fhircrackr: Download FHIR resources

2021-03-04

This vignette covers the following topics:

- Building FHIR search requests
- Downloading resources from a FHIR server
- Dealing with HTTP errors
- Dealing with large data sets
- Downloading the capability statement

Before running any of the following code, you need to load the `fhircrackr` package:

```r
library(fhircrackr)
```

FHIR search requests

To download FHIR resources from the server, you need to specify which resources you want with a *FHIR search request*. If you are already familiar and comfortable with FHIR search, you can just define your search request as a simple string that you provide to `fhir_search()`. In that case you can skip the following paragraph, as the first part of this vignette introduces the basics of FHIR search and some functions to build valid FHIR search requests with `fhircrackr`.

A FHIR search request will mostly have the form `[base]/[type]?parameter(s)`, where `[base]` is a URL to the FHIR endpoint you are trying to access, `[type]` refers to the type of resource you are looking for and `parameter(s)` characterize specific properties those resources should have. The function `fhir_build_request()` offers a solution to bring those three components together correctly, taking care of proper formatting for you. You use this function in conjunction with three sub-functions: `fhir_base()`, `fhir_resource()` and `fhir_key_value()`, will format the three components separately, the results of these functions are then fed to `fhir_build_request()`. We'll explain those subfunctions now:

The base URL

`fhir_base()` takes a string containing the base URL to your FHIR endpoint, removes white space and trailing slashes and names the string for `fhir_build_request()`:

```r
fhir_base(" http://hapi.fhir.org/baseR4/")
#                          base
# "http://hapi.fhir.org/baseR4"
```

The resource type

The information provided with this function determines the kind of resource you are getting back from the server and also determines the kinds of search parameters that are allowed. A list of all currently available resource types can be found at https://hl7.org/FHIR/resourcelist.html. The function `fhir_resource()` will take a string with a resource name, check that name against the list at hl7.org and will also do some of the formatting for you. For example `fhir_resource()` will convert all letters to capital that need to be capital letters.:

```
fhir_resource("medicationadministration")
#                     resource
# "MedicationAdministration"
```

If the resource you provided is not in the list at hl7.org, there will be a warning:

```
fhir_resource("inventedResource")
# Warning in fhir_resource("inventedResource"): It seems that the resource you
# provided is not one of the official resource types from https://hl7.org/FHIR/
# resourcelist.html. If you are sure this resource exists on your server you can
# ignore this warning.
#           resource
# "InventedResource"
```

The search parameters

You can add zero, one, or multiple search parameters to the request. If you don't give any parameters, the search will just return all resources of the specified type from the server. Search parameters generally come in the form `key = value`. There are a number of resource independent parameters that can be found under https://www.hl7.org/fhir/search.html#Summary. These parameters usually have a `_` at the beginning. `_sort = status` for examples sorts the results by their status, `_include = Observation:patient`, will include the linked Patient resources in a search for Observation resources.

Apart from the resource independent parameters, there are also resource dependent parameters referring to elements specific to that resource. These parameters come without a `_` and you can find a list of them at the end of every resource site e.g. at https://www.hl7.org/fhir/patient.html#search for the Patient resource. An example of such a parameter would be `birthdate = lt2000-01-01` for patients born before the year 2000 or `gender = female` to get female patients only. The function `fhir_key_value()` takes a pair of key and value and formats them correctly:

```
fhir_key_value(key = "birthdate", value = "lt2000-01-01", url_enc = TRUE)
#                     keyval
# "birthdate=lt2000-01-01"

fhir_key_value(key = "code", value = "http://loinc.org|1751-1", url_enc = TRUE )
#                                   keyval
# "code=http%3A%2F%2Floinc.org%7C1751-1"
```

The main use of this function is the URL encoding it performs, e.g. converting the `|` that is used to divide codesystem from code into `%7C`. This is not necessary with all servers, but some will fail if special characters aren't encoded properly.

Putting the request together

In practice you will never need to use the three functions above alone, but always in combination with a call to `fhir_build_request()`:

```
fhir_build_request(
    fhir_base(" http://hapi.fhir.org/baseR4/"),
    fhir_resource("patient"),
    fhir_key_value(key = "birthdate", value = "lt2000-01-01", url_enc = TRUE),
    fhir_key_value(key = "_count", value = "10")
)
# [1] "http://hapi.fhir.org/baseR4/Patient?birthdate=lt2000-01-01&_count=10"
```

You have to provide exactly one base and one resource and can provide none or as many key value pairs as you want.

