Package 'webfakes'

May 16, 2026

Title Fake Web Apps for HTTP Testing

Version 1.5.0

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 'httpbin.org' web service.

License MIT + file LICENSE

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

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

Depends R (>= 3.6)

Imports stats, tools, utils

Suggests brotli, callr, covr, curl, digest, glue, httpuv, httr, jsonlite, processx, testthat (>= 3.0.0), withr, xml2, zip (>= 2.3.0)

Biarch true

Config/Needs/website tidyverse/tidytemplate

Config/testthat/edition 3

Config/usethis/last-upkeep 2025-04-28

Encoding UTF-8

Config/roxygen2/version 8.0.0

NeedsCompilation yes

Author Gábor Csárdi [aut, cre],
Posit Software, PBC [cph, fnd] (ROR: <https://ror.org/03wc8by49>),
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] (ORCID: <https://orcid.org/0000-0002-2815-0399>)

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

Repository CRAN

Date/Publication 2026-05-16 21:50:02 UTC

Contents

`git_app` *Web app that acts as a git http server*

Description

It is useful for tests that need an HTTP git server.

Usage

```
git_app(
  git_root,
  git_cmd = "git",
  git_timeout = as.difftime(1, units = "mins"),
  filter = TRUE,
  cleanup = TRUE
)
```

Arguments

`git_root`	Path to the root of the directory tree to be served.
`git_cmd`	Command to call, by default it is `"git"`. It may also be a full path to git.
`git_timeout`	A `difftime` object, time limit for the git command.
`filter`	Whether to support the `filter` capability in the server.
`cleanup`	Whether to clean up `git_root` when the app is garbage collected.

Examples

```
dir.create(tmp <- tempfile())
setwd(tmp)
system("git clone --bare https://github.com/cran/crayon")
system("git clone --bare https://github.com/cran/glue")
app <- git_app(tmp)
git <- new_app_process(app)
system(paste("git ls-remote", git$url("/crayon")))
```

`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.

`how-to` *How to use webfakes in your tests*

Description

How to use webfakes in your tests

How to use webfakes in your tests

How do I use webfakes in my package?:

First, you need to add webfakes to the `DESCRIPTION` file of your package. Use the `Suggests` field, as webfakes is only needed for testing:

```
...
Suggests:
  webfakes,
  testthat
...
```

Then, unless the URL to the web service is an argument of your package functions, you might need to tweak your package code slightly to make sure every call to a real web service can be targeted at another URL instead (of a fake app). See next subsection.

Last but not least, you need to decide if you want a single web app for all your test cases. The alternative is to use different apps for some or all test files. Occasionally you may want to use a special app for a single test case. Each app runs in a new subprocess, and it takes typically about 100-400ms to start.

See the sections later on on writing tests with a single app or multiple apps.

How do I make my app connect to webfakes when the tests are running?:

In the typical scenario, you want your package to connect to the test app only when running the tests. If the URL to the web service is not an argument of the functions, one way to achieve this is to allow specifying the web server URL(s) via environment variables. E.g. when writing a GitHub API client, your package can check use `GITHUB_URL` environment variable.

E.g.

```
service_url <- function() {
    Sys.getenv("GITHUB_URL", "https://api.github.com")
}

# rest of the package code

foobar <- function() {
    httr::GET(service_url())
}
```

When this is not set, the package connects to the proper GitHub API. When testing, you can point it to your test app.

`new_app_process()` helps you setting up temporary environment variables. These are active while the process is running, and they are removed or reset in `$stop()`. For example:

In `$local_env()` environment variables, webfakes replaces `{url}` with the actual app URL. This is needed by default, because the web server process starts up only later, so the URL is not known yet.

```
http <- webfakes::local_app_process(webfakes::httpbin_app(), start = TRUE)
http$local_env(list(GITHUB_API = "{url}"))
Sys.getenv("GITHUB_API")
#> [1] "http://127.0.0.1:64362/"
http$stop()
Sys.getenv("GITHUB_API")
#> [1] ""
```

How can I write my own app?:

You create a new app with `new_app()`. This returns an object with methods to add middleware and API endpoints to it. For example, a simple app that returns the current time in JSON would look like this:

```
time <- webfakes::new_app()
time$get("/time", function(req, res) {
  res$send_json(list(time = format(Sys.time())), auto_unbox = TRUE)
})
```

Now you can start this app on a random port using `web$listen()`. Alternatively, you can start it in a subprocess with `new_app_process()`.

```
web <- webfakes::new_app_process(time)
web$url()
#> [1] "http://127.0.0.1:64364/"
```

Use `web$url()` to query the URL of the app. For example:

```
url <- web$url("/time")
httr::content(httr::GET(url))
#> $time
#> [1] "2025-01-14 10:07:38"
```

`web$stop()` stops the app and the subprocess as well:

```
web$stop()
web$get_state()
#> [1] "not running"
```

`local_app_process()` is similar to `new_app_process()`, but it stops the server process at the end of the calling block. This means that the process is automatically cleaned up at the end of a `test_that()` block or at the end of the test file.

