Package 'webfakes'

November 8, 2022

Title Fake Web Apps for HTTP Testing

Version 1.1.6

Description Create a web app that makes it easier to test web clients without using the internet. It includes a web app framework with path matching, parameters and templates. Can parse various 'HTTP' request bodies. Can send 'JSON' data or files from the disk. Includes a web app that implements the <https://httpbin.org> web service.

License MIT + file LICENSE

URL https://webfakes.r-lib.org/,
https://github.com/r-lib/webfakes#readme

BugReports https://github.com/r-lib/webfakes/issues

RoxygenNote 7.2.1.9000

Config/testthat/edition 3

Imports stats, tools, utils

Suggests callr, curl, glue, jsonlite, httr, httpuv, testthat (>= 3.0.0), withr, xml2

Encoding UTF-8

NeedsCompilation yes

Author Gábor Csárdi [aut, cre],
RStudio, Pbc. [cph],
Civetweb contributors [ctb] (see inst/credits/ciwetweb.md),
Redoc contributors [ctb] (see inst/credits/redoc.md),
L. Peter Deutsch [ctb] (src/md5.h),
Martin Purschke [ctb] (src/md5.h),
Aladdin Enterprises [cph] (src/md5.h),
Maëlle Salmon [ctb] (<https://orcid.org/0000-0002-2815-0399>)

Maintainer Gábor Csárdi <csardi.gabor@gmail.com>

Repository CRAN

Date/Publication 2022-11-08 13:10:02 UTC

R topics documented:

`glossary` *webfakes glossary*

Description

webfakes glossary

Webfakes glossary

The webfakes package uses vocabulary that is standard for web apps, especially those developed with Express.js, but not necessarily well known to all R package developers.

app:

(Also: fake web app, webfakes app.) A web application that can be served by webfakes's web server, typically in another process, an *app process*. Sometimes we call it a *fake* web app, to emphasize that we use it for testing real web apps and APIs.

You can create a webfakes app with the `new_app()` function. A webfakes app is an R object that you can save to disk with `saveRDS()` , and you can also include it in your package.

You can start an with its `$listen()` method. Since the main R process runs that test suite code, you usually run them in a subprocess, see `new_app_process()` or `local_app_process()`.

app process:

(Also: web server process, webfakes subprocess.) An app process is an R subprocess, started from the main R process, to serve a webfakes *app*.

You can create an app process object with `new_app_process()` or `local_app_process()`. By default the actual process does not start yet, when you create it. You can start it explicitly with the `$start` method of the app process object, or by querying its URL with `$url()` or its port with `$get_port()`.

For test cases, you typically start app processes at these places:

- In a `setup*.R` file, to start an app that the whole test suite can use.
- Alternatively, in a `helper*.R` file, to start an app that the whole test suite can use, and it works better for interactive development.
- At the beginning of a test file, to create an app for a single test file.
- Inside `test_that()`, to create an app for a single test block.

See the How-to for details about each.

handler:

(Or handler function.) A handler is a *route* or a *middleware*.

handler stack:

This is a stack of handler functions, which are called by the app one after the other, passing the request and response objects to them. Handlers typically manipulate the request and/or response objects. A terminal handler instructs the app to return the response to the HTTP client. A non-terminal handler tells the app to keep calling handlers, by returning the string `"next"`.

httpbin app:

This is an example app, which implements the excellent `https://httpbin.org/` web service. You can use it to simulate certain HTTP responses. It is most handy for HTTP clients, but potentially useful for other tools as well.

Use `httpbin_app()` to create an instance of this app.

middleware:

A middleware is a handler function that is not bound to a path. It is called by the router, like other handler functions. It may manipulate the request or the response, or can have a side effect. Some example built-in middleware functions in webfakes:

- `mw_json()` parses a request's JSON body into an R object.
- `mw_log()` logs requests and responses to the screen or to a file.
- `mw_static()` serves static files from the directory.

You can also write your own middleware functions.

path matching:

The router performs path matching when it goes over the handler stack. If the HTTP method and path of a *route* match the HTTP method and URL of the request, then the handler is called, otherwise it is not. Paths can have parameters and be regular expressions. See `?new_regexp()` for regular expressions and "Path parameters" in `?new_app()` for parameters.

route:

A route is a handler function that is bound to certain paths of you web app. If the request URL matches the path of the route, then the handler function is called, to give it a chance to send the appropriate response. Route paths may have parameters or they can be regular expressions in webfakes.

routing:

Routing is the process of going over the handlers stack, and calling handler functions, one after the other, until one handles the request. If a handler function is a *route*, then the router only calls it if its path matches the request URL.

`httpbin_app` *Generic web app for testing HTTP clients*

Description

A web app similar to https://httpbin.org. See its specific docs. You can also see these docs locally, by starting the app:

```
httpbin <- new_app_process(httpbin_app())
browseURL(httpbin$url())
```

Usage

```
httpbin_app(log = interactive())
```

Arguments

`log`	Whether to log requests to the standard output.

Value

A `webfakes_app`.

