fhircrackr: Handling HL7® FHIR® Resources in R

2020-12-17

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 covers the following topics:

- Prerequisites
- FHIR search requests
- Downloading and flattening resources from a FHIR server
- Processing data frames with multiple entries
- Saving and loading downloaded bundles
- Saving and reading designs
- Performance
- Downloading capability statements
- Further options

Prerequisites

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

[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

1. First of all, you need the endpoint 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 endpoint of a FHIR server is often referred to as [base] or [baseR4] for the HL7 R4 standard for instance.

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 paragraph on FHIR search requests.

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)
```

FHIR search requests

This paragraph introduces the basics of FHIR search and some functions to build valid FHIR search requests with `fhircrackr`. If you are already familiar and comfortable with FHIR search, you can skip this paragraph.

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()`, which we'll explain shortly 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()`:

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

The resource type

This information determines the kind of resource you are getting back from the server and also the kinds of search parameters that are allowed in the next step. A list of all currently available resource types can be found at https://hl7.org/FHIR/resourcelist.html. The function `fhir_resource()` will check the resource you provided against that list and also do some of the formatting for you:

```
fhir_resource("patient")
#>  resource
#> "Patient"
```

While `fhir_resource()` will convert the first letter of the word into a capital, it won't do so with other letters in the word. Mind that the case within the word does matter though, as for example `MedicationAdministration` is a valid resource but `Medicationadministration` is not. If the resource you provided is not in the list at hl7.org, there will be a warning:

```r
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. Please note that upper and lower cases within the word
#> matter. 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:

```r
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 is the URL encoding this function performs. 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()`:

```r
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()` (see below), the corresponding FHIR search request will be saved implicitly and can be accessed like this:

```r
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:

```r
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:

```r
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 and flatten FHIR Resources from a server

1. Download Patient Resources

We will start with a very simple example and use `fhir_search()` to download Patient resources from a public HAPI server:

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

If you want to build the request using `fhir_build_request()` you can also do this:

```r
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)
```

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`.

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 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.

2. Flatten FHIR 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`).

In general, `design` has to be 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. In the following we will first describe the different elements of a data.frame description and will then provide several examples.

The data.frame description itself is again a list, with 3 elements:

1. *resource*
   A string containing an XPath expression to the resource you want to extract, e.g. `"//Patient"`. If your bundles are the result of a regular FHIR search request, the correct XPath expression will always be `"//<resource name>"`.

2. *cols*
   Can be NULL, a string or a list describing the columns your data frame is going to have.

- If *cols* is NULL, all attributes available in the resources will be extracted and put in one column each, the column names will be chosen automatically and reflect the position of the attribute in the resource.
- If *cols* is a string with an XPath expression indicating a certain level in the bundle, all attributes on this specific level will be extracted. `"./*"` e.g. will extract all attributes that are located (exactly) one level below the root level given by `"//Patient"`.
- If *cols* is a named list of XPath expressions, each element is taken to be the description for one column. `family_name = "name/family"` for example creates a column named family_name which contains the values for the attribute indicated by the XPath expression `"name/family"`.

3. *style*
   Can be NULL or a list of length 3 with the following named elements:

- *sep*: A string defining the seperator used when multiple entries to the same attribute are pasted together, e.g. `"|"`.
- *brackets*: Either NULL or a character vector of length 2. If NULL, multiple entries will be pasted together without indices. If character, the two strings provided here are used as brackets for automatically

generated indices to sort out multiple entries (see paragraph Multiple Entries). `brackets = c("[", "]")` e.g. will lead to indices like `[1.1]`.

- *rm_empty_cols*: Logical. If `TRUE`, columns containing only `NA` values will be removed, if `FALSE`, these columns will be kept.

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.

We will now work through examples using designs of different complexity.

Extract all available attributes Lets start with an example where we only provide the (mandatory) `resource` component of the data.frame description that is called `Patients` in our example. In this case, `fhir_crack()` will extract all available attributes and use default values for the `style` component:

