HTTP mocking and testing in R

built with vcr v0.5.0 / webmockr v0.6.0 / crul v0.9.0 / curl v4.3

built on 2020-03-24

Contents

1 Preamble **7**

I introduction 9

2 Introduction **11**

2.1 What is webmockr? 11

2.2 What is vcr? 11

2.3 Why crul? 12

2.4 Use cases 12

II webmockr 13

3 Mocking HTTP Requests **15**

3.1 Package documentation 15

3.2 Features 15

3.3 How webmockr works in detail 15

3.4 Basic usage 17

4 stubs **19**

4.1 Writing to disk 21

5 testing **23**

6 utilities **25**

6.1 Managing stubs 25

6.2 Managing stubs 25

6.3 Managing requests 25

III vcr **27**

7 Caching HTTP requests **29**

7.1 Package documentation 29

7.2 Terminology 29

7.3 Basic usage 30

7.4 vcr enabled testing 32

8 vcr usage **35**

8.1 Mocking writing to disk 35

9 vcr configuration **39**

9.1 Get your configuration 39

9.2 Set configuration variables 41

9.3 Re-set to defaults 42

9.4 dir - directory of where cassettes are stored 42

9.5 record - record mode 43

9.6 match_requests_on - customize how vcr matches requests 43

9.7 allow_unused_http_interactions - Allow HTTP connections when no cassette 43

9.8 serialize_with - which serializer to use 44

9.9 persist_with - which persister to use 44

9.10 ignore requests 45

9.11 uri_parser - which uri parser to use 46

9.12 preserve_exact_body_bytes 46

9.13 allow_http_connections_when_no_cassette 47

9.14 write_disk_path 47

10 record modes **49**

10.1 once . . . 49

10.2 none . . . 49

10.3 new_episodes . . . 50

10.4 all . . . 50

11 request matching **51**

11.1 matching on method . . . 51

11.2 matching on uri . . . 51

11.3 matching on host . . . 51

11.4 matching on path . . . 52

11.5 matching on query string . . . 52

11.6 matching on body . . . 52

11.7 matching on headers . . . 52

11.8 Playback repeats . . . 52

12 logging **53**

12.1 Setup logging . . . 53

12.2 The log file . . . 53

13 security **55**

13.1 API keys and such . . . 55

13.2 API keys and tests run in varied contexts . . . 56

13.3 Other security . . . 57

14 Turning vcr on & off **59**

14.1 turned_off . . . 60

14.2 turn_off/turn_on . . . 60

14.3 turned_on . . . 61

14.4 Environment variables . . . 62

15 managing cassettes **63**

15.1 gitignore cassettes 63

15.2 Rbuildignore cassettes 63

15.3 deleting cassettes 64

15.4 cassette file types 64

16 gotchas **65**

16.1 Correct line identification 66

IV Appendix **67**

17 Testing considerations **69**

18 session info **71**

Chapter 1

Preamble

This book is intended as a detailed guide to using a particular suite of packages for HTTP mocking and testing in R code and/or packages.

Info

- Source: https://github.com/ropensci-books/http-testing
- Issues/Bug reports: https://github.com/ropensci-books/http-testing/issues
- pkgdown site for crul: https://docs.ropensci.org/crul/
- pkgdown site for webmockr: https://docs.ropensci.org/webmockr/
- pkgdown site for vcr: https://docs.ropensci.org/vcr/

Packages

- `crul`: https://github.com/ropensci/crul/
- `webmockr`: https://github.com/ropensci/webmockr/
- `vcr`: https://github.com/ropensci/vcr/
- `curl`: https://github.com/jeroen/curl/

Installation

Stable version from CRAN

```
install.packages(c("crul", "webmockr", "vcr", "httr"))
```

none of `crul`, `webmockr` or `vcr` 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.

Part I

introduction

Chapter 2

Introduction

2.1 What is webmockr?

`webmockr` 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 match on
- You also can set rules for what 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 will return what you requested be returned
- While `webmockr` is in use, real HTTP interactions are not allowed
- There is no recording interactions to disk at all, just mocked responses given as the user specifies in the R session

`webmockr` works with both the `crul` package and the `httr` package.

Read more about `webmockr` in Section 2.

2.2 What is vcr?

The short version is: `vcr` helps you stub HTTP requests so you don't have to repeat HTTP requests.

The main use case is for unit tests of R packages.

`vcr` works with both the `crul` package and the `httr` package.

`vcr` works by hooking into `webmockr`. However, when `webmockr` finds a match, we then look for a recorded interaction on disk. If one is not found, we record the request and response. If one is found, we use that recorded interaction to construct a real response as the R client expects.

Read more about `vcr` in Section 3.

2.3 Why crul?

`crul` is just one of the HTTP clients in R. It's the one that I maintain though, so was easiest to get started with adding mocking integration.

There is now integration with `httr` for both `webmockr` and `vcr`.

The major feature that `httr` has that `crul` does not have is OAuth support, but that's not an important use case for me so is not a high priority for `crul`.

A major reason to use `crul` over `httr` is that it uses more of an object oriented interface. That is, you create objects that you can call methods on and retrieve variables/results from calls/etc. It's a different approach than `httr` which focuses on passing things to functions.

2.4 Use cases

2.4.1 mocking use cases

- one
- two
- three

2.4.2 caching use cases

- one
- two
- three

Part II

webmockr

Chapter 3

Mocking HTTP Requests

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

3.1 Package documentation

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

3.2 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

3.3 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 anythning.

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 (right now only `crul`).

