HTTP testing in R

Scott Chamberlain, Maëlle Salmon

2025-10-24

Contents

1 Preamble **9**

I Introduction 11

2 HTTP in R 101 **13**

2.1 What is HTTP? 13

2.2 HTTP requests in R: what package? 14

3 Graceful HTTP R packages **19**

3.1 Choose the HTTP resource wisely 19

3.2 User-facing grace (how your package actually works) 20

3.3 Graceful vignettes and examples 20

3.4 Graceful code 21

3.5 Graceful tests 21

3.6 Conclusion 21

4 Packages for HTTP testing **23**

4.1 Why do we need special packages for HTTP testing? 23

4.2 webmockr 23

4.3 What is vcr? 24

4.4 What is httptest? 24

4.5 What is httptest2? 25

4.6 What is webfakes? 25

4.7 testthat 25
4.8 Conclusion 26

II Whole Game(s) 27

5 Introduction 29
5.1 Our example packages 29
5.2 Conclusion 35

6 Use vcr (& webmockr) 37
6.1 Setup 37
6.2 Actual testing 38
6.3 Also testing for real interactions 41
6.4 Summary 42
6.5 PS: Where to put use_cassette() 42

7 Use httptest 45
7.1 Setup 45
7.2 Actual testing 46
7.3 Also testing for real interactions 48
7.4 Summary 48

8 vcr and httptest 51
8.1 Setting up the infrastructure 51
8.2 Calling mock files 51
8.3 Naming mock files 52
8.4 Matching requests 52
8.5 Handling secrets 52
8.6 Recording, playing back 53
8.7 Testing for API errors 54
8.8 Conclusion 54

9 Use httptest2 **55**

9.1 Setup 55

9.2 Actual testing 56

9.3 Also testing for real interactions 58

9.4 Summary 58

10 Use webfakes **61**

10.1 Setup 61

10.2 Actual testing 62

10.3 Also testing for real interactions 65

10.4 Summary 65

11 vcr (& webmockr), httptest, webfakes **67**

11.1 What HTTP client can you use (curl, httr, httr2, crul) 67

11.2 Sustainability of the packages 67

11.3 Test writing experience 68

11.4 Test debugging experience 69

11.5 Conclusion 69

III Advanced Topics **71**

12 Making real requests **73**

12.1 What can change? 73

12.2 How to make real requests 73

12.3 Why not make only or too many real requests? 74

12.4 A complement to real requests: API news! 74

13 CRAN- (and Bioconductor) preparedness for your tests **77**

13.1 Running tests on CRAN? 77

13.2 Skipping a few tests on CRAN? 78

13.3 Skipping all tests on CRAN? 78

13.4 Stress-test your package 78

14 Security **79**

14.1 Managing secrets securely 79

14.2 Sensitive recorded responses? 83

14.3 Further resources 83

15 Faking HTTP errors **85**

15.1 How to test for API errors (e.g. 503) 85

15.2 How to test for sequence of responses (e.g. 503 then 200) 85

16 Contributor friendliness **87**

16.1 Taking notes about encryption 87

16.2 Providing a sandbox 87

16.3 Switching between accounts depending on the development mode 88

16.4 Documenting HTTP testing 88

IV Conclusion **89**

17 Conclusion **91**

V webmockr details **93**

18 Mocking HTTP Requests **95**

18.1 Package documentation 95

18.2 Features 95

18.3 How webmockr works in detail 95

18.4 Basic usage 97

19 stubs **99**

19.1 Writing to disk 101

20 testing **103**

21 utilities **105**

21.1 Managing stubs 105

21.2 Managing stubs 105

21.3 Managing requests 105

VI vcr details 107

22 Caching HTTP requests **109**

22.1 Package documentation 109

22.2 Design 109

22.3 Basic usage 109

23 Advanced vcr usage **111**

23.1 Mocking writing to disk 111

24 Configure vcr **115**

25 Record modes **117**

26 Request matching **119**

27 Debugging your tests that use vcr **121**

28 Security with vcr **123**

28.1 Keeping secrets safe 123

28.2 API keys and tests run in varied contexts 124

28.3 Other security 125

29 Turning vcr on and off **127**

30 Managing cassettes **129**

30.1 gitignore cassettes 129

30.2 Rbuildignore cassettes 129

30.3 sharing cassettes 130

30.4 deleting cassettes 130

30.5 cassette file types 130

31 Gotchas **131**

31.1 Correct line identification 132

32 Session info **133**

32.1 Session info 133

32.2 Full session info 135

Chapter 1

Preamble

Are you working on an R package accessing resources on the web, be it a cat facts API, a scientific data source or your system for Customer relationship management? As with all other packages, appropriate unit testing can make your code more robust. The unit testing of a package interacting with web resources, however, brings special challenges: dependence of tests on a good internet connection, testing in the absence of authentication secrets, etc. Having tests fail due to resources being down or slow, during development or on CRAN, means a time loss for everyone involved (slower development, messages from CRAN). Although some packages accessing remote resources are well tested, there is a lack of resources around best practices.

This book is meant to be a free, central reference for developers of R packages accessing web resources, to help them have a faster and more robust development. Our aim is to develop a useful guide to go with the great recent tools `{vcr}`, `{webmockr}`, `{httptest}`, `{httptest2}` and `{webfakes}`.

We expect you to know package development basics, and git.

Note related to previous versions: this book was intended as a detailed guide to using a particular suite of packages for HTTP mocking and testing in R code and/or packages, namely those maintained by Scott Chamberlain (`{crul}`, `{webmockr}`, `{vcr}`), but its scope has been extended to generalize the explanation of concepts to similar packages.

You can also read the PDF version or epub version of this book.

Thanks to contributors to the book: Alex Whan, Aurèle, Christophe Dervieux, Daniel Possenriede, Hugo Gruson, Jon Harmon, Lluís Revilla Sancho, Xavier A.

Project funded by rOpenSci (Scott Chamberlain's work) & the R Consortium (Maëlle Salmon's work).

Part I

Introduction

Chapter 2

HTTP in R 101

2.1 What is HTTP?

HTTP means HyperText Transport Protocol, but you were probably not just looking for a translation of the abbreviation. HTTP is a way for you to exchange information with a remote server. In your package, if information is going back and forth between the R session and the internet, you are using some sort of HTTP tooling. Your package is making *requests* and receives *responses*.

2.1.1 HTTP requests

The HTTP request is what your package makes. It has a method (are you fetching information via `GET`? are you sending information via `POST`?), different parts of a URL (domain, endpoint, query string), and headers (containing for instance your secret identifiers). It can contain a body. For instance, you might be sending data as JSON. In that case one of the headers will describe the content.

How do you know what request to make from your package? Hopefully you are interacting with a well documented web resource that will explain to you what methods are associated with what endpoints.

2.1.2 HTTP responses

The HTTP response is what the remote server provides, and what your package parses. A response has a status code indicating whether the request succeeded, response headers, and (optionally) a response body.

Hopefully the documentation of the web API or web resource you are working with shows good examples of responses. In any case you'll find yourself experimenting with different requests to see what the response "looks like".

2.1.3 More resources about HTTP

How do you get started with interacting with HTTP in R?

2.1.3.1 General HTTP resources

- Mozilla Developer Network docs about HTTP (recommended in the zine mentioned hereafter)
- (*not free*) Julia Evans' Zine "HTTP: Learn your browser's language!"
- The docs of the web API you are aiming to work with, and a search engine to understand the words that are new.

2.1.3.2 HTTP with R

- The docs of the R package you end up choosing!
- Digging into the source code of another package that does similar things.

2.2 HTTP requests in R: what package?

In R, to interact with web resources, it is recommended to use `{curl}`; or its higher-level interfaces `{httr}` (pronounced *hitter* or *h-t-t-r*), `{httr2}` or `{crul}`.

Do not use RCurl, because it is not actively maintained!

When writing a package interacting with web resources, you will probably use `{httr2}`, `{httr}` or `{crul}`.

- `{httr}` is the most popular and oldest of the three packages, and supports OAuth. `{httr}` docs feature a vignette called Best practices for API packages
- `{httr2}` *"is a ground-up rewrite of httr that provides a pipeable API with an explicit request object that solves more problems felt by packages that wrap APIs (e.g. built-in rate-limiting, retries, OAuth, secure secrets, and more)"* so it might be a good idea to adopt it rather than `{httr}` for a new package. It has a vignette about Wrapping APIs.
- `{crul}` does not support OAuth but it uses an object-oriented interface, which you might like. `{crul}` has a set of clients, or ways to perform requests, that might be handy. `{crul}` also has a vignette about API package best practices.

Below we will try to programmatically access the status of GitHub, the open-source platform provided by the company of the same name. We will access the same information with {httr2} and {crul} If you decide to try the low-level curl, feel free to contribute an example. The internet has enough examples for httr.

```
github_url <- "https://kctbh9vrtdwd.statuspage.io/api/v2/status.json"
```

The URL above leaves no doubt as to what format the data is provided in, JSON!

Let's first use {httr2}.

```
library("magrittr")
response <- httr2::request(github_url) %>%
  httr2::req_perform()

# Check the response status
httr2::resp_status(response)
```

```
## [1] 200
```

```
# Or in a package you'd write
httr2::resp_check_status(response)

# Parse the content
httr2::resp_body_json(response)
```

```
## $page
## $page$id
## [1] "kctbh9vrtdwd"
##
## $page$name
## [1] "GitHub"
##
## $page$url
## [1] "https://www.githubstatus.com"
##
## $page$time_zone
## [1] "Etc/UTC"
##
## $page$updated_at
## [1] "2025-10-24T06:06:05.819Z"
##
##
## $status
```

```
## $status$indicator
## [1] "none"
##
## $status$description
## [1] "All Systems Operational"
```

```
# In case you wonder, the format was obtained from a header
httr2::resp_header(response, "content-type")
```

```
## [1] "application/json; charset=utf-8"
```

Now, the same with {crul}.

```
# Create a client and get a response
client <- crul::HttpClient$new(github_url)
response <- client$get()

# Check the response status
response$status_http()
```

```
## <Status code: 200>
##   Message: OK
##   Explanation: Request fulfilled, document follows
```

```
# Or in a package you'd write
response$raise_for_status()

# Parse the content
response$parse()
```

```
## No encoding supplied: defaulting to UTF-8.
```

```
## [1] "{\"page\":{\"id\":\"kctbh9vrtdwd\",\"name\":\"GitHub\",\"url\":\"https://www.g
```

```
jsonlite::fromJSON(response$parse())
```

```
## No encoding supplied: defaulting to UTF-8.
```

```
## $page
## $page$id
## [1] "kctbh9vrtdwd"
##
```

```
## $page$name
## [1] "GitHub"
##
## $page$url
## [1] "https://www.githubstatus.com"
##
## $page$time_zone
## [1] "Etc/UTC"
##
## $page$updated_at
## [1] "2025-10-24T06:06:05.819Z"
##
##
## $status
## $status$indicator
## [1] "none"
##
## $status$description
## [1] "All Systems Operational"
```

Hopefully these very short snippets give you an idea of what syntax to expect when choosing one of these packages.

Note that the choice of a package will constrain the HTTP testing tools you can use. However, the general ideas will remain the same. You could switch your package backend from, say, `{crul}` to `{httr}` *without changing your tests*, if your tests do not test too many specificities of internals.

Chapter 3

Graceful HTTP R packages

Based on the previous chapter, your package interacting with a web resource has a dependency on `{curl}`, `{httr}`, `{httr2}` or `{crul}`. You have hopefully read the docs of the dependency you chose, including, in the case of `{httr}`, `{httr2}` and `{crul}`, the vignette about best practices for HTTP packages. Now, in this chapter we want to give more tips aimed at making your HTTP R package graceful, part of which you'll learn more about in this very book!

Why write a *graceful* HTTP R package? First of all, graceful is a nice adjective. ⍰⍰Second, graceful is the adjective used in CRAN repository policy *"Packages which use Internet resources should fail gracefully with an informative message if the resource is not available or has changed (and not give a check warning nor error)."* Therefore, let's review how to make your R package graceful from this day forward, in success and in failure.

3.1 Choose the HTTP resource wisely

First of all, your life and the life of your package's users will be easier if the web service you're wrapping is well maintained and well documented. When you have a choice, try not to rely on a fragile web service. Moreover, if you can, try to communicate with the API providers (telling them about your package; reporting feature requests and bug reports in their preferred way).

3.2 User-facing grace (how your package actually works)

0. If you can, do not request the API every time the user asks for something; cache data instead. No API call, no API call failure! ⊠ See the R-hub blog post "Caching the results of functions of your R package". To remember answers across sessions, see approaches presented in the R-hub blog post "Persistent config and data for R packages". Caching behavior should be well documented for users, and there should probably be an expiration time for caches that's based on how often data is updated on the remote service.
1. Try to send correct requests by knowing what the API expects and validating user inputs; at the correct rate.

- For instance, don't even try interacting with a web API requiring authentication if the user does not provide authentication information.
- For limiting rate (not sending too many requests), automatically wait. If the API docs allow you to define an ideal or maximal rate, set the request rate in advance using the ratelimitr package (or, with `{httr2}`, `httr2::req_throttle()`).

2. If there's a status API (a separate API indicating whether the web resource is up or down), use it. If it tells you the API is down, `stop()` (or `rlang::abort()`) with an informative error message.
3. If the API indicates an error, depending on the actual error,
    - If the *server* seems to be having issues, re-try with an exponential backoff. In `{httr2}` there is `httr2::req_retry()`.
    - Otherwise, transform the error into a useful error.
    - If you used retry and nothing was sent after the maximal number of retries, show an informative error message.

That was it for aspects the user will care about. Now, what might be more problematic for your package's fate on CRAN are the automatic checks that happen there at submission and then regularly.

3.3 Graceful vignettes and examples

4. Pre-compute vignettes in some way. Don't use them as tests; they are a showcase. Of course have a system to prevent them from going stale, maybe even simple reminders (potentially in the unexported `release_questions()` function). Don't let vignettes run on a system where a failure has bad consequences.