```
#define design
design1 <- list(

     Patients = list(

        resource =  "//Patient"
     )
)

#Convert resources
list_of_tables <- fhir_crack(bundles = patient_bundles, design = design1, verbose = 0)

#have look at part of the results
list_of_tables$Patients[1:5,1:5]
#>     id meta.versionId              meta.lastUpdated text.status   text.div.div
#> 1 1282              1 2019-03-05T11:33:15.214+01:00   generated hapiHeaderText
#> 2  267              2 2018-05-13T10:17:40.800+02:00   generated hapiHeaderText
#> 3  722              1 2018-09-02T17:24:17.083+02:00   generated hapiHeaderText
#> 4  731              1 2018-09-02T17:28:16.838+02:00   generated hapiHeaderText
#> 5  736              1 2018-09-02T17:34:50.955+02:00   generated hapiHeaderText
```

As you can see, this can easily become a rather wide and sparse data frame. This is due to the fact that every attribute appearing in at least one of the resources will be turned into a variable (i.e. column), even if none of the other resources contain this attribute. For those resources, the value on that attribute will be set to `NA`. Depending on the variability of the resources, the resulting data frame can contain a lot of `NA` values. If a resource has multiple entries for an attribute, these entries will pasted together using the string provided in `sep` as a separator. The column names in this option are automatically generated by pasting together the path to the respective attribute, e.g. `name.given.value`.

Extract all attributes at certain levels We can extract all attributes that are found on a certain level of the resource if we specify this level in an XPath expression and provide it in the `cols` argument of the data.frame description:

```
#define design
design2 <- list(

    Patients = list(
```

```
        resource =  "//Patient",

        cols = "./*"
     )
)

#Convert resources
list_of_tables <- fhir_crack(bundles = patient_bundles, design = design2, verbose = 0)

#have look at the results
head(list_of_tables$Patients)
#>     id  birthDate gender
#> 1 1282       <NA>   <NA>
#> 2  267 1960-10-04   <NA>
#> 3  722 1982-01-01   male
#> 4  731 1982-01-01   male
#> 5  736 1982-01-01   male
#> 6  737 1982-01-01   male
```

"./*" tells fhir_crack() to extract all attributes that are located (exactly) one level below the root level. The column names are still automatically generated.

Extract specific attributes If we know exactly which attributes we want to extract, we can specify them in a named list and provide it in the cols component of the data.frame description:

```
#define design
design3 <- 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"
        )
    )
)
#Convert resources
list_of_tables <- fhir_crack(bundles = patient_bundles, design = design3, verbose = 0)

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

This option will usually return the most tidy and clear data frames, because you have full control over the extracted columns including their name in the resulting data frame. You should always extract the resource id, because this is used to link to other resources you might also extract.

If you are not sure which attributes are available or where they are located in the resource, it can be helpful to start by extracting all available attributes. If you are more comfortable with xml, you can also use `xml2::xml_structure` on one of the bundles from your bundle list, this will print the complete xml structure into your console. Then you can get an overview over the available attributes and their location and continue by doing a second, more targeted extraction to get your final data frame.

Set `style` component Even though our example won't show any difference if we change it, here is what a `design` with a complete data.frame description would look like:

```
design4 <- 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
        )
    )
)
```

The `style` component will become more important in the example for multiple entries later on.

Internally, `fhir_crack()` will always complete the `design` you provided so that it contains `resource`, `cols` and `style` with its elements `sep`, `brackets` and `rm_empty_cols`, even if you left out `cols` and `style` completely. You can retrieve the completed `design` of you last call to `fhir_crack()` with the function `fhir_canonical_design()`:

```
fhir_canonical_design()
#> $Patients
#> $Patients$resource
#> [1] "//Patient"
#>
#> $Patients$cols
#> $Patients$cols$PID
#> [1] "id/@value"
#>
#> $Patients$cols$use_name
#> [1] "name/use/@value"
#>
#> $Patients$cols$given_name
```

```
#> [1] "name/given/@value"
#>
#> $Patients$cols$family_name
#> [1] "name/family/@value"
#>
#> $Patients$cols$gender
#> [1] "gender/@value"
#>
#> $Patients$cols$birthday
#> [1] "birthDate/@value"
#>
#>
#> $Patients$style
#> $Patients$style$sep
#> [1] " "
#>
#> $Patients$style$brackets
#> NULL
#>
#> $Patients$style$rm_empty_cols
#> [1] TRUE
```

