fhircrackr Intro: Handling HL7® FHIR® Resources in R

2021-03-09

Introduction

`fhircrackr` is a package designed to help analyzing HL7 FHIR[1] resources.

FHIR stands for *Fast Healthcare Interoperability Resources* and is a standard describing data formats and elements (known as "resources") as well as an application programming interface (API) for exchanging electronic health records. The standard was created by the Health Level Seven International (HL7) health-care standards organization. For more information on the FHIR standard, visit https://www.hl7.org/fhir/.

While FHIR is a very useful standard to describe and exchange medical data in an interoperable way, it is not at all useful for statistical analyses of data. This is due to the fact that FHIR data is stored in many nested and interlinked resources instead of matrix-like structures.

Thus, to be able to do statistical analyses a tool is needed that allows converting these nested resources into data frames. This process of tabulating FHIR resources is not trivial, as the unpredictable degree of nesting and connectedness of the resources makes generic solutions to this problem not feasible.

We therefore implemented a package that makes it possible to download FHIR resources from a server into R and to tabulate these resources into (multiple) data frames.

The package is still under development. The CRAN version of the package contains all functions that are already stable, for more recent (but potentially unstable) developments, the development version of the package can be downloaded from GitHub using `devtools::install_github("POLAR-fhiR/fhircrackr")`.

This vignette is an introduction on the basic functionalities of the `fhircrackr` and should give you a broad overview over what the package can do. For more detailed instructions on each subtopic please have a look the other vignettes. This introduction covers the following topics:

- Prerequisites
- Downloading resources from a FHIR server
- Flattening resources
- Multiple entries
- Saving and loading downloaded bundles

Prerequisites

The complexity of the problem requires a couple of prerequisites both regarding your knowledge and access to data. We will shortly list the preconditions for using the `fhircrackr` package here:

1. First of all, you need the base URL of the FHIR server you want to access. If you don't have your own FHIR server, you can use one of the available public servers, such as `https://hapi.fhir.org/baseR4` or `http://fhir.hl7.de:8080/baseDstu3`. The base URL of a FHIR server is often referred to as [base].

[1] FHIR is the registered trademark of HL7 and is used with the permission of HL7. Use of the FHIR trademark does not constitute endorsement of this product by HL7

2. To download resources from the server, you should be familiar with FHIR search requests. FHIR search allows you to download sets of resources that match very specific requirements. The `fhircrackr` package offers some help building FHIR search requests, for this please see the vignette on downloading FHIR resources.

3. In the first step, `fhircrackr` downloads the resources in xml format into R. To specify which elements from the FHIR resources you want in your data frame, you should have at least some familiarity with XPath expressions. A good tutorial on XPath expressions can be found here.

In the following we'll go through a typical workflow with `fhircrackr` step by step. The first and foremost step is of course, to install and load the package:

```
install.packages("fhircrackr")
library(fhircrackr)
```

Downloading resources

To download resources from a FHIR server, you need to send a FHIR search request using `fhir_search()`. This introduction will not go into the details of building a valid FHIR search request. For that, please see the vignette on downloading FHIR resources. Here we will use a simple example of downloading all Patient resources from a public HAPI server:

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

The minimum information `fhir_search()` requires is a string containing the full FHIR search request in the argument `request` which you can either provide explicitly or by a call to `fhir_build_url()` before. 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. If the argument `max_bundles` is set to its default `Inf`, `fhir_search()` will return all available bundles, 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.

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

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 following functions anyway, the package provides two example lists of bundles containing Patient and MedicationStatement resources. See `?patient_bundles` and `?medication_bundles` for how to use them.

Flattening resources

Now we know that inside these xml objects there is the patient data somewhere. To get it out, we will use `fhir_crack()`. The most important argument `fhir_crack()` takes is `bundles`, the list of bundles that is returned by `fhir_search()`. The second important argument is `design`, an object that tells the function

which data to extract from the bundle. `fhir_crack()` returns a list of data.frames (the default) or a list of data.tables (if argument `data.tables=TRUE`).

The proper format of a `design` in `fhir_crack()` is described in detail in the vignette on flattening resources. Please refer to this document for more information, as we will just use one example of a design here.