5. Don't run examples on CRAN. Now, for a first submission, CRAN maintainers might complain if there is no example. In that case, you might want to add some minimal example, e.g.

```
if (crul::ok("some-url")) {
  my_fun() # some eg that uses some-url
}
```

These two precautions ensure that CRAN checks won't end with some WARNINGs, e.g. because an example failed when the API was down.

3.4 Graceful code

For simplifying your own life and those of contributors, make sure to re-use code in your package by e.g. defining helper functions for making requests, handling responses etc. It will make it easier for you to support interactions with more parts of the web API. Writing DRY (don't repeat yourself) code means less lines of code to test, and less API calls to make or fake!

Also, were you to export a function à la `gh::gh()`, you'll help users call any endpoint of the web API even if you haven't written any high-level helper for it yet.

3.5 Graceful tests

We're getting closer to the actual topic of this book!

6. Read the rest of this book! Your tests should ideally run without needing an actual internet connection nor the API being up. Your tests that do need to interact with the API should be skipped on CRAN. `testthat::skip_on_cran()` (or `skip_if_offline()` that skips if the test is run offline or on CRAN) will ensure that.
7. Do not only test "success" behavior! Test for the behavior of your package in case of API errors, which shall also be covered later in the book.

3.6 Conclusion

In summary, to have a graceful HTTP package, make the most of current best practice for the user interface; escape examples and vignettes on CRAN; make tests independent of actual HTTP requests. Do not forget CRAN's "graceful failure" policy is mostly about ensuring a clean R CMD check result on CRAN platforms (0 ERROR, 0 WARNING, 0 NOTE) even when the web service you're wrapping has some hiccups.

Chapter 4

Packages for HTTP testing

A brief presentation of packages you'll "meet" again later in this book!

4.1 Why do we need special packages for HTTP testing?

Packages for HTTP testing are useful because there are challenges to HTTP testing. Packages for HTTP testing help you solve these challenges, rather than letting you solve them with some homegrown solutions (you can still choose to do that, of course).

What are the challenges of HTTP testing?

- Having tests depend on an internet connection is not ideal.
- Having tests depend on having secrets for authentication at hand is not ideal.
- Having tests for situations that are hard to trigger (e.g. the failure of a remote server) is tricky.

4.2 webmockr

`{webmockr}`, maintained by Scott Chamberlain, is an R package to help you "mock" HTTP requests. What does "mock" mean? Mock refers to the fact that we're faking the response. Here is how it works:

- You "stub" a request. That is, you set rules for what HTTP request you'd like to respond to with a fake response. E.g. a rule might be a method, or a URL.

- You also can set rules for what fake response you'd like to respond with, if anything (if nothing, then we give you `NULL`).
- Then you make HTTP requests, and those that match your stub i.e. set of rules will return what you requested be returned.
- While {webmockr} is in use, real HTTP interactions are not allowed. Therefore you need to stub all possible HTTP requests happening via your code. You'll get error messages for HTTP requests not covered by any stub.
- There is no recording interactions to disk at all, just mocked responses given as the user specifies in the R session.

{webmockr} works with {crul}, {httr}, and {httr2}.

{webmockr} is quite low-level and not the first tool you'll use directly in your day-to-day HTTP testing. You may never use it directly but if you use {vcr} it's one of its foundations.

{webmockr} was inspired by the Ruby webmock gem.

4.3 What is vcr?

The short version is {vcr}, maintained by Scott Chamberlain, helps you stub HTTP requests so you don't have to repeat HTTP requests, mostly in your unit tests. It uses the power of {webmockr}, with a higher level interface.

When using {vcr} in tests, the first time you run a test, the API response is stored in a YAML or JSON file. All subsequent runs of the test use that local file instead of really calling the API. Therefore tests work independently of an internet connection.

{vcr} was inspired by the Ruby vcr gem.

{vcr} works for packages using {httr}, {httr2} or {crul}.

Direct link to {vcr} (& {webmockr}) demo

4.4 What is httptest?

{httptest}, maintained by Neal Richardson, uses mocked API responses (like {vcr}). It *"enables one to test all of the logic on the R sides of the API in your package without requiring access to the remote service."*

Contrary to {vcr}, {httptest} also lets you define mock files by hand (copying from API docs, or dumbing down real responses), whereas with {vcr} all mock files come from recording real interactions (although you can choose to edit {vcr} mock files after recording).

{httptest} works for packages using {httr}.

Direct link to {httptest} demo

The differences and similarities between {httptest} and {vcr} will become clearer in the chapters where we provide the whole games for each of them.

4.5 What is httptest2?

{httptest2}, maintained by Neal Richardson, is like {httptest}, but for {httr2}.

Direct link to {httptest2} demo

With {vcr}, {httptest} and {httptest2} the tests will use some sort of fake API responses.

In {vcr} they are called **fixtures** or **cassettes**. In {httptest} and {httptest2} they are called **mock files**.

4.6 What is webfakes?

{webfakes}, maintained by Gábor Csárdi, provides an alternative (complementary?) tool for HTTP testing. It will let you fake a whole web service, potentially outputting responses from mock files you'll have created. It does not help with recording fake responses. Because it runs a fake web service, you can even interact with said web service in your browser or with curl in the command line.

{webfakes} works with packages using any HTTP package (i.e. it works with {curl}, {crul}, {httr}, or {httr2}).

Direct link to {webfakes} demo

4.7 testthat

{testthat}, maintained by Hadley Wickham, is not a package specifically for HTTP testing; it is a package for general-purpose unit testing of R packages. In this book we will assume that is what you use, because of its popularity.

If you use an alternative like {tinytest},

- {httptest} won't work as it's specifically designed as a complement to {testthat};
- {vcr} might work;
- {webfakes} can work.

4.8 Conclusion

Now that you have an idea of the tools we can use for HTTP testing, we'll now create a minimal package and then amend it in three versions tested with

- {vcr} and {webmockr};
- {httptest};
- {httptest2};
- {webfakes}.

Our minimal package will use {httr} (except for {httptest2}, where we'll use {httr2}). However, it will help you understand concepts even if you end up using {crul} or {curl}.[1]

[1] If you end up using {crul}, you can use {vcr} and {webmockr}; or {webfakes}; but not {httptest}. If you end up using {curl} you can only use {webfakes}.

Part II

Whole Game(s)

Chapter 5

Introduction

Similar to the Whole Game chapter in the R packages book by Hadley Wickham and Jenny Bryan, we will go through how to add HTTP tests to a minimal package. However, we will do it *four* times to present alternative approaches: with vcr, httptest, httptest2, webfakes. After that exercise, we will compare approaches: we will compare both packages that involve mocking i.e. vcr vs. httptest; and all three HTTP packages in a last chapter. The next section will then present single topics such as "how to deal with authentication" in further details.

5.1 Our example packages

Our minimal packages, `exemplighratia` and `exemplighratia2`, access the GitHub status API and one endpoint of GitHub V3 REST API. They are named after the Latin phrase *exempli gratia* that means "for instance", with an H for GH. If you really need to interact with GitHub V3 API, we recommend the gh package. We also recommend looking at the source of the gh package, and at the docs of GitHub V3 API, in particular about authentication.

Our example packages call web *APIs* but the tools and concepts are applicable to packages wrapping any web resource, even poorly documented ones.[1]

GitHub V3 API works without authentication too, but at a lower rate. For the sake of having an example of a package *requiring* authentication we will assume the API is *not* usable without authentication. Authentication is, here, the setting of a token in a HTTP header (so quite simple, compared to e.g. OAuth).

GitHub Status API, on the contrary, does not necessitate authentication at all.

[1] An interesting post to read about an R package wrapping an undocumented web API is "One-Hour Package" by Neal Richardson.

So we will create two functions, one that works without authentication, one that works with authentication.

How did we create the packages? You are obviously free to use your own favorite workflow tools, but below we share our workflow using the usethis package.

- We followed usethis setup article.

5.1.1 exemplighratia2 (httr2)

Then we ran

- `usethis::create_package("path/to/folder/exemplighratia2")` to create and open the package project;
- `usethis::use_mit_license()` to add an MIT license;
- `usethis::use_package("httr2")` to add a dependency on httr2;
- `usethis::use_package("purrr")` to add a dependency on purrr;
- `use_r("api-status.R")` to add the first function whose code is written below;

```
status_url <- function() {
  "https://kctbh9vrtdwd.statuspage.io/api/v2/components.json"
}

#' GitHub APIs status
#'
#' @description Get the status of requests to GitHub APIs
#'
#' @importFrom magrittr `%>%`
#'
#' @return A character vector, one of "operational", "degraded_performance",
#' "partial_outage", or "major_outage."
#'
#' @details See details in https://www.githubstatus.com/api#components.
#' @export
#'
#' @examples
#' \dontrun{
#' gh_api_status()
#' }
gh_api_status <- function() {
  response <- status_url() %>%
    httr2::request() %>%
    httr2::req_perform()
```

```
  # Check status
  httr2::resp_check_status(response)

  # Parse the content
  content <- httr2::resp_body_json(response)

  # Extract the part about the API status
  components <- content$components
  api_status <- components[purrr::map_chr(components, "name") == "API Requests"][[1]]

  # Return status
  api_status$status

}
```

- `use_test("api-status")` (and using testthat latest edition so setting `Config/testthat/edition: 3` in `DESCRIPTION`) to add a simple test whose code is below.

```
test_that("gh_api_status() works", {
  testthat::expect_type(gh_api_status(), "character")
})
```

- `use_r("organizations.R")` (and `usethis::use_package("cli")`) to add a second function. Note that an ideal version of this function would have some sort of callback in the retry, to call the `gh_api_status()` function (maybe with `httr2::req_retry()`'s `is_transient` argument).

```
gh_v3_url <- function() {
  "https://api.github.com/"
}

#' GitHub organizations
#'
#' @description Get logins of GitHub organizations.
#'
#' @param since The integer ID of the last organization that you've seen.
#'
#' @return A character vector of at most 30 elements.
#' @export
#'
#' @details Refer to https://developer.github.com/v3/orgs/#list-organizations
#'
```

```
#' @examples
#' \dontrun{
#' gh_organizations(since = 42)
#' }
gh_organizations <- function(since = 1) {

  if (!gh::gh_token_exists()) {
    cli::cli_abort("No token provided! {.url https://usethis.r-lib.org/articles/git-cre
  }

  token <- gh::gh_token()

  response <- httr2::request(gh_v3_url()) %>%
    httr2::req_url_path_append("organizations") %>%
    httr2::req_url_query(since = since) %>%
    httr2::req_headers("Authorization" = paste("token", token)) %>%
    httr2::req_retry(max_tries = 3, max_seconds = 120) %>%
    httr2::req_perform()

  content <- httr2::resp_body_json(response)

  purrr::map_chr(content, "login")

}
```

- `use_test("organizations")` to add a simple test.

```
test_that("gh_organizations works", {
  expect_type(gh_organizations(), "character")
})
```

5.1.2 exemplighratia (httr)

Then we ran

- `usethis::create_package("path/to/folder/exemplighratia")` to create and open the package project;
- `usethis::use_mit_license()` to add an MIT license;
- `usethis::use_package("httr")` to add a dependency on httr;
- `usethis::use_package("purrr")` to add a dependency on purrr;
- `usethis::use_r("api-status.R")` to add the first function whose code is written below;

```
status_url <- function() {
  "https://kctbh9vrtdwd.statuspage.io/api/v2/components.json"
}

#' GitHub APIs status
#'
#' @description Get the status of requests to GitHub APIs
#'
#' @return A character vector, one of "operational", "degraded_performance",
#' "partial_outage", or "major_outage."
#'
#' @details See details in https://www.githubstatus.com/api#components.
#' @export
#'
#' @examples
#' \dontrun{
#' gh_api_status()
#' }
gh_api_status <- function() {
  response <- httr::GET(status_url())

  # Check status
  httr::stop_for_status(response)

  # Parse the content
  content <- httr::content(response)

  # Extract the part about the API status
  components <- content$components
  api_status <- components[purrr::map_chr(components, "name") == "API Requests"][[1]]

  # Return status
  api_status$status

}
```

- `use_test("api-status")` (and using testthat latest edition so setting `Config/testthat/edition: 3` in `DESCRIPTION`) to add a simple test whose code is below.

```
test_that("gh_api_status() works", {
  testthat::expect_type(gh_api_status(), "character")
})
```

- `usethis::use_r("organizations.R")` to add a second function. Note that

an ideal version of this function would have some sort of callback in the retry, to call the `gh_api_status()` function (which seems easier to implement with crul's retry method).

```
gh_v3_url <- function() {
  "https://api.github.com/"
}

#' GitHub organizations
#'
#' @description Get logins of GitHub organizations.
#'
#' @param since The integer ID of the last organization that you've seen.
#'
#' @return A character vector of at most 30 elements.
#' @export
#'
#' @details Refer to https://developer.github.com/v3/orgs/#list-organizations
#'
#' @examples
#' \dontrun{
#' gh_organizations(since = 42)
#' }
gh_organizations <- function(since = 1) {
  url <- httr::modify_url(
    gh_v3_url(),
    path = "organizations",
    query = list(since = since)
    )

  token <- Sys.getenv("GITHUB_PAT")

  if (!nchar(token)) {
    stop("No token provided! Set up the GITHUB_PAT environment variable please.")
  }

  response <- httr::RETRY(
    "GET",
    url,
    httr::add_headers("Authorization" = paste("token", token))
  )

  httr::stop_for_status(response)

  content <- httr::content(response)
```

```
  purrr::map_chr(content, "login")

}
```

- `use_test("organizations")` to add a simple test.

```
test_that("gh_organizations works", {
  testthat::expect_type(gh_organizations(), "character")
})
```

5.2 Conclusion

All good, now our package has 100% test coverage and passes R CMD Check (granted, our tests could be more thorough, but remember this is a minimal example). But what if we try working without a connection? In the following chapters, we'll add more robust testing infrastructure to this minimal package, and we will do that *four* times to compare packages/approaches: once with vcr, once with httptest, once with httptest2, and once with webfakes.

Chapter 6

Use vcr (& webmockr)

In this chapter we aim at adding HTTP testing infrastructure to exemplighratia2 using [vcr(#what-vcr)] (& webmockr).

