fhircrackr: Flatten FHIR resources

2021-05-12

This vignette covers the following topics:

- Extracting one resource type
- The design
- Extracting more than one resource type
- Multiple Entries
- Processing tables with multiple entries

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

```r
library(fhircrackr)
```

Preparation

In the vignette `fhircrackr: Download FHIR resources` you saw how to download FHIR resources into R. Now we'll have a look at how to flatten them into data.frames/data.tables. For rest of the vignette, we'll work with the two example data sets from `fhircrackr`, which can be made accessible like this:

```r
pat_bundles <- fhir_unserialize(patient_bundles)
med_bundles <- fhir_unserialize(medication_bundles)
```

See `?patient_bundles` and `?medication_bundles` for the FHIR search request that generated them.

There are two extraction scenarios when you want to flatten FHIR bundles: Either you want to extract just one resource type, or you want to extract several resource types. Because the structure of different resource types is quite dissimilar, it makes sense to create one table per resource type. Therefore the result of the flattening process in `fhircrackr` can be either a single table (when extracting just one resource type) or a list of tables (when extracting more than one resource type). Both scenarios are realized with a call to `fhir_crack()`. We will now explain the two scenarios individually.

Extracting one resource type

We'll start with `pat_bundles`, which only contains Patient resources. To transform them into a table, we will use `fhir_crack()`. The most important argument `fhir_crack()` takes is `bundles`, an object of class `fhir_bundle_list` that is returned by `fhir_search()`. The second important argument is `design`, which tells the function which data to extract from the bundle. When we want to extract just one resource type, we can use a `fhir_df_description` in the argument `design`.`fhir_crack()` then returns a single data.frame or data.table (if argument `data.tables=TRUE`).

We'll show you an example of how it works first and then go on to explain the `fhir_df_description` in more detail.

```r
pat_df_description <- fhir_df_description(resource = "Patient",
                                          cols = list(id = "id",
                                                      gender = "gender",
                                                      name = "name/family",
                                                      city = "address/city"))
```

```
table <- fhir_crack(bundles = pat_bundles,
                    design = pat_df_description,
                    verbose = 0)

head(table)
#            id gender             name          city
# 1     1837602   male           Jacobs          <NA>
# 2 example-r4   male Chalmers Windsor PleasantVille
# 3     1837624   <NA>             <NA>          <NA>
# 4     1837626   male             <NA>          <NA>
# 5     1837631   male          paredes          <NA>
# 6     1837716   male             <NA>          <NA>
```

The df_description

A `fhir_df_description` holds all the information `fhir_crack()` needs to create a table from resources of a certain type. It is created with `fhir_df_description()` and generally consists of the three elements that can be provided in the three following arguments:

The `resource` argument This is basically a string that defines the resource type (e.g. Patient or Observation) to extract. You set it like this:

```
fhir_df_description(resource = "Patient")
# A fhir_df_description with the following elements:
#
# fhir_resource_type: Patient
#
# fhir_columns:
# An empty fhir_columns object
#
# fhir_style:
# sep: ' '
# brackets: character(0)
# rm_empty_cols: TRUE
```

Internally, `fhir_df_description()` calls `fhir_resource_type()` which checks the type you provided against list of all currently available resource types can be found at https://hl7.org/FHIR/resourcelist.html. Case errors are corrected automatically and the function throws a warning, if the resource type doesn't match the list under hl7.org.

As you can see in the above output, there are two more elements in a `fhir_df_description` which are filled automatically by `fhir_df_description()`.