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

3.4 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:
#>   to_return:
#>     status:
#>     body:
#>     response_headers:
#>   should_timeout: FALSE
#>   should_raise: FALSE
```

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

```
#> <crul response>
#>   url: https://httpbin.org/get
#>   request_headers:
#>     User-Agent: libcurl/7.64.1 r-curl/4.3 crul/0.9.0
#>     Accept-Encoding: gzip, deflate
#>     Accept: application/json, text/xml, application/xml, */*
#>   response_headers:
#>   status: 200
```

Chapter 4

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:
#>   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?hello=world
```

```
#>   request_headers:
#>     User-Agent: libcurl/7.64.1 r-curl/4.3 crul/0.9.0
#>     Accept-Encoding: gzip, deflate
#>     Accept: application/json, text/xml, application/xml, */*
#>   response_headers:
#>   params:
#>     hello: world
#>   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
#>   to_return:
#>     status:
#>     body:
#>     response_headers:
#>   should_timeout: FALSE
#>   should_raise: FALSE
```

```
stub_registry()
```

```
#> <webmockr stub registry>
#>  Registered Stubs
#>    GET: https://httpbin.org/get
#>    GET: https://httpbin.org/get?hello=world   | to_return:   with status 418
#>    GET: https://httpbin.org/get?hello=world   with headers {"User-Agent":"libcurl/7
```

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

```
#> <crul response>
#>   url: https://httpbin.org/get?hello=world
```

```
#>   request_headers:
#>     User-Agent: libcurl/7.64.1 r-curl/4.3 crul/0.9.0
#>     Accept-Encoding: gzip, deflate
#>     Accept: application/json, text/xml, application/xml, */*
#>   response_headers:
#>   params:
#>     hello: world
#>   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.
```

4.1 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 or httr 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 5

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:
#>   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 20
```

```
# 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 6

utilities

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

6.1 Managing stubs

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

6.2 Managing stubs

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

6.3 Managing requests

- `request_registry()`

Part III

vcr

Chapter 7

Caching HTTP requests

Record HTTP calls and replay them

7.1 Package documentation

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

7.2 Terminology

- **http**: hyptertext transfer protocol
- **vcr**: the name comes from the idea that we want to record something and play it back later, like a VCR
- **cassette**: A *thing* to record HTTP interactions to. Right now the only option is file system, but in the future could be other things, e.g. a key-value store like Redis
- **Persister**: defines how to save requests - currently only option is the file system
- **Serializers**: defines how to serialize the HTTP response; that is, how the data is stored on whatever the persister is (right now only file system). Currently only option is YAML; other options in the future could include e.g. JSON
- **insert cassette**: create a cassette (all HTTP interactions will be recorded to this cassette). once a cassette is inserted, we don't allow insertion of additional cassettes
- **eject cassette**: eject the cassette (no longer recording to that cassette). however, if any interactions were written to disk, those are still stored there

- **replay**: refers to using a cached result of an http request that was recorded earlier
- **recording**: this means you've set vcr in a mode in which we can record HTTP interactions. sometimes recording can be not possible given user configuration or otherwise

7.3 Basic usage

```
library(vcr)
library(crul)

cli <- crul::HttpClient$new(url = "https://api.crossref.org")
system.time(
  use_cassette(name = "helloworld", {
    cli$get("works", query = list(rows = 3))
  })
)
#>    user  system elapsed
#>   0.082   0.007   1.629
```

The request gets recorded, and all subsequent requests of the same form used the cached HTTP response, and so are much faster

```
system.time(
  use_cassette(name = "helloworld", {
    cli$get("works", query = list(rows = 3))
  })
)
#>    user  system elapsed
#>   0.073   0.004   0.077
```

Importantly, your unit test deals with the same inputs and the same outputs - but behind the scenes you use a cached HTTP resonse - thus, your tests run faster.

The cached response looks something like (condensed for brevity):

```
http_interactions:
- request:
    method: get
    uri: https://api.crossref.org/works?rows=3
    body:
      encoding: ''
```

```
      string: ''
    headers:
      User-Agent: libcurl/7.54.0 r-curl/3.2 crul/0.5.2
      Accept-Encoding: gzip, deflate
      Accept: application/json, text/xml, application/xml, */*
  response:
    status:
      status_code: '200'
      message: OK
      explanation: Request fulfilled, document follows
    headers:
      status: HTTP/1.1 200 OK
      content-type: application/json;charset=UTF-8
      vary: Accept
      access-control-allow-origin: '*'
      access-control-allow-headers: X-Requested-With
      content-length: '5360'
      server: http-kit
      date: Sat, 28 Apr 2018 15:12:29 GMT
      x-rate-limit-limit: '50'
      x-rate-limit-interval: 1s
      connection: close
    body:
      encoding: UTF-8
      string: '{"status":"ok","message-type":"work-list","message-version":"1.0.0","message":{"fa
        BV","issue":"3","license":[{"URL":"http:\/\/www.elsevier.com\/tdm\/userlicense\/1.0\/","s
        in Planning"],"published-print":{"date-parts":[[1999,10]]},"DOI":"10.1016\/s0305-9006(99)
        in Planning"],"link":[{"URL":"http:\/\/api.elsevier.com\/content\/article\/PII:S030590069
        Planning and Development"]},{"indexed":{"date-parts":[[2017,10,23]],"date-time":"2017-10-
        BV","issue":"4","license":[{"URL":"http:\/\/www.elsevier.com\/tdm\/userlicense\/1.0\/","s
        in Planning"],"published-print":{"date-parts":[[1998,12]]},"DOI":"10.1016\/s0305-9006(98)
        in Planning"],"link":[{"URL":"http:\/\/api.elsevier.com\/content\/article\/PII:S030590069
        Planning and Development"]},{"indexed":{"date-parts":[[2017,10,23]],"date-time":"2017-10-
        BV","issue":"1","license":[{"URL":"http:\/\/www.elsevier.com\/tdm\/userlicense\/1.0\/","s
        Technology"],"published-print":{"date-parts":[[1999,1]]},"DOI":"10.1016\/s0032-5910(98)00
        diversion by embedding in crushed salt"],"prefix":"10.1016","volume":"101","author":[{"gi
        Technology"],"link":[{"URL":"http:\/\/api.elsevier.com\/content\/article\/PII:S003259109
        Chemical Engineering"]}],"items-per-page":3,"query":{"start-index":0,"search-terms":null}
  recorded_at: 2018-04-28 15:12:29 GMT
  recorded_with: vcr/0.0.8.9521, webmockr/0.2.2.9119, crul/0.5.2
```