In general, `design` is a named list containing one element per data frame that will be created. We call these elements *data.frame descriptions*. The names of the data.frame descriptions in `design` are also going to be the names of the resulting data frames. It usually makes sense to create one data frame per type of resource. Because we have just downloaded resources of the type Patient, the `design` here would be a list of length 1, containing just one data.frame description. A full data.frame description is a list with the elements *resource*, *cols* and *style* and can look as follows:

```
#define design
design <- list(

    Patients = list(

        resource = "//Patient",

        cols = list(
            PID           = "id",
            use_name      = "name/use",
            given_name    = "name/given",
            family_name   = "name/family",
            gender        = "gender",
            birthday      = "birthDate"
        ),

        style = list(
            sep = "|",
            brackets = c("[","]"),
            rm_empty_cols = FALSE
        )
    )
)
```

All three elements of `style` can also be controlled directly by the `fhir_crack()` arguments `sep`, `brackets` and `remove_empty_columns`. If the function arguments are `NULL` (their default), the values provided in `style` are used, if they are not NULL, they will overwrite any values in `style`. If both the function arguments and the `style` component of the data.frame description are NULL, default values(`sep=" "`, `brackets = NULL`, `rm_empty_cols=TRUE`) will be assumed.

After it is defined, the design can be used in `fhir_crack()` like this:

```
#flatten resources
list_of_tables <- fhir_crack(bundles = patient_bundles, design = design, verbose = 0)

#have look at the results
head(list_of_tables$Patients)
#>       PID      use_name    given_name  family_name  gender      birthday
#> 1 [1]1282 [1.1]official      [1.1]Sam [1.1]Fhirman    <NA>          <NA>
#> 2  [1]267          <NA> [1.1]Testfall   [1.1]Nr. 1    <NA> [1]1960-10-04
#> 3  [1]722          <NA>     [1.1]Rick [1.1]Sanchez [1]male [1]1982-01-01
#> 4  [1]731          <NA>     [1.1]Rick [1.1]Sanchez [1]male [1]1982-01-01
#> 5  [1]736          <NA>     [1.1]Rick [1.1]Sanchez [1]male [1]1982-01-01
#> 6  [1]737          <NA>     [1.1]Rick [1.1]Sanchez [1]male [1]1982-01-01
```

Extract more than one resource type

Of course the previous example is using just one resource type. If you are interested in several types of resources, `design` will contain several data.frame descriptions and the result will be a list of several data frames.

Consider the following example where we want to download MedicationStatements referring to a certain medication we specify with its SNOMED CT code and also the Patient resources these MedicationStatements are linked to.

When the FHIR search request gets longer, it can be helpful to build up the request piece by piece like this:

```
search_request  <- paste0(
  "https://hapi.fhir.org/baseR4/", #base url
  "MedicationStatement?", #look for MedicationsStatements
  "code=http://snomed.info/ct%7C429374003", #only choose resources with this snomed code
  "&_include=MedicationStatement:subject") #include the corresponding Patient resources
```

Then we can download the resources:

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

Now our `design` needs two data.frame descriptions (called `MedicationStatement` and `Patients` in our example), one for the MedicationStatement resources and one for the Patient resources:

```
design <- list(

    MedicationStatement = list(

        resource = "//MedicationStatement",

        cols = list(
            MS.ID              = "id",
            STATUS.TEXT        = "text/status",
            STATUS             = "status",
            MEDICATION.SYSTEM  = "medicationCodeableConcept/coding/system",
            MEDICATION.CODE    = "medicationCodeableConcept/coding/code",
            MEDICATION.DISPLAY = "medicationCodeableConcept/coding/display",
            DOSAGE             = "dosage/text",
            PATIENT            = "subject/reference",
            LAST.UPDATE        = "meta/lastUpdated"
        ),

        style = list(
            sep = "|",
            brackets = NULL,
            rm_empty_cols = FALSE
        )
    ),

    Patients = list(

        resource = "//Patient",
        cols = "./*"
    )
)
```

In this example, we have spelled out the data.frame description MedicationStatement completely, while we

have used a short form for Patients. We can now use this `design` for `fhir_crack()`:

```
list_of_tables <- fhir_crack(bundles = medication_bundles, design = design, verbose = 0)

head(list_of_tables$MedicationStatement)
#>   MS.ID STATUS.TEXT STATUS     MEDICATION.SYSTEM MEDICATION.CODE
#> 1 30233   generated active http://snomed.info/ct       429374003
#> 2 42012   generated active http://snomed.info/ct       429374003
#> 3 42091   generated active http://snomed.info/ct       429374003
#> 4 45646   generated active http://snomed.info/ct       429374003
#> 5 45724   generated active http://snomed.info/ct       429374003
#> 6 45802   generated active http://snomed.info/ct       429374003
#>   MEDICATION.DISPLAY           DOSAGE       PATIENT
#> 1   simvastatin 40mg 1 tab once daily Patient/30163
#> 2   simvastatin 40mg 1 tab once daily Patient/41945
#> 3   simvastatin 40mg 1 tab once daily Patient/42024
#> 4   simvastatin 40mg 1 tab once daily Patient/45579
#> 5   simvastatin 40mg 1 tab once daily Patient/45657
#> 6   simvastatin 40mg 1 tab once daily Patient/45735
#>                     LAST.UPDATE
#> 1 2019-09-26T14:34:44.543+00:00
#> 2 2019-10-09T20:12:49.778+00:00
#> 3 2019-10-09T22:44:05.728+00:00
#> 4 2019-10-11T16:17:42.365+00:00
#> 5 2019-10-11T16:30:24.411+00:00
#> 6 2019-10-11T16:32:05.206+00:00

head(list_of_tables$Patients)
#>      id gender  birthDate
#> 1 60096   male 2019-11-13
#> 2 49443 female 1970-10-19
#> 3 46213 female 2019-10-11
#> 4 45735   male 1970-10-11
#> 5 42024 female 1979-10-09
#> 6 58504   male 2019-11-08
```

As you can see, the result now contains two data frames, one for Patient resources and one for MedicationStatement resources.