The cols argument The `cols` argument takes the column names and XPath (1.0) expressions defining the columns to create from the FHIR resources. If this element is empty, `fhir_crack()` will extract all available elements of the resource and name the columns automatically. To explicitly define columns, you can provide a (named) character or a (named) list with XPath expressions like this:

```
fhir_df_description(resource = "Patient",
                    cols = list(gender = "gender",
                                name = "name/family",
                                city = "address/city"))
# A fhir_df_description with the following elements:
#
# fhir_resource_type: Patient
```

```
#
# fhir_columns:
# column name | xpath expression
#  ------------------------
# gender      | gender
# name        | name/family
# city        | address/city
#
# fhir_style:
# sep: ' '
# brackets: character(0)
# rm_empty_cols: TRUE
```

In this case a table with three columns called gender, name and city will be created. They will be filled with the value attribute that can be found under the respective xpath expressions in the resource. In the rare cases where you want to extract another attribute (besides *value* FHIR currently only allows *id* and *url* attributes), you have to add the attribute to the xpath expression like this: `"node/node/@id"`. Note that this happens extremely rarely.

Internally, `fhir_df_description()` calls `fhir_columns()` to check the validity of the XPath expressions and assign column names. You can provide the XPath expressions in a named or unnamed character vector or a named or unnamed list. If you choose the unnamed version, the names will be set automatically and reflect the respective XPath expression:

```
#custom column names
fhir_columns(c(gender = "gender",
               name = "name/family",
               city = "address/city"))
# column name | xpath expression
#  ------------------------
# gender      | gender
# name        | name/family
# city        | address/city

#automatic column names
fhir_columns(c("gender", "name/family", "address/city"))
# column name  | xpath expression
#  -------------------------
# gender       | gender
# name.family  | name/family
# address.city | address/city
```

We strongly advise to only use fully specified relative XPath expressions here, e.g. `"ingredient/strength/numerator/code"` and not search paths like `"//code"`, as those can generate unexpected results especially if the searched element appears on different levels of the resource.

The style argument The final element of a `fhir_df_description` is a `fhir_style` object. This element controls how `fhir_crack()` deals with multiple entries to the same element and with columns that are completely empty, i.e. have only `NA` values. In the outputs above you can see, that the style takes some default values if you skip it in `fhir_df_description()`. You can change the defaults like this:

```
fhir_df_description(resource = "Patient",

                    cols = list(gender = "gender",
                                name = "name/family",
                                city = "address/city"),

                    style = fhir_style(sep = "||",
                                       brackets = c("[","]"),
                                       rm_empty_cols = FALSE))
# A fhir_df_description with the following elements:
#
# fhir_resource_type: Patient
#
# fhir_columns:
# column name | xpath expression
#  -------------------------
# gender      | gender
# name        | name/family
# city        | address/city
#
# fhir_style:
# sep: ||
# brackets: '[' ']'
# rm_empty_cols: FALSE
```

As you can see, the style is created by a call to `fhir_style()` which can be used outside of `fhir_df_description()`, too:

```
fhir_style(sep = "||",
           brackets = c("[","]"),
           rm_empty_cols = FALSE)
# sep: ||
# brackets: '[' ']'
# rm_empty_cols: FALSE
```

The `sep` element is a string defining the separator used when multiple entries to the same attribute are pasted together. This could for example happen if there is more than one address entry in a Patient resource. Examples of this are shown further down under the heading **Multiple entries**.

The `brackets` element is either an empty character (of length 0) or a character vector of length 2. If it is empty, multiple entries will be pasted together without indices. If it is of length 2, 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]`.

The `rm_empty_cols`: Can be `TRUE` or `FALSE`. 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`.

We will now work through examples using `fhir_df_descriptions` of different complexity.

Examples

Extract all available attributes Lets start with an example where we only provide the (mandatory) `resource` component of the df_description. In this case, `fhir_crack()` will extract all available attributes and use default values for the `style` component:

```
#define design
df_description1 <- fhir_df_description(resource = "Patient")
```

```
#Convert resources
table<- fhir_crack(bundles = pat_bundles, design = df_description1, verbose = 0)

#have look at part of the results
table[1:5,1:5]
#           id meta.versionId              meta.lastUpdated       meta.source
# 1    1837602              1 2021-01-26T13:19:57.198+00:00 #rlPNWRjKJIXpIViv
# 2 example-r4             16 2021-05-05T14:35:25.900+00:00 #SjWscZfVfkMgcAGw
# 3    1837624              1 2021-01-26T13:34:05.655+00:00 #SHEY4NcGvuguei38
# 4    1837626              1 2021-01-26T14:04:12.384+00:00 #E88fut6cJLAp8inR
# 5    1837631              1 2021-01-26T15:34:28.829+00:00 #eKbD4whsoZcSI3eR
#   text.status
# 1   generated
# 2   generated
# 3   generated
# 4   generated
# 5   generated
```

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 (e.g. several addresses in a Patient resource), 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`.

