fhircrackr: Flatten FHIR resources

2021-03-04

This vignette covers the following topics:

- Extracting one resource type
- The design
- Extracting more than one resource type
- Multiple Entries
- Processing data.frames with multiple entries

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

```
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:

```
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.

Bundles with one resource type

We'll start with `pat_bundles`. We know that this is a list containing xml objects that hold the patient data. 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`).

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

```
design <- list(
    Patients = list(resource = "//Patient",
                    cols = list(id = "id",
                                gender = "gender",
                                name = "name/family",
                                city = "address/city")
                    )
)

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

list_of_tables$Patients
```

```
#       id gender          name       city
# 1   1282   <NA>       Fhirman       <NA>
# 2    267   <NA>         Nr. 1       <NA>
# 3    722   male       Sanchez Osnabrück
# 4    731   male       Sanchez Osnabrück
# 5    736   male       Sanchez Osnabrück
# 6    737   male       Sanchez Osnabrück
# 7    721   male       Sanchez       <NA>
# 8     56   <NA>         Nr. 5       <NA>
# 9    213   <NA>      Sandfrau       <NA>
# 10   118   male        Nr. 13       <NA>
# 11    94   <NA> Maxdata-Demo       <NA>
# 12   151   <NA>         Nr. 8       <NA>
# 13   724   <NA>         Wayne    Gotham
# 14   477   <NA>        Nr. 14       <NA>
# 15   316 female        Nr. 17       <NA>
# 16   175   <NA>        Nr. 10       <NA>
# 17   224   <NA>      Testmann       <NA>
# 18   384   <NA>         Nr. 6       <NA>
# 19  1283   <NA>       Fhirman       <NA>
# 20   280   <NA>         Nr. 4       <NA>
```

The design

Structure

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 in the above example 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:

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>"`.

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. `name = "name/family"` for example creates a column named name which contains the values for the attribute indicated by the XPath expression `"name/family"`.

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

- *sep*: A string defining the separator 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.

Building designs

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 = pat_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 (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 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 = pat_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 = pat_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
```

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(design1, 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"))
# $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
```

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, `design` will contain several data.frame descriptions and the result will be a list of several data frames. Consider `med_bundles`where we have downloaded MedicationStatements referring to a certain medication as well as the Patient resources these MedicationStatements are linked to.

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

```
design <- list(

    MedicationStatement = list(

        resource = "//MedicationStatement",

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

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

    Patients = list(

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

In this example, we have spelled out the data.frame description MedicationStatement completely, while we have used a short form for Patients. We can now use this `design` for `fhir_crack()`:

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

head(list_of_tables$MedicationStatement)
#   MS.ID STATUS.TEXT STATUS     MEDICATION.SYSTEM MEDICATION.CODE
# 1 30233   generated active http://snomed.info/ct       429374003
# 2 42012   generated active http://snomed.info/ct       429374003
# 3 42091   generated active http://snomed.info/ct       429374003
# 4 45646   generated active http://snomed.info/ct       429374003
# 5 45724   generated active http://snomed.info/ct       429374003
# 6 45802   generated active http://snomed.info/ct       429374003
#   MEDICATION.DISPLAY           DOSAGE       PATIENT
```

```
# 1   simvastatin 40mg 1 tab once daily Patient/30163
# 2   simvastatin 40mg 1 tab once daily Patient/41945
# 3   simvastatin 40mg 1 tab once daily Patient/42024
# 4   simvastatin 40mg 1 tab once daily Patient/45579
# 5   simvastatin 40mg 1 tab once daily Patient/45657
# 6   simvastatin 40mg 1 tab once daily Patient/45735
#                       LAST.UPDATE
# 1 2019-09-26T14:34:44.543+00:00
# 2 2019-10-09T20:12:49.778+00:00
# 3 2019-10-09T22:44:05.728+00:00
# 4 2019-10-11T16:17:42.365+00:00
# 5 2019-10-11T16:30:24.411+00:00
# 6 2019-10-11T16:32:05.206+00:00

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

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

Multiple entries

A particularly complicated problem in flattening FHIR resources is caused by the fact that there can be multiple entries to an attribute. 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>
            <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>"
)

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 name 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 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`:

```
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 [1.1]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.

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 [1.1]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


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 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

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]home    [1]Berlin         <NA>            <NA>
# 5 [1]id3     [1]work    [1]London    [1]postal      [1]England
# 6 [1]id3        <NA>         <NA>    [1]postal       [1]France
#              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$Patients, 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>     [1]work    [1]London    [1]postal      [1]England
# 6  <NA>        <NA>         <NA>    [1]postal       [1]France
#              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$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]home    [1]Berlin         <NA>            <NA>
# 5 [1]id3     [1]work    [1]London    [1]postal      [1]England
# 6 [1]id3        <NA>         <NA>    [1]postal       [1]France
#              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     [1]work    [1]London    [1]postal      [1]England   [1]Frank
# 7 [1]id3     [1]work    [1]London    [1]postal      [1]England     [1]Max
# 8 [1]id3        <NA>         <NA>    [1]postal       [1]France   [1]Frank
# 9 [1]id3        <NA>         <NA>    [1]postal       [1]France     [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 cross 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_crack() when the cols element of the design is omitted or set to `NULL`.

```
fhir_melt_all(df2$Patients, 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        work       London       postal         England      Frank
# 7 id3        work       London       postal         England        Max
# 8 id3        <NA>         <NA>       postal          France      Frank
# 9 id3        <NA>         <NA>       postal          France        Max
```

As you can see `fhir_melt_all()` removes the indices by default. If you need the indices for turning your data frame back into resources as described in the respective vignette *fhircrackr:Creating FHIR resources*, you can set `rm_indices = FALSE`:

```
fhir_melt_all(df2$Patients, 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   [3.1]work    [3.1]London   [2.1]postal     [2.1]England [1.1]Frank
# 7 [1]id3   [3.1]work    [3.1]London   [2.1]postal     [2.1]England   [2.1]Max
# 8 [1]id3        <NA>          <NA>   [3.1]postal      [3.1]France [1.1]Frank
# 9 [1]id3        <NA>          <NA>   [3.1]postal      [3.1]France   [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        work       London       postal         England      Frank
# 7 id3        work       London       postal         England        Max
# 8 id3        <NA>         <NA>       postal          France      Frank
# 9 id3        <NA>         <NA>       postal          France        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.