You can create your app at the beginning of your test file. Or, if you want to use the same app in multiple test files, use a testthat helper file. Sometimes it useful if your users can create and use your test app, for example to create reproducible examples. You can include a (possibly internal) function in your package, that creates the app.

See `?new_app()`, `?new_app_process()` and `?local_app_process` for more details.

How do I use `httpbin_app()` (or another app) with testthat?:

You can use testthat's setup files. You start the app in a setup file and also register a teardown expression for it. `local_app_process()` can do both in one go. Your `tests/testthat/setup-http.R` may look like this:

```
http <- webfakes::local_app_process(
  webfakes::httpbin_app(),
  .local_envir = testthat::teardown_env()
)
```

(Before testthat 3.0.0, you had to write the teardown expression in a `tests/testthat/teardown-http.R` file. That still works, but a single setup file is considered to be better practice, see this testthat vignette.)

In the test cases you can query the `http` app process to get the URLs you need to connect to:

```
test_that("fails on 404", {
  url <- http$url("/status/404")
  response <- httr::GET(url)
  expect_error(
    httr::stop_for_status(response),
    class = "http_404"
  )
})
#> Test passed with 1 success.
```

When writing your tests interactively, you may create a `http` app process in the global environment, for convenience. You can `source()` your `setup-http.R` file for this. Alternatively, you can start the app process in a helper file. See "How do I start the app when writing the tests?" just below.

How do I start the app when writing the tests?:

It is convenient to start the webfakes server process(es) when working on the tests interactively, e.g. when using `devtools::load_all()`. With `local_app_process()` in the testthat `setup*.R` file this is not automatic, because `devtools::load_all()` does not run these files. So you would need to source the `setup*.R` files manually, which is error prone.

One solution is to create server processes in the testthat `helper*.R` files. `load_all()` executes the helper files by default. So instead of using a setup file, you can simply do this in the `helper-http.R` file:

```
httpbin <- local_app_process(httpbin_app())
```

If the app process is created in the helper file, then it is ready use after `load_all()`, and (by default) the actual process will be started at the first `$url()` or `$get_port()` call. You can also start it manually with `$start()`.

Processes created in helper files are not cleaned up automatically at the end of the test suite, unless you clean them up by registering a `$stop()` call in a setup file, like this:

```
withr::defer(httpbin$stop(), testthat::teardown_env())
```

In practice this is not necessary, because `R CMD check` runs the tests in a separate process, and when that finishes, the webfakes processes are cleaned up as well.

When running `devtools::test()`, `testthat::test_local()` or another testthat function to run (part of) the test suite in the current session, the `helper*.R` files are (re)loaded first. This will terminate the currently running app processes, if any, and create new app process objects. Should the test suite auto-start some of the test processes from `helper*.R`, these will not be cleaned up at the end of the test suite, but only at the next `load_all()` or `test()` call, or at the end of the R session. This lets you run your test code interactively, either via `test()` or manually, without thinking too much about the webfakes processes.

Can I have an app for a single testthat test file?:

To run a web app for a single test file, start it with `new_app_process()` at the beginning of the file, and register its cleanup using `withr::defer()`. Even simpler, use `local_app_process()` which is the same as `new_app_process()` but it automatically stops the web server process, at the end of the test file:

```
app <- webfakes::new_app()
app$get("/hello/:user", function(req, res) {
  res$send(paste0("Hello ", req$params$user, "!"))
})

web <- webfakes::local_app_process(app)
```

Then in the test cases, use `web$url()` to get the URL to connect to.

```
test_that("can use hello API", {
  url <- web$url("/hello/Gabor")
  expect_equal(httr::content(httr::GET(url)), "Hello Gabor!")
})
#> No encoding supplied: defaulting to UTF-8.
#> Test passed with 1 success.
```

Can I use an app for a single testthat test?:

Sure. For this you need to create the app process within the `testthat::test_that()` test case. `local_app_process()` automatically cleans it up at the end of the block. It goes like this:

```
test_that("query works", {
  app <- webfakes::new_app()
  app$get("/hello", function(req, res) res$send("hello there"))
  web <- webfakes::local_app_process(app)
```

```
  echo <- httr::content(httr::GET(web$url("/hello")))
  expect_equal(echo, "hello there")
})
#> No encoding supplied: defaulting to UTF-8.
#> Test passed with 1 success.
```

How do I test a sequence of requests?:

To test a sequence of requests, the app needs state information that is kept between requests. `app$locals` is an environment that belongs to the app, and it can be used to record information and then retrieve it in future requests. You could store anything in `app$locals`, something simple like a counter variable, something fancier like a sqlite database. You can add something to `app$locals` via methods or directly after creating the app.

```
store <- webfakes::new_app()
store$locals$packages <- list("webfakes")
ls(store$locals)
#> [1] "packages"
store$locals$packages
#> [[1]]
#> [1] "webfakes"
```

E.g. here is an end point that fails three times, then succeeds once, fails again three times, etc. Note that the `counter` created by the code below starts at 0, not 1.

```
flaky <- webfakes::new_app()
flaky$get("/unstable", function(req, res) {
  if (identical(res$app$locals$counter, 3L)) {
    res$app$locals$counter <- NULL
    res$send_json(object = list(result = "ok"))
  } else {
    res$app$locals$counter <- c(res$app$locals$counter, 0L)[[1]] + 1L
    res$send_status(401)
  }
})
```

Let's run this app in another process and connect to it:

```
pr <- webfakes::new_app_process(flaky)
url <- pr$url("/unstable")
httr::RETRY("GET", url, times = 4)
#> Request failed [401]. Retrying in 1 seconds...
#> Request failed [401]. Retrying in 1 seconds...
#> Request failed [401]. Retrying in 2.1 seconds...
#> Response [http://127.0.0.1:64374/unstable]
#>   Date: 2025-01-14 10:07
#>   Status: 200
#>   Content-Type: application/json
#>   Size: 17 B
```

Another example where we send information to an app and then retrieve it. On a POST request we store the `name` query parameter in `app$locals$packages`, which can be queried with a GET request.

```
store <- webfakes::new_app()
# Initial "data" for the app
store$locals$packages <- list("webfakes")
# Get method
store$get("/packages", function(req, res) {
  res$send_json(res$app$locals$packages, auto_unbox = TRUE)
})
# Post method, store information from the query
store$post("/packages", function(req, res) {
  res$app$locals$packages <- c(res$app$locals$packages, req$query$name)
  res$send_json(res$app$locals$packages, auto_unbox = TRUE)
})
```

Now we start the app in a subprocess, and run a GET query against it.

```
web <- webfakes::local_app_process(store, start = TRUE)
# Get current information
get_packages <- function() {
  httr::content(
    httr::GET(
      httr::modify_url(
        web$url(),
        path = "packages"
        )
      )
    )
}
get_packages()
#> [[1]]
#> [1] "webfakes"
```

Let's POST some new information.

```
post_package <- function(name) {
  httr::POST(
  httr::modify_url(
    web$url(),
    path = "packages",
    query = list(name = name)
  )
)
}
post_package("vcr")
#> Response [http://127.0.0.1:64380/packages?name=vcr]
#>   Date: 2025-01-14 10:07
#>   Status: 200
#>   Content-Type: application/json
#>   Size: 18 B
# Get current information
get_packages()
```

```
#> [[1]]
#> [1] "webfakes"
#>
#> [[2]]
#> [1] "vcr"

post_package("httptest")
#> Response [http://127.0.0.1:64380/packages?name=httptest]
#>   Date: 2025-01-14 10:07
#>   Status: 200
#>   Content-Type: application/json
#>   Size: 29 B
# Get current information
get_packages()
#> [[1]]
#> [1] "webfakes"
#>
#> [[2]]
#> [1] "vcr"
#>
#> [[3]]
#> [1] "httptest"
```

Stop the app process:

```
web$stop()
```

How can I debug an app?:

To debug an app, it is best to run it in the main R process, i.e. *not* via `new_app_process()`. You can add breakpoints, or `browser()` calls to your handler functions, and then invoke your app from another process. You might find the `curl` command line tool to send HTTP requests to the app, or you can just use another R process. Here is an example. We will simply print the incoming request object to the screen now. For a real debugging session you probably want to place a `browser()` command there.

```
app <- webfakes::new_app()
app$get("/debug", function(req, res) {
  print(req)
  res$send("Got your back")
})
```

Now start the app on port 3000:

```
app$listen(port = 3000)
```

```
#> Running webfakes web app on port 3000
```