All components of both the request and response are preserved, so that the HTTP client (in this case `crul`) can reconstruct its own response just as it would if it wasn't using `vcr`.

7.4 vcr enabled testing

7.4.1 check vs. test

TLDR: Run `devtools::test()` before running `devtools::check()`

When running tests or checks of your whole package, note that you'll get different results with `devtools::check()` vs. `devtools::test()`. 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()` to create your vcr cassettes.

The same goes for using the RStudio IDE buttons/keyboard shortcuts.

7.4.2 CRAN

There is no one right answer to how to manage your tests for CRAN. The following is a discussion of the various considerations - which should give the reader enough information to make an educated decision.

You can run vcr enabled tests on CRAN. CRAN is okay with files associated with tests, and so in general you can run your vcr enabled tests on CRAN just as you run them locally.

If you do run your vcr enabled tests on CRAN be aware of a few things:

- If your vcr enabled tests require any environment variables or R options, they won't be available on CRAN. In these cases you likely want to skip these tests.
- If your vcr enabled tests have cassettes 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'll want to read the gitignore cassettes section

If you are worried at all about problems with vcr enabled tests on CRAN you can use `testthat::skip_on_cran()` to skip specific tests.

7.4.3 CI sites: Travis, Appveyor, etc.

Similar considerations are involved when dealing with Travis, Appveyor, CircleCI, etc.

There are a few differences. With CI services you can manage environment variables so you can run tests that require API keys, etc. In addition, if tests fail you aren't threatened with your package being taken down.

Chapter 8

vcr usage

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

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

8.1 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")
)
```

```
#> <vcr configuration>
#>   Cassette Dir: /var/folders/24/8k48jl6d249_n_qfxwsl6xvm0000gn/T//RtmprGVhdo/fixtures
#>   Record: once
#>   URI Parser: crul::url_parse
#>   Match Requests on: method, uri
#>   Preserve Bytes?: FALSE
#>   Logging?: FALSE
#>   ignored hosts:
```

```
#>   ignore localhost?: FALSE
#>   Write disk path: /var/folders/24/8k48jl6d249_n_qfxwsl6xvm0000gn/T//RtmprGVhdo/file
```

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.

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 9

vcr configuration

vcr configuration

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

9.1 Get your configuration

Use vcr_configuration() to get the current configuration

```
vcr_configuration()
```

```
#> <vcr configuration>
#>   Cassette Dir: .
#>   Record: once
#>   URI Parser: crul::url_parse
#>   Match Requests on: method, uri
#>   Preserve Bytes?: FALSE
#>   Logging?: FALSE
#>   ignored hosts:
#>   ignore localhost?: FALSE
#>   Write disk path:
```

You can get the default configuration variables via vcr_config_defaults()

```
vcr_config_defaults()
```

```
#> $write_disk_path
```

```
#> NULL
#>
#> $filter_sensitive_data
#> NULL
#>
#> $log_opts
#> $log_opts$file
#> [1] "vcr.log"
#>
#> $log_opts$log_prefix
#> [1] "Cassette"
#>
#> $log_opts$date
#> [1] TRUE
#>
#>
#> $log
#> [1] FALSE
#>
#> $linked_context
#> NULL
#>
#> $cassettes
#> list()
#>
#> $allow_http_connections_when_no_cassette
#> [1] FALSE
#>
#> $clean_outdated_http_interactions
#> [1] FALSE
#>
#> $re_record_interval
#> NULL
#>
#> $turned_off
#> [1] FALSE
#>
#> $preserve_exact_body_bytes
#> [1] FALSE
#>
#> $uri_parser
#> [1] "crul::url_parse"
#>
#> $ignore_request
#> NULL
#>
```

```
#> $ignore_localhost
#> [1] FALSE
#>
#> $ignore_hosts
#> NULL
#>
#> $persist_with
#> [1] "FileSystem"
#>
#> $serialize_with
#> [1] "yaml"
#>
#> $allow_unused_http_interactions
#> [1] TRUE
#>
#> $match_requests_on
#> [1] "method" "uri"
#>
#> $record
#> [1] "once"
#>
#> $dir
#> [1] "."
```

These defaults are set when you load `vcr` - you can override any of them as described below.

9.2 Set configuration variables

Use `vcr_configure()` to set configuration variables.

For example, set a single variable:

```
vcr_configure(
  dir = "foobar/vcr_cassettes"
)
```

```
#> <vcr configuration>
#>   Cassette Dir: foobar/vcr_cassettes
#>   Record: once
#>   URI Parser: crul::url_parse
#>   Match Requests on: method, uri
#>   Preserve Bytes?: FALSE
#>   Logging?: FALSE
```