Examples

```
app <- httpbin_app()
proc <- new_app_process(app)
url <- proc$url("/get")
resp <- curl::curl_fetch_memory(url)
curl::parse_headers_list(resp$headers)
cat(rawToChar(resp$content))
proc$stop()
```

local_app_process *App process that is cleaned up automatically*

Description

You can start the process with an explicit `$start()` call. Alternatively it starts up at the first `$url()` or `$get_port()` call.

Usage

```
local_app_process(app, ..., .local_envir = parent.frame())
```

Arguments

app	webfakes_app object, the web app to run.
...	Passed to new_app_process().
.local_envir	The environment to attach the process cleanup to. Typically a frame. When this frame finishes, the process is stopped.

See Also

new_app_process() for more details.

mw_etag *Middleware that add an* Etag *header to the response*

Description

This middleware handles the `If-None-Match` headers, and it sets the status code of the response to 304 if `If-None-Match` matches the `Etag`. It also removes the response body in this case.

Usage

```
mw_etag(algorithm = "crc32")
```

Arguments

algorithm	Checksum algorithm to use. Only "crc32" is implemented currently.

Value

Handler function.

See Also

Other middleware: mw_json(), mw_log(), mw_multipart(), mw_raw(), mw_static(), mw_text(), mw_urlencoded()

Examples

```
app <- new_app()
app$use(mw_etag())
app
```

---

mw_json *Middleware to parse a JSON body*

Description

Adds the parsed object as the json element of the request object.

Usage

```
mw_json(type = "application/json", simplifyVector = FALSE, ...)
```

Arguments

type	Content type to match before parsing. If it does not match, then the request object is not modified.
simplifyVector	Whether to simplify lists to vectors, passed to jsonlite::fromJSON().
...	Arguments to pass to jsonlite::fromJSON(), that performs the JSON parsing.

Value

Handler function.

See Also

Other middleware: mw_etag(), mw_log(), mw_multipart(), mw_raw(), mw_static(), mw_text(), mw_urlencoded()

Examples

```
app <- new_app()
app$use(mw_json())
app
```

`mw_log` *Log requests to the standard output or other connection*

Description

A one line log entry for every request. The output looks like this:

```
GET http://127.0.0.1:3000/image 200 3 ms - 4742
```

and contains

- the HTTP method,
- the full request URL,
- the HTTP status code of the response,
- how long it took to process the response, in ms,
- and the size of the response body, in bytes.

Usage

```
mw_log(format = "dev", stream = "stdout")
```

Arguments

`format`	Log format. Not implemented currently.
`stream`	R connection to log to. `"stdout"` means the standard output, `"stderr"` is the standard error. You can also supply a connection object, but then you need to be sure that it will be valid when the app is actually running.

Value

Handler function.

See Also

Other middleware: `mw_etag()`, `mw_json()`, `mw_multipart()`, `mw_raw()`, `mw_static()`, `mw_text()`, `mw_urlencoded()`

Examples

```
app <- new_app()
app$use(mw_log())
app
```

mw_multipart *Parse a multipart HTTP request body*

Description

Adds the parsed form fields in the `form` element of the request and the parsed files to the `files` element.

Usage

```
mw_multipart(type = "multipart/form-data")
```

Arguments

`type`	Content type to match before parsing. If it does not match, then the request object is not modified.

Value

Handler function.

See Also

Other middleware: `mw_etag()`, `mw_json()`, `mw_log()`, `mw_raw()`, `mw_static()`, `mw_text()`, `mw_urlencoded()`

Examples

```
app <- new_app()
app$use(mw_multipart())
app
```

mw_raw *Middleware to read the raw body of a request*

Description

Adds the raw body, as a raw object to the `raw` field of the request.

Usage

```
mw_raw(type = "application/octet-stream")
```

Arguments

`type`	Content type to match. If it does not match, then the request object is not modified.

Value

Handler function.

See Also

Other middleware: mw_etag(), mw_json(), mw_log(), mw_multipart(), mw_static(), mw_text(), mw_urlencoded()

Examples

```
app <- new_app()
app$use(mw_raw())
app
```

`mw_static` *Middleware function to serve static files*

Description

The content type of the response is set automatically from the extension of the file. Note that this is a terminal middleware handler function. If a file is served, then the rest of the handler functions will not be called. If a file was not found, however, the rest of the handlers are still called.

Usage

```
mw_static(root, set_headers = NULL)
```

Arguments

`root`	Root path of the served files. Everything under this directory is served automatically. Directory lists are not currently supports.
`set_headers`	Callback function to call before a file is served.

Value

Handler function.

See Also

Other middleware: mw_etag(), mw_json(), mw_log(), mw_multipart(), mw_raw(), mw_text(), mw_urlencoded()

Examples

```
root <- system.file(package = "webfakes", "examples", "static", "public")
app <- new_app()
app$use(mw_static(root = root))
app
```

mw_text *Middleware to parse a plain text body*

Description

Adds the parsed object as the `text` element of the request object.

Usage

```
mw_text(default_charset = "utf-8", type = "text/plain")
```

Arguments

`default_charset`	Encoding to set on the text.
`type`	Content type to match before parsing. If it does not match, then the request object is not modified.