Extract specific attributes If we know 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
df_description2 <- fhir_df_description(

        resource = "Patient",

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

#Convert resources
table <- fhir_crack(bundles = pat_bundles, design = df_description2, verbose = 0)

#have look at the results
head(table)
#          PID              use_name                  given_name      family_name
# 1    1837602              official                     Jeffrey           Jacobs
# 2 example-r4 official usual maiden Peter James Jim Peter James Chalmers Windsor
# 3    1837624                  <NA>                        <NA>             <NA>
# 4    1837626                  <NA>                        <NA>             <NA>
# 5    1837631              official                          juan          paredes
# 6    1837716                  <NA>                          <NA>             <NA>
#   gender   birthday
# 1   male 1996-07-08
# 2   male 1974-12-25
# 3   <NA>       <NA>
# 4   male 1972-10-13
# 5   male 2021-01-26
# 6   male 2021-01-20
```

This option will return more tidy and clear data frames, because you have full control over the extracted columns including their name in the resulting table. 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.

If you want to have a look at how the design looked that was actually used in the last call to `fhir_crack()` you can retrieve it with `fhir_canonical_design()`.

```
fhir_canonical_design()
# A fhir_df_description with the following elements:
#
# fhir_resource_type: Patient
#
# fhir_columns:
# column name | xpath expression
#  ------------------------------
# PID         | id/@value
# use_name    | name/use/@value
# given_name  | name/given/@value
# family_name | name/family/@value
# gender      | gender/@value
# birthday    | birthDate/@value
#
# fhir_style:
# sep: ' '
# brackets: character(0)
# rm_empty_cols: TRUE
```

As you can see the value attributes have been amended by `fhir_crack()`. This will be done automatically, whenever the XPath expressions don't state the attribute explicitly, so you don't have to mind that part.

Extracting 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, you need one `fhir_df_description` per resource type. You can bundle a bunch of `fhir_df_descriptions` in a `fhir_design`. This is basically a named list of `fhir_df_descriptions`, and when you pass it to `fhir_crack()`, the result will be a named list of tables with the same names as the design. Consider an example where we have downloaded MedicationStatements referring to a certain medication as well as the Patient resources these MedicationStatements are linked to.

The design to extract both resource types could look like this:

```
meds <- fhir_df_description(resource = "MedicationStatement",

                            cols = list(
                                ms_id              = "id",
                                status_text        = "text/status",
                                status             = "status",
                                med_system  = "medicationCodeableConcept/coding/system",
                                med_code    = "medicationCodeableConcept/coding/code",
                                med_display = "medicationCodeableConcept/coding/display",
                                dosage             = "dosage/text",
                                patient            = "subject/reference",
                                last_update        = "meta/lastUpdated"
                            ),

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

pat <- fhir_df_description(resource = "Patient")

design <- fhir_design(meds, pat)
```

In this example, we have spelled out the df_description MedicationStatement completely, while we have used a short form for Patients. It looks like this:

```
design
# A fhir_design with 2 df_descriptions:
# =====================================================
# Name: meds
#
# Resource type: MedicationStatement
#
# Columns:
# column name | xpath expression
#  ------------------------------------------------------
# ms_id       | id
# status_text | text/status
# status      | status
# med_system  | medicationCodeableConcept/coding/system
# med_code    | medicationCodeableConcept/coding/code
# med_display | medicationCodeableConcept/coding/display
# dosage      | dosage/text
# patient     | subject/reference
# last_update | meta/lastUpdated
#
# Style:
# sep: |
# brackets: character(0)
# rm_empty_cols: FALSE
# =====================================================
# Name: pat
#
```