Connect to the app from another R or `curl` process:

```
curl -v http://127.0.0.1:3000/debug
```

```
#> *   Trying 127.0.0.1...
#> * TCP_NODELAY set
#> * Connected to 127.0.0.1 (127.0.0.1) port 3000 (#0)
#> > GET /debug HTTP/1.1
#> > Host: 127.0.0.1:3000
#> > User-Agent: curl/7.54.0
#> > Accept: */*
#> >
#> < HTTP/1.1 200 OK
#> < Content-Type: text/plain
#> < Content-Length: 13
#> <
#> * Connection #0 to host 127.0.0.1 left intact
#> Got your back
```

Your main R session will print the incoming request:

```
#> <webfakes_request>
#> method:
#>   get
#> url:
#>   http://127.0.0.1:3000/debug
#> client:
#>   127.0.0.1
#> query:
#> headers:
#>   Host: 127.0.0.1:3000
#>   User-Agent: curl/7.54.0
#>   Accept: */*
#> fields and methods:
#>   app                    # the webfakes_app the request belongs to
#>   headers                # HTTP request headers
#>   hostname               # server hostname, the Host header
#>   method                 # HTTP method of request (lowercase)
#>   path                   # server path
#>   protocol               # http or https
#>   query_string           # raw query string without '?'
#>   query                  # named list of query parameters
#>   remote_addr            # IP address of the client
#>   url                    # full URL of the request
#>   get_header(field)      # get a request header
#>  # see ?webfakes_request for details
```

Press `CTRL+C` or `ESC` to interrupt the app in the main session.

How can I test HTTPS requests?:

Serving HTTPS from `localhost` or `127.0.0.1` instead of HTTP is easy, all you need to do is

1. Set the port to an HTTPS port by adding an `"s"` suffix to the port number. Use `"0s"` for an OS assigned free port:

```
new_app_process(app, port = "0s")
```

By default webfakes uses the server key + certificate in the file at

```
system.file("cert/localhost/server.pem", package = "webfakes")
```

This certificate includes `localhost`, `127.0.0.1` and `localhost.localdomain`. If you need another domain or IP address, you'll need to create your own certificate. The `generate.sh` file in the same directory helps with that.

2. Specify the certificate bundle to the HTTP client you are using. For the default server key use the `ca.crt` file from the webfakes package:

   ```
   system.file("cert/localhost/ca.crt", package = "webfakes")
   ```

   See examples for HTTP clients below.

If you are using the curl package, use the `ca_info` option in `curl::new_handle()` or `curl::handle_setopt()`:

```
cainfo <- system.file("cert/localhost/ca.crt", package = "webfakes")
curl::curl_fetch_memory(
  http$url("/path/to/endpoint"),
  handle = curl::new_handle(cainfo = cainfo)
)
```

For the httr package, use `httr::config(cainfo = ...)`:

```
httr::GET(
  http$url("/headers", https = TRUE),
  httr::config(cainfo = cainfo)
)
```

For the httr2 package:

```
httr2::request("https://example.com") |>
  httr2::req_options(cainfo = cainfo) |>
  httr2::req_perform()
```

For `utils::download.file` point the `CURL_CA_BUNDLE` environment variable to the `ca.crt` file. Don't forget to undo this, once the HTTP request is done.

```
Sys.setenv(
  CURL_CA_BUNDLE = system.file("cert/localhost/ca.crt", package = "webfakes")
)
download.file(http$url("/path/to/endpoint"), res <- tempfile())
```

Special considerations for tests on Windows:
Unfortunately things are not that simple on Windows, for the HTTP clients. As far as I can tell, it is not easily possible to make the HTTP clients accept a new self-signed certificate. It is possible with libcurl, though, but you need to set the `CURL_SSL_BACKEND=openssl` environment variable. (And libcurl must be built with openssl support of course.) You need to set this env var *before* loading libcurl, so it is best to set it before starting R. One way to do this in the tests is to run the tests in a subprocess, with the callr package. Look at the `test-https.R` file in webfakes for a complete, current example. The tests use this helper function, defined in `helper.R`:

```
callr_curl <- function(url, options = list()) {
  callr::r(
    function(url, options) {
```

```
      h <- curl::new_handle()
      curl::handle_setopt(h, .list = options)
      curl::curl_fetch_memory(url, handle = h)
    },
    list(url = url, options = options),
    env = c(
      callr::rcmd_safe_env(),
      CURL_SSL_BACKEND = "openssl",
      CURL_CA_BUNDLE = if ("cainfo" %in% names(options)) options$cainfo
    )
  )
}
```

Example test case:

```
# ...
  cainfo <- system.file("cert/localhost/ca.crt", package = "webfakes")
  resp <- if (.Platform$OS.type == "windows") {
    callr_curl(http$url("/hello"), list(cainfo = cainfo))
  } else {
    curl::curl_fetch_memory(
      http$url("/hello"),
      handle = curl::new_handle(cainfo = cainfo)
    )
  }
# ...
```

It seems like a good idea to `skip_on_cran()` HTTPS tests, at least on Windows, because this setup is not yet tested enough to consider it robust.
webfakes uses Mbed TLS for serving HTTPS.