```
#>   ignored hosts:
#>   ignore localhost?: FALSE
#>   Write disk path:
```

Or many at once:

```
vcr_configure(
  dir = "foobar/vcr_cassettes",
  record = "all"
)
```

```
#> <vcr configuration>
#>   Cassette Dir: foobar/vcr_cassettes
#>   Record: all
#>   URI Parser: crul::url_parse
#>   Match Requests on: method, uri
#>   Preserve Bytes?: FALSE
#>   Logging?: FALSE
#>   ignored hosts:
#>   ignore localhost?: FALSE
#>   Write disk path:
```

9.3 Re-set to defaults

```
vcr_configure_reset()
```

9.4 dir - directory of where cassettes are stored

```
vcr_configure(dir = "new/path")
```

```
#> <vcr configuration>
#>   Cassette Dir: new/path
#>   Record: once
#>   URI Parser: crul::url_parse
#>   Match Requests on: method, uri
#>   Preserve Bytes?: FALSE
#>   Logging?: FALSE
#>   ignored hosts:
#>   ignore localhost?: FALSE
#>   Write disk path:
```

9.5 record - record mode

One of: 'all', 'none', 'new_episodes', 'once'. See `?recording` for info on the options

```
vcr_configure(record = "new_episodes")
```

```
#> <vcr configuration>
#>   Cassette Dir: new/path
#>   Record: new_episodes
#>   URI Parser: crul::url_parse
#>   Match Requests on: method, uri
#>   Preserve Bytes?: FALSE
#>   Logging?: FALSE
#>   ignored hosts:
#>   ignore localhost?: FALSE
#>   Write disk path:
```

9.6 match_requests_on - customize how vcr matches requests

```
vcr_configure(match_requests_on = c('query', 'headers'))
```

```
#> <vcr configuration>
#>   Cassette Dir: new/path
#>   Record: new_episodes
#>   URI Parser: crul::url_parse
#>   Match Requests on: query, headers
#>   Preserve Bytes?: FALSE
#>   Logging?: FALSE
#>   ignored hosts:
#>   ignore localhost?: FALSE
#>   Write disk path:
```

9.7 allow_unused_http_interactions - Allow HTTP connections when no cassette

Default is `TRUE`, and thus does not error when http interactions are unused. You can set to `FALSE` in which case vcr errors when a cassette is ejected and not all http interactions have been used.

```
vcr_configure(allow_unused_http_interactions = FALSE)
```

```
#> <vcr configuration>
#>   Cassette Dir: new/path
#>   Record: new_episodes
#>   URI Parser: crul::url_parse
#>   Match Requests on: query, headers
#>   Preserve Bytes?: FALSE
#>   Logging?: FALSE
#>   ignored hosts:
#>   ignore localhost?: FALSE
#>   Write disk path:
```

9.8 serialize_with - which serializer to use

Right now the only option is `yaml`

```
vcr_configure(serialize_with = "yaml")
```

```
#> <vcr configuration>
#>   Cassette Dir: new/path
#>   Record: new_episodes
#>   URI Parser: crul::url_parse
#>   Match Requests on: query, headers
#>   Preserve Bytes?: FALSE
#>   Logging?: FALSE
#>   ignored hosts:
#>   ignore localhost?: FALSE
#>   Write disk path:
```

9.9 persist_with - which persister to use

Right now the only option is `FileSystem`

```
vcr_configure(persist_with = "FileSystem")
```

```
#> <vcr configuration>
#>   Cassette Dir: new/path
#>   Record: new_episodes
#>   URI Parser: crul::url_parse
```

```
#>   Match Requests on: query, headers
#>   Preserve Bytes?: FALSE
#>   Logging?: FALSE
#>   ignored hosts:
#>   ignore localhost?: FALSE
#>   Write disk path:
```

9.10 ignore requests

9.10.1 ignore_hosts - specify particular hosts to ignore

```
vcr_configure(ignore_hosts = "google.com")
```

```
#> <vcr configuration>
#>   Cassette Dir: new/path
#>   Record: new_episodes
#>   URI Parser: crul::url_parse
#>   Match Requests on: query, headers
#>   Preserve Bytes?: FALSE
#>   Logging?: FALSE
#>   ignored hosts: google.com
#>   ignore localhost?: FALSE
#>   Write disk path:
```

9.10.2 ignore_localhost - ignore all localhost flavors

```
vcr_configure(ignore_localhost = TRUE)
```

```
#> <vcr configuration>
#>   Cassette Dir: new/path
#>   Record: new_episodes
#>   URI Parser: crul::url_parse
#>   Match Requests on: query, headers
#>   Preserve Bytes?: FALSE
#>   Logging?: FALSE
#>   ignored hosts: google.com
#>   ignore localhost?: TRUE
#>   Write disk path:
```

9.10.3 ignore_request - ignore any request for which function is true

```
vcr_configure(ignore_request = function(x) x == 5)
```

```
#> <vcr configuration>
#>   Cassette Dir: new/path
#>   Record: new_episodes
#>   URI Parser: crul::url_parse
#>   Match Requests on: query, headers
#>   Preserve Bytes?: FALSE
#>   Logging?: FALSE
#>   ignored hosts: google.com
#>   ignore localhost?: TRUE
#>   Write disk path:
```

9.11 uri_parser - which uri parser to use

By default we use `httr::parse_url`, but you can use a different one. Remember to pass in the function quoted, and namespaced.