```
# Resource type: Patient
#
# Columns:
# An empty fhir_columns object
#
# Style:
# sep: ' '
# brackets: character(0)
# rm_empty_cols: TRUE
```

As you can see, each df_description is identified by a name, which will also be the name of the corresponding table in the result of `fhir_crack()`.

You can assign the names explicitly, if you prefer:

```
design <- fhir_design(meds, pat, names = c("Medications", "Patients"))
design
# A fhir_design with 2 df_descriptions:
# =====================================================
# Name: Medications
#
# Resource type: MedicationStatement
#
# Columns:
# column name | xpath expression
#  ------------------------------------------------------
# ms_id       | id
# status_text | text/status
# status      | status
# med_system  | medicationCodeableConcept/coding/system
# med_code    | medicationCodeableConcept/coding/code
# med_display | medicationCodeableConcept/coding/display
# dosage      | dosage/text
# patient     | subject/reference
# last_update | meta/lastUpdated
#
# Style:
# sep: |
# brackets: character(0)
# rm_empty_cols: FALSE
# =====================================================
# Name: Patients
#
# Resource type: Patient
#
# Columns:
# An empty fhir_columns object
#
# Style:
# sep: ' '
# brackets: character(0)
# rm_empty_cols: TRUE
```

And you can also extract single df_descriptions by their name:

```
design$Patients
# A fhir_df_description with the following elements:
#
# fhir_resource_type: Patient
#
# fhir_columns:
# An empty fhir_columns object
#
# fhir_style:
# sep: ' '
# brackets: character(0)
# rm_empty_cols: TRUE
```

We can use the `design` for `fhir_crack()`:

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

head(list_of_tables$Medications)
#     ms_id status_text status            med_system  med_code      med_display
# 1 2084775   generated active http://snomed.info/ct 429374003 simvastatin 40mg
# 2 2084671   generated active http://snomed.info/ct 429374003 simvastatin 40mg
# 3 2084572   generated active http://snomed.info/ct 429374003 simvastatin 40mg
# 4 2084493   generated active http://snomed.info/ct 429374003 simvastatin 40mg
# 5 2084411   generated active http://snomed.info/ct 429374003 simvastatin 40mg
# 6 2083903   generated active http://snomed.info/ct 429374003 simvastatin 40mg
#             dosage         patient                   last_update
# 1 1 tab once daily Patient/2084708 2021-05-10T05:23:41.686+00:00
# 2 1 tab once daily Patient/2084604 2021-05-10T03:14:24.264+00:00
# 3 1 tab once daily Patient/2084505 2021-05-09T20:09:07.446+00:00
# 4 1 tab once daily Patient/2084426 2021-05-09T18:06:22.183+00:00
# 5 1 tab once daily Patient/2084344 2021-05-09T15:29:57.406+00:00
# 6 1 tab once daily Patient/2083836 2021-05-07T18:48:53.657+00:00

head(list_of_tables$Patients)
#        id meta.versionId              meta.lastUpdated       meta.source
# 1 2082559              1 2021-05-06T23:19:31.967+00:00 #wjSG0x8YGkFzMzav
# 2 2083743              1 2021-05-07T17:53:07.707+00:00 #uTNjj6EX3iU5pKw2
# 3 2081756              1 2021-05-05T23:32:34.605+00:00 #kWCVkuLJ9rQSAYwj
# 4 2083836              1 2021-05-07T18:48:48.888+00:00 #c3JUhMltFV87nsAu
# 5 2084604              1 2021-05-10T03:14:21.154+00:00 #0FuL46MT7dmyDT7v
# 6 2084505              1 2021-05-09T20:09:02.361+00:00 #TFHQXArZ50FoRBmc
#   text.status                                 identifier.system
# 1   generated http://clinfhir.com/fhir/NamingSystem/identifier
# 2   generated http://clinfhir.com/fhir/NamingSystem/identifier
# 3   generated http://clinfhir.com/fhir/NamingSystem/identifier
# 4   generated http://clinfhir.com/fhir/NamingSystem/identifier
# 5   generated                                              <NA>
# 6   generated                                              <NA>
#              identifier.value name.use
# 1             Kaushal.Kishore9 official
# 2 Karlina.Kavi@kaviglobal.com official
# 3    marcelagillr@hotmail.com official
# 4                marcelagillr official
# 5                        <NA> official
# 6                         <NA> official
#
# 1
# 2
# 3
# 4
# 5
# 6 <U+0417><U+0434><U+043E><U+0440><U+043E><U+0432><U+0435><U+043D><U+044C><U+043A><U+043E> <U+0410><U
#
# 1
# 2
# 3
# 4
# 5
# 6 <U+0410><U+0434><U+0430><U+043C> <U+041C><U+0438><U+043A><U+043E><U+043B><U+0430><U+0439><U+043E><U
#                                                                                  name.given
# 1                                                                                  Kaushal
# 2                                                                                  Karlina
# 3                                                                                  Marcela
# 4                                                                                  Marcela
# 5                                                                                    Vicky
# 6 <U+0417><U+0434><U+043E><U+0440><U+043E><U+0432><U+0435><U+043D><U+044C><U+043A><U+043E>
#   gender  birthDate
# 1   male 2000-05-06
# 2 female 2015-05-07
# 3 female 1965-09-10
# 4 female 1965-09-10
# 5   male 2021-05-09
# 6   male 1980-09-10
```