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/", #server endpoint
  "MedicationStatement?", #look for MedicationsStatements
  "code=http://snomed.info/ct|429374003", #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",
```

```r
            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()`:

```r
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.

3. Multiple entries

A particularly complicated problem in flattening FHIR resources is caused by the fact that there can be multiple entries to an attribute. The profile according to which your FHIR resources have been built defines how often a particular attribute can appear in a resource. This is called the *cardinality* of the attribute. For example the Patient resource defined here can have zero or one birthdates but arbitrarily many addresses. 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. In most cases this will work just fine, but there are some special cases that require a little more attention.

Let's have a look at the following example, where we have a bundle containing just three 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>
            <birthDate value='1992-02-06'/>
        </Patient>

        <Patient>
            <id value='id2'/>
            <address>
                <use value='home'/>
                <city value='Rome'/>
                <type value='physical'/>
                <country value='Italy'/>
            </address>
            <address>
                <use value='work'/>
                <city value='Stockholm'/>
                <type value='postal'/>
                <country value='Sweden'/>
            </address>
            <birthDate value='1980-05-23'/>
        </Patient>

        <Patient>
            <id value='id3.1'/>
            <id value='id3.2'/>
            <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>
            <birthDate value='1974-12-25'/>
        </Patient>

    </Bundle>"
)

bundle_list <- list(bundle)
```

This bundle contains three Patient resources. The first resource has just one entry for the address attribute. The second Patient resource has two entries containing the same elements for the address attribute. The third Patient resource has a rather messy address attribute, with three entries containing different elements and also two entries for the id attribute.

Let's see what happens if we extract all attributes:

```
design1 <- list(
    Patients = list(
        resource = "//Patient",
        cols = NULL,
        style = list(
            sep = " | ",
            brackets  = NULL,
            rm_empty_cols = TRUE
        )
    )
)

df1 <- fhir_crack(bundles = bundle_list, design = design1, verbose = 0)
df1$Patients
#>              id address.use     address.city      address.type  address.country
#> 1           id1        home        Amsterdam          physical      Netherlands
#> 2           id2 home | work Rome | Stockholm physical | postal   Italy | Sweden
#> 3 id3.1 | id3.2 home | work  Berlin | London   postal | postal France | England
#>    birthDate
#> 1 1992-02-06
#> 2 1980-05-23
#> 3 1974-12-25
```

As you can see, multiple entries for the same attribute (address and id) are pasted together. This works fine for Patient 2, but for Patient 3 you can see a problem with the number of entries that are displayed. The original Patient resource had *three* (incomplete) `address` entries, but because the first two of them use complementary elements (`use` and `city` vs. `type` and `country`), the resulting pasted entries look like there had just been two entries for the `address` attribute.

You can counter this problem by setting `brackets`:

```
design2 <- list(
    Patients = list(
        resource = "//Patient",
        cols = NULL,
        style = list(
            sep = " | ",
            brackets  = c("[", "]"),
            rm_empty_cols = TRUE
        )
    )
)

df2 <- fhir_crack(bundles = bundle_list, design = design2, verbose = 0)
df2$Patients
#>                      id            address.use               address.city
#> 1                [1]id1              [1.1]home            [1.1]Amsterdam
#> 2                [1]id2 [1.1]home | [2.1]work [1.1]Rome | [2.1]Stockholm
#> 3 [1]id3.1 | [2]id3.2 [1.1]home | [3.1]work  [1.1]Berlin | [3.1]London
#>                    address.type            address.country     birthDate
#> 1                 [1.1]physical           [1.1]Netherlands [1]1992-02-06
#> 2 [1.1]physical | [2.1]postal    [1.1]Italy | [2.1]Sweden [1]1980-05-23
#> 3   [2.1]postal | [3.1]postal [2.1]France | [3.1]England [1]1974-12-25
```

Now the indices display the entry the value belongs to. That way you can see that Patient resource 3 had three entries for the attribute `address` and you can also see which attributes belong to which entry.