```
vcr_configure(uri_parser = "urltools::url_parse")
```

```
#> <vcr configuration>
#>   Cassette Dir: new/path
#>   Record: new_episodes
#>   URI Parser: urltools::url_parse
#>   Match Requests on: query, headers
#>   Preserve Bytes?: FALSE
#>   Logging?: FALSE
#>   ignored hosts: google.com
#>   ignore localhost?: TRUE
#>   Write disk path:
```

9.12 preserve_exact_body_bytes

Some HTTP servers are not well-behaved and respond with invalid data. Set `preserve_exact_body_bytes` to `TRUE` to base64 encode the result body in order to preserve the bytes exactly as-is. `vcr` does not do this by default, since base64-encoding the string removes the human readibility of the cassette.

```
vcr_configure(preserve_exact_body_bytes = TRUE)
```

```
#> <vcr configuration>
#>   Cassette Dir: new/path
#>   Record: new_episodes
#>   URI Parser: urltools::url_parse
#>   Match Requests on: query, headers
#>   Preserve Bytes?: TRUE
#>   Logging?: FALSE
#>   ignored hosts: google.com
#>   ignore localhost?: TRUE
#>   Write disk path:
```

9.13 allow_http_connections_when_no_cassette

Determines how `vcr` treats HTTP requests that are made when no cassette is in use. When `TRUE`, requests made when there is no vcr cassette in use will be allowed. When `FALSE` (default), an [UnhandledHTTPRequestError] error will be raised for any HTTP request made when there is no cassette in use

```
vcr_configure(allow_http_connections_when_no_cassette = TRUE)
```

```
#> <vcr configuration>
#>   Cassette Dir: new/path
#>   Record: new_episodes
#>   URI Parser: urltools::url_parse
#>   Match Requests on: query, headers
#>   Preserve Bytes?: TRUE
#>   Logging?: FALSE
#>   ignored hosts: google.com
#>   ignore localhost?: TRUE
#>   Write disk path:
```

9.14 write_disk_path

The path to write files to for any requests that write responses to disk. By default this parameter is `NULL`; if you don't set this and you do http requests while using vcr, you'll get an error that you need to set this config variable (or you may just get an invalid path error).

For testing a package, you'll probably want this path to be in your `tests/` directory, perhaps next to your cassettes directory, e.g., if your cassettes are

in `tests/fixtures`, then put your files from requests that write to disk in `tests/files`. Note: in the below example, `vcr_configure` is run from within `tests/testthat`, so you set the paths relative to that location.

```
vcr_configure(dir = "../fixtures", write_disk_path = "../files")
```

```
#> <vcr configuration>
#>   Cassette Dir: ../fixtures
#>   Record: once
#>   URI Parser: crul::url_parse
#>   Match Requests on: method, uri
#>   Preserve Bytes?: FALSE
#>   Logging?: FALSE
#>   ignored hosts:
#>   ignore localhost?: FALSE
#>   Write disk path: ../files
```

If you want to ignore these files in your installed package, add them to `.Rinstignore`. If you want these files ignored on build then add them to `.Rbuildignore`. However, adding these files to `.Rbuildignore` will make tests that depend on these files break because the files won't be found; so you'll likely have to skip the associated tests as well.

Chapter 10

record modes

Record modes dictate under what circumstances http requests/responses are recorded to cassettes (disk). Set the recording mode with the parameter `record` in the `use_cassette()` and `insert_cassette()` functions.

10.1 once

The `once` record mode will:

- Replay previously recorded interactions.
- Record new interactions if there is no cassette file.
- Cause an error to be raised for new requests if there is a cassette file.

It is similar to the `new_episodes` record mode, but will prevent new, unexpected requests from being made (i.e. because the request URI changed or whatever).

`once` is the default record mode, used when you do not set one.

10.2 none

The `none` record mode will:

- Replay previously recorded interactions.
- Cause an error to be raised for any new requests.

This is useful when your code makes potentially dangerous HTTP requests. The `none` record mode guarantees that no new HTTP requests will be made.

10.3 new_episodes

The `new_episodes` record mode will:

- Record new interactions.
- Replay previously recorded interactions.

It is similar to the `once` record mode, but will **always** record new interactions, even if you have an existing recorded one that is similar (but not identical, based on the `match_request_on` option).

10.4 all

The `all` record mode will:

- Record new interactions.
- Never replay previously recorded interactions.

This can be temporarily used to force vcr to re-record a cassette (i.e. to ensure the responses are not out of date) or can be used when you simply want to log all HTTP requests.

Chapter 11

request matching

There are a number of options, some of which are on by default, some of which can be used together, and some alone.

11.1 matching on method

Use the **method** request matcher to match requests on the HTTP method (i.e. GET, POST, PUT, DELETE, etc). You will generally want to use this matcher. The **method** matcher is used (along with the **uri** matcher) by default if you do not specify how requests should match.

11.2 matching on uri

Use the **uri** request matcher to match requests on the request URI. The **uri** matcher is used (along with the **method** matcher) by default if you do not specify how requests should match.

11.3 matching on host

Use the **host** request matcher to match requests on the request host. You can use this (alone, or in combination with **path**) as an alternative to **uri** so that non-deterministic portions of the URI are not considered as part of the request matching.

11.4 matching on path

Use the **path** request matcher to match requests on the path portion of the request URI. You can use this (alone, or in combination with **host**) as an alternative to **uri** so that non-deterministic portions of the URI

11.5 matching on query string

Use the **query** request matcher to match requests on the query string portion of the request URI. You can use this (alone, or in combination with others) as an alternative to **uri** so that non-deterministic portions of the URI are not considered as part of the request matching.

11.6 matching on body

Use the **body** request matcher to match requests on the request body.

11.7 matching on headers

Use the **headers** request matcher to match requests on the request headers.

11.8 Playback repeats

still in progress …

Chapter 12

logging

Use logging to set an IO-like object that `vcr` will log output to. This is a useful way to troubleshoot what `vcr` is doing.

12.1 Setup logging

To set up logging, see `?vcr_logging` use `vcr_configure()`