Corresponding pull request to exemplighratia2. Feel free to fork the repository to experiment yourself!

6.1 Setup

Before working on all this, we need to install `{vcr}`.

First, we need to run `usethis::use_package("vcr", type = "Suggests")` (in the exemplighratia directory) to add vcr as a dependency to `DESCRIPTION`, under Suggests just like testthat.

Then we need to create a helper file using `usethis::use_test_helper("vcr")`. A new file is created under `tests/testthat/helper-vcr.R`.

When testthat runs tests, files whose name start with "helper" are always run first. They are also loaded by `devtools::load_all()`, so the vcr setup is loaded when developing and testing interactively. See the table in the R-hub blog post "Helper code and files for your testthat tests".

We have to tweak the vcr setup for our needs.

- We need to ensure that we set up a fake API key when there is no API token around. Why? Because if you remember well, the code of our function `gh_organizations()` checks for the presence of a token. With mock responses around, we don't need a token but we still need to fool our own package in contexts where there is no token (e.g. in continuous integration checks for a fork of a GitHub repository).

- We do not need to tweak the configuration to prevent our API from leaking, because with vcr 2.0.0 and above, the `Authorization` header is never recorded.

Below is the updated setup file saved under `tests/testthat/helper-vcr.R`.

```
vcr_dir <- vcr::vcr_test_path("_vcr")
if (!gh::gh_token_exists()) {
  if (dir.exists(vcr_dir)) {
    # Fake API token to fool our package
    Sys.setenv("GITHUB_PAT" = "foobar")
  } else {
    # If there's no mock files nor API token, impossible to run tests
    stop("No API key nor cassettes, tests cannot be run.", call. = FALSE)
  }
}
```

So this was just setup, now on to adapting our tests!

6.2 Actual testing

The most important function will be `vcr::local_cassette("cassette-informative-and-unique-name")` which tells vcr to create a mock file to store all API responses for API calls occurring in the current test (what's inside `test_that()`). The `vcr::local_cassette()` function might remind you of the `withr::local_` functions or of `rlang::local_options()`.

Let's tweak the test for `gh_api_status`, it now becomes

```
test_that("gh_api_status() works", {
  vcr::local_cassette("gh_api_status")
  status <- gh_api_status()
  expect_type(status, "character")
})
```

If you had used `vcr::use_cassette()` instead of the newer `vcr::local_cassette()`, you might notice the code feels less nested!

If we run this test, for instance through `devtools::test_active_file()`,

- the first time, vcr creates a cassette (mock file) under `tests/testthat/_vcr/gh_api_status.yml` where it stores the API response It contains all the information related to requests and responses, headers included.

```yaml
http_interactions:
- request:
    method: GET
    uri: https://kctbh9vrtdwd.statuspage.io/api/v2/components.json
  response:
    status: 200
    headers:
      content-type: application/json; charset=utf-8
      date: Tue, 09 Sep 2025 10:09:28 GMT
      x-download-options: noopen
      x-permitted-cross-domain-policies: none
      referrer-policy: strict-origin-when-cross-origin
      x-statuspage-version: b27f95704d5835d5f1a3ccc341503d3f7f22f502
      strict-transport-security: max-age=259200
      x-statuspage-skip-logging: 'true'
      access-control-allow-origin: '*'
      cache-control: max-age=3, public
      x-pollinator-metadata-service: status-page-web-pages
      etag: W/"81603fa0d180d761ce3b3c3c02d68741"
      x-runtime: '0.049065'
      server: AtlassianEdge
      accept-ranges: bytes
      x-content-type-options: nosniff
      x-xss-protection: 1; mode=block
      atl-traceid: daa0d6503446464e9748ed3a8634ead0
      atl-request-id: daa0d650-3446-464e-9748-ed3a8634ead0
      report-to: '{"endpoints": [{"url": "https://dz8aopenkvv6s.cloudfront.net"}],
        "group": "endpoint-1", "include_subdomains": true, "max_age": 600}'
      nel: '{"failure_fraction": 0.001, "include_subdomains": true, "max_age": 600,
        "report_to": "endpoint-1"}'
      content-encoding: br
      server-timing: atl-edge;dur=129,atl-edge-internal;dur=5,atl-edge-upstream;dur=125,atl-edge-
      vary: Accept,Accept-Encoding
      x-cache: Miss from cloudfront
      via: 1.1 dd239fa6f06e5b3ae1437460dbc3d6a6.cloudfront.net (CloudFront)
      x-amz-cf-pop: MRS53-P3
      x-amz-cf-id: K63D5rR4HMl9WRamA6Yzh8R6SuF2vfDOuh8W-bgc1kUo-xGdrVZp9g==
    body:
      string: '{"page":{"id":"kctbh9vrtdwd","name":"GitHub","url":"https://www.githubstatus.com",
        Operations","status":"operational","created_at":"2017-01-31T20:05:05.370Z","updated_at":"
        of git clones, pulls, pushes, and associated operations","showcase":false,"start_date":nu
        time HTTP callbacks of user-generated and system events","showcase":false,"start_date":nu
        www.githubstatus.com for more information","status":"operational","created_at":"2018-12-0
        Requests","status":"operational","created_at":"2017-01-31T20:01:46.621Z","updated_at":"20
        for GitHub APIs","showcase":false,"start_date":null,"group_id":null,"page_id":"kctbh9vrtd
```

```
        for Issues on GitHub.com","showcase":false,"start_date":null,"group_id":null,"
        Requests","status":"operational","created_at":"2020-09-02T15:39:06.329Z","upda
        for Pull Requests on GitHub.com","showcase":false,"start_date":null,"group_id"
        Compute and Orchestration for GitHub Actions","showcase":false,"start_date":nu
        requests and webhook delivery for GitHub Packages","showcase":false,"start_dat
        application and API servers for Pages builds","showcase":false,"start_date":nu
        and Compute for GitHub Codespaces","showcase":false,"start_date":"2021-08-11",
  recorded_at: 2025-09-09 10:09:28
recorded_with: VCR-vcr/2.0.0
```

- all the times after that, unless we delete the cassette (mock file), vcr simply uses the cassette instead of actually calling the API.

Let's tweak our other test, of `gh_organizations()`. Here things get more exciting or complicated, as we also set out to adding a test of the error behavior. This inspired us to change error behavior a bit with a slightly more specific error message i.e. `httr::stop_for_status(response)` became `httr::stop_for_status(response, task = "get data from the API, oops")`.

The test file `tests/testthat/test-organizations.R` is now:

```
test_that("gh_organizations works", {
  vcr::local_cassette("gh_organizations")
  orgs <- gh_organizations()
  expect_type(orgs, "character")
})

test_that("gh_organizations errors when the API doesn't behave", {
  webmockr::enable()
  stub <- webmockr::stub_request(
    "get",
    "https://api.github.com/organizations?since=1"
  )
  webmockr::to_return(stub, status = 502)
  expect_snapshot(error = TRUE, gh_organizations())
  webmockr::disable()
})
```

The first test is similar to what we did for `gh_api_status()`. In the second test there is more to unpack.

- We enable the use of {`webmockr`} at the beginning with `webmockr::enable()`. Why webmockr? Because it can help mock a failure scenario.

- We explicitly write that a request to `https://api.github.com/organizations?since=1` should return a status of 502.

```
stub <- webmockr::stub_request("get", "https://api.github.com/organizations?since=1")
webmockr::to_return(stub, status = 502)
```

- We then test for the error message with `expect_snapshot(error = TRUE, gh_organizations())`.
- We disable webmockr with `webmockr::disable()`.

Instead of using webmockr for creating a fake API eror, we could have

- recorded a normal cassette;
- edited it to replace the status code.

Read pros and cons of this approach in the vcr vignette *Why and how edit your vcr cassettes?*, especially if you don't find the webmockr approach enjoyable.

Without the HTTP testing infrastructure, testing for behavior of the package in case of API errors would be more difficult. However, we could resort to testthat's mocking tools.

Regarding our secret API token, the first time we run the test file, vcr creates a cassette where we *do not see our token*.

6.3 Also testing for real interactions

What if the API responses change? Hopefully we'd notice that thanks to following API news. However, sometimes web APIs change without any notice. Therefore it is important to run tests against the real web service once in a while.

The vcr package provides various methods to turn vcr use on and off to allow real requests i.e. ignoring mock files. See `?vcr::lightswitch`.

In the case of exemplighratia2, we added a GitHub Actions workflow that will run on schedule once a week, for which one of the build has vcr turned off via the `VCR_TURN_OFF` environment variable. We chose to have one build with vcr turned on and otherwise the same configuration to make it easier to assess what broke in case of failure (if both builds fail, the web API is probably not the culprit). Compared to continuous integration builds where vcr is turned on, this one build needs to have access to a `GITHUB_PAT` secret environment variable. Furthermore, it is slower.

One could imagine other strategies:

- Always having one continuous integration build with vcr turned off but skipping it in contexts where there isn't any token (pull requests from forks for instance?);
- Only running tests with vcr turned off locally once in a while.

6.4 Summary

- We set up vcr usage in our package exemplighratia2 by registering vcr as a dependency and creating a file to protect our secret API key and to fool our own package that needs an API token.
- Inside `test_that()` blocks, we call `vcr::local_cassette(<cassette-name>)` and ran the tests a first time to generate mock files that hold all information about the API interactions.
- In one of the tests, we used webmockr to create an environment where only fake requests are allowed. We defined that the request that `gh_organizations()` makes should get a 502 status. We were therefore able to test for the error message `gh_organizations()` returns in such cases.

Now, how do we make sure this works?

- Turn off wifi, run the tests again. It works! Turn on wifi again.
- Open .Renviron (`usethis::edit_r_environ()`), edit "GITHUB_PAT" into "byeGITHUB_PAT", re-start R, run the tests again. It works! Fix your "GITHUB_PAT" token in .Renviron. (to store your credentials, you should actually use gitcreds for Git credentials, and keyring for other credentials).

So we now have tests that no longer rely on an internet connection nor on having API credentials.

We also added a continuous integration workflow for having a build using real interactions once every week, as it is important to regularly make sure the package still works against the latest API responses.

For the full list of changes applied to exemplighratia in this chapter, see the pull request diff on GitHub.

How do we get there with other packages? Let's try httptest in the next chapter!

6.5 PS: Where to put use_cassette()

Where do we put the `vcr::use_cassette()` call? Well, as written in the manual page of that function, *There's a few ways to get correct line numbers for failed tests and one way to not get correct line numbers:* What's correct?

- Wrapping the whole `testthat::test_that()` call (do not do that if your test contains for instance 'skip_on_cran()");

```
vcr::use_cassette("thing", {
  testthat::test_that("thing", {
    lala <- get_foo()
    expect_true(lala)
  })
})
```

- Wrapping a few lines inside `testthat::test_that()` **excluding the expectations** `expect_blabla()`

```
testthat::test_that("thing", {
  vcr::use_cassette("thing", {
    lala <- get_foo()
  })
    expect_true(lala)
})
```

What's incorrect?

```
testthat::test_that("thing", {
  vcr::use_cassette("thing", {
    lala <- get_foo()
    expect_true(lala)
  })
})
```

We used the solution of only wrapping the lines containing API calls in `vcr::use_cassette()`, but it is up to you to choose what you prefer.

Chapter 7

Use httptest

In this chapter we aim at adding HTTP testing infrastructure to exemplighratia using httptest. For this, we start from the initial state of exemplighratia again. Back to square one!

Note that the `httptest::with_mock_dir()` function is only available in httptest version >= 4.0.0 (released on CRAN on 2021-02-01).

Corresponding pull request to exemplighratia. Feel free to fork the repository to experiment yourself!

7.1 Setup

Before working on all this, we need to install `{httptest}`.

First, we need to run `httptest::use_httptest()` which has a few effects:

- Adding httptest as a dependency to `DESCRIPTION`, under Suggests just like testthat.
- Creating a setup file under `tests/testthat/setup`,

```
library(httptest)
```

When testthat runs tests, files whose name starts with "setup" are always run first. The setup file added by httptest loads httptest.

We will tweak it a bit to fool our package into believing there is an API token around in contexts where there is not. Since tests will use recorded responses when we are not recording, we do not need an actual API token when not recording, but we need `gh_organizations()` to not stop because `Sys.getenv("GITHUB_PAT")` returns nothing.

```
library(httptest)

# for contexts where the package needs to be fooled
# (CRAN, forks)
# this is ok because the package will used recorded responses
# so no need for a real secret
if (!nzchar(Sys.getenv("GITHUB_PAT"))) {
  Sys.setenv(GITHUB_PAT = "foobar")
}
```

So this was just setup, now on to adapting our tests!

7.2 Actual testing

The key function will be `httptest::with_mock_dir("dir", {code-block})` which tells httptest to create mock files under `tests/testthat/dir` to store all API responses for API calls occurring in the code block. We are allowed to tweak the mock files by hand, and we will do that in some cases.

Let's tweak the test file for `gh_status_api`, it becomes

```
with_mock_dir("gh_api_status", {
  test_that("gh_api_status() works", {
    expect_type(gh_api_status(), "character")
    expect_equal(gh_api_status(), "operational")
  })
})
```

We only had to wrap the whole test in `httptest::with_mock_dir()`.

If we run this test (in RStudio clicking on "Run test"),

- the first time, httptest creates a mock file under `tests/testthat/gh_api_status/kctbh9vrtdwd.sta` where it stores the API response. We however dumbed it down by hand, to

```
{"components":[{"name":"API Requests","status":"operational"}]}
```

- all the times after that, httptest simply uses the mock file instead of actually calling the API.

Let's tweak our other test, of `gh_organizations()`.

Here things get more exciting or complicated, as we also set out to adding a test of the error behavior. This inspired us to change error behavior a bit with a slightly

more specific error message i.e. `httr::stop_for_status(response)` became `httr::stop_for_status(response, task = "get data from the API, oops")`.

