Web Services

Jim Harner

10/4/2020

The `httr` package is part of the `tidyverse`. It is a wrapper for the `curl` package, which provide requests for working with the `http` protocol.

```
library(httr)
library(jsonlite)
```

2.6 Web Services

Web Services is a recently introduced phrase. The *Web* and the *HyperText Transfer Protocol (HTTP)* that underlies the communication of data on the Web have become a vital part of our information network and day-to-day environment. Thus, being able to access various forms of data using HTTP is an important facility in a general programming language. We want to be able to:

- download files,
- get data via HTML forms,
- "scrape" HTML page content as data itself, and
- use REST APIs

HTTP is a stateless protocol, i.e., no information is kept between message exchanges. The communication usually takes place over TCP/IP, but other transports can be used.

Communication between a host and a client occurs through a request/response pair. The client initiates an HTTP request message, which is serviced through a HTTP response message in return.

The request message is sent by Uniform Resource Locators (URLs). URLs have the following structure:

`http://www.domain.com:1234/path/to/resource? a=b&x=y`

where:

- `http` is the protocol;
- `www.domain.com` is the host;
- `1234` is the port;
- `path/to/resource` is the resource path;
- `a=b&x=y` is the query.

The protocol `https` is used for secure communications. The default port is 80, but others can be specified. The resource path is the local path to the resource on the server.

The action to be performed on the host is specified via `HTTP` *verbs*. The common request verbs are:

- `GET`: fetch an existing resource. The URL contains all the necessary information the server needs to locate and return the resource.
- `POST`: create a new resource. `POST` requests usually carry a payload that specifies the data for the new resource.
- `PUT`: update an existing resource. The payload may contain the updated data for the resource.
- `DELETE`: delete an existing resource.

`PUT` and `DELETE` can be considered specialized versions of the `POST` verb, i.e., they can be packaged as POST requests with the payload containing the exact action: `create`, `update` or `delete`.

The client can initiate requests to the server. In return, the server responds with *status codes* and *message payloads*.

The status codes are:

- 1xx: Informational Messages—provisional;
- 2xx: Successful—the request was successfully processed;
- 3xx: Redirection—the client must take additional action;
- 4xx: Client Error— the client is at fault, either by requesting an invalid resource or making a bad request;
- 5xx: Server Error—a server failure while processing the request.

The request or response message has the following generic structure:

```
message = <start-line>
          *(<message-header>)
          CRLF
          [<message-body>]
```

where

```
<start-line> = Request-Line | Status-Line
<message-header> = Field-Name ':' Field-Value
```

Chrome's WebKit inspector can monitor HTTP communications, but other paid applications provide more functionality.

cURL is an open source software project providing a library and command-line tool for transferring data using various protocols. It is written in C and is cross-platform

`libcurl` is a free URL transfer library, supporting FTP, FTPS, HTTP (with HTTP/2 support), HTTPS, and many other protocols. The library supports HTTPS certificates, HTTP POST, HTTP PUT, FTP uploading, Kerberos, HTTP form-based upload, and other services.

`curl` is a command line tool for getting or sending files using URL syntax. It supports a range of common Internet protocols, currently including HTTP, HTTPS, FTP, FTPS, SCP, SFTP, TFTP, LDAP, and others.

Wikipedia gives a more complete reference: https://en.wikipedia.org/wiki/CURL
To experiment with `curl` use RStudio's shell and type in the following (note: `curl` is not installed, but libcurl is.):

```
curl --version
curl http://httpbin.org/ip
curl http://httpbin.org/user-agent
```

We can get the content of http://httpbin.org/ by:

```
curl http://httpbin.org/get
{
  "args": {},
  "headers": {
    "Accept": "*/*",
    "Host": "httpbin.org",
    "User-Agent": "curl/7.29.0"
  },
  "origin": "157.182.3.5",
  "url": "http://httpbin.org/get"
}
```

The later uses the `get` verb directly using `curl` from the command line. The output is given for reference.

The `curl` command-line tool is easy enough to use, but we often want to extract text, parse it, and then use various text manipulation functions or machine learning algorithms to get meaning from the text. This is why our focus will be on R packages that use the `libcurl` library. We can build workflows to extract, parse, manipulate, etc. directly in R.

2.6.1 `curl` R Package

The `curl` package is a modern interface to the `cURL` library. It implements R's connection interface with support for encryption (https:// and ftps://), `gzip` compression, authentication, and other `libcurl` features. `curl` is meant to be a replacement of the aging `Rcurl` package.

The best introduction is given by its CRAN vignette.

The `curl` package has been made more accessible by the `httr` package.

2.6.2 `httr` R Package

The `httr` package provides a wrapper for the `curl` package.

The following two package vignettes describe how the package works:

1. httr quickstart guide
2. api-package

The text in this subsection is extracted from the `httr` quickstart guide.