As you can see, the result is a list of two data frames, one for Patient resources and one for MedicationStatement resources. When you use `fhir_crack()` with a `fhir_desgn()` instead of a `fhir_df_description`, the result is an object of class `fhir_df_list` or `fhir_dt_list` that also has the design attached. You can extract the design from a list like this using `fhir_design()`:

```
fhir_design(list_of_tables)
# A fhir_design with 2 df_descriptions:
# =====================================================
# Name: Medications
#
# Resource type: MedicationStatement
#
# Columns:
# column name | xpath expression
#  ------------------------------------------------------------
# ms_id       | id/@value
# status_text | text/status/@value
# status      | status/@value
# med_system  | medicationCodeableConcept/coding/system/@value
# med_code    | medicationCodeableConcept/coding/code/@value
# med_display | medicationCodeableConcept/coding/display/@value
# dosage      | dosage/text/@value
# patient     | subject/reference/@value
# last_update | meta/lastUpdated/@value
```

```
#
# Style:
# sep: |
# brackets: character(0)
# rm_empty_cols: FALSE
# =====================================================
# Name: Patients
#
# Resource type: Patient
#
# Columns:
# An empty fhir_columns object
#
# Style:
# sep: ' '
# brackets: character(0)
# rm_empty_cols: TRUE
```

Note that this doesn't work on single tables created with a `fhir_df_description`.

Saving and reading 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:

```
temp_dir <- tempdir()
fhir_save_design(design, file = paste0(temp_dir,"/design.xml"))
```

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

```
fhir_load_design(paste0(temp_dir,"/design.xml"))
# A fhir_design with 2 df_descriptions:
# =====================================================
# Name: Medications
#
# Resource type: MedicationStatement
#
# Columns:
# column name | xpath expression
#  ------------------------------------------------------
# ms_id       | id
# status_text | text/status
# status      | status
# med_system  | medicationCodeableConcept/coding/system
# med_code    | medicationCodeableConcept/coding/code
# med_display | medicationCodeableConcept/coding/display
# dosage      | dosage/text
# patient     | subject/reference
# last_update | meta/lastUpdated
#
# Style:
# sep: |
# brackets: character(0)
# rm_empty_cols: FALSE
# =====================================================
# Name: Patients
#
# Resource type: Patient
#
# Columns:
# An empty fhir_columns object
#
# Style:
# sep: ' '
# brackets: character(0)
# rm_empty_cols: TRUE
```

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 an example bundle containing just three Patient resources. You can make it available in your workspace like this:

```
bundle <- fhir_unserialize(example_bundles2)
```

This is how the xml looks:

```
<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='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>
        <name>
            <given value='Susie'/>
        </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>
```

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 name attribute.