The test file `tests/testthat/test-organizations.R` is now:

```
with_mock_dir("gh_organizations", {
  test_that("gh_organizations works", {
    expect_type(gh_organizations(), "character")
  })
})

with_mock_dir("gh_organizations_error", {
  test_that("gh_organizations errors if the API doesn't behave", {
    expect_error(gh_organizations())
  })
},
simplify = FALSE)
```

The first test is similar to what we did for `gh_api_status()` except we didn't touch the mock file this time, out of laziness. In the second test there is more to unpack: how do we get a mock file corresponding to an error?

- We first run the test as is. It fails because there is no error, which we expected. Note the `simplify = FALSE` that means the mock file also contains headers for the response.
- We replaced `200L` with `502L` and removed the body, to end up with a very simple mock file under `tests/testthat/gh_organizations_error/api.github.com/organizations-5377e8.R`

```
structure(list(
    url = "https://api.github.com/organizations?since=1",
    status_code = 502L,
    headers = NULL),
  class = "response")
```

- We re-run the tests. We got the expected error message.

Without the HTTP testing infrastructure, testing for behavior of the package in case of API errors would be more difficult.

Regarding our secret API token, since httptest doesn't save the requests, and since the responses don't contain the token, it is safe without our making any effort.

In this demo we used `httptest::with_mock_dir()` but there are other ways to use httptest, e.g. using `httptest::with_mock_api()` that does not require naming a directory (you'd still need to use a separate directory for mocking the error response).

Find out more in the main httptest vignette.

7.3 Also testing for real interactions

What if the API responses change? Hopefully we'd notice that thanks to following API news. However, sometimes web APIs change without any notice. Therefore it is important to run tests against the real web service once in a while.

As with vcr we setup a GitHub Actions workflow that runs once a week with tests against the real web service. The difference is what and where these tests are. As some tests with custom made mock files can be more specific (e.g. testing for actual values, whereas the latest responses from the API will have different values), instead of turning off mock files usage, we use our old original tests that we put in a folder called `real-tests`. Most of the time `real-tests` is .Rbuildignored but in the scheduled run, before checking the package we replace the content of `tests` with `real-tests`. An alternative would be to use `testthat::test_dir()` on that directory but in case of failures we would not get artifacts as we do with `R CMD check` (at least not without further effort).

Again, one could imagine other strategies, but in all cases it is important to keep checking the package against the real web service fairly regularly.

7.4 Summary

- We set up httptest usage in our package exemplighratia by running `use_httptest()` and tweaking the setup file to fool our own package that needs an API token.
- We wrapped `test_that()` into `httptest::with_mock_dir()` and ran the tests a first time to generate mock files that hold all information about the API responses. In some cases we modified these mock files to make them smaller or to make them correspond to an API error.

Now, how do we make sure this works?

- Turn off wifi, run the tests again. It works! Turn on wifi again.
- Open .Renviron (`usethis::edit_r_environ()`), edit "GITHUB_PAT" into "byeGITHUB_PAT", re-start R, run the tests again. It works! Fix your "GITHUB_PAT" token in .Renviron.

So we now have tests that no longer rely on an internet connection nor on having API credentials.

We also added a continuous integration workflow for having a build using real interactions once every week, as it is important to regularly make sure the package still works against the latest API responses.

For the full list of changes applied to exemplighratia in this chapter, see the pull request diff on GitHub.

How do we get there with yet another package? We'll try webfakes but first let's compare vcr and httptest as they both use mocking.

Chapter 8

vcr and httptest

We have just followed very similar processes to add HTTP testing infrastructure involving mock files to exemplighratia

- Adding a package as a Suggests dependency;
- Creating a helper file that in particular loads this package before each test;
- Tweaking tests, in some cases wrapping our tests into functions that allows to record API responses in mock files and to play them back from said mock files; in other cases (only with httptest), creating mock files ourselves.

Now, there were a few differences. We won't end up advocating for one package in particular since both have their merits, but we do hope to help you differentiate the two packages.

8.1 Setting up the infrastructure

To set up the HTTP testing infrastructure, in one case you need to run `vcr::use_vcr()` and in another case you need to run `httptest::use_httptest()`. Not too hard to remember.

8.2 Calling mock files

As mentioned before, vcr and httptest both use mock files but they call them differently.

In vcr they are called both **fixtures** and **cassettes**. In httptest they are called **mock files**. Note that fixtures is not as specific as cassettes and mock files: cassettes and

mock files are fixtures, but anything (a csv file of input for instance) you use to consistently test your package is a fixture.

8.3 Naming mock files

With vcr the `use_cassette()` call needs to include a name that will be used to create the filename of the mock file. The help of `?use_cassette` explains some criteria for naming them, such as the fact that cassette names need to be unique. Now if you wrap your whole `test_that()` block in them you might just as well use a name similar to the test name, and you already make those meaningful, right?

With httptest the mock filepaths are translated from requests according to several rules that incorporate the request method, URL, query parameters, and body. If you use `with_mock_dir()` you need a name for the directory under which the mock files are saved, and you can make it meaningful.

Also note that with vcr one file can (but does not have to) contain several HTTP interactions (requests and responses) whereas with httptest one file contains one response only (and the filename helps matching it to a request).

8.4 Matching requests

With httptest as the mock file name includes everything that's potentially varying about a request, each mock file corresponds to one request only.

With vcr, there are different possible configurations for matching a request to a saved interaction but by default you can mostly expect that one saved interaction corresponds to one request only.

8.5 Handling secrets

With vcr, since everything from the HTTP interactions is recorded, you always need to add some sort of configuration to be sure to wipe your API tokens from the mock files.

With httptest, only responses are saved, and most often, only their bodies. Most often, responses don't contain secrets e.g. they don't contain your API token. If the response contains secrets, refer to httptest's article about "Redacting sensitive information".

8.6 Recording, playing back

When using mock files for testing, first you need to record responses in mock files; and then you want to use the mock files instead of real HTTP interactions (that's the whole point).

With vcr, the recording vs playing back modes happen automatically depending on the existence of the cassette. If you write `vcr::use_cassette("blabla", )` and there's no cassette called blabla, vcr will create it. Note that if you change the HTTP interactions in the code block, you'll have to re-record the cassette which is as simple as deleting it then running the test. *Note that you can also change the way vcr behaves by looking into `?vcr::vcr_configure`'s "Cassette Options".*

With httptest, there is a lot of flexibility around how to record mock files. It is because httptest doesn't assume that every API mock came from a real request to a real server; maybe you copy some of the mocks directly from the API docs.

Note that nothing prevents you from editing vcr cassettes by hand, but you'll have to be careful not re-recording them by mistake.

httptest flexiblity comes from original design principles of httptest

> *"[httptest] doesn't assume that every API mock came from a real request to a real server, and it is designed so that you are able to see and modify test fixtures. Among the considerations:*
>
> *1. In many cases, API responses contain way more content than is necessary to test your R code around them: 100 records when 2 will suffice, request metadata that you don't care about and can't meaningfully assert things about, and so on. In the interest of minimally reproducible examples, and of making tests readable, it often makes sense to take an actual API response and delete a lot of its content, or even to fabricate one entirely.*
>
> *2. And then it's good to keep that API mock fixed so you know exactly what is in it. If I re-recorded a Twitter API response of, say, the most recent 10 tweets with #rstats, the specific content will change every time I record it, so my tests can't say much about what is in the response without having to rewrite them every time too.*
>
> *3. Some conditions (rate limiting, server errors, e.g.) are difficult to test with real responses, but if you can hand-create a API mock with, say, a 503 response status code and test how your code handles it, you can have confidence of how your package will respond when that rare event happens with the real API.*
>
> *4. Re-recording all responses can make for a huge code diff, which can blow up your repository size and make code review harder."*

Now, creating mock files by hand (or inventing some custom scripts to create them) involves more elbow grease, so it's a compromise.

8.7 Testing for API errors

In your test suite you probably want to check how things go if the server returns 502 or so, and you cannot trigger such a response to record it.

With httptest, to test for API errors, you need to create one or several fake mock file(s). The easiest way to do that might be to use `httptest::with_mock_dir()` that will create mock files with the expected filenames and locations, that you can then tweak. Or reading the error message of `httptest::with_mock_ap()` helps you know where to create a mock file.

With vcr, you either

- use webmockr as we showed in our demo. On the one hand it's more compact than creating a fake mock file, on the other hand it's a way to test that's different from the vcr cassette.

```
test_that("gh_organizations errors when the API doesn't behave", {
  webmockr::enable()
  stub <- webmockr::stub_request("get", "https://api.github.com/organizations?since=1")
  webmockr::to_return(stub, status = 502)
  expect_error(gh_organizations(), "oops")
  webmockr::disable()
})
```

- or you edit a cassette by hand which would be similar to testing for API errors with httptest. If you did that, you'd need to skip the test when vcr is off, as when vcr is off real requests are made. For that you can use `vcr::skip_if_vcr_off()`.

8.8 Conclusion

Both vcr and httptest are similar packages in that they use mock files for allowing easier HTTP testing. They are a bit different in their design philosophy and features, which might help you choose one of them.

And now, to make things even more complex, or fun, we shall explore a third HTTP testing package that does not *mock* requests but instead spins up a local fake web service.

Chapter 9

Use httptest2

In this chapter we aim at adding HTTP testing infrastructure to exemplighratia2 using httptest2. For this, we start from the initial state of exemplighratia2 again. Back to square one!

Corresponding pull request to exemplighratia2 Feel free to fork the repository to experiment yourself!

9.1 Setup

Before working on all this, we need to install `{httptest2}`.

First, we need to run `httptest2::use_httptest2()` which has a few effects:

- Adding httptest2 as a dependency to `DESCRIPTION`, under Suggests just like testthat.
- Creating a setup file under `tests/testthat/setup`,

```
library(httptest2)
```

When testthat runs tests, files whose name starts with "setup" are always run first. The setup file added by httptest2 loads httptest2.

We shall tweak it a bit to fool our package into believing there is an API token around in contexts where there is not. Since tests will use recorded responses when we are not recording, we do not need an actual API token when not recording, but we need `gh_organizations()` to not stop because `Sys.getenv("GITHUB_PAT")` returns nothing.

```
library(httptest2)

# for contexts where the package needs to be fooled
# (CRAN, forks)
# this is ok because the package will used recorded responses
# so no need for a real secret
if (!nzchar(Sys.getenv("GITHUB_PAT"))) {
  Sys.setenv(GITHUB_PAT = "foobar")
}
```

So this was just setup, now on to adapting our tests!

9.2 Actual testing

The key function will be `httptest2::with_mock_dir("dir", {code-block})` which tells httptest to create mock files under `tests/testthat/dir` to store all API responses for API calls occurring in the code block. We are allowed to tweak the mock files by hand, and we will do that in some cases.

Let's tweak the test file for `gh_status_api`, it becomes

```
with_mock_dir("gh_api_status", {
  test_that("gh_api_status() works", {
    testthat::expect_type(gh_api_status(), "character")
    testthat::expect_equal(gh_api_status(), "operational")
  })
})
```

We only had to wrap the whole test in `httptest2::with_mock_dir()`.

If we run this test (in RStudio clicking on "Run test"),

- the first time, httptest2 creates a mock file under `tests/testthat/gh_api_status/kctbh9vrtdwd.st` where it stores the API response. We however dumbed it down by hand, to

```
{"components":[{"name":"API Requests","status":"operational"}]}
```

- all the times after that, httptest2 simply uses the mock file instead of actually calling the API.

Let's tweak our other test, of `gh_organizations()`.

Here things get more exciting or complicated, as we also set out to adding a test of the error behavior.

The test file `tests/testthat/test-organizations.R` is now:

```
with_mock_dir("gh_organizations", {
  test_that("gh_organizations works", {
    testthat::expect_type(gh_organizations(), "character")
  })
})

with_mock_dir("gh_organizations_error", {
  test_that("gh_organizations errors if the API doesn't behave", {
    testthat::expect_snapshot_error(gh_organizations())
  })
},
simplify = FALSE)
```

The first test is similar to what we did for `gh_api_status()` except we didn't touch the mock file this time, out of laziness. In the second test there is more to unpack: how do we get a mock file corresponding to an error?

- We first run the test as is. It fails because there is no error, which we expected. Note the `simplify = FALSE` that means the mock file also contains headers for the response.
- We replaced `200L` with `502L` and removed the body, to end up with a simpler mock file under `tests/testthat/gh_organizations_error/api.github.com/organizations-5377e8.R`

```
structure(list(method = "GET", url = "https://api.github.com/organizations?since=1",
    status_code = 502L, headers = structure(list(server = "GitHub.com",
        date = "Thu, 17 Feb 2022 12:40:29 GMT", `content-type` = "application/json; charset=utf-8
        `cache-control` = "private, max-age=60, s-maxage=60",
        vary = "Accept, Authorization, Cookie, X-GitHub-OTP",
        etag = "W/\"d56e867402a909d66653b6cb53d83286ba9a16eef993dc8f3cb64c43b66389f4\"",
        `x-oauth-scopes` = "gist, repo, user, workflow", `x-accepted-oauth-scopes` = "",
        `x-github-media-type` = "github.v3; format=json", link = "<https://api.github.com/organiz
        `x-ratelimit-limit` = "5000", `x-ratelimit-remaining` = "4986",
        `x-ratelimit-reset` = "1645104327", `x-ratelimit-used` = "14",
        `x-ratelimit-resource` = "core", `access-control-expose-headers` = "ETag, Link, Location,
        `access-control-allow-origin` = "*", `strict-transport-security` = "max-age=31536000; inc
        `x-frame-options` = "deny", `x-content-type-options` = "nosniff",
        `x-xss-protection` = "0", `referrer-policy` = "origin-when-cross-origin, strict-origin-wh
        `content-security-policy` = "default-src 'none'", vary = "Accept-Encoding, Accept, X-Requ
        `content-encoding` = "gzip", `x-github-request-id` = "A4BA:12D5C:178438:211160:620E423C")
    body = charToRaw("")), class = "httr2_response")
```

- We re-run the tests. We got the expected error message.

Without the HTTP testing infrastructure, testing for behavior of the package in case of API errors would be more difficult.

Regarding our secret API token, since httptest2 doesn't save the requests[1], and since the responses don't contain the token, it is safe without our making any effort.