How can I run a server on multiple ports?:

You can specify multiple port numbers, in a vector. webfakes will then listen on all those ports. You can also mix HTTP and HTTP ports.

To redirect from an HTTP port to an HTTPS port, append an `"r"` suffix to the HTTP port number. THis port will be redirected to the next HTTPS port. E.g.

```
new_app_process(app, port = c("3000r", "3001s"))
```

will redirect HTTP from port 3000 to HTTPS on port 3001.

To redirect from an OS assigned HTTP port to an OS assigned HTTPS port, use zeros for the port numbers:

```
http <- new_app_process(app, port = c("0r", "0s"))
```

Then you can use `http$get_ports()` to query all port numbers.

You can also use

```
http$url(..., https = TRUE)
```

to get an HTTPS URL instead of the default one (the one with the first port).

Can I test asynchronous or parallel HTTP requests?:

R is single threaded and a webfakes app runs an R interpreter, so it cannot process multiple requests at the same time. The web server itself runs in a separate thread, and it can also process each request in a separate thread, but at any time only one request can use the R interpreter.

This is important, because sometimes test requests may take longer to process. For example the `/delay/:secs` end point of `httpbin_app()` wait for the specified number of seconds before responding, to simulate a slow web server. If this wait is implemented via the standard `Sys.sleep()` R function, then no other requests can be processed until the sleep is over. To avoid this, webfakes can put the waiting request on hold, return from the R interpreter, and respond to other incoming requests. Indeed, the `/delay/` end point is implemented using this feature.

However, the request thread of the web server is still busy while on hold, so to take advantage of this, you need to allow multiple threads. The `num_threads` argument of the `$listen()` method of `webfakes_app` lets you specify the number of request threads the web server will use. Similarly, the `num_threads` argument of `local_app_process()` lets you modify the number of threads.

When testing asynchronous or parallel code, that might invoke multiple, possibly delayed requests, it is best to increase the number of threads. The code below calls the same API request concurrently, three times. Each request takes 1 second to answer, but if the web server has more than three threads, together they'll still take about 1 second.

```
web <- webfakes::local_app_process(
  webfakes::httpbin_app(),
  opts = webfakes::server_opts(num_threads = 6, enable_keep_alive = TRUE)
)

testthat::test_that("parallel requests", {
  url <- web$url("/delay/0.5")
  p <- curl::new_pool()
  handles <- replicate(3, curl::new_handle(url = url))
  resps <- list()
  for (handle in handles) {
    curl::multi_add(
      handle,
      done = function(x) resps <<- c(resps, list(x)),
      fail = stop,
      pool = p
    )
  }
  st <- system.time(curl::multi_run(timeout = 3, pool = p))
  testthat::expect_true(st[["elapsed"]] < 1.5)
})
#> Test passed with 1 success.
```

(If this should fail for you and webfakes appears to process the requests sequentially, see issue #108 for possible workarounds.)

How to make sure that my code works with the *real* API?:

Indeed, if you use webfakes for your test cases, then they never touch the real web server. As you might suspect, this is not ideal, especially when you do not control the server. The web service might change their API, and your test cases will fail to warn you.

One practical solution is to write (at least some) flexible tests, that can run against a local fake webserver, or a real one, and you have a quick switch to change their behavior. I have found that environment variables work great for this.

E.g. if the FAKE_HTTP_TESTS environment variable is not set, the tests run with the real web server, otherwise they use a fake one. Another solution, that works best is the HTTP requests are in the downstream package code, is to introduce one environment variable for each API you need to connect to. These might be set to the real API servers, or to the fake ones.

Once you have some tests that can use both kinds or servers, you can set up your continuous integration (CI) framework, to run the tests agains the real server (say) once a day. This special CI run makes sure that your code works well with the real API. You can run all the other tests, locally and in the CI, against the fake local web server.

See the question on how webfakes helps you setting environment variables that point to your local server.