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

```
desc1 <- fhir_df_description(resource = "Patient",
                             style = fhir_style(sep = " | "))


df1 <- fhir_crack(bundles = bundle, design = desc1, verbose = 0)
df1
#    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 home | work  Berlin | London   postal | postal France | England
#    name.given
# 1       Marie
# 2       Susie
# 3 Frank | Max
```

As you can see, multiple entries for the same attribute (address and name) 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`:

```
desc2 <- fhir_df_description(resource = "Patient",
                             style = fhir_style(sep = " | ",
                                                brackets = c("[", "]"))
                             )


df2 <- fhir_crack(bundles = bundle, design = desc2, verbose = 0)
df2
#       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 [1.1]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 [1.1]physical | [2.1]postal   [1.1]Italy | [2.1]Sweden            [1.1]Susie
# 3   [2.1]postal | [3.1]postal [2.1]France | [3.1]England [1.1]Frank | [2.1]Max
```

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.

Of course the above example is a very specific case that only occurs if your resources have multiple entries with complementary elements. In the majority of cases multiple entries in one resource will have the same structure, thus making numbering of those entries superfluous. But the indices also help to disentangle those entries and put them in separate rows, as you'll see in the next paragraph.

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(df2, 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         <NA>                   3
# 6    [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, 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         <NA>         <NA>    [1]postal       [1]France                   3
# 6    [1]London      [1]work    [1]postal      [1]England                   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, 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, 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]home    [1]Berlin         <NA>            <NA>
# 5 [1]id3        <NA>         <NA>    [1]postal       [1]France
# 6 [1]id3     [1]work    [1]London    [1]postal      [1]England
#               name.given resource_identifier
# 1             [1.1]Marie                   1
# 2             [1.1]Susie                   2
# 3             [1.1]Susie                   2
# 4 [1.1]Frank | [2.1]Max                    3
# 5 [1.1]Frank | [2.1]Max                    3
# 6 [1.1]Frank | [2.1]Max                    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, columns = cols, brackets = c("[","]"),
          sep=" | ", all_columns = TRUE)
#      id address.use address.city address.type address.country
# 1 []id1     [1]home [1]Amsterdam  [1]physical  [1]Netherlands
# 2 []id2     [1]home      [1]Rome  [1]physical        [1]Italy
# 3  <NA>     [1]work [1]Stockholm    [1]postal       [1]Sweden
# 4 []id3     [1]home    [1]Berlin         <NA>            <NA>
# 5  <NA>        <NA>         <NA>    [1]postal       [1]France
# 6  <NA>     [1]work    [1]London    [1]postal      [1]England
#               name.given resource_identifier
# 1             [1.1]Marie                   1
# 2             [1.1]Susie                   2
# 3             [1.1]Susie                   2
# 4 [1.1]Frank | [2.1]Max                    3
# 5 [1.1]Frank | [2.1]Max                    3
# 6 [1.1]Frank | [2.1]Max                    3
```

Instead, melt the attributes one after another:

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

molten_1 <- fhir_melt(df2, 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]home    [1]Berlin         <NA>            <NA>
# 5 [1]id3        <NA>         <NA>    [1]postal       [1]France
# 6 [1]id3     [1]work    [1]London    [1]postal      [1]England
#              name.given resource_identifier
# 1             [1.1]Marie                   1
# 2             [1.1]Susie                   2
# 3             [1.1]Susie                   2
# 4 [1.1]Frank | [2.1]Max                    3
# 5 [1.1]Frank | [2.1]Max                    3
# 6 [1.1]Frank | [2.1]Max                    3

molten_2 <- fhir_melt(molten_1, columns = "name.given", brackets = c("[","]"),
                      sep=" | ", all_columns = TRUE)
molten_2
#       id address.use address.city address.type address.country name.given
# 1 [1]id1     [1]home [1]Amsterdam  [1]physical  [1]Netherlands   [1]Marie
# 2 [1]id2     [1]home      [1]Rome  [1]physical        [1]Italy   [1]Susie
# 3 [1]id2     [1]work [1]Stockholm    [1]postal       [1]Sweden   [1]Susie
# 4 [1]id3     [1]home    [1]Berlin         <NA>            <NA>   [1]Frank
# 5 [1]id3     [1]home    [1]Berlin         <NA>            <NA>     [1]Max
# 6 [1]id3        <NA>         <NA>    [1]postal       [1]France   [1]Frank
# 7 [1]id3        <NA>         <NA>    [1]postal       [1]France     [1]Max
# 8 [1]id3     [1]work    [1]London    [1]postal      [1]England   [1]Frank
# 9 [1]id3     [1]work    [1]London    [1]postal      [1]England     [1]Max
#   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 product of all multiple entries.

If you just want all multiple entries molten correctly, you can use `fhir_melt_all()`. This function will find all columns containing multiple entries and melt them appropriately. Note that this will only work if the column names reflect the path to the corresponding resource element with . as a separator, e.g. `name.given`. These names are produced automatically by `fhir_df_description()` when the cols element is unnamed or omitted.