```
vcr::vcr_configure(
  log = TRUE,
  log_opts = list(file = "vcr.log", log_prefix = "Cassette", date = TRUE)
)
```

- The `log` parameter is a boolean to indicate whether `vcr` should log or not
- The `log_opts` parameter is a named list with various options:
  - `file`: the log file path (it does not get put in the cassette directory, but is at whatever this path is)
  - `log_prefix`: prefix to put in each log entry. the default is `Cassette`
  - `date`: whether to include a time stamp in each log entry or not. format is `YYYY-MM-DD HH:MM:SS`

12.2 The log file

The following is an example log file:

```
[Cassette: 'foobar'] - 2018-04-27 08:36:28 - Init. HTTPInteractionList w/ request matcl
[Cassette: 'foobar'] - 2018-04-27 08:36:28 - Initialized with options: {name: foobar, r
[Cassette: 'foobar'] - 2018-04-27 08:36:28 - Handling request: get http://www.marinespe
[Cassette: 'foobar'] - 2018-04-27 08:36:28 - Checking if {get http://www.marinespecies
[Cassette: 'foobar'] - 2018-04-27 08:36:28 -   method matched: current request [get htt
[Cassette: 'foobar'] - 2018-04-27 08:36:28 -   uri matched: current request [get http:/
[Cassette: 'foobar'] - 2018-04-27 08:36:28 - Identified request type: (stubbed_by_vcr)
[Cassette: 'foobar'] - 2018-04-27 08:36:28 - Checking if {get http://www.marinespecies
[Cassette: 'foobar'] - 2018-04-27 08:36:28 -   method matched: current request [get htt
[Cassette: 'foobar'] - 2018-04-27 08:36:28 -   uri matched: current request [get http:/
[Cassette: 'foobar'] - 2018-04-27 08:36:28 - Found matching interaction for get http:/
```

Internally `vcr` logs certain actions that we think are important steps in the process, including:

- initializing an `HTTPInteractionList` object that holds HTTP interactions
- initializing a `Cassette` with whatever options the user passes in
- declaring what request is being handled
- what request is being checked
- whether there is a match found or not
- when an interaction is recorded, or pulled from a stub, etc.

If you turn off the date, you won't get date entries:

```
vcr::vcr_configure(
  log = TRUE,
  log_opts = list(file = "vcr.log", log_prefix = "Cassette", date = FALSE)
)
```

```
[Cassette: 'foobar'] - {{message}}
```

And you can change the prefix from `Cassette` to something else:

```
vcr::vcr_configure(
  log = TRUE,
  log_opts = list(file = "vcr.log", log_prefix = "Unicorn", date = FALSE)
)
```

```
[Unicorn: 'foobar'] - {{message}}
```

Chapter 13

security

13.1 API keys and such

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.

The before record replacement happens in an internal function `write_interactions()`, while before playback replacement happens in internal function `YAML$deserialize_path()`

```
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.

It's a good idea to not in place of `thing_to_replace` put your actual sensitive key thing, because that defeats the purpose of trying to protect your private data. This is why we highly recommend setting your API keys as environment variables, then you can as seen above just put a call to `Sys.getenv()`, which we'll use internally to get your key, find it anywhere in the HTTP responses, and replace it with your placeholder string.

The reason you want to do this is because you may on purpose or on accident push your cassettes to the public web, and when that happens you don't want your private keys in those cassettes.

Note that the way this is implemented in `vcr` is not super elegant and is not general with respect to the serializer. We only support YAML serializing right now, but when we support other serializers we'll need to change the implementation.

13.2 API keys and tests run in varied contexts

When `vcr` enabled tests are run in different contexts (laptops, CI services, containers, etc.), and those tests can optionally use authentication you can run into problems. An example will best illustrate the issue.

Given an R package `foo`, the package maintainer sets up tests with `vcr`. Functions in package `foo` can optionally use an API key supplied by the user. For the purposes of this example, the API key when given is included as a query parameter (it is not good pratice to pass API keys as query parameter, but its not uncommon).

When the maintainer runs tests locally on their own machine they can unset the environment variable that holds their API key so its not included in vcr cassettes. Additionally, when they run tests on CI systems (e.g., Travis-CI), they do not have an API key set. When tests are run in either of these two locations, the API key is not included in the requests, and thus not included in the cassettes.

Now, consider a contributor that forks the repository and as a first run through the package installs the package, then runs tests. If this contributor does not have an API key set, the tests should run fine. However, once the contributor sets their API key and attempts to run tests (or imagine another contributor that already has an API key set), the tests will fail because the URI now contains an API key as a query parameter in the URL.

There's various combinations of the above problem.

One option for dealing with this is requiring an API key to be set. If you do this you likely want to make sure your actual key is not in the cassettes that will be pushed to the public web; see the bit about `filter_sensitive_data` above for that. If you do require an API key, then you can ensure that whenever tests are run an API key is required; that way there's no way that tests will be run with AND without a key - which can lead to problems. Requiring an API key means you can't run tests in environments where it's not possible to safely set a key, e.g., on CRAN (in which case you must skip tests).

There's no magic way around this problem ~ it's a good thing to be aware of, especially if other people contribute to your package.