In this demo we used `httptest2::with_mock_dir()` but there are other ways to use httptest2, e.g. using `httptest2::with_mock_api()` that does not require naming a directory (you'd still need to use a separate directory for mocking the error response).

Find out more in the main httptest2 vignette.

9.3 Also testing for real interactions

What if the API responses change? Hopefully we'd notice that thanks to following API news. However, sometimes web APIs change without any notice. Therefore it is important to run tests against the real web service once in a while.

One could use the same strategy as the one we demonstrated for httptest i.e. with a different test folder.

Again, one could imagine other strategies, but in all cases it is important to keep checking the package against the real web service fairly regularly.

9.4 Summary

- We set up httptest2 usage in our package exemplighratia by running `use_httptest2()` and tweaking the setup file to fool our own package that needs an API token.
- We wrapped `test_that()` into `httptest2::with_mock_dir()` and ran the tests a first time to generate mock files that hold all information about the API responses. In some cases we modified these mock files to make them smaller or to make them correspond to an API error.

Now, how do we make sure this works?

- Turn off wifi, run the tests again. It works! Turn on wifi again.
- Open .Renviron (`usethis::edit_r_environ()`), edit "GITHUB_PAT" into "byeGITHUB_PAT", re-start R, run the tests again. It works! Fix your "GITHUB_PAT" token in .Renviron.

[1] `httr2_response` objects, unlike the equivalent in httr, don't include the request.

So we now have tests that no longer rely on an internet connection nor on having API credentials.

We also added a continuous integration workflow for having a build using real interactions once every week, as it is important to regularly make sure the package still works against the latest API responses.

For the full list of changes applied to exemplighratia in this chapter, see the pull request diff on GitHub.

How do we get there with yet another package? We'll try webfakes.

Chapter 10

Use webfakes

In this chapter we aim at adding HTTP testing infrastructure to exemplighratia using webfakes.

10.1 Setup

Before working on all this, we need to install `{webfakes}`, with `install.packages("webfakes")`.

Then, we need to add webfakes as a Suggests dependency of our package, potentially via running `usethis::use_package("webfakes", type = "Suggests")`.

Last but not least, we create a setup file at `tests/testthat/setup.R`. When testthat runs tests, files whose name starts with "setup" are always run first. We need to ensure that we set up a fake API key when there is no API token around. Why? Because if you remember well, the code of our function `gh_organizations()` checks for the presence of a token. When using our own fake web service, we obviously don't really need a token but we still need to fool our own package in contexts where there is no token (e.g. in continuous integration checks for a fork of a GitHub repository).

```
if(!nzchar(Sys.getenv("REAL_REQUESTS"))) {
  Sys.setenv("GITHUB_PAT" = "foobar")
}
```

The setup file could also load webfakes, but in our demo we will namespace webfakes functions instead.

10.2 Actual testing

With webfakes we will be spinning local fake web services that we will want our package to interact with instead of the real APIs. Therefore, we first need to amend the code of functions returning URLs to services to be able to change them via an environment variable. They become:

```
status_url <- function() {

  env_url <- Sys.getenv("EXEMPLIGHRATIA_GITHUB_STATUS_URL")

  if (nzchar(env_url)) {
    return(env_url)
  }

  "https://kctbh9vrtdwd.statuspage.io/api/v2/components.json"
}
```

and

```
gh_v3_url <- function() {

  api_url <- Sys.getenv("EXEMPLIGHRATIA_GITHUB_API_URL")

  if (nzchar(api_url)) {
    return(api_url)
  }

  "https://api.github.com/"
}
```

Having these two switches is crucial.

Then, let's tweak our test of `gh_api_status()`.

```
test_that("gh_api_status() works", {
  if (!nzchar(Sys.getenv("REAL_REQUESTS"))) {
   app <- webfakes::new_app()
      app$get("/", function(req, res) {
        res$send_json(
          list( components =
          list(
            list(
            name = "API Requests",
            status = "operational"
```

```
            )
          )
          ),
          auto_unbox = TRUE
        )
      })
    web <-  webfakes::local_app_process(app, start = TRUE)
    web$local_env(list(EXEMPLIGHRATIA_GITHUB_STATUS_URL = "{url}"))
  }

  testthat::expect_type(gh_api_status(), "character")
})
```

So what's happening here?

- When we're not asking for requests to the real service to be made (`Sys.getenv("REAL_REQUESTS")`), we prepare a new app via `webfakes::new_app()`. It's a very simple one, that returns, for GET requests, a list corresponding to what we're used to getting out of the status API, except that a) it's much smaller and b) the "operational" status is hard-coded.
- When then create a local app process via `webfakes::local_app_process(, start = TRUE)`. It will start right away thanks to `start=TRUE` but we could have chosen to start it later via calling e.g. `web$url()` (see `?webfakes::local_app_process`); and most importantly it will be stopped automatically after the test. No mess made!
- We set the `EXEMPLIGHRATIA_GITHUB_STATUS_URL` variable to the URL of the local app process. This is what connects our code to our fake web service.

It might seem like a lot of overhead code but

- It means no real requests are made which is our ultimate goal.
- We will get used to it.
- We can write helper code in a testthat helper file to not repeat ourselves in further test files; there could even be an app shared between all test files depending on your package.

Now, let's add a test for error behavior. This inspired us to change error behavior a bit with a slightly more specific error message i.e. `httr::stop_for_status(response)` became `httr::stop_for_status(response, task = "get API status, ouch!")`.

```
test_that("gh_api_status() errors when the API does not behave", {
  app <- webfakes::new_app()
  app$get("/", function(req, res) {
```

```
    res$send_status(502L)
  })
  web <-  webfakes::local_app_process(app, start = TRUE)
  web$local_env(list(EXEMPLIGHRATIA_GITHUB_STATUS_URL = "{url}"))
  testthat::expect_error(gh_api_status(), "ouch")
})
```

It's a similar process to the earlier test:

- setting up a new app;
- having it return something we chose, in this case a 502 status;
- launching a local app process;
- connecting our code to it via setting the `EXEMPLIGHRATIA_GITHUB_STATUS_URL` environment variable to the URL of the fake service;
- test.

Last but not least let's convert our test of `gh_organizations()`,

```
test_that("gh_organizations works", {

  if (!nzchar(Sys.getenv("REAL_REQUESTS"))) {
    app <- webfakes::new_app()
    app$get("/organizations", function(req, res) {
      res$send_json(
        jsonlite::read_json(
          testthat::test_path(
            file.path("responses", "organizations.json")
          )
        ),
        auto_unbox = TRUE
        )
    })
    web <-  webfakes::local_app_process(app, start = TRUE)
    web$local_env(list(EXEMPLIGHRATIA_GITHUB_API_URL = "{url}"))
  }

  testthat::expect_type(gh_organizations(), "character")
})
```

As before we

- create a new app;
- have it returned something we chose for a GET request of the `/organizations` endpoint. In this case, we have it return the content of a JSON file we created

at `tests/testthat/responses/organizations.json` by copy-pasting a real response from the API;
- launch a local app process;
- set its URL as the `EXEMPLIGHRATIA_GITHUB_API_URL` environment variable;
- test.

10.3 Also testing for real interactions

What if the API responses change? Hopefully we'd notice that thanks to following API news. However, sometimes web APIs change without any notice. Therefore it is important to run tests against the real web service once in a while.

In our tests we have used the condition

```
if (!nzchar(Sys.getenv("REAL_REQUESTS"))) {
```

before launching the app and using its URL as URL for the service. So if our tests are generic enough, we can add a CI build where the environment variable `REAL_REQUESTS` is set to true. If they are not generic enough, we can use theapproach exemplified in the chapter about httptest.

- set up a folder real-tests with tests interacting with the real web service;
- add it to Rbuildignore;
- in a CI build, delete tests/testthat and replace it with real-tests, before running R CMD check.

10.4 Summary

- We set up webfakes usage in our package exemplighratia by adding a dependency on webfakes and by adding a setup file to fool our own package that needs an API token.
- We created and launched fake apps in our test files.

Now, how do we make sure this works?

- Turn off wifi, run the tests again. It works! Turn on wifi again.
- Open .Renviron (`usethis::edit_r_environ()`), edit "GITHUB_PAT" into "byeGITHUB_PAT", re-start R, run the tests again. It works! Fix your "GITHUB_PAT" token in .Renviron.

So we now have tests that no longer rely on an internet connection nor on having API credentials.

For the full list of changes applied to exemplighratia in this chapter, see the pull request diff on GitHub.

In the next chapter, we shall compare the three approaches to HTTP testing we've demo-ed.

Chapter 11

vcr (& webmockr), httptest, webfakes

We're now at a nice stage where we have made a demo of usage for each of the HTTP testing packages, in our exemplighratia package. Of course, the choice of strategy in the demo is a bit subjective, but we hope it showed the best of each tool.

A first message that's important to us: if you're learning about HTTP testing and using it in a branch of your own package sounds daunting, create a minimal package for playing!

11.1 What HTTP client can you use (curl, httr, httr2, crul)

- httptest only works with httr (the most popular HTTP R client);
- vcr (& webmockr) works with httr, httr2, and crul (the three "high-level" HTTP R clients);
- webfakes works with any R HTTP client, even base R if you wish.

11.2 Sustainability of the packages

All packages (vcr, webmockr, httptest, webfakes) are actively maintained. During the writing of this book, issues and pull requests were tackled rather quickly, and always in a very nice way.

11.3 Test writing experience

In all cases having HTTP tests, i.e. tests that work independently from any internet connection, depends on

- setup, which is mainly adding a dependency on the HTTP testing packages in DESCRIPTION, and a setup or helper file;
- providing responses from the API.

The difference between packages, the *test writing experience* depends on how you can provide responses from the API, both real ones and fake ones.

With vcr and httptest for tests testing *normal behavior*, after set up (for which there is a helper function), testing is just a function away (`vcr::use_cassette()`, `httptest::with_mock_dir()`, `httptest::with_mock_api()`). Recording happens automatically during the first run of tests. You might also provide fake recorded response or dumb down the existing ones. For creating *API errors*, and API sequence of responses (e.g. 502 then 200), you end up either using webmockr, or amending mock files, see vcr and httptest related docs.[1]

With webfakes you need to create an app. There could be one per test, per test file or for the whole test suite. It might seem like more overhead code but being able to share an app between different tests reduces this effort. You can test for an API sequence of responses (e.g. 502 then 200) by following an how-to. The one thing that's not supported in webfakes yet is a smooth workflow for recording responses, so at the time of writing you might need to write your own workflow for recording responses.

In general setup&test writing might be easier for packages with mocking (vcr and httptest) but you might be able to replicate more complex behavior with webfakes (such as an OAuth dance).

11.3.1 The special case of secrets

With webfakes as no authentication is needed at any point, you have less chance of exposing a secret.

With httptest only the body of responses is saved, so unless it contains secrets, no further effort is needed. If you *need* to redact mock files, see the corresponding vignette.

With vcr as all HTTP interactions, including request URLs and headers, are saved to disk, you will most often have to use the `filter_sensitive_data`, `filter_request_header` and/or `filter_response_header` arguments of `vcr::vcr_configure()`.

[1] Sequence of requests are not supported smoothly yet by httptest.

11.3.2 How about making real requests

In all three cases, switching back to real requests might be an environment variable away (turning vcr off, setting the URL of the real web service as URL to be connected to instead of a webfakes fake web service). However, your tests using fixed/fake responses / a fake web service might not work with real requests as you can't trigger an API error, and as you might test for specific values in your tests using mock files whereas the API returns something different every day. Therefore, and it's a challenge common to all three packages, you might need to choose to have *distinct* tests as integration tests/contract tests. See also our chapter about making real requests.

11.4 Test debugging experience

Sadly sometimes one needs to run code from the tests in an interactive session, either to debug tests after making a code change, or to learn how to write HTTP tests.

With webfakes, debugging works this way: load the helper or test file where

- the app is created,
- the environment variable connecting your package code to the fake web service is changed.

Then run your code. To debug *webfakes apps*, follow the guidance.

With vcr, refer to the debugging vignette: you'll have to load the helper file or source the setup file after making sure the paths use in it work both from `tests/testthat/` and the package root (see `?vcr::vcr_test_path`), and then use `vcr::inject_cassette()`; don't forget to run `vcr::eject_cassette()` afterwards. With webmockr debugging is quite natural, run the code that's in the test, in particular `webmockr::enable()` and `webmockr::disable()`.

With httptest, the process is similar as with vcr except the key functions are

- `use_mock_api()`
- .mockPaths.

11.5 Conclusion

In this chapter we compared the three R packages that make HTTP testing easier. If you are still unsure which one to pick, first try packages out without commitment, in branches or so, but then choose one and commit to your lock-in.

> “Every piece of code written in a given language or framework is a step away from any other language, and five more minutes you’ll have to spend migrating it to something else. That’s fine. You just have to decide what you’re willing to be locked into.
>
> (…)
>
> Code these days becomes obsolete so quickly, regardless of what’s chosen. By the time your needs change, by the time the latest framework is obsolete, all of the code will be rotten anyway
>
> (…)
>
> The most dangerous feature about these articles examining cloud lock-in is that they introduce a kind of paralysis into teams that result in applications never being completely fleshed out or finished.”
>
> Vicki Boykis, “Commit to your lock-in”.

Part III

Advanced Topics

Chapter 12

Making real requests

As touched upon in the Whole Games section, it's good to have *some* tests against the real API. Indeed, the web resource can change.

12.1 What can change?

What can happen?

- An API introducing rate-limiting;
- A web resource disappearing;
- etc.

12.2 How to make real requests

Maybe you can just run the same tests without using the mock files.

- with vcr, this behavior is one environment variable away (namely, `VCR_TURN_OFF`).
- with httptest or httptest2 you can create the same kind of behavior.
- with webfakes you can also create that behavior.

Now this means assuming *all* your tests work with real requests.