It is possible to set the `style` separately for every data.frame description you have. If you want to have the same style specifications for all the data frames, you can supply them in as function arguments to `fhir_crack()`. The values provided there will be automatically filled in in the design, as you can see, when you check with `fhir_canonical_design()`:

```
design3 <- list(
    Patients = list(
        resource = "//Patient"
    )
)

df3 <- fhir_crack(bundles = bundle_list,
                  design = design3,
                  sep = " | ",
                  brackets = c("[", "]"))
#>
#>  Patients
#>  1...
#> FHIR-Resources cracked.


df3$Patients
#>                      id            address.use               address.city
#> 1                [1]id1              [1.1]home            [1.1]Amsterdam
#> 2                [1]id2 [1.1]home | [2.1]work [1.1]Rome | [2.1]Stockholm
#> 3 [1]id3.1 | [2]id3.2 [1.1]home | [3.1]work  [1.1]Berlin | [3.1]London
#>                    address.type            address.country     birthDate
#> 1                 [1.1]physical           [1.1]Netherlands [1]1992-02-06
```

```
#> 2 [1.1]physical | [2.1]postal   [1.1]Italy | [2.1]Sweden [1]1980-05-23
#> 3   [2.1]postal | [3.1]postal [2.1]France | [3.1]England [1]1974-12-25
```

```
fhir_canonical_design()
#> $Patients
#> $Patients$resource
#> [1] "//Patient"
#>
#> $Patients$cols
#> NULL
#>
#> $Patients$style
#> $Patients$style$sep
#> [1] " | "
#>
#> $Patients$style$brackets
#> [1] "[" "]"
#>
#> $Patients$style$rm_empty_cols
#> [1] TRUE
```

Of course the above example is a very specific case that only occurs if your resources have multiple entries with complementary elements. In the vast majority of cases multiple entries in one resource will have the same structure, thus making numbering of those entries superfluous.

Process Data Frames with multiple Entries

1. 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(df2$Patients, columns = "address.city", brackets = c("[","]"),
          sep=" | ", all_columns = FALSE)
#>   address.city resource_identifier
#> 1 [1]Amsterdam                   1
#> 2      [1]Rome                   2
#> 3 [1]Stockholm                   2
#> 4    [1]Berlin                   3
#> 5    [1]London                   3
```

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(df2$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]Rome     [1]home  [1]physical        [1]Italy                    2
#> 3 [1]Stockholm     [1]work    [1]postal       [1]Sweden                    2
#> 4    [1]Berlin     [1]home         <NA>            <NA>                    3
#> 5    [1]London     [1]work    [1]postal      [1]England                    3
#> 6         <NA>        <NA>    [1]postal       [1]France                    3
```

If the names of the variables in your data frame have been generated automatically with `fhir_crack()` you can find all variable names belonging to the same attribute with `fhir_common_columns()`:

```
cols <- fhir_common_columns(df2$Patients, column_names_prefix = "address")
cols
#> [1] "address.use"     "address.city"    "address.type"    "address.country"
```

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:

```
fhir_melt(df2$Patients, columns = cols, brackets = c("[","]"),
          sep=" | ", all_columns = TRUE)
#>                     id address.use address.city address.type address.country
#> 1               [1]id1     [1]home [1]Amsterdam  [1]physical  [1]Netherlands
#> 2               [1]id2     [1]home      [1]Rome  [1]physical        [1]Italy
#> 3               [1]id2     [1]work [1]Stockholm    [1]postal       [1]Sweden
#> 4 [1]id3.1 | [2]id3.2     [1]home    [1]Berlin         <NA>            <NA>
#> 5 [1]id3.1 | [2]id3.2     [1]work    [1]London    [1]postal      [1]England
#> 6 [1]id3.1 | [2]id3.2        <NA>         <NA>    [1]postal       [1]France
#>       birthDate resource_identifier
#> 1 [1]1992-02-06                   1
#> 2 [1]1980-05-23                   2
#> 3 [1]1980-05-23                   2
#> 4 [1]1974-12-25                   3
#> 5 [1]1974-12-25                   3
#> 6 [1]1974-12-25                   3
```

Values on the other variables will just repeat in the newly created rows.