13.3 Other security

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

Chapter 14

Turning vcr on & off

Sometimes you may need to turn off `vcr`, either for individual function calls, individual test blocks, whole test files, or for the entire package. The following attempts to break down all the options.

`vcr` has the following four exported functions:

- `turned_off()` - Turns vcr off for the duration of a code block
- `turn_off()` - Turns vcr off completely, so that it no longer handles every HTTP request
- `turn_on()` - turns vcr on; the opposite of `turn_off()`
- `turned_on()` - Asks if vcr is turned on, returns a boolean

Instead of using the above four functions, you could use environment variables to achieve the same thing. This way you could enable/disable `vcr` in non-interactive environments such as continuous integration, Docker containers, or running R non-interactively from the command line. The full set of environment variables `vcr` uses, all of which accept only `TRUE` or `FALSE`:

- `VCR_TURN_OFF`: turn off vcr altogether; set to `TRUE` to skip any vcr usage; default: `FALSE`
- `VCR_TURNED_OFF`: set the `turned_off` internal package setting; this does not turn off vcr completely as does `VCR_TURN_OFF` does, but rather is looked at together with `VCR_IGNORE_CASSETTES`
- `VCR_IGNORE_CASSETTES`: set the `ignore_cassettes` internal package setting; this is looked at together with `VCR_TURNED_OFF`

14.1 turned_off

`turned_off()` lets you temporarily make a real HTTP request without completely turning `vcr` off, unloading it, etc.

What happens internally is we turn off `vcr`, run your code block, then on exit turn `vcr` back on - such that `vcr` is only turned off for the duration of your code block. Even if your code block errors, `vcr` will be turned back on due to use of `on.exit(turn_on())`

```
library(vcr)
library(crul)
turned_off({
  con <- HttpClient$new(url = "https://httpbin.org/get")
  con$get()
})
```

```
#> <crul response>
#>   url: https://httpbin.org/get
#>   request_headers:
#>     User-Agent: libcurl/7.54.0 r-curl/4.3 crul/0.9.0
#>     Accept-Encoding: gzip, deflate
#>     Accept: application/json, text/xml, application/xml, */*
#>   response_headers:
#>     status: HTTP/1.1 200 OK
#>     date: Fri, 14 Feb 2020 19:44:46 GMT
#>     content-type: application/json
#>     content-length: 365
#>     connection: keep-alive
#>     server: gunicorn/19.9.0
#>     access-control-allow-origin: *
#>     access-control-allow-credentials: true
#>   status: 200
```

14.2 turn_off/turn_on

`turn_off()` is different from `turned_off()` in that `turn_off()` is not aimed at a single call block, but rather it turns `vcr` off for the entire package. `turn_off()` does check first before turning `vcr` off that there is not currently a cassette in use. `turn_off()` is meant to make R ignore `vcr::insert_cassette()` and `vcr::use_cassette()` blocks in your test suite - letting the code in the block run as if they were not wrapped in `vcr` code - so that all you have to do to run your tests with cached requests/responses AND with real HTTP requests is toggle a single R function or environment variable.

```
library(vcr)
vcr_configure(dir = tempdir())
# real HTTP request works - vcr is not engaged here
crul::HttpClient$new(url = "https://eu.httpbin.org/get")$get()
# wrap HTTP request in use_cassette() - vcr is engaged here
use_cassette("foo_bar", {
  crul::HttpClient$new(url = "https://eu.httpbin.org/get")$get()
})
# turn off & ignore cassettes - use_cassette is ignored, real HTTP request made
turn_off(ignore_cassettes = TRUE)
use_cassette("foo_bar", {
  crul::HttpClient$new(url = "https://eu.httpbin.org/get")$get()
})
# if you turn off and don't ignore cassettes, error thrown
turn_off(ignore_cassettes = FALSE)
use_cassette("foo_bar", {
  res2=crul::HttpClient$new(url = "https://eu.httpbin.org/get")$get()
})
# vcr back on - now use_cassette behaves as before
turn_on()
use_cassette("foo_bar3", {
  res2=crul::HttpClient$new(url = "https://eu.httpbin.org/get")$get()
})
```

14.3 turned_on

`turned_on()` does what it says on the tin - it tells you if `vcr` is turned on or not.

```
library(vcr)
turn_on()
turned_on()
```

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

```
turn_off()
turned_on()
```

```
#> [1] FALSE
```

14.4 Environment variables

The `VCR_TURN_OFF` environment variable can be used within R or on the command line to turn off `vcr`. For example, you can run tests for a package that uses `vcr`, but ignore any `use_cassette/insert_cassette` usage, by running this on the command line in the root of your package:

```
VCR_TURN_OFF=true Rscript -e "devtools::test()"
```

Or, similarly within R:

```
Sys.setenv(VCR_TURN_OFF = TRUE)
devtools::test()
```

The `VCR_TURNED_OFF` and `VCR_IGNORE_CASSETTES` environment variables can be used in combination to achieve the same thing as `VCR_TURN_OFF`:

```
VCR_TURNED_OFF=true VCR_IGNORE_CASSETTES=true Rscript -e "devtools::test()"
```

Chapter 15

managing cassettes

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

15.1 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.

15.2 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.

15.3 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.

15.4 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 16

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 (13)
- **API key issues**: Running vcr enabled tests in different contexts when API keys are used can have some rough edges. See (13.2)
- **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 (7.4.1)
- **ignored files**: See (15)

16.1 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.

Part IV

Appendix

Chapter 17

Testing considerations

When should you use `webmockr` vs. `vcr` vs. testing real HTTP interactions?