Value

Handler function.

See Also

Other middleware: `mw_etag()`, `mw_json()`, `mw_log()`, `mw_multipart()`, `mw_raw()`, `mw_static()`, `mw_urlencoded()`

Examples

```
app <- new_app()
app$use(mw_text())
app
```

mw_urlencoded *Middleware to parse an url-encoded request body*

Description

This is typically data from a form. The parsed data is added as the `form` element of the request object.

Usage

```
mw_urlencoded(type = "application/x-www-form-urlencoded")
```

Arguments

type	Content type to match before parsing. If it does not match, then the request object is not modified.

Value

Handler function.

See Also

Other middleware: mw_etag(), mw_json(), mw_log(), mw_multipart(), mw_raw(), mw_static(), mw_text()

Examples

```
app <- new_app()
app$use(mw_urlencoded())
app
```

---

new_app *Create a new web application*

Description

Create a new web application

Usage

```
new_app()
```

Details

The typical workflow of creating a web application is:

1. Create a webfakes_app object with new_app().
2. Add middleware and/or routes to it.
3. Start is with the webfakes_app$listen() method, or start it in another process with new_app_process().
4. Make queries to the web app.
5. Stop it via CTRL+C / ESC, or, if it is running in another process, with the $stop() method of new_app_process().

A web application can be

- restarted,
- saved to disk,
- copied to another process using the callr package, or a similar way,

- embedded into a package,
- extended by simply adding new routes and/or middleware.

The webfakes API is very much influenced by the express.js project.

Create web app objects:

`new_app()`

`new_app()` returns a `webfakes_app` object the has the methods listed on this page.
An app is an environment with S3 class `webfakes_app`.

The handler stack:

An app has a stack of handlers. Each handler can be a route or middleware. The differences between the two are:

- A route is bound to one or more paths on the web server. Middleware is not (currently) bound to paths, but run for all paths.
- A route is usually (but not always) the end of the handler stack for a request. I.e. a route takes care of sending out the response to the request. Middleware typically performs some action on the request or the response, and then the next handler in the stack is invoked.

Routes:

The following methods define routes. Each method corresponds to the HTTP verb with the same name, except for `app$all()`, which creates a route for all HTTP methods.

```
app$all(path, ...)
app$delete(path, ...)
app$get(path, ...)
app$head(path, ...)
app$patch(path, ...)
app$post(path, ...)
app$put(path, ...)
... (see list below)
```

- `path` is a path specification, see 'Path specification' below.
- `...` is one or more handler functions. These will be placed in the handler stack, and called if they match an incoming HTTP request. See 'Handler functions' below.

webfakes also has methods for the less frequently used HTTP verbs: `CONNECT`, `MKCOL`, `OPTIONS`, `PROPFIND`, `REPORT`. (The method names are always in lowercase.)
If a request is not handled by any routes (or handler functions in general), then webfakes will send a simple HTTP 404 response.

Middleware:

`app$use()` adds a middleware to the handler stack. A middleware is a handler function, see 'Handler functions' below. webfakes comes with middleware to perform common tasks:

- `mw_etag()` adds an `Etag` header to the response.
- `mw_log()` logs each requests to standard output, or another connection.
- `mw_raw()` parses raw request bodies.
- `mw_text()` parses plain text request bodies.

- `mw_json()` parses JSON request bodies.
- `mw_multipart()` parses multipart request bodies.
- `mw_static()` serves static files from a directory.
- `mw_urlencoded()` parses URL encoded request bodies.

```
app$use(..., .first = FALSE)
```

- `...` is a set of (middleware) handler functions. They are added to the handler stack, and called for every HTTP request. (Unless an HTTP response is created before reaching this point in the handler stack.)
- `.first` set to `TRUE` is you want to add the handler function to the bottom of the stack.

Handler functions:

A handler function is a route or middleware. A handler function is called by webfakes with the incoming HTTP request and the outgoing HTTP response objects (being built) as arguments. The handler function may query and modify the members of the request and/or the response object. If it returns the string `"next"`, then it is *not* a terminal handler, and once it returns, webfakes will move on to call the next handler in the stack.

A typical route:

```
app$get("/user/:id", function(req, res) {
  id <- req$params$id
  ...
  res$
    set_status(200L)$
    set_header("X-Custom-Header", "foobar")$
    send_json(response, auto_unbox = TRUE)
})
```

- The handler belongs to an API path, which is a wildcard path in this case. It matches `/user/alice`, `/user/bob`, etc. The handler will be only called for GET methods and matching API paths.
- The handler receives the request (`req`) and the response (`res`).
- It sets the HTTP status, additional headers, and sends the data. (In this case the `webfakes_response$send_json()` method automatically converts `response` to JSON and sets the `Content-Type` and `Content-Length` headers.
- This is a terminal handler, because it does *not* return `"next"`. Once this handler function returns, webfakes will send out the HTTP response.

A typical middleware:

```
app$use(function(req, res) {
  ...
  "next"
})
```

- There is no HTTP method and API path here, webfakes will call the handler for each HTTP request.
- This is not a terminal handler, it does return `"next"`, so after it returns webfakes will look for the next handler in the stack.