- If a few tests won't work with real requests (say they have a fixture mimicking an API error, or specific answer as if today were a given date) then you can skip

these tests when mocking/faking the web service is off. With vcr this means using `vcr::skip_if_vcr_off()`; with httptest and webfakes you'd create your custom skipper.
- If most tests won't work with real requests, then creating a different folder for tests making real requests makes sense. It might be less unit-y as you could view these tests as integration/contract tests. Maybe they could use testthat's snapshot testing (so you could view what's different in the API).

12.2.1 When to make real requests?

Locally, you might want to make real requests once in a while, in particular before a CRAN release.

On continuous integration you have to learn how to trigger workflows and configure build matrices to e.g.

- Have one build in your build matrix using real requests at each commit (this might be too much, see next section);
- Have one scheduled workflow once a day or once a week using real requests.

12.3 Why not make only or too many real requests?

The reasons why you can't only make real requests in your tests are the reasons why you are reading these book:

- they are slower;
- you can't test for API errors;
- etc.

Now no matter what your setup is you don't want to make *too many* real requests as it can be bad for the web resource and bad for you (e.g. using all your allowed requests!). Regarding allowed requests, if possible you could however increase them by requesting for some sort of special development account if such a thing exists for the API you are working with.

12.4 A complement to real requests: API news!

Running real requests is important to notice if something changes in the API (expected requests, responses). Now, you can and should also follow the news of the web resource you are using in case there is something in place.

- Subscribe to the API newsletter if there's one;
- Read API changelogs if they are public;
- In particular, if the API is developed on GitHub/GitLab/etc. you could watch the repo or subscribe to releases, so that you might automatically get notified.

Chapter 13

CRAN- (and Bioconductor) preparedness for your tests

There is no one right answer to how to manage your tests for CRAN, except that you do want a clean check result on CRAN at all times. This probably applies to Bioconductor too. The following is a discussion of the various considerations - which should give you enough information to make an educated decision.

13.1 Running tests on CRAN?

You can run vcr/httptest/httptest2/webfakes enabled tests on CRAN. CRAN is okay with files associated with tests, and so in general on CRAN you can run your tests that use cassettes, mock files or recorded responses on CRAN. Another aspect is the presence of dependencies: make sure the HTTP testing package you use is listed as `Suggests` dependency in DESCRIPTION! With webfakes this might mean also listing optional dependencies in DESCRIPTION. With webfakes, your tests if run on CRAN should not assume the availability of a given port.

When running HTTP tests on CRAN, be aware of a few things:

- If your tests require any secret environment variables or R options (apart from the "foobar" ones used to fool your package when using a saved response), they won't be available on CRAN. In these cases you likely want to skip these tests with `testthat::skip_on_cran()`.
- If your tests have cassettes, mock files or recorded responses with sensitive information in them, you probably do not want to have those cassettes on the internet, in which case you won't be running vcr enabled tests on CRAN either.

In the case of sensitive information, you might want to Rbuildignore the cassettes, mock files or recorded responses (and to gitignore them or make your package development repository private).

- There is a maximal size for package sources so you will want your cassettes, mock files or recorded responses to not be too big. There are three ways to limit their size
  - Make requests that do not generate a huge response (e.g. tweak the time range);
  - Edit the recorded responses (why not even copy-paste responses from the API docs as those are often short) — see vcr docs about editing cassettes for pros and cons;
  - Share cassettes / mock files / recorded responses between tests.

Do not *compress* cassettes, mock files or recorded responses: CRAN submissions are already compressed; compressed files will make git diffs hard to use.

13.2 Skipping a few tests on CRAN?

If you are worried at all about problems with HTTP tests on CRAN you can use `testthat::skip_on_cran()` to skip specific tests. Make sure your tests run somewhere else (on continuous integration) regularly!

We'd recommend not running tests making real requests on CRAN, even when interacting with an API without authentication.

13.3 Skipping all tests on CRAN?

If you have a good continuous integration setup (several operating systems, scheduled runs, etc.) why not skip all tests on CRAN?

13.4 Stress-test your package

To stress-test your package before a CRAN submission, use `rhub::check_for_cran()` without passing any environment variable to the function, and use WinBuilder.

Chapter 14

Security

When developing a package that uses secrets (API keys, OAuth tokens) and produces them (OAuth tokens, sensitive data),

- You want the secrets to be usable by you, collaborators and CI services, without being readable by anyone else;
- You want tests and checks (e.g. vignette building) that use the secrets to be turned off in environments where secrets won't be available (CRAN, forks of your development repository).

Your general attitude should be to think about:

- what are my secrets (an API key, an OAuth2.0 access token and the refresh token, etc.) and where/how exactly are there used (in the query part of an URL? as a header? which header, Authentication or something else?) – packages like httr or httr2 might abstract some of the complexity for you but you need to really know where secrets are used and could be leaked,
- what could go wrong (e.g. your token ending up being published),
- how to prevent that (save your unedited token outside of your package, make sure it is not printed in logs or present in package check artefacts),
- how to fix mistakes (how do you deactivate a token and how do you check no one used it in the meantime).

14.1 Managing secrets securely

14.1.1 Follow best practice when developing your package

This book is about testing but security starts with how you develop your package. To better protect your users' secret,

- It might be best not to let users pass API keys as parameters. It's best to have them save them in `.Renviron` or e.g. using the keyring package. This way, API keys are not in scripts. The opencage package might provide some inspiration.
- If the API you are working with lets you pass keys either in the request headers or query string, prefer to use request headers.

14.1.2 Share secrets with continuous integration services

You need to share secrets with continuous integration services… for real requests only! For tests using vcr, httptest, httptest2 or webfakes, you at most need a fake secret, e.g. "foobar" as API key – except for recording cassettes and mock files, but that is something you do locally.

In GitHub repositories, when storing a new secret, do not save it with quotes. I.e. if your secret is "blabla", the field should contain `blabla`, not `"blabla"` nor `'blabla'`.

```
knitr::include_graphics("secret.png")
```

alt=Screenshot of the interface for adding secrets in a GitHub repository, showing how the secret is stored without any quote.

14.1.2.1 API keys

For API keys, you can use something like GitHub repo secrets if you use GitHub Actions. Then for the secret to be accessible as environment variable from your workflow in GitHub Actions as explained in gargle docs you need to add a line like

```
env:
  PACKAGE_PASSWORD: ${{ secrets.PACKAGE_PASSWORD }}
```

14.1.2.2 More complex objects

If your secret is an OAuth token, you might be able to re-create it from pieces, where the pieces are strings you can store as repo secrets much like what you'd do for an API key. E.g. if your secret is an OAuth token, the actual secrets are the access token and refresh token.

```
env:
  ACCESS_TOKEN: ${{ secrets.ACCESS_TOKEN }}
  REFRESH_TOKEN: ${{ secrets.REFRESH_TOKEN }}
```

Therefore you could re-create it using e.g. the `credentials` argument of `httr::oauth2.0_token()`. The re-creation using environment variables `Sys.getenv("ACCESS_TOKEN")` and `Sys.getenv("REFRESH_TOKEN")` would happen in a testthat helper file.

14.1.3 Secret files

For files, you will need to use encryption and to store a text-version of the encryption key/passwords as GitHub repo secret if you use GitHub Actions. Read the documentation of the continuous integration service your are using to find out how secrets are protected and how you can use them in your builds.

See gargle vignette about securely managing tokens.

The approach is:

- Create your OAuth token locally, either outside of your package folder, or inside of it if you really want to, but **gitignored and Rbuildignored**.
- Encrypt it using e.g. the user-friendly cyphr package. Save the code for this and for the step before in a file e.g. inst/secrets.R for when you need to re-create a token as even refresh tokens expire.
- For encrypting you need some sort of password. You will want to save it securely as *text* in your user-level .Renviron and in your GitHub repo secrets (or equivalent secret place for other CI services). E.g. create a key via `sodium_key <- sodium::keygen()` and get its text equivalent via `sodium::bin2hex(sodium_key)`. E.g. the latter command might give me `e46b7faf296e3f0624e6240a6efafe3dfb17b92ae0089c7e51952934b60749f2` and I will save this in .Renviron

```
MEETUPR_PWD="e46b7faf296e3f0624e6240a6efafe3dfb17b92ae0089c7e51952934b60749f2"
```

Example of a script creating and encrypting an OAuth token (for the Meetup API).

```
# thanks Jenny Bryan https://github.com/r-lib/gargle/blob/4fcf142fde43d107c6a20f905052f24859133c3

token_path <- testthat::test_path(".meetup_token.rds")
use_build_ignore(token_path)
use_git_ignore(token_path)

meetupr::meetup_auth(
  token = NULL,
  cache = TRUE,
  set_renv = FALSE,
  token_path = token_path
)
```

```
# sodium_key <- sodium::keygen()
# set_renv("MEETUPR_PWD" = sodium::bin2hex(sodium_key))
# set_renv being an internal function taken from rtweet
# that saves something to .Renviron

# get key from environment variable
key <- cyphr::key_sodium(sodium::hex2bin(Sys.getenv("MEETUPR_PWD")))

cyphr::encrypt_file(
  token_path,
  key = key,
  dest = testthat::test_path("secret.rds")
)
```

- In tests you have a setup / helper file with code like below.

```
key <- cyphr::key_sodium(sodium::hex2bin(Sys.getenv("MEETUPR_PWD")))

temptoken <- tempfile(fileext = ".rds")

cyphr::decrypt_file(
  testthat::test_path("secret.rds"),
  key = key,
  dest = temptoken
)
```

Now what happens in contexts where `MEETUPR_PWD` is not available? Well there should be no tests using it! See our chapter about making real requests.

14.1.4 Do not store secrets in the cassettes, mock files, recorded responses

- With vcr make sure to configure vcr correctly.
- With httptest and httptest2 only the response body (and headers, but not by default) are recorded. If those contains secrets, refer to the documentation about redacting sensitive information (for httptest2).
- With webfakes you will be creating recorded responses yourself, make sure this process does not leak secrets. If you test something related to authentication, use fake secrets.

If the API you are interacting with uses OAuth for instance, make sure you are not leaking access tokens nor *refresh tokens*.

14.1.5 Escape tests that require secrets

This all depends on your setup for testing real requests. You have to be sure no test requiring secrets will be run on CRAN for instance.

14.2 Sensitive recorded responses?

In that case you might want to gitignore the cassettes / mock files / recorded responses, and skip the tests using them on continuous integration (e.g. `testthat::skip_on_ci()` or something more involved). You'd also Rbuildignore the cassettes / mock files / recorded responses, as you do not want to release them to CRAN.

14.3 Further resources

Some tools might help you detect leaks or prevent them.

- shhgit's goal is "Find secrets in your code. Secrets detection for your GitHub, GitLab and Bitbucket repositories".
- Yelp's detect-secret is "An enterprise friendly way of detecting and preventing secrets in code.".
- git-secret is a "bash tool to store your private data inside a git repo".

Chapter 15

Faking HTTP errors

With HTTP testing you can test the behavior of your package in case of an API error without having to actually trigger an API error. This is important for testing your package's gracefulness (informative error message for the user) and robustness (if you e.g. use retrying in case of API errors).

15.1 How to test for API errors (e.g. 503)

Different possibilities:

- Use webmockr as in our demo.
- Edit a vcr cassette; be careful to skip this test when vcr is off with `vcr::skip_if_vcr_is_off()`.
- With httptest or httptest2, edit a mock file as in our demo, or create it from scratch.
- With webfakes, choose what to return, have a specific app for the test, see our demo.

15.2 How to test for sequence of responses (e.g. 503 then 200)

Different possibilities:

- Use webmockr.
- Edit a vcr cassette; be careful to skip this test when vcr is off with `vcr::skip_if_vcr_is_off()`

- With httptest, this is not easy yet (httptest2 issue)
- With webfakes, follow the docs. Also have a specific app for the test as this is not the behavior you want in all your tests.‘

Chapter 16

Contributor friendliness

How do you make your package wrapping an HTTP resource contributor-friendly?

rOpenSci has some general advice on contributor-friendliness.

Now, there are some more aspects when dealing with HTTP testing.

16.1 Taking notes about encryption

In your contributing guide, make sure you note how you e.g. created an encrypted token for the tests. Link to a script that one could run to re-create it. Good for future contributors including yourself!

16.2 Providing a sandbox

It might be very neat to provide a **sandbox**, even if just for yourself.

- If interacting with say Twitter API you might want to create a Twitter account dedicated to this.
- If interacting with some sort of web platform you might want to create an account special for storing test data.
- Some web APIs provide a test API key, a test account that one can request access to.

Make sure to take notes on how to create / request access to a sandbox, in your contributing guide.

16.3 Switching between accounts depending on the development mode

Your package might have some behaviour to load a default token for instance, placed in an app dir. Now, for testing, you might want it to load another token, and you probably also want the token choice to be as automatic as possible.

The rtweet package has such logic.

- It detects testing/dev mode.

```
is_testing <- function() {
  identical(Sys.getenv("TESTTHAT"), "true")
}
is_dev_mode <- function() {
  exists(".__DEVTOOLS__", .getNamespace("rtweet"))
}
```

- If some environment variables are present it is able to create a testing token.

```
rtweet_test <- function() {
  access_token <- Sys.getenv("RTWEET_ACCESS_TOKEN")
  access_secret <- Sys.getenv("RTWEET_ACCESS_SECRET")

  if (identical(access_token, "") || identical(access_secret, "")) {
    return()
  }

  rtweet_bot(
    "7rX1CfEYOjrtZenmBhjljPzO3",
    "rM3HOLDqmjWzr9UN4cvscchlkFprPNNg99zJJU5R8iYtpC0P0q",
    access_token,
    access_secret
  )
}
```

- The testing token or a default token is loaded depending on the development mode.

16.4 Documenting HTTP testing

Contributors to the package might not be familiar with the HTTP testing package(s) you use (this is true of any non-trivial test setup). Make sure your contributing guide mentions pre-requisites and link to resources (maybe even this very book?).

Part IV

Conclusion

Chapter 17

Conclusion

Once you get here you will have read about basic HTTP (testing) concepts in R, discovered five great packages in demos (httptest2, vcr&webmockr, httptest, webfakes), and dived into more advanced topics like security.

What's next? Applying those tools in your package(s), of course!