Accessing the current request

Whenever you call `fhir_build_request()` or `fhir_search()`, the corresponding FHIR search request will be saved implicitly and can be accessed like this:

```
fhir_current_request()
# [1] "http://hapi.fhir.org/baseR4/Patient?birthdate=lt2000-01-01&_count=10"
```

You can update it with new search parameters using `fhir_update_request()`. If you set the argument `append=FALSE`, the key value pairs in the current request are overwritten:

```
fhir_update_request(fhir_key_value(key = "gender", value = "male"),
                    append = FALSE,
                    return_request = TRUE)
# [1] "http://hapi.fhir.org/baseR4/Patient?gender=male"
```

If you set `append=TRUE`, the new pairs are appended to the current ones:

```
fhir_current_request()
# [1] "http://hapi.fhir.org/baseR4/Patient?gender=male"
fhir_update_request(fhir_key_value(key = "birthdate", value = "lt2000-01-01"),
                    append = TRUE,
                    return_request = FALSE)
fhir_current_request()
# [1] "http://hapi.fhir.org/baseR4/Patient?gender=male&birthdate=lt2000-01-01"
```

You can save the requests you build explicitly in an object and provide this object to the `request` argument of `fhir_search()`. If you call `fhir_search()` without providing an explicit request however, the function will automatically call `fhir_current_request()`.

Download FHIR resources from a server

To download resources from a server, you use the function `fhir_search()` and provide a FHIR search request as a string, either one you have written by hand or one you have created with `fhir_build_request()`.

Basic request

We will start with a very simple example and use `fhir_search()` to download Patient resources from a public HAPI server using an explicitly "handwritten" request:

```
patient_bundles <- fhir_search(request="http://fhir.hl7.de:8080/baseDstu3/Patient",
                               max_bundles=2, verbose = 0)
```

You could also build the request using `fhir_build_request()` like this:

```
fhir_build_request(fhir_base("http://fhir.hl7.de:8080/baseDstu3"),
                   fhir_resource("Patient"))

#implicitly calls fhir_current_request()
patient_bundles <- fhir_search(max_bundles=2, verbose = 0)
```

In the latter case `fhir_search()` will automatically look for the last request that was either used in `fhir_search()` or built with `fhir_build_request()`. To have a look at this request, you can use `fhir_current_request()`:

```
fhir_build_request(fhir_base("http://fhir.hl7.de:8080/baseDstu3"),
                   fhir_resource("Patient"))
# [1] "http://fhir.hl7.de:8080/baseDstu3/Patient"
```

```
fhir_current_request()
# [1] "http://fhir.hl7.de:8080/baseDstu3/Patient"
```

In general, a FHIR search request returns a *bundle* of the resources you requested. If there are a lot of resources matching your request, the search result isn't returned in one big bundle but distributed over several of them, sometimes called *pages*, the size of which is determined by the FHIR server. If the argument `max_bundles` is set to its default `Inf`, `fhir_search()` will return all available bundles/pages, meaning all resources matching your request. If you set it to `2` as in the example above, the download will stop after the first two bundles. Note that in this case, the result *may not contain all* the resources from the server matching your request, but it can be useful to first look at the first couple of search results before you download all of them.

If you want to connect to a FHIR server that uses basic authentication, you can supply the arguments `username` and `password`.

Because endpoints can sometimes be hard to reach, `fhir_search()` will start five attempts to connect to the endpoint before it gives up. With the arguments `max_attempts` and `delay_between_attempts` you can control this number as well the time interval between attempts.

As you can see in the next block of code, `fhir_search()` returns a list of xml objects where each list element represents one bundle of resources, so a list of two xml objects in our case:

```
length(patient_bundles)
# [1] 2
str(patient_bundles[[1]])
# List of 2
#  $ node:<externalptr>
#  $ doc :<externalptr>
#  - attr(*, "class")= chr [1:2] "xml_document" "xml_node"
```

If for some reason you cannot connect to a FHIR server at the moment but want to explore the bundles anyway, the package provides an example list of bundles containing Patient resources. See `?patient_bundles` for how to use it.

More than one resource type

In many cases, you will want to download different types of FHIR resources belonging together. For example you might want to download all MedicationStatement resources with the snomed code `429374003` and also download the Patient resources these MedicationStatements refer to. The FHIR search request to do this can be built in two ways. Either you write it down yourself without URL encoding, which will be fine for most but not all servers. For long requests it makes sense to split up the string:

```
request <- paste0("https://hapi.fhir.org/baseR4/",
                  "MedicationStatement?",
                  "code=http://snomed.info/ct|429374003",
                  "&_include=MedicationStatement:subject")
```

Or you use `fhir_build_request()`:

```
request <- fhir_build_request(
            fhir_base("https://hapi.fhir.org/baseR4"),
            fhir_resource("MedicationStatement"),
            fhir_key_value(key = "code", value = "http://snomed.info/ct|429374003"),
            fhir_key_value(key = "_include", value = "MedicationStatement:subject")
)
```

Then you provide the request to `fhir_search()`:

```r
medication_bundles <- fhir_search(request = request, max_bundles = 3)
```

These bundles now contain two types of resources, MedicationStatement resources as well as Patient resources. If you want to have a look at the bundles, it is not very useful to print them to the console. Instead just save them as xml-files to a directory of you choice and look at the resources there:

```r
fhir_save(medication_bundles, directory = "MyProject/medicationBundles")
```

If you want to have a look at a bundle like this but don't have access to a FHIR server at the moment, check out `?medication_bundles`.