To make a request call `GET()` with a `url`:

```
r <- GET("http://httpbin.org/get")
```

This gives you a response object. Printing a response object gives you useful information: the actual `url` used, the `http` status, the file (content) type, the size, and if it's a text file, the first few lines of output.

```
r
```

```
## Response [http://httpbin.org/get]
##   Date: 2020-10-04 19:10
##   Status: 200
##   Content-Type: application/json
##   Size: 364 B
## {
##   "args": {},
##   "headers": {
##     "Accept": "application/json, text/xml, application/xml, */*",
```

```
##     "Accept-Encoding": "deflate, gzip",
##     "Host": "httpbin.org",
##     "User-Agent": "libcurl/7.58.0 r-curl/4.3 httr/1.4.1",
##     "X-Amzn-Trace-Id": "Root=1-5f7a1e0b-79761f3b77fbf06627e0eb32"
##   },
##   "origin": "67.165.80.93",
## ...
```

The R output is similar to `curl` output, but more complete. You can pull out important parts of the response with helper methods (see below).

You can also use the `GET()`, `HEAD()`, `POST()`, `PATCH()`, `PUT()` and `DELETE()` verbs. `GET()` is used by your browser when requesting a page, and `POST()` is usually used when submitting a form to a server. `PUT()`, `PATCH()` and `DELETE()` are used most often by web APIs.

The Response The data sent back from the server consists of three parts:

- the status line,
- the headers, and
- the body.

The most important part of the status line is the http status code which tells you whether or not the request was successful.

The status code is a three digit number that summaries whether or not the request was successful (as defined by the server). You can access the status code along with a descriptive message using `http_status()` or more compactly `status_code()`.

```
r <- GET("http://httpbin.org/get")
status_code(r)
```

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

```
# Or just access the raw code: r$status_code
```

A successful request always returns a status of 200. Common errors are 404 (file not found) and 403 (permission denied).

The output of `headers` is a named list. For example, the content type is given by:

```
headers(r)$`content-type`
```

```
## [1] "application/json"
```

There are three ways to access the body of the request (`text`, `raw`, and `parsed`), all using `content()`. `content(r, as = "text")` returns the body as a character vector:

```
content(r, as = "text")
```

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

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

`httr` will automatically decode content from the server using the encoding supplied in the content-type HTTP header. The output type that is most readable is `parsed`.

```
str(content(r, as = "parsed"))
```

```
## List of 4
##  $ args   : Named list()
```

```
##  $ headers:List of 5
##   ..$ Accept         : chr "application/json, text/xml, application/xml, */*"
##   ..$ Accept-Encoding: chr "deflate, gzip"
##   ..$ Host           : chr "httpbin.org"
##   ..$ User-Agent     : chr "libcurl/7.58.0 r-curl/4.3 httr/1.4.1"
##   ..$ X-Amzn-Trace-Id: chr "Root=1-5f7a1e0b-652884927c8bdb7b534cf433"
##  $ origin : chr "67.165.80.93"
##  $ url    : chr "http://httpbin.org/get"
```

A common way of sending simple key-value pairs to the server is the query string: e.g. `http://httpbin.org/get?key=val`. `httr` allows you to provide these arguments as a named list with the query argument. For example, if you wanted to pass `key1=value1` and `key2=value2` to `http://httpbin.org/get` you could do:

```
r <- GET("http://httpbin.org/get",
  query = list(key1 = "value1", key2 = "value2")
)
content(r)$args
```

```
## $key1
## [1] "value1"
##
## $key2
## [1] "value2"
```

Cookies are simple key-value pairs like the query string, but they persist across multiple requests in a session (because they're sent back and forth every time). To send your own cookies to the server, use `set_cookies()`:

```
r <- GET("http://httpbin.org/cookies", set_cookies("MeWant" = "cookies"))
content(r)$cookies
```

```
## $MeWant
## [1] "cookies"
```

`POST()` can include data in the body of the request. `httr` allows you to supply this in a number of different ways. The most common way is a named list. You can use the `encode` argument to determine how this data is sent to the server:

```
url <- "http://httpbin.org/post"
body <- list(a = 1, b = 2, c = 3)

# Form encoded
r <- POST(url, body = body, encode = "form")
r
```

```
## Response [http://httpbin.org/post]
##   Date: 2020-10-04 19:10
##   Status: 200
##   Content-Type: application/json
##   Size: 561 B
## {
##   "args": {},
##   "data": "",
##   "files": {},
##   "form": {
##     "a": "1",
##     "b": "2",
##     "c": "3"
##   },
```

```
##   "headers": {
## ...
```

Multi-part encoding (`encode = "multiport"`) is the default, but it is equivalue to `form` here.