How do I simulate a slow internet connection?:

You need to use the `throttle` server option when you start your web app. This means that you can run the very same app with different connection speed. This is how it goes:

```
library(webfakes)
slow <- new_app_process(
  httpbin_app(),
  opts = server_opts(throttle = 100)
)
resp <- curl::curl_fetch_memory(slow$url("/bytes/200"))
resp$times
#>      redirect    namelookup       connect   pretransfer starttransfer
#>      0.000000      0.000087      0.000266      0.000284      0.004878
#>         total
#>      2.013756
```

`throttle` gives the number of bytes per second, so downloading 200 random bytes from the fake app will take about 2 seconds.

`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()
```

---

http_time_stamp *Format a time stamp for HTTP*

Description

Format a time stamp for HTTP

Usage

```
http_time_stamp(t = Sys.time())
```

Arguments

t	Date-time value to format, defaults to the current date and time. It must be a POSIXct object.

Value

Character vector, formatted date-time.

`introduction` *Happy HTTP testing with webfakes*

Description

Happy HTTP testing with webfakes

Happy HTTP testing with webfakes

What is webfakes?:

Webfakes is an R package that can spin up web servers on your machine to facilitate testing R code. R code that needs an HTTP connection is not trivial to test:

- Connectivity problems might prevent the tests from accessing the web server.
- The web server might need authentication, and it is not easy to convey login information to the test suite in a secure way.
- The web server might have rate limits, i.e, it limits the number of queries per hour or day, causing some spurious test failures.
- You might want to test in non-normal conditions, e.g. with low bandwidth, or when the client is rate limited. These conditions don't normally happen on the web server and they are hard to trigger.

With webfakes you can easily start a custom web app, that is running on the local machine.

- Webfakes does not need a network connection.
- Webfakes does not need authentication. Well, unless you want it to.
- Webfakes does not have rate limits.
- Webfakes can simulate low bandwidth, or a broken connection.

Webfakes vs mocking:

Mocking is a general technique to mimic the behavior of a function or object that is needed in test case. In the case of HTTP requests, this typically means that both the request and its response are recorded when the tests run the first time, and saved to disk. Subsequent test runs intercept the HTTP requests, match them against the recorded requests and then replay the corresponding recorded response. See for example the vcr and httptest R packages.

The advantages of using your own webfakes server, over mocking:

- Simpler infrastructure. No separate recording and replaying phases, no recorded files. No request matching.
- You can use any web client you want. E.g. curl and base R's HTTP functions do not explicitly support mocking currently.
- No need to worry about sensitive information in recorded requests or responses.
- Often easier to use when testing non-normal conditions, e.g. errors that are hard to trigger, low bandwidth, or rate limits.
- Works if you stream data from a HTTP connection, instead of reading the whole response at once.
- You can reuse the same app for multiple tests, in multiple packages.

- Easier to use for tests that require multiple rounds of requests.
- Comes with a built-in `https://httpbin.org` compatible app, chances are, you don't even need to write your testing app, just start writing tests right away.
- Better test writing experience. This is subjective, and your mileage may vary.

Webfakes vs the real API:

- No network needed. No more `skip_if_offline()`.
- Much faster.
- No rate limits. But you can simulate one if you want to.
- You can write your custom app.
- Simulate low bandwidth or a broken connection.

Webfakes vs httpbin.org:

- No network needed. No more `skip_if_offline()`.
- Much faster.
- You can use the built-in `webfakes::httpbin_app()` app, so it is easy to switch from httpbin.org.
- You can write your custom app, httpbin.org might not have what you need.

Using `webfakes::httpbin_app()` with testthat:

You can use testthat's setup files. You start the app in a setup file and also register a teardown expression for it. `local_app_process()` can do both in one go. Your `tests/testthat/setup-http.R` may look like this:

```
http <- webfakes::local_app_process(
  webfakes::httpbin_app(),
  .local_envir = testthat::teardown_env()
)
```

(Before testthat 3.0.0, you had to write the teardown expression in a `tests/testthat/teardown-http.R` file. That still works, but a single setup file is considered to be better practice, see this testthat vignette.)

In the test cases you can query the `http` app process to get the URLs you need to connect to:

```
test_that("fails on 404", {
  url <- http$url("/status/404")
  response <- httr::GET(url)
  expect_error(
    httr::stop_for_status(response),
    class = "http_404"
  )
})
#> Test passed with 1 success.
```

When writing your tests interactively, you may create a `http` app process in the global environment, for convenience. You can `source()` your `setup-http.R` file for this. Alternatively, you can start the app process in a helper file. See "How do I start the app when writing the tests?" just below.

You can also create a web server for a test file, or even for a single test case. See `vignette("how-to")` for details how.

Writing apps:

If the builtin `httpbin_app()` is not appropriate for your tests, you can write your own app. You can also extend the `httpbin_app()` app, if you don't want to start from scratch.

You create a new app with `new_app()`. This returns an object with methods to add middleware and API endpoints to it. For example, a simple app that returns the current time in JSON would look like this:

```
time <- webfakes::new_app()
time$get("/time", function(req, res) {
  res$send_json(list(time = format(Sys.time())), auto_unbox = TRUE)
})
```

Now you can start this app on a random port using `web$listen()`. Alternatively, you can start it in a subprocess with `new_app_process()`.