- Pick one or several HTTP testing package(s) for your package. Examples of combinations:
  - vcr for testing normal behavior, webmockr for testing behavior in case of web resource errors.
  - vcr or httptest2 for most tests, webfakes for more advanced things like OAuth2.0 flows or slow internet connection.
- Read all the docs of the HTTP testing package(s) you choose – a very worthy use of your time. For vcr and webmockr you can even stay here in this book and take advantage of the "vcr details" and "webmockr details" sections.
- For more examples, you could also look at the reverse dependencies of the HTTP testing package(s) you use to see how they are used by other developers.
- Follow developments of the HTTP testing package(s) you choose. As all five packages we presented are developed on GitHub, you could e.g. release-watch their repositories. They are also all distributed on CRAN, so you might use your usual channel for learning about CRAN updates.
- Participate in the development of the HTTP testing package(s) you choose? Your bug reports, feature requests, contributions might be helpful. Make sure to read the contributing guide and to look at current activity in the repositories.
- Report any feedback about this book, your experience HTTP testing, tips, etc.

- in the GitHub repository of the book,
- or in rOpenSci forum.

Happy HTTP testing!

Part V

webmockr details

Chapter 18

Mocking HTTP Requests

The very very short version is: webmockr helps you stub HTTP requests so you don't have to repeat yourself.

18.1 Package documentation

Check out https://docs.ropensci.org/webmockr/ for documentation on `webmockr` functions.

18.2 Features

- Stubbing HTTP requests at low http client lib level
- Setting and verifying expectations on HTTP requests
- Matching requests based on method, URI, headers and body
- Support for `testthat` via vcr
- Can be used for testing or outside of a testing context

18.3 How webmockr works in detail

You tell `webmockr` what HTTP request you want to match against and if it sees a request matching your criteria it doesn't actually do the HTTP request. Instead, it gives back the same object you would have gotten back with a real request, but only with the bits it knows about. For example, we can't give back the actual data you'd get from a real HTTP request as the request wasn't performed.

In addition, if you set an expectation of what `webmockr` should return, we return that. For example, if you expect a request to return a 418 error (I'm a Teapot), then that's what you'll get.

What you can match against

- HTTP method (required)

Plus any single or combination of the following:

- URI
  - Right now, we can match directly against URI's, and with regex URI patterns. Eventually, we will support RFC 6570 URI templates.
  - We normalize URI paths so that URL encoded things match URL un-encoded things (e.g. `hello world` to `hello%20world`)
- Query parameters
  - We normalize query parameter values so that URL encoded things match URL un-encoded things (e.g. `message = hello world` to `message = hello%20world`)
- Request headers
  - We normalize headers and treat all forms of same headers as equal. For example, the following two sets of headers are equal:
    * `list(H1 = "value1", content_length = 123, X_CuStOm_hEAder = "foo")`
    * `list(h1 = "value1", "Content-Length" = 123, "x-cuSTOM-HeAder" = "foo")`
- Request body

Real HTTP requests

There's a few scenarios to think about when using `webmockr`:

After doing

```
library(webmockr)
```

`webmockr` is loaded but not turned on. At this point `webmockr` doesn't change anything.

Once you turn on `webmockr` like

```
webmockr::enable()
```

`webmockr` will now by default not allow real HTTP requests from the http libraries that adapters are loaded for (`crul`, `httr`, `httr2`).

You can optionally allow real requests via `webmockr_allow_net_connect()`, and disallow real requests via `webmockr_disable_net_connect()`. You can check whether you are allowing real requests with `webmockr_net_connect_allowed()`.

Certain kinds of real HTTP requests allowed: We don't suppoprt this yet, but you can allow localhost HTTP requests with the `allow_localhost` parameter in the `webmockr_configure()` function.

Storing actual HTTP responses

`webmockr` doesn't do that. Check out vcr

18.4 Basic usage

```
library("webmockr")
# enable webmockr
webmockr::enable()
```

Stubbed request based on uri only and with the default response

```
stub_request("get", "https://httpbin.org/get")
```

```
#> <webmockr stub>
#>   method: get
#>   uri: https://httpbin.org/get
#>   with:
#>     query:
#>     body:
#>     request_headers:
#>     auth:
#>   to_return:
```

```
library("crul")
x <- HttpClient$new(url = "https://httpbin.org")
x$get('get')
```

```
#> <crul response>
```

```
#>   url: https://httpbin.org/get
#>   request_headers:
#>     User-Agent: libcurl/8.5.0 r-curl/7.0.0 crul/1.6.0
#>     Accept-Encoding: gzip, deflate
#>     Accept: application/json, text/xml, application/xml, */*
#>   response_headers:
#>   status: 200
```

Chapter 19

stubs

```
library("webmockr")
```

set return objects

```
stub_request("get", "https://httpbin.org/get") %>%
  wi_th(
    query = list(hello = "world")) %>%
    to_return(status = 418)
```

```
#> <webmockr stub>
#>   method: get
#>   uri: https://httpbin.org/get
#>   with:
#>     query: hello=world
#>     body:
#>     request_headers:
#>     auth:
#>   to_return:
#>   - status: 418
#>     body:
#>     response_headers:
#>     should_timeout: FALSE
#>     should_raise: FALSE
```

```
x$get('get', query = list(hello = "world"))
```

```
#> <crul response>
```

```
#>   url: https://httpbin.org/get
#>   request_headers:
#>     User-Agent: libcurl/8.5.0 r-curl/7.0.0 crul/1.6.0
#>     Accept-Encoding: gzip, deflate
#>     Accept: application/json, text/xml, application/xml, */*
#>   response_headers:
#>   status: 418
```

Stubbing requests based on method, uri and query params

```
stub_request("get", "https://httpbin.org/get") %>%
  wi_th(query = list(hello = "world"),
        headers = list('User-Agent' = 'libcurl/7.51.0 r-curl/2.6 crul/0.3.6',
                       'Accept-Encoding' = "gzip, deflate"))
```

```
#> <webmockr stub>
#>   method: get
#>   uri: https://httpbin.org/get
#>   with:
#>     query: hello=world
#>     body:
#>     request_headers: User-Agent=libcurl/7.51.0 r-cur..., Accept-Encoding=gzip, defla
#>     auth:
#>   to_return:
```

```
stub_registry()
```

```
#> <webmockr stub registry>
#>  Registered Stubs
#>   GET: https://httpbin.org/get
#>   GET: https://httpbin.org/get  with query params hello=world   | to_return:    with
#>   GET: https://httpbin.org/get  with query params hello=world   with headers {"User-
```

```
x <- HttpClient$new(url = "https://httpbin.org")
x$get('get', query = list(hello = "world"))
```

```
#> <crul response>
#>   url: https://httpbin.org/get
#>   request_headers:
#>     User-Agent: libcurl/8.5.0 r-curl/7.0.0 crul/1.6.0
#>     Accept-Encoding: gzip, deflate
#>     Accept: application/json, text/xml, application/xml, */*
#>   response_headers:
#>   status: 418
```

Stubbing requests and set expectation of a timeout

```
stub_request("post", "https://httpbin.org/post") %>% to_timeout()
x <- HttpClient$new(url = "https://httpbin.org")
x$post('post')
#> Error: Request Timeout (HTTP 408).
#>   - The client did not produce a request within the time that the server was prepared
#>     to wait. The client MAY repeat the request without modifications at any later time.
```

Stubbing requests and set HTTP error expectation

```
library(fauxpas)
stub_request("get", "https://httpbin.org/get?a=b") %>% to_raise(HTTPBadRequest)
x <- HttpClient$new(url = "https://httpbin.org")
x$get('get', query = list(a = "b"))
#> Error: Bad Request (HTTP 400).
#>   - The request could not be understood by the server due to malformed syntax.
#>     The client SHOULD NOT repeat the request without modifications.
```

19.1 Writing to disk

There are two ways to deal with mocking writing to disk. First, you can create a file with the data you'd like in that file, then tell crul, httr, or httr2 where that file is. Second, you can simply give webmockr a file path (that doesn't exist yet) and some data, and webmockr can take care of putting the data in the file.

Here's the first method, where you put data in a file as your mock, then pass the file as a connection (with `file(<file path>)`) to `to_return()`.

```
## make a temp file
f <- tempfile(fileext = ".json")
## write something to the file
cat("{\"hello\":\"world\"}\n", file = f)
## make the stub
invisible(stub_request("get", "https://httpbin.org/get") %>%
  to_return(body = file(f)))
## make a request
out <- HttpClient$new("https://httpbin.org/get")$get(disk = f)
## view stubbed file content
readLines(file(f))
```

```
#> [1] "{\"hello\":\"world\"}"
```

With the second method, use `webmockr::mock_file()` to have `webmockr` handle file and contents.

```
g <- tempfile(fileext = ".json")
## make the stub
invisible(stub_request("get", "https://httpbin.org/get?a=b") %>%
  to_return(body = mock_file(path = g, payload = "{\"hello\":\"mars\"}\n")))
## make a request
out <- crul::HttpClient$new("https://httpbin.org/get?a=b")$get(disk = g)
## view stubbed file content
readLines(out$content)
```

```
#> [1] "{\"hello\":\"mars\"}" ""
```

`webmockr` also supports `httr::write_disk()`, here letting `webmockr` handle the mock file creation:

```
library(httr)
httr_mock()
## make a temp file
f <- tempfile(fileext = ".json")
## make the stub
invisible(stub_request("get", "https://httpbin.org/get?cheese=swiss") %>%
  to_return(
    body = mock_file(path = f, payload = "{\"foo\": \"bar\"}"),
    headers = list('content-type' = "application/json")
  ))
## make a request
out <- GET("https://httpbin.org/get?cheese=swiss", write_disk(f, TRUE))
## view stubbed file content
readLines(out$content)
```

```
#> [1] "{\"foo\": \"bar\"}"
```

Chapter 20

testing

```
library("webmockr")
library("crul")
library("testthat")

stub_registry_clear()

# make a stub
stub_request("get", "https://httpbin.org/get") %>%
   to_return(body = "success!", status = 200)
```

```
#> <webmockr stub>
#>   method: get
#>   uri: https://httpbin.org/get
#>   with:
#>     query:
#>     body:
#>     request_headers:
#>     auth:
#>   to_return:
#>   - status: 200
#>     body: success!
#>     response_headers:
#>     should_timeout: FALSE
#>     should_raise: FALSE
```

```
# check that it's in the stub registry
stub_registry()
```

```
#> <webmockr stub registry>
#>  Registered Stubs
#>   GET: https://httpbin.org/get    | to_return:   with body "success!"  with status 2
```

```
# make the request
z <- crul::HttpClient$new(url = "https://httpbin.org")$get("get")

# run tests (nothing returned means it passed)
expect_is(z, "HttpResponse")
expect_equal(z$status_code, 200)
expect_equal(z$parse("UTF-8"), "success!")
```

Chapter 21

utilities

```
library("webmockr")
```

21.1 Managing stubs

- `enable()`
- `enabled()`
- `disable()`
- `httr_mock()`

21.2 Managing stubs

- `stub_registry()`
- `stub_registry_clear()`
- `remove_request_stub()`

21.3 Managing requests

- `request_registry()`

Part VI

vcr details

Chapter 22

Caching HTTP requests

Record HTTP calls and replay them

22.1 Package documentation

Check out https://docs.ropensci.org/vcr/ for documentation on `vcr` functions.

22.2 Design

https://docs.ropensci.org/vcr/articles/design.html

22.3 Basic usage

https://docs.ropensci.org/vcr/articles/vcr.html ## vcr enabled testing {#vcr-enabled-testing}

22.3.1 check vs. test

> TLDR: Run `devtools::test()` before running `devtools::check()` for recording your cassettes.

When running tests or checks of your whole package, note that you'll get different results with `devtools::check()` (check button of RStudio build pane)

vs. `devtools::test()` (test button of RStudio build pane). This arises because `devtools::check()` runs in a temporary directory and files created (vcr cassettes) are only in that temporary directory and thus don't persist after `devtools::check()` exits.

However, `devtools::test()` does not run in a temporary directory, so files created (vcr cassettes) are in whatever directory you're running it in.

Alternatively, you can run `devtools::test_file()` (or the "Run test" button in RStudio) to create your vcr cassettes one test file at a time.

22.3.2 CI sites: GitHub Actions, Appveyor, etc.

Refer to the security chapter.

Chapter 23

Advanced vcr usage

Now that we've covered basic `vcr` usage, it's time for some more advanced usage topics.

```
library("vcr")
```

23.1 Mocking writing to disk

If you have http requests for which you write the response to disk, then use `vcr_configure()` to set the `write_disk_path` option. See more about the write_disk_path configuration option.

Here, we create a temporary directory, then set the fixtures

```
tmpdir <- tempdir()
vcr_configure(
  dir = file.path(tmpdir, "fixtures"),
  write_disk_path = file.path(tmpdir, "files")
)
```

Then pass a file path (that doesn't exist yet) to crul's `disk` parameter. `vcr` will take care of handling writing the response to that file in addition to the cassette.

```
library(crul)
## make a temp file
f <- tempfile(fileext = ".json")
## make a request
cas <- use_cassette("test_write_to_disk", {
```

```
  out <- HttpClient$new("https://httpbin.org/get")$get(disk = f)
})
file.exists(out$content)
```

```
#> [1] TRUE
```

```
out$parse()
```

```
#> [1] "{\n  \"args\": {}, \n  \"headers\": {\n    \"Accept\": \"application/json, text
```

This also works with `httr`. The only difference is that you write to disk with a function `httr::write_disk(path)` rather than a parameter.