Dealing with HTTP Errors

`fhir_search()` internally sends a `GET` request to the server. If anything goes wrong, e.g. because you request wasn't valid or the server caused an error, the result of you request will be a HTTP error. `fhir_search()` will print the error code along with some suggestions for the most common errors to the console.

To get more detailed information on the error response, you can set the argument `log_errors` to `1` or `2`. This will write a log file with error information to your working directory, either as a csv-file or as a xml-file.

```r
medication_bundles <- fhir_search(request = request, max_bundles = 3, log_errors = 2)
```

Dealing with large data sets

If you want to download a lot of resources from a server, you might run into several problems.

First of all, downloading a lot of resources will require a lot of time, depending on the performance of your FHIR server. Because `fhir_search()` essentially runs a loop pulling bundle after bundle, downloads can usually be accelerated if the bundle size is increased, because that way we can lower the number of requests to the server. You can achieve this by adding `_count=` parameter to your FHIR search request. `http://hapi.fhir.org/baseR4/Patient?_count=500` for example will pull patient resources in bundles of 500 resources from the server.

A problem that is also related to the number of requests to the server is that sometimes servers might crash, when too many requests are sent to them in a row. The third problem is that large amounts of resources can at some point exceed the working memory you have available. There are two solutions to the problem of crashing servers and working memory:

1. Use the save_to_disc argument of fhir_search()

If you set `save_to_disc=TRUE` in your call to `fhir_search()`, the bundles will not be combined in a bundle list that is returned when the downloading is done, but will instead be saved as xml-files to the directory specified in the argument `directory` one by one. This way, the R session will only have to keep one bundle at a time in the working memory and if the server crashes halfway trough, all bundles up to the crash are safely saved in your directory. You can later load them using `fhir_load()`:

```r
fhir_search("http://hapi.fhir.org/baseR4/Patient", max_bundles = 10,
            save_to_disc=TRUE, directory ="MyProject/downloadedBundles"))

bundles<- fhir_load("MyProject/downloadedBundles")
```

2. Use fhir_next_bundle_url()

Alternatively, you can also use `fhir_next_bundle_url()`. This function returns the url to the next bundle from you most recent call to `fhir_search()`:

```r
fhir_next_bundle_url()
# [1] "http://hapi.fhir.org/baseR4?_getpages=0be4d713-a4db-4c27-b384-b772deabcbc4&_getpagesoffset=200&_
```

To get a better overview, we can split this very long link along the `&`:

```
strsplit(fhir_next_bundle_url(), "&")
# [[1]]
# [1] "http://hapi.fhir.org/baseR4?_getpages=0be4d713-a4db-4c27-b384-b772deabcbc4"
# [2] "_getpagesoffset=200"
# [3] "_count=20"
# [4] "_pretty=true"
# [5] "_bundletype=searchset"
```

You can see two interesting numbers: `_count=20` tells you that the queried hapi server has a default bundle size of 20. `getpagesoffset=200` tells you that the bundle referred to in this link starts after resource no. 200, which makes sense since the `fhir_search()` request above downloaded 10 bundles with 20 resources each, i.e. 200 resources. If you use this link in a new call to `fhir_search`, the download will start from this bundle (i.e. the 11th bundle with resources 201-220) and will go on to the following bundles from there.

When there is no next bundle (because all available resources have been downloaded), `fhir_next_bundle_url()` returns `NULL`.

If a download with `fhir_search()` is interrupted due to a server error somewhere in between, you can use `fhir_next_bundle_url()` to see where the download was interrupted.

You can also use this function to avoid memory issues. Th following block of code utilizes `fhir_next_bundle_url()` to download all available Observation resources in small batches of 10 bundles that are immediately cracked and saved before the next batch of bundles is downloaded. Note that this example can be very time consuming if there are a lot of resources on the server, to limit the number of iterations uncomment the lines of code that have been commented out here:

```
#Starting fhir search request
url <- "http://hapi.fhir.org/baseR4/Observation?_count=500"

#count <- 0

while(!is.null(url)){

    #load 10 bundles
    bundles <- fhir_search(url, max_bundles = 10)

    #crack bundles
    dfs <- fhir_crack(bundles, list(Obs=list(resource = "//Observation")))

    #save cracked bundle to RData-file (can be exchanged by other data type)
    save(tables, file = paste0(temp_dir,"/table_", count, ".RData"))

    #retrieve starting point for next 10 bundles
    url <- fhir_next_bundle_url()

    #count <- count + 1
    #if(count >= 20) {break}
}
```

Download Capability Statement

The capability statement documents a set of capabilities (behaviors) of a FHIR Server for a particular version of FHIR. You can download this statement using the function `fhir_capability_statement()`:

```r
cap <- fhir_capability_statement("http://hapi.fhir.org/baseR4/", verbose = 0)
```

`fhir_capability_statement()` takes a FHIR server endpoint and returns a list of data frames containing all information from the capability statement of this server. This information can be useful to determine, for example, which FHIR search parameters are implemented in you FHIR server.

Next steps

To learn about how `fhircrackr` allows you to convert the downloaded FHIR resources into data.frames/data.tables, see the vignette on flattening FHIR resources.