If you try to melt several variables that don't belong to the same attribute in one call to `fhir_melt()`, this will cause problems, because the different attributes won't be combined correctly:

```
cols <- c(cols, "id")
fhir_melt(df2$Patients, columns = cols, brackets = c("[","]"),
          sep=" | ", all_columns = TRUE)
#>        id address.use address.city address.type address.country     birthDate
#> 1   []id1     [1]home [1]Amsterdam  [1]physical  [1]Netherlands [1]1992-02-06
#> 2   []id2     [1]home      [1]Rome  [1]physical        [1]Italy [1]1980-05-23
#> 3    <NA>     [1]work [1]Stockholm    [1]postal       [1]Sweden [1]1980-05-23
#> 4 []id3.1     [1]home    [1]Berlin         <NA>            <NA> [1]1974-12-25
#> 5    <NA>     [1]work    [1]London    [1]postal      [1]England [1]1974-12-25
#> 6 []id3.2        <NA>         <NA>    [1]postal       [1]France [1]1974-12-25
#>   resource_identifier
#> 1                   1
#> 2                   2
#> 3                   2
#> 4                   3
#> 5                   3
#> 6                   3
```

Instead, melt the attributes one after another:

```
cols <- fhir_common_columns(df2$Patients, "address")

molten_1 <- fhir_melt(df2$Patients, columns = cols, brackets = c("[","]"),
                      sep=" | ", all_columns = TRUE)
molten_1
#>                     id address.use address.city address.type address.country
#> 1              [1]id1     [1]home [1]Amsterdam  [1]physical  [1]Netherlands
#> 2              [1]id2     [1]home      [1]Rome  [1]physical        [1]Italy
#> 3              [1]id2     [1]work [1]Stockholm    [1]postal       [1]Sweden
#> 4 [1]id3.1 | [2]id3.2     [1]home    [1]Berlin         <NA>            <NA>
#> 5 [1]id3.1 | [2]id3.2     [1]work    [1]London    [1]postal      [1]England
#> 6 [1]id3.1 | [2]id3.2        <NA>         <NA>    [1]postal       [1]France
#>       birthDate resource_identifier
#> 1 [1]1992-02-06                   1
#> 2 [1]1980-05-23                   2
#> 3 [1]1980-05-23                   2
#> 4 [1]1974-12-25                   3
#> 5 [1]1974-12-25                   3
#> 6 [1]1974-12-25                   3

molten_2 <- fhir_melt(molten_1, columns = "id", brackets = c("[","]"),
                      sep=" | ", all_columns = TRUE)
molten_2
#>        id address.use address.city address.type address.country     birthDate
#> 1   []id1     [1]home [1]Amsterdam  [1]physical  [1]Netherlands [1]1992-02-06
#> 2   []id2     [1]home      [1]Rome  [1]physical        [1]Italy [1]1980-05-23
#> 3   []id2     [1]work [1]Stockholm    [1]postal       [1]Sweden [1]1980-05-23
#> 4 []id3.1     [1]home    [1]Berlin         <NA>            <NA> [1]1974-12-25
#> 5 []id3.2     [1]home    [1]Berlin         <NA>            <NA> [1]1974-12-25
#> 6 []id3.1     [1]work    [1]London    [1]postal      [1]England [1]1974-12-25
#> 7 []id3.2     [1]work    [1]London    [1]postal      [1]England [1]1974-12-25
#> 8 []id3.1        <NA>         <NA>    [1]postal       [1]France [1]1974-12-25
#> 9 []id3.2        <NA>         <NA>    [1]postal       [1]France [1]1974-12-25
#>   resource_identifier
#> 1                   1
#> 2                   2
#> 3                   3
#> 4                   4
#> 5                   4
#> 6                   5
#> 7                   5
#> 8                   6
#> 9                   6
```

This will give you the appropriate cross product of all multiple entries.