Multiple entries

A particularly complicated problem in flattening FHIR resources is caused by the fact that there can be multiple entries to an attribute. For a more detailed description of this problem, please see the vignette on flattening resources.

In general, `fhir_crack()` will paste multiple entries for the same attribute together in the data frame, using the separator provided by the `sep` argument.

Let's have a look at the following simple example, where we have a bundle containing just two Patient resources:

```
bundle <- xml2::read_xml(
    "<Bundle>

        <Patient>
            <id value='id1'/>
```

```
            <address>
                <use value='home'/>
                <city value='Amsterdam'/>
                <type value='physical'/>
                <country value='Netherlands'/>
            </address>
            <name>
                <given value='Marie'/>
            </name>
        </Patient>

        <Patient>
            <id value='id3'/>
            <address>
                <use value='home'/>
                <city value='Berlin'/>
            </address>
            <address>
                <type value='postal'/>
                <country value='France'/>
            </address>
            <address>
                <use value='work'/>
                <city value='London'/>
                <type value='postal'/>
                <country value='England'/>
            </address>
            <name>
                <given value='Frank'/>
            </name>
            <name>
                <given value='Max'/>
            </name>
        </Patient>

    </Bundle>"
)

bundle_list <- list(bundle)
```

The first resource has just one entry for the address attribute. The second Patient resource has an address attribute with three entries containing different elements and also two entries for the name attribute.

This is where the `style` element of the design comes into play:

```
design <- list(
    Patients = list(
        resource = "//Patient",
        cols = NULL,
        style = list(
            sep = " | ",
            brackets  = c("[", "]"),
            rm_empty_cols = TRUE
        )
    )
```

```
)

dfs <- fhir_crack(bundles = bundle_list, design = design, verbose = 0)
dfs$Patients
#>       id           address.use              address.city
#> 1 [1]id1             [1.1]home            [1.1]Amsterdam
#> 2 [1]id3 [1.1]home | [3.1]work [1.1]Berlin | [3.1]London
#>                address.type            address.country            name.given
#> 1             [1.1]physical           [1.1]Netherlands             [1.1]Marie
#> 2 [2.1]postal | [3.1]postal [2.1]France | [3.1]England [1.1]Frank | [2.1]Max
```

Multiple entries are pasted together with the specified seperator in between and the indices (inside the specified brackets) display the entry the value belongs to. That way you can see that Patient resource 2 had three entries for the attribute `address` and you can also see which attributes belong to which entry.

Process Data Frames with multiple Entries

Melt data frames with multiple entries

If the data frame produced by `fhir_crack()` contains multiple entries, you'll probably want to divide these entries into distinct observations at some point. This is where `fhir_melt()` comes into play. `fhir_melt()` takes an indexed data frame with multiple entries in one or several `columns` and spreads (aka melts) these entries over several rows:

```
fhir_melt(dfs$Patients, columns = "address.city", brackets = c("[","]"),
          sep=" | ", all_columns = FALSE)
#>   address.city resource_identifier
#> 1 [1]Amsterdam                   1
#> 2    [1]Berlin                   2
#> 3    [1]London                   2
```

The new variable `resource_identifier` maps which rows in the created data frame belong to which row (usually equivalent to one resource) in the original data frame. `brackets` and `sep` should be given the same character vectors that have been used to build the indices in `fhir_melt()`. `columns` is a character vector with the names of the variables you want to melt. You can provide more than one column here but it makes sense to only have variables from the same repeating attribute together in one call to `fhir_melt()`:

```
cols <- c("address.city", "address.use", "address.type",
          "address.country")

fhir_melt(dfs$Patients, columns = cols, brackets = c("[","]"),
          sep=" | ", all_columns = FALSE)
#>   address.city address.use address.type address.country resource_identifier
#> 1 [1]Amsterdam     [1]home  [1]physical  [1]Netherlands                   1
#> 2    [1]Berlin     [1]home         <NA>            <NA>                   2
#> 3    [1]London     [1]work    [1]postal      [1]England                   2
#> 4         <NA>        <NA>    [1]postal       [1]France                   2
```