Errors:

If a handler function throws an error, then the web server will return a HTTP 500 `text/plain` response, with the error message as the response body.

Request and response objects:

See webfakes_request and webfakes_response for the methods of the request and response objects.

Path specification:

Routes are associated with one or more API paths. A path specification can be

- A "plain" (i.e. without parameters) string. (E.g. `"/list"`.)
- A parameterized string. (E.g. `"/user/:id"`.)
- A regular expression created via `new_regexp()` function.
- A list or character vector of the previous ones. (Regular expressions must be in a list.)

Path parameters:

Paths that are specified as parameterized strings or regular expressions can have parameters.

For parameterized strings the keys may contain letters, numbers and underscores. When webfakes matches an API path to a handler with a parameterized string path, the parameters will be added to the request, as `params`. I.e. in the handler function (and subsequent handler functions, if the current one is not terminal), they are available in the `req$params` list.

For regular expressions, capture groups are also added as parameters. It is best to use named capture groups, so that the parameters are in a named list.

If the path of the handler is a list of parameterized strings or regular expressions, the parameters are set according to the first matching one.

Templates:

webfakes supports templates, using any template engine. It comes with a template engine that uses the glue package, see `tmpl_glue()`.

`app$engine()` registers a template engine, for a certain file extension. The `$render()` method of webfakes_response can be called from the handler function to evaluate a template from a file.

```
app$engine(ext, engine)
```

- `ext`: the file extension for which the template engine is added. It should not contain the dot. E.g. `"html"', "brew"‘`.
- `engine`: the template engine, a function that takes the file path (`path`) of the template, and a list of local variables (`locals`) that can be used in the template. It should return the result.

An example template engine that uses glue might look like this:

```
app$engine("txt", function(path, locals) {
  txt <- readChar(path, nchars = file.size(path), useBytes = TRUE)
  glue::glue_data(locals, txt)
})
```

(The built-in `tmpl_glue()` engine has more features.)

This template engine can be used in a handler:

```
app$get("/view", function(req, res) {
 txt <- res$render("test")
 res$
   set_type("text/plain")$
   send(txt)
})
```

The location of the templates can be set using the `views` configuration parameter, see the `$set_config()` method below.

In the template, the variables passed in as `locals`, and also the response local variables (see `locals` in webfakes_response), are available.

Starting and stopping:

```
app$listen(port = NULL, opts = server_opts(), cleanup = TRUE)
```

- `port`: port to listen on. When `NULL`, the operating system will automatically select a free port.
- `opts`: options to the web server. See `server_opts()` for the list of options and their default values.
- `cleanup`: stop the server (with an error) if the standard input of the process is closed. This is handy when the app runs in a `callr::r_session` subprocess, because it stops the app (and the subprocess) if the main process has terminated.

This method does not return, and can be interrupted with `CTRL+C` / `ESC` or a SIGINT signal. See `new_app_process()` for interrupting an app that is running in another process.

When `port` is `NULL`, the operating system chooses a port where the app will listen. To be able to get the port number programmatically, before the listen method blocks, it advertises the selected port in a `webfakes_port` condition, so one can catch it:

webfakes by default binds only to the loopback interface at 127.0.0.1, so the webfakes web app is never reachable from the network.

```
withCallingHandlers(
  app$listen(),
  "webfakes_port" = function(msg) print(msg$port)
)
```

Logging:

webfakes can write an access log that contains an entry for all incoming requests, and also an error log for the errors that happen while the server is running. This is the default behavior for local app (the ones started by `app$listen()` and for remote apps (the ones started via `new_app_process()`:

- Local apps do not write an access log by default.
- Remote apps write an access log into the `<tmpdir>/webfakes/<pid>/access.log` file, where `<tmpdir>` is the session temporary directory of the *main process*, and `<pid>` is the process id of the *subprocess*.
- Local apps write an error log to `<tmpdir>/webfakes/error.log`, where `<tmpdir>` is the session temporary directory of the current process.
- Remote app write an error log to the `<tmpdir>/webfakes/<pid>/error.log`, where `<tmpdir>` is the session temporary directory of the *main process* and `<pid>` is the process id of the *subprocess*‘.

See `server_opts()` for changing the default logging behavior.

Shared app data:

`app$locals`

It is often useful to share data between handlers and requests in an app. `app$locals` is an environment that supports this. E.g. a middleware that counts the number of requests can be implemented as:

```
app$use(function(req, res) {
  locals <- req$app$locals
  if (is.null(locals$num)) locals$num <- 0L
  locals$num <- locals$num + 1L
  "next"
})
```

webfakes_response objects also have a `locals` environment, that is initially populated as a copy of `app$locals`.

Configuration:

```
app$get_config(key)
app$set_config(key, value)
```

- `key`: configuration key.
- `value`: configuration value.

Currently used configuration values:

- `views`: path where webfakes searches for templates.

Value

A new `webfakes_app`.

See Also

webfakes_request for request objects, webfakes_response for response objects.