2. 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_2, brackets=c("[","]"))
#>        id address.use address.city address.type address.country  birthDate
#> 1   []id1        home    Amsterdam     physical     Netherlands 1992-02-06
```

```
#> 2   []id2        home         Rome     physical           Italy 1980-05-23
#> 3   []id2        work    Stockholm       postal          Sweden 1980-05-23
#> 4 []id3.1        home       Berlin         <NA>            <NA> 1974-12-25
#> 5 []id3.2        home       Berlin         <NA>            <NA> 1974-12-25
#> 6 []id3.1        work       London       postal         England 1974-12-25
#> 7 []id3.2        work       London       postal         England 1974-12-25
#> 8 []id3.1        <NA>         <NA>       postal          France 1974-12-25
#> 9 []id3.2        <NA>         <NA>       postal          France 1974-12-25
#>   resource_identifier
#> 1                   1
#> 2                   2
#> 3                   3
#> 4                   4
#> 5                   4
#> 6                   5
#> 7                   5
#> 8                   6
#> 9                   6
```

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.

1. 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:

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

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

#have a look
head(bundles[[1]])
#> $node
#> <pointer: 0x0000000012508d00>
#>
#> $doc
#> <pointer: 0x000000001527f900>
```

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.

2. 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.

```r
#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()`:

```r
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()`.

Save and read designs

If you want to save a design for later or to share with others, you can do so using the `fhir_save_design()`. This function takes a design and saves it as an xml file:

```r
fhir_save_design(design1, file = paste0(temp_dir,"/design.xml"))
```

To read the design back into R, you can use `fhir_load_design()`:

```r
fhir_load_design(paste0(temp_dir,"/design.xml"))
#> $Patients
#> $Patients$resource
#> [1] "//Patient"
#>
#> $Patients$cols
#> NULL
#>
#> $Patients$style
#> $Patients$style$sep
#> [1] " | "
#>
#> $Patients$style$brackets
```

```
#> NULL
#>
#> $Patients$style$rm_empty_cols
#> [1] TRUE
```

Performance

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. In that case `fhir_search()` will throw an error. If you set the argument `log_errors` accordingly, you can however retrieve the server errors that caused `fhir_search()` to crash.

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:

```
fhir_search("http://hapi.fhir.org/baseR4/Patient", max_bundles = 10,
            save_to_disc=TRUE, directory = paste0(temp_dir, "/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()`:

```
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()`:

```
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.

Further Options

Extract data below resource level

While we recommend extracting exactly one data frame per resource, it is technically possible to choose a different level per data frame:

```
design <- list(
    MedCodes=list(resource = "//medicationCodeableConcept/coding")
)
```

```r
df <- fhir_crack(medication_bundles, design, verbose=0)

head(df$MedCodes)
#>                   system      code          display
#> 1 http://snomed.info/ct 429374003 simvastatin 40mg
#> 2 http://snomed.info/ct 429374003 simvastatin 40mg
#> 3 http://snomed.info/ct 429374003 simvastatin 40mg
#> 4 http://snomed.info/ct 429374003 simvastatin 40mg
#> 5 http://snomed.info/ct 429374003 simvastatin 40mg
#> 6 http://snomed.info/ct 429374003 simvastatin 40mg
```

The above example shows that instead of the MedicationStatement resource, we can choose the Medication-CodeableConcept as the root level for our extraction. This can be useful to get a quick and relatively clean overview over the types of codes used on this level of the resource. It is however important to note that this mode of extraction makes it impossible to recognize if each row belongs to one resource or if several of these rows came from the same resource. This of course also means that you cannot link this information to data from other resources because this extraction mode discards of that information.

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.