…New serializer

vcr now has two serializer options: YAML (which we had before) and JSON (the new one). Serializing here refers to what format the HTTP request and response are put in before being written to a file. If you're a keen observer of these two data formats, you'll know these are not that different, with YAML a superset of JSON. The clear advantage of JSON straight away is that it avoids the common problem with vcr enabled tests on Windows platform where using YAML leads to errors due to something about the `yaml` package. The `jsonlite` package, which does the heavy lifting when using JSON, does not suffer from the same problems on Windows.

Keep in mind that by default the JSON serializer does not pretty print the json to disk to save disk space by eliminating the newlines; you can turn pretty printing on by setting `json_pretty=TRUE`.

With `json_pretty=FALSE` (the default) all the JSON is on one line:

```
{"http_interactions":[{"request":{"method":"get","uri":"https://api.catalogueoflife.
org/dataset/1000/taxon/1/children","headers":{"User-Agent":"r-curl/4.3 crul/1.0.0
rOpenSci(rcol/0.1.0)"}},"response":{"status":{"status_
code":"200","message":"OK","explanation":"Request fulfilled, document
follows"},"body":{"encoding":"UTF-8","file":false,"string":"{\"result
\":[{\"datasetKey\":1000,\"id\":\"2\",\"verbatimKey\":856}"}},"
recorded_at":"2020-12-09 02:22:12 GMT","recorded_with":"vcr/0.5.8.91,
webmockr/0.7.4"}]}
```

When `json_pretty=TRUE` the JSON is on many lines and easier to read, but resulting files will be larger:

```
{
"http_interactions": [
{
"request": {
"method": "get",
"uri": "https://api.catalogueoflife.org/dataset/1000/taxon/1/children",
"headers": {
"User-Agent": "r-curl/4.3 crul/1.0.0 rOpenSci(rcol/0.1.0)"
}
},
"response": {
"status": {
"status_code": "200",
"message": "OK",
"explanation": "Request fulfilled, document follows"
},
"body": {
"encoding": "UTF-8",
"file": false,
"string": "{\"result\":[{\"datasetKey\":1000,\"id\":\"2\",\"
verbatimKey\":856}"
}
},
"recorded_at": "2020-12-09 02:22:12 GMT",
"recorded_with": "vcr/0.5.8.91, webmockr/0.7.4"
}
]
}
```

To see the JSON serializer in action, the rcol package (not on CRAN) uses the JSON serializer – an example JSON fixture.

If you are currently using the YAML serializer and want to use the JSON serializer, delete all of your YAML based cassettes and update your vcr configuration globally:

```
vcr::vcr_configure(dir = "../fixtures", serialize_with = "json")
```

Or, set per cassette:

```
vcr::use_cassette("some_neat_code", {
some_neat_code()
}, serialize_with = "json")
```

Secrets

vcr has two new options for managing secrets: `filter_request_headers` and `filter_response_headers`. They both can be set globally with `vcr_configure()` or per cassette. They both allow for completely removing specific headers as well as replacing the value of a specific header with a value you specify.

These two new filters work differently than `filter_sensitive_data`; the two new ones only remove headers or replace values of headers, whereas `filter_sensitive_data` uses regex to replace values anywhere in the request or response. The three filters can be used in combination or alone.

See the Security section of the HTTP Testing in R book for more on vcr secrets.

Request matching

Two new request matchers now work: `host` and `path`.

Matching on host means that you are matching on just the hostname, e.g., in https://google.com, the hostname or host is google.com. That is, if you match on host you ignore the scheme (so requests could have http or https), and you ignore anything else in the url after the hostname. So if you do requests to https://google.com/foo and https://google.com/bar they will match even though their paths components do not match.

Matching on path means that you are matching on any path components of a URL. For example, if a URL is https://google.com/foo/bar, the path is `foo/bar`. So if you do requests to https://google.com/foo/bar and https://duckduckgo.com/foo/bar they will match even though their hostnames do not match.

See the Configure vcr request matching for more on request matching.

The insert/eject workflow

Previous to this version the only way to use vcr was through use_cassette(). Now you have another option.

The combination of insert_cassette() and eject_cassette() now allow more flexibility in vcr usage.

Whereas `use_cassette()` requires you to pass code that uses vcr into a code block within the function call, `insert_cassette/eject_cassette` allow you to turn on vcr, run any code you want to use vcr without embedding in a code block, then turn it off when you're done. That looks like:

```
insert_cassette("leothelion")
con <- crul::HttpClient$new(url = "https://httpbin.org")
con$get("get") # does use vcr
eject_cassette("leothelion")
con$get("get") # does NOT use vcr
```

Alternatively, you can assign the output of `insert_cassette()` to an object and call `$eject()` on that object.

```
x <- insert_cassette("leothelion")
con <- crul::HttpClient$new(url = "https://httpbin.org")
con$get("get") # does use vcr
x$eject()
con$get("get") # does NOT use vcr
```

This inject/eject workflow has already been used for caching HTTP requests in vignettes.

Better debugging experience

There is a new vcr article debugging that helps guide you through debugging tests that use vcr. Paths work sightly different when running tests hitting a button in RStudio or running `devtools::test()` vs. running tests line by line in an interactive R session. The article includes discussion of a new function `vcr::vcr_test_path()` - based on `testthat::test_path()` - which will set the correct path to vcr files whether you are in an interactive R session or not. The article also includes guidance on logging to help in your debugging process.