There is no right answer to this question, but rather a range of considerations.

On one hand, it seems appropriate to think about testing your R package in a way that is not sensitive to the remote service the package interacts with being down/etc. However, part of the user experience of using your package will be dealing with intermittent server side issues, for which ideally your package is robust to, and/or at least fails well in reponse to. Of course you can still test your package against failure scenarios without testing real interactions.

In addition to intermittent server side issues, your tests may be performing queries with cached (vcr) or mocked (webmockr) responses that are no longer valid with the current state of the remote service. It may be harmless, for example, the response to some query now returns no data because the data for that entity was removed. But it could be more serious in that the remove service changed their API such that an API route is no longer available or the route name has changed, or similar.

Testing real HTTP interactions should be the slowest option, but has the benefit of not adding any (permanent) files to your package. Mocking tests can be very lite weight, though you can include very heavy responses. Using vcr to cache real responses can lead to many files and sometimes files of large size. It's worth thinking about this trade-off between speed of tests, what can be tested, and files added to your package.

If you're working with in a team, and if you're using vcr, you need to consider where files are stored and make sure access to those files doesn't vary by team member.

A last option is to create a very minimal fake service to run your tests against. The consultancy Thoughtbot wrote a nice post on How to Stub External Services in Tests. Using Ruby, they briefly described mocking with webmock (similar

to `webmockr` in R), using vcr (similar to `vcr` in R), and creating a "fake" (fake service).

Some have found they really don't like having the added vcr cassettes in their projects, and thus prefer mocking.

Chapter 18

session info

Session info for this book

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

```
#> - Session info ---------------------------------------------------------------
#>  setting  value
#>  version  R version 3.6.3 (2020-02-29)
#>  os       macOS Catalina 10.15.3
#>  system   x86_64, darwin15.6.0
#>  ui       X11
#>  language (EN)
#>  collate  en_US.UTF-8
#>  ctype    en_US.UTF-8
#>  tz       UTC
#>  date     2020-03-24
#>
#> - Packages -------------------------------------------------------------------
#>  package     * version date       lib source
#>  assertthat    0.2.1   2019-03-21 [1] CRAN (R 3.6.0)
#>  base64enc     0.1-3   2015-07-28 [1] CRAN (R 3.6.0)
#>  bookdown      0.18    2020-03-05 [1] CRAN (R 3.6.0)
#>  cli           2.0.2   2020-02-28 [1] CRAN (R 3.6.0)
#>  crayon        1.3.4   2017-09-16 [1] CRAN (R 3.6.0)
#>  crul        * 0.9.0   2019-11-06 [1] CRAN (R 3.6.0)
#>  curl          4.3     2019-12-02 [1] CRAN (R 3.6.0)
#>  digest        0.6.25  2020-02-23 [1] CRAN (R 3.6.0)
#>  evaluate      0.14    2019-05-28 [1] CRAN (R 3.6.0)
#>  fansi         0.4.1   2020-01-08 [1] CRAN (R 3.6.0)
#>  fauxpas       0.2.0   2018-03-01 [1] CRAN (R 3.6.0)
```

```
#>  glue          1.3.2   2020-03-12 [1] CRAN (R 3.6.0)
#>  htmltools     0.4.0   2019-10-04 [1] CRAN (R 3.6.0)
#>  httpcode      0.2.0   2016-11-14 [1] CRAN (R 3.6.0)
#>  httr        * 1.4.1   2019-08-05 [1] CRAN (R 3.6.0)
#>  jsonlite      1.6.1   2020-02-02 [1] CRAN (R 3.6.0)
#>  knitr         1.28    2020-02-06 [1] CRAN (R 3.6.0)
#>  lazyeval      0.2.2   2019-03-15 [1] CRAN (R 3.6.0)
#>  magrittr      1.5     2014-11-22 [1] CRAN (R 3.6.0)
#>  R6            2.4.1   2019-11-12 [1] CRAN (R 3.6.0)
#>  Rcpp          1.0.4   2020-03-17 [1] CRAN (R 3.6.0)
#>  rlang         0.4.5   2020-03-01 [1] CRAN (R 3.6.0)
#>  rmarkdown     2.1     2020-01-20 [1] CRAN (R 3.6.0)
#>  rstudioapi    0.11    2020-02-07 [1] CRAN (R 3.6.0)
#>  sessioninfo   1.1.1   2018-11-05 [1] CRAN (R 3.6.0)
#>  stringi       1.4.6   2020-02-17 [1] CRAN (R 3.6.0)
#>  stringr       1.4.0   2019-02-10 [1] CRAN (R 3.6.0)
#>  testthat    * 2.3.2   2020-03-02 [1] CRAN (R 3.6.0)
#>  triebeard     0.3.0   2016-08-04 [1] CRAN (R 3.6.0)
#>  urltools      1.7.3   2019-04-14 [1] CRAN (R 3.6.0)
#>  vcr         * 0.5.0   2020-03-04 [1] CRAN (R 3.6.0)
#>  webmockr    * 0.6.0   2020-03-02 [1] CRAN (R 3.6.0)
#>  withr         2.1.2   2018-03-15 [1] CRAN (R 3.6.0)
#>  xfun          0.12    2020-01-13 [1] CRAN (R 3.6.0)
#>  yaml          2.2.1   2020-02-01 [1] CRAN (R 3.6.0)
#>
#> [1] /Users/runner/runners/2.165.2/work/_temp/Library
#> [2] /Library/Frameworks/R.framework/Versions/3.6/Resources/library
```