Examples

```
# see example web apps in the `/examples` directory in
system.file(package = "webfakes", "examples")

app <- new_app()
app$use(mw_log())

app$get("/hello", function(req, res) {
  res$send("Hello there!")
})

app$get(new_regexp("^/hi(/.*)?$"), function(req, res) {
  res$send("Hi indeed!")
```

```
})

app$post("/hello", function(req, res) {
  res$send("Got it, thanks!")
})

app

# Start the app with: app$listen()
# Or start it in another R session: new_app_process(app)
```

`new_app_process` *Run a webfakes app in another process*

Description

Runs an app in a subprocess, using callr::r_session.

Usage

```
new_app_process(
  app,
  port = NULL,
  opts = server_opts(remote = TRUE),
  start = FALSE,
  auto_start = TRUE,
  process_timeout = NULL,
  callr_opts = NULL
)
```

Arguments

`app`	`webfakes_app` object, the web app to run.
`port`	Port to use. By default the OS assigns a port.
`opts`	Server options. See `server_opts()` for the defaults.
`start`	Whether to start the web server immediately. If this is `FALSE`, and `auto_start` is `TRUE`, then it is started as neeed.
`auto_start`	Whether to start the web server process automatically. If `TRUE` and the process is not running, then `$start()`, `$get_port()` and `$url()` start the process.
`process_timeout`	How long to wait for the subprocess to start, in milliseconds.
`callr_opts`	Options to pass to `callr::r_session_options()` when setting up the subprocess.

Value

A `webfakes_app_process` object.

Methods:

The `webfakes_app_process` class has the following methods:

```
get_app()
get_port()
stop()
get_state()
local_env(envvars)
url(path = "/", query = NULL)
```

- `envvars`: Named list of environment variables. The `{url}` substring is replaced by the URL of the app.
- `path`: Path to return the URL for.
- `query`: Additional query parameters, a named list, to add to the URL.

`get_app()` returns the app object.

`get_port()` returns the port the web server is running on.

`stop()` stops the web server, and also the subprocess. If the error log file is not empty, then it dumps its contents to the screen.

`get_state()` returns a string, the state of the web server:

- `"not running"` the server is not running (because it was stopped already).
- `"live"` means that the server is running.
- `"dead"` means that the subprocess has quit or crashed.

`local_env()` sets the given environment variables for the duration of the app process. It resets them in `$stop()`. Webfakes replaces `{url}` in the value of the environment variables with the app URL, so you can set environment variables that point to the app.

`url()` returns the URL of the web app. You can use the `path` parameter to return a specific path.

See Also

`local_app_process()` for automatically cleaning up the subprocess.

Examples

```
app <- new_app()
app$get("/foo", function(req, res) {
  res$send("Hello world!")
})

proc <- new_app_process(app)
url <- proc$url("/foo")
resp <- curl::curl_fetch_memory(url)
cat(rawToChar(resp$content))

proc$stop()
```

new_regexp *Create a new regular expression to use in webfakes routes*

Description

Note that webfakes uses PERL regular expressions.

Usage

```
new_regexp(x)
```

Arguments

x	String scalar containing a regular expression.

Details

As R does not have data type or class for regular expressions, you can use `new_regexp()` to mark a string as a regular expression, when adding routes.

Value

String with class `webfakes_regexp`.

See Also

The 'Path specification' and 'Path parameters' chapters of the manual of `new_app()`.

Examples

```
new_regexp("^/api/match/(?<pattern>.*)$")
```

oauth2_httr_login *Helper function to use httr's OAuth2.0 functions non-interactively, e.g. in test cases*

Description

To perform an automatic acknowledgement and log in for a local OAuth2.0 app, run by httr, wrap the expression that obtains the OAuth2.0 token in `oauth2_httr_login()`.

Usage

```
oauth2_httr_login(expr)
```

Arguments

expr	Expression that calls `httr::oauth2.0_token()`, either directly, or indirectly.

Details

In interactive sessions, `oauth2_httr_login()` overrides the `browser` option, and when httr opens a browser page, it calls `oauth2_login()` in a subprocess.

In non-interactive sessions, httr does not open a browser page, only messages the user to do it manually. `oauth2_httr_login()` listens for these messages, and calls `oauth2_login()` in a subprocess.

Value

The return value of `expr`.

See Also

See `?vignette("oauth", package = "webfakes")` for a case study that uses this function.

Other OAuth2.0 functions: `oauth2_login()`, `oauth2_resource_app()`, `oauth2_third_party_app()`

---

oauth2_login *Helper function to log in to a third party OAuth2.0 app without a browser*

Description

It works with `oauth2_resource_app()`, and any third party app, including the fake `oauth2_third_party_app()`.

Usage

```
oauth2_login(login_url)
```

Arguments

login_url	The login URL of the third party app.

Details

See `test-oauth.R` in webfakes for an example.

Value

A named list with

- `login_response` The curl HTTP response object for the login page.
- `token_response` The curl HTTP response object for submitting the login page.