With the argument `all_columns` you can control whether the resulting data frame contains only the molten columns or all columns of the original data frame:

```
molten <- fhir_melt(dfs$Patients, columns = cols, brackets = c("[","]"),
                    sep=" | ", all_columns = TRUE)
molten
#>       id address.use address.city address.type address.country
#> 1 [1]id1     [1]home [1]Amsterdam  [1]physical  [1]Netherlands
```

```
#> 2 [1]id3     [1]home    [1]Berlin         <NA>            <NA>
#> 3 [1]id3     [1]work    [1]London    [1]postal      [1]England
#> 4 [1]id3        <NA>        <NA>    [1]postal       [1]France
#>              name.given resource_identifier
#> 1            [1.1]Marie                   1
#> 2 [1.1]Frank | [2.1]Max                   2
#> 3 [1.1]Frank | [2.1]Max                   2
#> 4 [1.1]Frank | [2.1]Max                   2
```

Values on the other variables will just repeat in the newly created rows. For more information, e.g. on how to melt all multiple entries in a data.rame at once, please see the vignette on flattening resources.

Remove indices

Once you have sorted out the multiple entries, you might want to get rid of the indices in your data.frame. This can be achieved using `fhir_rm_indices()`:

```
fhir_rm_indices(molten, brackets=c("[","]"))
#>    id address.use address.city address.type address.country  name.given
#> 1 id1        home    Amsterdam     physical     Netherlands       Marie
#> 2 id3        home       Berlin         <NA>            <NA> Frank | Max
#> 3 id3        work       London       postal         England Frank | Max
#> 4 id3        <NA>         <NA>       postal          France Frank | Max
#>   resource_identifier
#> 1                   1
#> 2                   2
#> 3                   2
#> 4                   2
```

Again, `brackets` and `sep` should be given the same character vector that was used for `fhir_crack()` and `fhir_melt()`respectively.

Save and load downloaded bundles

Since `fhir_crack()` discards of all the data not specified in `design`, it makes sense to store the original search result for reproducibility and in case you realize later on that you need elements from the resources that you haven't extracted at first.

There are two ways of saving the FHIR bundles you downloaded: Either you save them as R objects, or you write them to an xml file.

Save and load bundles as R objects

If you want to save the list of downloaded bundles as an `.rda` or `.RData` file, you can't just use R's `save()` or `save_image()` on it, because this will break the external pointers in the xml objects representing your bundles. Instead, you have to serialize the bundles before saving and unserialize them after loading. For single xml objects the package `xml2` provides serialization functions. For convenience, however, `fhircrackr` provides the functions `fhir_serialize()` and `fhir_unserialize()` that can be used directly on the list of bundles returned by `fhir_search()`:

```
#serialize bundles
serialized_bundles <- fhir_serialize(patient_bundles)

#have a look at them
head(serialized_bundles[[1]])
#> [1] 58 0a 00 00 00 03
```

```
#create temporary directory for saving
temp_dir <- tempdir()

#save
save(serialized_bundles, file=paste0(temp_dir, "/bundles.rda"))
```

If you load this bundle again, you have to unserialize it before you can work with it:

```
#load bundles
load(paste0(temp_dir, "/bundles.rda"))
```

```
#unserialize
bundles <- fhir_unserialize(serialized_bundles)

#have a look
head(bundles[[1]])
#> $node
#> <pointer: 0x00000000124370a0>
#>
#> $doc
#> <pointer: 0x00000000123f31e0>
```

After unserialization, the pointers are restored and you can continue to work with the bundles. Note that the example bundles `medication_bundles` and `patient_bundles` that are provided with the `fhircrackr` package are also provided in their serialized form and have to be unserialized as described on their help page.

Save and load bundles as xml files

If you want to store the bundles in xml files instead of R objects, you can use the functions `fhir_save()` and `fhir_load()`. `fhir_save()` takes a list of bundles in form of xml objects (as returned by `fhir_search()`) and writes them into the directory specified in the argument `directory`. Each bundle is saved as a separate xml-file. If the folder defined in `directory` doesn't exist, it is created in the current working directory.

```
#save bundles as xml files
fhir_save(patient_bundles, directory=temp_dir)
```

To read bundles saved with `fhir_save()` back into R, you can use `fhir_load()`:

```
bundles <- fhir_load(temp_dir)
```

`fhir_load()` takes the name of the directory (or path to it) as its only argument. All xml-files in this directory will be read into R and returned as a list of bundles in xml format just as returned by `fhir_search()`.

Acknowledgements

This work was carried out by the SMITH consortium and the cross-consortium use case POLAR_MI; both are part of the German Initiative for Medical Informatics and funded by the German Federal Ministry of Education and Research (BMBF), grant no. 01ZZ1803A , 01ZZ1803C and 01ZZ1910A.