` Automatically populate the response if formatting fails

`default` The name of the default formatter, which will be used if none match. Setting this will avoid autofailing with 406 as a formatter is always selected

Method `compress()`**:** 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.

Usage:

```
Response$compress(
  priority = c("gzip", "deflate", "br", "identity"),
  force = FALSE,
  limit = NULL
)
```

Arguments:

`priority` A vector of compression types ranked by the servers priority

`force` Should compression be done even if the type is known to be uncompressible

`limit` The size limit in bytes for performing compression. If `NULL` then the `compression_limit` setting from the initialization of the request is used

Method `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.

Usage:

```
Response$content_length()
```

Method `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. Will apply the formatter set by `set_formatter()` unless the body has already been formatted. Will add a Date header if none exist.

Usage:

```
Response$as_list()
```

Method `as_message()`**:** Prints a HTTP representation of the response to the output stream.

Usage:

```
Response$as_message()
```

Method `encode_string()`**:** base64-encode a string. If a key has been provided during initialisation the string is first encrypted and the final result is a combination of the encrypted text and the nonce, both base64 encoded and combined with a `"_"`.

Usage:

```
Response$encode_string(val)
```

Arguments:

`val` A single string to encrypt

Method `decode_string()`**:** base64-decodes a string. If a key has been provided during initialisation the input is first split by `"_"` and then the two parts are base64 decoded and decrypted. Otherwise the input is base64-decoded as-is. It will always hold that `val == decode_string(encode_string(val))`.

Usage:

```
Response$decode_string(val)
```

Arguments:

`val` A single string to encrypt

Method `reset()`**:** Resets the content of the response. Is mainly used by the `clear()` method of the associated request, and should seldom be called directly

Usage:

```
Response$reset()
```

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()
```

---

session_cookie *Collect settings for a session cookie*

Description

A session cookie is just like any other cookie, but reqres treats this one different, parsing it's value and making it available in the `$session` field. However, the same settings as any other cookies applies and can be given during request initialisation using this function.

Usage

```
session_cookie(
  name = "reqres",
  expires = NULL,
  max_age = NULL,
  path = NULL,
  secure = NULL,
  same_site = NULL
)

is_session_cookie_settings(x)
```

Arguments

name	The name of the cookie
expires	A POSIXct object given the expiration time of the cookie
max_age	The number of seconds to elapse before the cookie expires
path	The URL path this cookie is related to
secure	Should the cookie only be send over https
same_site	Either "Lax", "Strict", or "None" indicating how the cookie can be send during cross-site requests. If this is set to "None" then secure *must* also be set to TRUE
x	An object to test

Value

A session_cookie_settings object that can be used during request initialisation. Can be cached and reused for all requests in a server

Note

As opposed to regular cookies the session cookie is forced to be HTTP only which is why this argument is missing.

Examples

```
session_cookie <- session_cookie()

rook <- fiery::fake_request("http://example.com")

# A key must be provided for session_cookie to be used
Request$new(rook, key = random_key(), session_cookie = session_cookie)
```

`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