See Also

Other OAuth2.0 functions: `oauth2_httr_login()`, `oauth2_resource_app()`, `oauth2_third_party_app()`

`oauth2_resource_app` *Fake OAuth 2.0 resource and authorization app*

Description

The webfakes package comes with two fake apps that allow to imitate the OAuth2.0 flow in your test cases. (See Aaron Parecki's tutorial for a good introduction to OAuth2.0.) One app (`oauth2_resource_app()`) is the API server that serves both as the resource and provides authorization. `oauth2_third_party_app()` plays the role of the third-party app. They are useful when testing or demonstrating code handling OAuth2.0 authorization, token caching, etc. in a package. The apps can be used in your tests directly, or you could adapt one or both of them to better mimic a particular OAuth2.0 flow.

Usage

```
oauth2_resource_app(
  access_duration = 3600L,
  refresh_duration = 7200L,
  refresh = TRUE,
  seed = NULL,
  authorize_endpoint = "/authorize",
  token_endpoint = "/token"
)
```

Arguments

`access_duration`	After how many seconds should access tokens expire.
`refresh_duration`	After how many seconds should refresh tokens expire (ignored if `refresh` is `FALSE`).
`refresh`	Should a refresh token be returned (logical).
`seed`	Random seed used when creating tokens. If `NULL`, we rely on R to provide a seed. The app uses its own RNG stream, so it does not affect reproducibility of the tests.
`authorize_endpoint`	The authorization endpoint of the resource server. Change this from the default if the real app that you are faking does not use `/authorize`.
`token_endpoint`	The endpoint to request tokens. Change this if the real app that you are faking does not use `/token`.

Details

The app has the following endpoints:

- `GET /register` is the endpoint that you can use to register your third party app. It needs to receive the `name` of the third party app, and its `redirect_uri` as query parameters, otherwise returns an HTTP 400 error. On success it returns a JSON dictionary with entries `name` (the name of the third party app), `client_id`, `client_secret` and `redirect_uri`.

- `GET /authorize` is the endpoint where the user of the third party app is sent. You can change the URL of this endpoint with the `authorize_endpoint` argument. It needs to receive the `client_id` of the third party app, and its correct `redirect_uri` as query parameters. It may receive a `state` string as well, which can be used by a client to identify the request. Otherwise it generates a random `state` string. On error it fails with a HTTP 400 error. On success it returns a simple HTML login page.
- `POST /authorize/decision` is the endpoint where the HTML login page generated at `/authorize` connects back to, either with a positive or negative result. The form on the login page will send the `state` string and the user's choice in the `action` variable. If the user authorized the third party app, then they are redirected to the `redirect_uri` of the app, with a temporary `code` and the `state` string supplied as query parameters. Otherwise a simple HTML page is returned.
- `POST /token` is the endpoint where the third party app requests a temporary access token. It is also uses for refreshing an access token with a refresh token. You can change the URL of this endpoint with the `token_endpoint` argument. To request a new token or refresh an existing one, the following data must be included in either a JSON or an URL encoded request body:
  - `grant_type`, this must be `authorization_code` for new tokens, and `refresh_token` for refreshing.
  - `code`, this must be the temporary code obtained from the `/authorize/decision` redirection, for new tokens. It is not needed when refreshing.
  - `client_id` must be the client id of the third party app.
  - `client_secret` must be the client secret of the third party app.
  - `redirect_uri` must be the correct redirection URI of the third party app. It is not needed when refreshing tokens.
  - `refresh_token` must be the refresh token obtained previously, when refreshing a token. It is not needed for new tokens. On success a JSON dictionary is returned with entries: `access_token`, `expiry` and `refresh_token`. (The latter is omitted if the `refresh` argument is `FALSE`).
- `GET /locals` returns a list of current apps, access tokens and refresh tokens.
- `GET /data` is an endpoint that returns a simple JSON response, and needs authorization.

Notes:

- Using this app in your tests requires the glue package, so you need to put it in `Suggests`.
- You can add custom endpoints to the app, as needed.
- If you need authorization in your custom endpoint, call `app$is_authorized()` in your handler:

  ```
  if (!app$is_authorized(req, res)) return()
  ```

  `app$is_authorized()` returns an HTTP 401 response if the client is not authorized, so you can simply return from your handler.

For more details see `vignette("oauth", package = "webfakes")`.

Value

a `webfakes` app

webfakes app

`oauth2_resource_app()`

App representing the API server (resource/authorization)

See Also

Other OAuth2.0 functions: `oauth2_httr_login()`, `oauth2_login()`, `oauth2_third_party_app()`

`oauth2_third_party_app`

App representing the third-party app

Description

The webfakes package comes with two fake apps that allow to imitate the OAuth2.0 flow in your test cases. (See Aaron Parecki's tutorial for a good introduction to OAuth2.0.) One app (`oauth2_resource_app()`) is the API server that serves both as the resource and provides authorization. `oauth2_third_party_app()` plays the role of the third-party app. They are useful when testing or demonstrating code handling OAuth2.0 authorization, token caching, etc. in a package. The apps can be used in your tests directly, or you could adapt one or both of them to better mimic a particular OAuth2.0 flow.