Writing to disk with {`httr2`} does not yet work with {`vcr`} – see https://github.com/ropensci/vcr/issues/270

Note that when you write to disk when using `vcr`, the cassette is slightly changed. Instead of holding the http response body itself, the cassette has the file path with the response body.

```
http_interactions:
- request:
    method: get
    uri: https://httpbin.org/get
  response:
    headers:
      status: HTTP/1.1 200 OK
      access-control-allow-credentials: 'true'
    body:
      encoding: UTF-8
      file: yes
      string: /private/var/folders/fc/n7g_vrvn0sx_st0p8lxb3ts40000gn/T/Rtmp5W4olr/files
```

And the file has the response body that otherwise would have been in the `string` yaml field above:

```
{
  "args": {},
  "headers": {
    "Accept": "application/json, text/xml, application/xml, */*",
    "Accept-Encoding": "gzip, deflate",
    "Host": "httpbin.org",
    "User-Agent": "libcurl/7.54.0 r-curl/4.3 crul/0.9.0"
  },
```

```
  "origin": "24.21.229.59, 24.21.229.59",
  "url": "https://httpbin.org/get"
}
```

Chapter 24

Configure vcr

https://docs.ropensci.org/vcr/reference/vcr_configure.html

Chapter 25

Record modes

https://docs.ropensci.org/vcr/articles/debugging.html

Chapter 26

Request matching

https://docs.ropensci.org/vcr/articles/debugging.html

Chapter 27

Debugging your tests that use vcr

https://docs.ropensci.org/vcr/articles/debugging.html

Chapter 28

Security with vcr

Refer to the security chapter for more general guidance.

28.1 Keeping secrets safe

To keep your secrets safe, you need to use parameters of `vcr::vcr_configure()` that tell vcr either where secrets are (and what to put in their place), or what secrets are (and what to put in their place). It is best if you know how secrets are used in requests: e.g. is the API key passed as a header or part of the query string? Maybe you will need different strategies for the different secrets (e.g. an OAuth2.0 access token will be set as Authorization header but an OAuth2.0 refresh token might be in a query string).

In all cases, it is crucial to look at your cassettes before putting them on the public web, just to be sure you got the configuration right!

28.1.1 If the secret is in a request header

You can use `filter_request_headers`!

There are different ways to use it.

```
# Remove one header from the cassettes
vcr_configure(
  filter_request_headers = "Authorization"
)

# Remove two headers from the cassettes
```

```
vcr_configure(
  filter_request_headers = c("Authorization", "User-Agent")
)

# Replace one header with a given string
vcr_configure(
  filter_request_headers = list(Authorization = "<<<not-my-bearer-token>>>")
)
```

28.1.2 If the secret is in a response header

You can use `filter_response_headers` that works like `filter_request_headers`.

28.1.3 If the secret is somewhere else

In this case you need to tell vcr what the secret string is via `filter_sensitive_data`. Do not write the secret string directly in the configuration, that'd defeat the purpose of protecting it! Have the secret in an environment variable for instance and tell vcr to read it from there.

The configuration parameter `filter_sensitive_data` accepts a named list.

Each element in the list should be of the following format:

`thing_to_replace_it_with = thing_to_replace`

We replace all instances of `thing_to_replace` with `thing_to_replace_it_with`.

Before recording (writing to a cassette) we do the replacement and then when reading from the cassette we do the reverse replacement to get back to the real data.

```
vcr_configure(
  filter_sensitive_data = list("<<<my_api_key>>>" = Sys.getenv('API_KEY'))
)
```

You want to make the string that replaces your sensitive string something that won't be easily found elsewhere in the response body/headers/etc.

28.2 API keys and tests run in varied contexts

- For real requests a real API key is needed.
- For requests using cassettes a fake API key is needed to fool your package. That is why in our demo of vcr we set a fake API key in a test setup file.

28.3 Other security

Let us know about any other security concerns! Surely there's things we haven't considered yet.

Chapter 29

Turning vcr on and off

https://docs.ropensci.org/vcr/reference/lightswitch.html

Chapter 30

Managing cassettes

https://docs.ropensci.org/vcr/articles/vcr.html?q=editing#cassette-files

Be aware when you add your cassettes to either `.gitignore` and/or `.Rbuildignore`.

30.1 gitignore cassettes

The .gitignore file lets you tell [git][] what files to ignore - those files are not tracked by git and if you share the git repository to the public web, those files in the `.gitignore` file won't be shared in the public version.

When using `vcr` you may want to include your cassettes in the `.gitignore` file. You may wan to when your cassettes contain sensitive data that you don't want to have on the internet & dont want to hide with filter_sensitive_data.

You may want to have your cassettes included in your GitHub repo, both to be present when tests run on CI, and when others run your tests.

There's no correct answer on whether to gitignore your cassettes. Think about security implications and whether you want CI and human contributors to use previously created cassettes or to create/use their own.

30.2 Rbuildignore cassettes

The .Rbuildignore file is used to tell R to ignore certain files/directories.

There's not a clear use case for why you'd want to add vcr cassettes to your `.Rbuildignore` file, but if you do be aware that will affect your vcr enabled tests.

30.3 sharing cassettes

Sometimes you may want to share or re-use cassettes across tests, for example to reduce the size for package sources or to test different functionality of your package functions that make the same query under the hood.

To do so, you can use the same cassette name for multiple `vcr::use_cassette()` calls. `vcr::check_cassette_names()` will complain about duplicate cassette names, preventing you from accidentally re-using cassettes, however. To allow duplicates, you can provide a character vector of the cassette names you want to re-use to the `allowed_duplicates` argument of `vcr::check_cassette_names()`. That way you can use the same cassette across multiple tests.

30.4 deleting cassettes

Removing a cassette is as easy as deleting in your file finder, or from the command line, or from within a text editor or RStudio.

If you delete a cassette, on the next test run the cassette will be recorded again.

If you do want to re-record a test to a cassette, instead of deleting the file you can toggle record modes.

30.5 cassette file types

For right now the only persistence option is yaml. So all files have a `.yml` extension.

When other persister options are added, additional file types may be found. The next persister type is likely to be JSON, so if you use that option, you'd have `.json` files instead of `.yml` files.

Chapter 31

Gotchas

There's a few things to watch out for when using `vcr`.

- **Security**: Don't put your secure API keys, tokens, etc. on the public web. See the Security chapter and the vcr security chapter.
- **API key issues**: Running vcr enabled tests in different contexts when API keys are used can have some rough edges.
- **Dates**: Be careful when using dates in tests with `vcr`. e.g. if you generate todays date, and pass that in to a function in your package that uses that date for an HTTP request, the date will be different from the one in the matching cassette, causing a `vcr` failure.
- **HTTP errors**: It's a good idea to test failure behavior of a web service in your test suite. Sometimes `vcr` can handle that and sometimes it cannot. Open any issues about this because ideally i think `vcr` could handle all cases of HTTP failures.
- **Very large response bodies**: A few things about large response bodies. First, `vcr` may give you trouble with very large response bodies as we've see yaml parsing problems already. Second, large response bodies means large cassettes on disk - so just be aware of the file size if that's something that matters to you. Third, large response bodies will take longer to load into R, so you may still have a multi second test run even though the test is using a cached HTTP response.
- **Encoding**: We haven't dealt with encoding much yet at all, so we're likely to run into encoding issues. One blunt instrument for this for now is to set `preserve_exact_body_bytes = TRUE` when running `vcr::use_cassette()` or `vcr::insert_cassette()`, which stores the response body as base64.
- **devtools::check vs. devtools::test**: See (22.3.1)
- **ignored files**: See (30)

31.1 Correct line identification

To get the actual lines where failures occur, you can wrap the `test_that` block in a `use_cassette()` block:

```
library(testthat)
vcr::use_cassette("rl_citation", {
  test_that("my test", {
    aa <- rl_citation()

    expect_is(aa, "character")
    expect_match(aa, "IUCN")
    expect_match(aa, "www.iucnredlist.org")
  })
})
```

OR put the `use_cassette()` block on the inside, but make sure to put `testthat` expectations outside of the `use_cassette()` block:

```
library(testthat)
test_that("my test", {
  vcr::use_cassette("rl_citation", {
    aa <- rl_citation()
  })

  expect_is(aa, "character")
  expect_match(aa, "IUCN")
  expect_match(aa, "www.iucnredlist.org")
})
```

Do not wrap the `use_cassette()` block inside your `test_that()` block with `testthat` expectations inside the `use_cassette()` block, as you'll only get the line number that the `use_cassette()` block starts on on failures.

Chapter 32

Session info

32.1 Session info

```
library("magrittr")

dependencies <- attachment::att_from_rmds(".")
dependencies <- dependencies[!dependencies %in% c("attachment", "bookdown", "knitr")]

sessioninfo::package_info(
  pkgs = dependencies
  ) %>%
  as.data.frame() %>%
  .[, c("package", "ondiskversion")] %>%
  knitr::kable()
```

	package	ondiskversion
askpass	askpass	1.2.1
brio	brio	1.1.5
callr	callr	3.7.6
cli	cli	3.6.5
crayon	crayon	1.5.3
crul	crul	1.6.0
curl	curl	7.0.0
desc	desc	1.4.3
diffobj	diffobj	0.3.6
digest	digest	0.6.37
evaluate	evaluate	1.0.5
fauxpas	fauxpas	0.5.2
fs	fs	1.6.6
glue	glue	1.8.0
httpcode	httpcode	0.3.0
httr	httr	1.4.7
httr2	httr2	1.2.1
jsonlite	jsonlite	2.0.0
lifecycle	lifecycle	1.0.4
magrittr	magrittr	2.0.4
mime	mime	0.13
openssl	openssl	2.3.4
pkgbuild	pkgbuild	1.4.8
pkgload	pkgload	1.4.1
praise	praise	1.0.0
processx	processx	3.8.6
ps	ps	1.9.1
R6	R6	2.6.1
rappdirs	rappdirs	0.3.3
Rcpp	Rcpp	1.1.0
rlang	rlang	1.1.6
rprojroot	rprojroot	2.1.1
sessioninfo	sessioninfo	1.2.3
sys	sys	3.4.3
testthat	testthat	3.2.3
triebeard	triebeard	0.4.1
urltools	urltools	1.7.3.1
vcr	vcr	2.0.0
vctrs	vctrs	0.6.5
waldo	waldo	0.6.2
webmockr	webmockr	2.2.0
whisker	whisker	0.4.1
withr	withr	3.0.2
yaml	yaml	2.3.10

None of crul, webmockr, vcr, httptest have compiled code, but an underlying dependency of all of them, `curl` does. See curl's README for installation instructions in case you run into curl related problems. webfakes has compiled code.

32.2 Full session info

Session info for this book

```
sessioninfo::session_info()
```

```
#> - Session info ---------------------------------------------------------------
#>  setting  value
#>  version  R version 4.5.1 (2025-06-13)
#>  os       Ubuntu 24.04.3 LTS
#>  system   x86_64, linux-gnu
#>  ui       X11
#>  language (EN)
#>  collate  C.UTF-8
#>  ctype    C.UTF-8
#>  tz       UTC
#>  date     2025-10-24
#>  pandoc   3.1.11 @ /opt/hostedtoolcache/pandoc/3.1.11/x64/ (via rmarkdown)
#>  quarto   NA
#>
#> - Packages -------------------------------------------------------------------
#>  package     * version date (UTC) lib source
#>  attachment    0.4.5   2025-03-14 [1] RSPM
#>  bookdown      0.45    2025-10-03 [1] RSPM
#>  brio          1.1.5   2024-04-24 [1] RSPM
#>  cli           3.6.5   2025-04-23 [1] RSPM
#>  crul        * 1.6.0   2025-07-23 [1] RSPM
#>  curl          7.0.0   2025-08-19 [1] RSPM
#>  desc          1.4.3   2023-12-10 [1] RSPM
#>  digest        0.6.37  2024-08-19 [1] RSPM
#>  evaluate      1.0.5   2025-08-27 [1] RSPM
#>  fastmap       1.2.0   2024-05-15 [1] RSPM
#>  fauxpas       0.5.2   2023-05-03 [1] RSPM
#>  glue          1.8.0   2024-09-30 [1] RSPM
#>  htmltools     0.5.8.1 2024-04-04 [1] RSPM
#>  httpcode      0.3.0   2020-04-10 [1] RSPM
#>  httr        * 1.4.7   2023-08-15 [1] RSPM
#>  httr2         1.2.1   2025-07-22 [1] RSPM
#>  jsonlite      2.0.0   2025-03-27 [1] RSPM
```

```
#>  knitr         1.50    2025-03-16 [1] RSPM
#>  lifecycle     1.0.4   2023-11-07 [1] RSPM
#>  magrittr    * 2.0.4   2025-09-12 [1] RSPM
#>  pkgload       1.4.1   2025-09-23 [1] RSPM
#>  purrr         1.1.0   2025-07-10 [1] RSPM
#>  R6            2.6.1   2025-02-15 [1] RSPM
#>  rappdirs      0.3.3   2021-01-31 [1] RSPM
#>  Rcpp          1.1.0   2025-07-02 [1] RSPM
#>  rlang         1.1.6   2025-04-11 [1] RSPM
#>  rmarkdown     2.30    2025-09-28 [1] RSPM
#>  roxygen2      7.3.3   2025-09-03 [1] RSPM
#>  rprojroot     2.1.1   2025-08-26 [1] RSPM
#>  rstudioapi    0.17.1  2024-10-22 [1] RSPM
#>  sessioninfo   1.2.3   2025-02-05 [1] RSPM
#>  stringi       1.8.7   2025-03-27 [1] RSPM
#>  stringr       1.5.2   2025-09-08 [1] RSPM
#>  testthat    * 3.2.3   2025-01-13 [1] RSPM
#>  triebeard     0.4.1   2023-03-04 [1] RSPM
#>  urltools      1.7.3.1 2025-06-12 [1] RSPM
#>  vcr         * 2.0.0   2025-07-23 [1] RSPM
#>  vctrs         0.6.5   2023-12-01 [1] RSPM
#>  webmockr    * 2.2.0   2025-07-21 [1] RSPM
#>  whisker       0.4.1   2022-12-05 [1] RSPM
#>  xfun          0.53    2025-08-19 [1] RSPM
#>  xml2          1.4.0   2025-08-20 [1] RSPM
#>  yaml          2.3.10  2024-07-26 [1] RSPM
#>
#>  [1] /home/runner/work/_temp/Library
#>  [2] /opt/R/4.5.1/lib/R/site-library
#>  [3] /opt/R/4.5.1/lib/R/library
#>  * -- Packages attached to the search path.
#>
#> ------------------------------------------------------------------------------
```

Page not found. Use the table of contents or the search bar to find your way back.