```
fhir_melt_all(df2, brackets = c("[","]"), sep=" | ")
#    id address.use address.city address.type address.country name.given
# 1 id1        home    Amsterdam     physical     Netherlands      Marie
# 2 id2        home         Rome     physical           Italy      Susie
# 3 id2        work    Stockholm       postal          Sweden      Susie
# 4 id3        home       Berlin         <NA>            <NA>      Frank
# 5 id3        home       Berlin         <NA>            <NA>        Max
# 6 id3        <NA>         <NA>       postal          France      Frank
# 7 id3        <NA>         <NA>       postal          France        Max
# 8 id3        work       London       postal         England      Frank
# 9 id3        work       London       postal         England        Max
```

As you can see `fhir_melt_all()` removes the indices by default. If you need the indices for some reason, you can set `rm_indices = FALSE`:

```
fhir_melt_all(df2, brackets = c("[","]"), sep=" | ", rm_indices = FALSE)
#       id address.use   address.city  address.type  address.country name.given
# 1 [1]id1   [1.1]home [1.1]Amsterdam [1.1]physical [1.1]Netherlands [1.1]Marie
# 2 [1]id2   [1.1]home      [1.1]Rome [1.1]physical       [1.1]Italy [1.1]Susie
# 3 [1]id2   [2.1]work [2.1]Stockholm   [2.1]postal      [2.1]Sweden [1.1]Susie
# 4 [1]id3   [1.1]home    [1.1]Berlin          <NA>             <NA> [1.1]Frank
# 5 [1]id3   [1.1]home    [1.1]Berlin          <NA>             <NA>   [2.1]Max
# 6 [1]id3        <NA>           <NA>   [2.1]postal      [2.1]France [1.1]Frank
# 7 [1]id3        <NA>           <NA>   [2.1]postal      [2.1]France   [2.1]Max
# 8 [1]id3   [3.1]work    [3.1]London   [3.1]postal     [3.1]England [1.1]Frank
# 9 [1]id3   [3.1]work    [3.1]London   [3.1]postal     [3.1]England   [2.1]Max
```

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 name.given
# 1 id1        home    Amsterdam     physical     Netherlands      Marie
# 2 id2        home         Rome     physical           Italy      Susie
# 3 id2        work    Stockholm       postal          Sweden      Susie
# 4 id3        home       Berlin         <NA>            <NA>      Frank
# 5 id3        home       Berlin         <NA>            <NA>        Max
# 6 id3        <NA>         <NA>       postal          France      Frank
# 7 id3        <NA>         <NA>       postal          France        Max
# 8 id3        work       London       postal         England      Frank
# 9 id3        work       London       postal         England        Max
#   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.