Usage

```
oauth2_third_party_app(name = "Third-Party app")
```

Arguments

`name`	Name of the third-party app

Details

Endpoints:

- `POST /login/config` Use this endpoint to configure the client ID and the client secret of the app, received from `oauth2_resource_app()` (or another resource app). You need to send in a JSON or URL encoded body:
  - `auth_url`, the authorization URL of the resource app.
  - `token_url`, the token URL of the resource app.
  - `client_id`, the client ID, received from the resource app.
  - `client_secret` the client secret, received from the resource app.
- `GET /login` Use this endpoint to start the login process. It will redirect to the resource app for authorization and after the OAuth2.0 dance to `/login/redirect`.
- `GET /login/redirect`, `POST /login/redirect` This is the redirect URI of the third party app. (Some HTTP clients redirect a `POST` to a `GET`, others don't, so it has both.) This endpoint is used by the resource app, and it received the `code` that can be exchanged to an access token and the `state` which was generated in `/login`. It contacts the resource app to get an access token, and then stores the token in its `app$locals` local variables. It fails with HTTP

code 500 if it cannot obtain an access token. On success it returns a JSON dictionary with `access_token`, `expiry` and `refresh_token` (optionally) by default. This behavior can be changed by redefining the `app$redirect_hook()` function.

- `GET /locals` returns the tokens that were obtained from the resource app.
- `GET /data` is an endpoint that uses the obtained token(s) to connect to the `/data` endpoint of the resource app. The `/data` endpoint of the resource app needs authorization. It responds with the response of the resource app. It tries to refresh the access token of the app if needed.

For more details see `vignette("oauth", package = "webfakes")`.

Value

webfakes app

See Also

Other OAuth2.0 functions: `oauth2_httr_login()`, `oauth2_login()`, `oauth2_resource_app()`

---

server_opts *Webfakes web server options*

Description

Webfakes web server options

Usage

```
server_opts(
  remote = FALSE,
  port = NULL,
  num_threads = 1,
  interfaces = "127.0.0.1",
  enable_keep_alive = FALSE,
  access_log_file = remote,
  error_log_file = TRUE,
  tcp_nodelay = FALSE,
  throttle = Inf
)
```

Arguments

`remote`	Meta-option. If set to `TRUE`, webfakes uses slightly different defaults, that are more appropriate for a background server process.
`port`	Port to start the web server on. Defaults to a randomly chosen port.
`num_threads`	Number of request handler threads to use. Typically you don't need more than one thread, unless you run test cases in parallel or you make concurrent HTTP requests.

`interfaces` The network interfaces to listen on. Being a test web server, it defaults to the localhost. Only bind to a public interface if you know what you are doing. webfakes was not designed to serve public web pages.

`enable_keep_alive` Whether the server keeps connections alive.

`access_log_file` `TRUE`, `FALSE`, or a path. See 'Logging' below.

`error_log_file` `TRUE`, `FALSE`, or a path. See 'Logging' below.

`tcp_nodelay` if `TRUE` then packages will be sent as soon as possible, instead of waiting for a full buffer or timeout to occur.

`throttle` Limit download speed for clients. If not `Inf`, then it is the maximum number of bytes per second, that is sent to as connection.

Value

List of options that can be passed to `webfakes_app$listen()` (see `new_app()`), and `new_app_process()`.

Logging

- For `access_log_file`, `TRUE` means `<log-dir>/access.log`.
- For `error_log_file`, `TRUE` means `<log-dir>/error.log`.

`<log-dir>` is set to the contents of the `WEBFAKES_LOG_DIR` environment variable, if it is set. Otherwise it is set to `<tmpdir>/webfakes` for local apps and `<tmpdir>/<pid>/webfakes` for remote apps (started with `new_app_procss()`).
`<tmpdir>` is the session temporary directory of the *main process*.
`<pid>` is the process id of the subprocess.

Examples

```
# See the defaults
server_opts()
```

`tmpl_glue` *glue based template engine*

Description

Use this template engine to create pages with glue templates. See `glue::glue()` for the syntax.

Usage

```
tmpl_glue(
  sep = "",
  open = "{",
  close = "}",
  na = "NA",
```

```
  transformer = NULL,
  trim = TRUE
)
```

Arguments

sep	Separator used to separate elements.
open	The opening delimiter. Doubling the full delimiter escapes it.
close	The closing delimiter. Doubling the full delimiter escapes it.
na	Value to replace NA values with. If NULL missing values are propagated, that is an NA result will cause NA output. Otherwise the value is replaced by the value of na.
transformer	A function taking three parameters code, envir and data used to transform the output of each block before during or after evaluation.
trim	Whether to trim the input template with glue::trim() or not.

Value

Template function.

Examples

```
# See th 'hello' app at
hello_root <- system.file(package = "webfakes", "examples", "hello")
hello_root

app <- new_app()
app$engine("txt", tmpl_glue())
app$use(mw_log())


app$get("/view", function(req, res) {
  txt <- res$render("test")
  res$
    set_type("text/plain")$
    send(txt)
})

# Switch to the app's root: setwd(hello_root)
# Now start the app with: app$listen(3000L)
# Or start it in another process: new_process(app)
```

webfakes_request *A webfakes request object*

Description

webfakes creates a `webfakes_request` object for every incoming HTTP request. This object is passed to every matched route and middleware, until the response is sent. It has reference semantics, so handlers can modify it.