JSON encoding is a readable encoding, which puts the data in the `data` key.

```
r <- POST(url, body = body, encode = "json")
r
```

```
## Response [http://httpbin.org/post]
##   Date: 2020-10-04 19:10
##   Status: 200
##   Content-Type: application/json
##   Size: 561 B
## {
##   "args": {},
##   "data": "{\"a\":1,\"b\":2,\"c\":3}",
##   "files": {},
##   "form": {},
##   "headers": {
##     "Accept": "application/json, text/xml, application/xml, */*",
##     "Accept-Encoding": "deflate, gzip",
##     "Content-Length": "19",
##     "Content-Type": "application/json",
## ...
```

You can also send files from your local disk:

```
POST(url, body = upload_file("mypath.txt"))
```

`upload_file()` will guess the mime-type from the extension using the type argument. These uploads stream the data to the server: the data will be loaded in R in chunks then sent to the remote server. This means that you can upload files that are larger than memory.

The api-package vignette describes how to write an R client for a web API.

The REST API

This text is taken from Learn REST: a Tutorial. I recommend you look at the complete tutorial.

Increasingly REST with a JSON response is being used for web services, e.g., see Fetching JSON data from REST APIs.

REST (or ReST) stands for Representational State Transfer, which relies on a stateless, client-server communications protocol. It is a simple architecture for designing networked applications and nearly always uses the HTTP protocol to make calls between machines.

RESTful applications use HTTP requests to post data (create and/or update), read data (make queries), and delete data. Thus, REST uses HTTP for all four CRUD (Create/Read/Update/Delete) operations.

If we are simply fetching data, which is the most likely activity when extracting text from the web, the simplest way is to use `fromJSON` in the `jsonlite` package.

```
hadley_orgs <- fromJSON("https://api.github.com/users/hadley/orgs")
hadley_repos <- fromJSON("https://api.github.com/users/hadley/repos")
# gg_commits <- fromJSON("https://api.github.com/repos/hadley/ggplot2/commits")
gg_issues <- fromJSON("https://api.github.com/repos/hadley/ggplot2/issues")

# to see the structure of gg_issues (too long to print)
# str(gg_issues)
```

```r
# latest issues
paste(format(gg_issues$user$login), ":", gg_issues$title)
```

```
##  [1] "domq            : scale_x_binned() vs. POSIXct"
##  [2] "llrs            : Describe what is ymin, ymax"
##  [3] "swo             : Feature request: ggsave takes multiple filenames"
##  [4] "sieste          : Area units in midwest data frame"
##  [5] "geotheory       : ggsave to svg fails for certain themes"
##  [6] "JanaJarecki     : is scale_x_binned() supposed to not play with geom_boxplot() ?"
##  [7] "petrbouchal     : Top key missing in guide_bins when reverse = TRUE"
##  [8] "toobiwankenobi  : geom_sf() does not support scale_y_reverse()"
##  [9] "MartinEarle     : ggsave inproperly rendering geom_line with factored y-variable"
## [10] "morgan121       : Change alignment of only 1 facet title"
## [11] "teunbrand       : Datetime scales don't support out of bounds (oob) arguments"
## [12] "henrikmidtiby   : Title is placed at the bottom of the plot when coord_fixed and scale_y_revers
## [13] "vinay-swamy     :  scale_fill_discrete makes fills with grey when using a named vector "
## [14] "twest820        : documentation: values list for legend.box.just appears incomplete"
## [15] "yutannihilation : Backtransform data before mapping statistics"
## [16] "davidchall      : Feature request: support for tibble aesthetics"
## [17] "netique         : #4173 lambda functions in discrete scales & facets"
## [18] "twest820        : awkward behavior from scale_fill_viridis_c(trans = \"log\")"
## [19] "jtlandis        : geom_bar does not play well with scale_fill_binned"
## [20] "ignaczzs        : control over discrete_scale when faceting"
## [21] "tuberculo       : Density of each fill or color of weighted geom_density sum to one, but it wa
## [22] "edzer           : ggplot breaks when facet variable is `POSIXct` with `tz` attribute specified
## [23] "werkstattcodes  : Harmonize width and location of horizontal bars across facets with different
## [24] "netique         : Allow passing arguments to the custom labelling functions"
## [25] "bersbersbers    : hjust = \"outward\" not working for some angles"
## [26] "orrymr          : Change help message in geom_bindot() to not include reference to non-..."
## [27] "teunbrand       : Possible bug in `stage()`/`after_stat()` with scale transformations."
## [28] "yutannihilation : Expose x_aes and y_aes"
## [29] "apoorvalal      : Setting `options(ggplot2.discrete.fill)` overrides for aes(colour) with bad
## [30] "rlh1994         : New ggplot2.discrete.fill option doesn't work with ordered factor columns"
```

These results can use text analysis to analyze the issues.