```
web <- webfakes::new_app_process(time)
web$url()
#> [1] "http://127.0.0.1:64358/"
```

Use `web$url()` to query the URL of the app. For example:

```
url <- web$url("/time")
httr::content(httr::GET(url))
#> $time
#> [1] "2025-01-14 10:07:33"
```

`web$stop()` stops the app and the subprocess as well:

```
web$stop()
web$get_state()
#> [1] "not running"
```

`local_app_process()` is similar to `new_app_process()`, but it stops the server process at the end of the calling block. This means that the process is automatically cleaned up at the end of a `test_that()` block or at the end of the test file.

You can create your app at the beginning of your test file. Or, if you want to use the same app in multiple test files, use a testthat helper file. Sometimes it useful if your users can create and use your test app, for example to create reproducible examples. You can include a (possibly internal) function in your package, that creates the app.

See `?new_app()`, `?new_app_process()` and `?local_app_process` for more details.

`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_cgi *Middleware that calls a CGI script*

Description

You can use it as an unconditional middleware in app$use(), as a handler on app$get(), app$post(), etc., or you can call it from a handler. See examples below.

Usage

```
mw_cgi(command, args = character(), timeout = as.difftime(Inf, units = "secs"))
```

Arguments

command	External command to run.
args	Arguments to pass to the external command.
timeout	Timeout for the external command. If the command does not terminate in time, the web server kills it and returns an 500 response.

Value

A function with signature

```
function(req, res, env = character())
```

See RFC 3875 for details on the CGI protocol.

The request body (if any) is passed to the external command as standard intput. mw_cgi() sets CONTENT_LENGTH, CONTENT_TYPE, GATEWAY_INTERFACE, PATH_INFO, QUERY_STRING, REMOTE_ADDR, REMOTE_HOST, REMOTE_USER, REQUEST_METHOD, SERVER_NAME, SERVER_PORT, SERVER_PROTOCOL, SERVER_SOFTEWARE.

It does not currently set the AUTH_TYPE, PATH_TRANSLATED, REMOTE_IDENT, SCRIPT_NAME environment variables.

The standard output of the external command is used to set the response status code, the response headers and the response body. Example output from git's CGI:

```
Status: 200 OK
Expires: Fri, 01 Jan 1980 00:00:00 GMT
Pragma: no-cache
Cache-Control: no-cache, max-age=0, must-revalidate
Content-Type: application/x-git-upload-pack-advertisement

000eversion 2
0015agent=git/2.42.0
0013ls-refs=unborn
0020fetch=shallow wait-for-done
0012server-option
0017object-format=sha1
0010object-info
0000
```

See Also

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

Examples

```
app <- new_app()
app$use(mw_cgi("echo", "Status: 200\n\nHello"))
app

app2 <- new_app()
app2$get("/greet", mw_cgi("echo", "Status: 200\n\nHello"))
app2

# Using `mw_cgi()` in a handler, you can pass extra environment variables
app3 <- new_app()
cgi <- mw_cgi("echo", "Status: 200\n\nHello")
app2$get("/greet", function(req, res) {
  cgi(req, res, env = c("EXTRA_VAR" = "EXTRA_VALUE"))
})
app3
```

---

mw_cookie_parser *Middleware to parse Cookies*

Description

Adds the cookies as the cookies element of the request object.

Usage

```
mw_cookie_parser()
```

Details

It ignores cookies in an invalid format. It ignores duplicate cookies: if two cookies have the same name, only the first one is included.

Value

Handler function.

See Also

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

---

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

Description

If the response already has an ETag header, then it is kept.

Usage

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

Arguments

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

Details

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.

Value

Handler function.

See Also

Other middleware: mw_cgi(), mw_cookie_parser(), mw_json(), mw_log(), mw_multipart(), mw_range_parser(), 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_cgi()`, `mw_cookie_parser()`, `mw_etag()`, `mw_log()`, `mw_multipart()`, `mw_range_parser()`, `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_cgi()`, `mw_cookie_parser()`, `mw_etag()`, `mw_json()`, `mw_multipart()`, `mw_range_parser()`, `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_cgi()`, `mw_cookie_parser()`, `mw_etag()`, `mw_json()`, `mw_log()`, `mw_range_parser()`, `mw_raw()`, `mw_static()`, `mw_text()`, `mw_urlencoded()`

Examples

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

mw_range_parser *Middleware to parse a Range header*

Description

Adds the requested ranges to the `ranges` element of the request object. `request$ranges` is a data frame with two columns, `from` and `to`. Each row corresponds one requested interval.

Usage