Details

Fields and methods:

- `app`: The `webfakes_app` object itself.
- `headers`: Named list of HTTP request headers.
- `hostname`: The Host header, the server hostname and maybe port.
- `method`: HTTP method.
- `path`: Server path.
- `protocol`: `"http"` or `"https"`.
- `query_string`: The raw query string, without the starting `?`.
- `query`: Parsed query parameters in a named list.
- `remote_addr`: String, the domain name or IP address of the client. webfakes runs on the localhost, so this is `127.0.0.1`.
- `url`: The full URL of the request.
- `get_header(field)`: Function to query a request header. Returns `NULL` if the header is not present.

Body parsing middleware adds additional fields to the request object. See `mw_raw()`, `mw_text()`, `mw_json()`, `mw_multipart()` and `mw_urlencoded()`.

See Also

webfakes_response for the webfakes response object.

Examples

```
# This is how you can see the request and response objects:
app <- new_app()
app$get("/", function(req, res) {
  browser()
  res$send("done")
})
app

# Now start this app on a port:
# app$listen(3000)
```

```
# and connect to it from a web browser: http://127.0.0.1:3000
# You can also use another R session to connect:
# httr::GET("http://127.0.0.1:3000")
# or the command line curl tool:
# curl -v http://127.0.0.1:3000
# The app will stop while processing the request.
```

`webfakes_response` *A webfakes response object*

Description

webfakes creates a `webfakes_response` object for every incoming HTTP request. This object is passed to every matched route and middleware, until the HTTP response is sent. It has reference semantics, so handlers can modify it.

Details

Fields and methods:

- `app`: The `webfakes_app` object itself.
- `req`: The request object.
- `headers_sent`: Whether the response headers were already sent out.
- `locals`: Local variables, the are shared between the handler functions. This is for the end user, and not for the middlewares.
- `delay(secs)`: delay the response for a number of seconds. If a handler calls `delay()`, the same handler will be called again, after the specified number of seconds have passed. Use the `locals` environment to distinguish between the calls. If you are using `delay()`, and want to serve requests in parallel, then you probably need a multi-threaded server, see `server_opts()`.
- `add_header(field, value)`: Add a response header. Note that `add_header()` may create duplicate headers. You usually want `set_header()`.
- `get_header(field)`: Query the currently set response headers. If `field` is not present it return `NULL`.
- `on_response(fun)`: Run the `fun` handler function just before the response is sent out. At this point the headers and the body are already properly set.
- `redirect(path, status = 302)`: Send a redirect response. It sets the `Location` header, and also sends a `text/plain` body.
- `render(view, locals = list())`: Render a template page. Searches for the `view` template page, using all registered engine extensions, and calls the first matching template engine. Returns the filled template.
- `send(body)`. Send the specified body. `body` can be a raw vector, or HTML or other text. For raw vectors it sets the content type to `application/octet-stream`.

- `send_json(object = NULL, text = NULL, ...)`: Send a JSON response. Either `object` or `text` must be given. `object` will be converted to JSON using `jsonlite::toJSON()`. `...` are passed to `jsonlite::toJSON()`. It sets the content type appropriately.
- `send_file(path, root = ".")`: Send a file. Set `root = "/"` for absolute file names. It sets the content type automatically, based on the extension of the file, if it is not set already.
- `send_status(status)`: Send the specified HTTP status code, without a response body.
- `send_chunk(data)`: Send a chunk of a response in chunked encoding. The first chunk will automatically send the HTTP response headers. Webfakes will automatically send a final zero-lengh chunk, unless `$delay()` is called.
- `set_header(field, value)`: Set a response header. If the headers have been sent out already, then it throws a warning, and does nothing.
- `set_status(status)`: Set the response status code. If the headers have been sent out already, then it throws a warning, and does nothing.
- `set_type(type)`: Set the response content type. If it contains a `/` character then it is set as is, otherwise it is assumed to be a file extension, and the corresponding MIME type is set. If the headers have been sent out already, then it throws a warning, and does nothing.
- `write(data)`: writes (part of) the body of the response. It also sends out the response headers, if they haven't been sent out before.

Usually you need one of the `send()` methods, to send out the HTTP response in one go, first the headers, then the body.

Alternatively, you can use `$write()` to send the response in parts.

See Also

webfakes_request for the webfakes request object.

Examples

```
# This is how you can see the request and response objects:
app <- new_app()
app$get("/", function(req, res) {
  browser()
  res$send("done")
})
app

# Now start this app on a port:
# app$listen(3000)
# and connect to it from a web browser: http://127.0.0.1:3000
# You can also use another R session to connect:
# httr::GET("http://127.0.0.1:3000")
# or the command line curl tool:
# curl -v http://127.0.0.1:3000
# The app will stop while processing the request.
```

Index