```
mw_range_parser()
```

Details

When the last `n` bytes of the file are requested, the matrix row is set to `c(0, -n)`. When all bytes after a `p` position are requested, the matrix row is set to `c(p, Inf)`.

If the intervals overlap, then `ranges` is not set, i.e. the `Range` header is ignored.

If its syntax is invalid or the unit is not `bytes`, then the `Range` header is ignored.

Value

Handler function.

See Also

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

`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_cgi()`, `mw_cookie_parser()`, `mw_etag()`, `mw_json()`, `mw_log()`, `mw_multipart()`, `mw_range_parser()`, `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_cgi(), mw_cookie_parser(), mw_etag(), mw_json(), mw_log(), mw_multipart(), mw_range_parser(), 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_cgi(), mw_cookie_parser(), mw_etag(), mw_json(), mw_log(), mw_multipart(), mw_range_parser(), 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_cgi(), mw_cookie_parser(), mw_etag(), mw_json(), mw_log(), mw_multipart(), mw_range_parser(), 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_cookie_parser()` parses `Cookie` headers.
- `mw_etag()` adds an `ETag` header to the response.
- `mw_json()` parses JSON request bodies.
- `mw_log()` logs each requests to standard output, or another connection.
- `mw_multipart()` parses multipart request bodies.
- `mw_range_parser()` parses `Range` headers.
- `mw_raw()` parses raw request bodies.
- `mw_static()` serves static files from a directory.
- `mw_text()` parses plain text request bodies.
- `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))
  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. Add an `"s"` suffix to the port to use HTTPS. Use `"0s"` to use an OS assigned port with HTTPS. See the how-to manual page if you want to start the web server on more than one ports.
- `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"
})
```

As a shortcut, a handler function may declare a `locals` argument, in which case webfakes passes `app$locals` to it directly:

```
app$use(function(req, res, locals) {
  if (is.null(locals$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(s) to use. By default the OS assigns a port. Add an `"s"` suffix to the port to use HTTPS. Use `"0s"` to use an OS assigned port with HTTPS. See the how-to on how to run the web server on multiple ports.
`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()`, `$get_ports()` and `$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()
get_ports()
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 (first) port the web server is running on.

`get_ports()` returns all ports the web server is running on, and whether it uses SSL on those ports, in a data frame with columns `ipv4`, `ipv6`, `port` and `ssl`.

`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,
  keep_alive_timeout_ms = 5000,
  access_log_file = remote,
  error_log_file = TRUE,
  tcp_nodelay = FALSE,
  throttle = Inf,
  decode_url = TRUE,
  ssl_certificate = NULL
)
```

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.
`keep_alive_timeout_ms`	Idle timeout in milliseconds for keep-alive connections. If a client does not send another request within this window, the server closes the connection. If `enable_keep_alive` is not `TRUE` then it is ignored.
`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.

`decode_url` Whether the server should automatically decode URL-encodded URLs. If `TRUE` (the default), `/foo%2fbar` will be converted to `/foo/bar` automatically. If `FALSE`, URLs as not URL-decoded.

`ssl_certificate` Path to the SSL certificate of the server, needed if you want to server HTTPS requests.

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.
- `add_cookie(name, value, options)`: Adds a cookie to the response. `options` is a named list, and may contain:
  - `domain`: Domain name for the cookie, not set by default.
  - `expires`: Expiry date in GMT. It must be a POSIXct object, and will be formatted correctly.
  - 'http_only': if TRUE, then it sets the 'HttpOnly' attribute, so Javasctipt cannot access the cookie.
  - `max_age`: Maximum age, in number of seconds.
  - `path`: Path for the cookie, defaults to "/".
  - `same_site`: The 'SameSite' cookie attribute. Possible values are "strict", "lax" and "none".
  - `secure`: if TRUE, then it sets the 'Secure' attribute.
- `clear_cookie(name, options = list())`: clears a cookie. Typically, web browsers will only clear a cookie if the options also match.
- `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