fhircrackr: Creating FHIR resources

2021-03-04

This vignette covers the following topics:

- Motivation
- Flatten and manipulate original resources
- Recreate resources
- A note on referential integrety

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

```r
library(fhircrackr)
```

Motivation

In the vignette `fhircrackr: Flatten FHIR resources` you saw how to flatten FHIR resources into R. But sometimes you might want to go this path in the reverse direction, turning data.frames back into FHIR resources. One use case could be for example if you want to anonymize a bunch of FHIR resources. You could download them from the server, flatten them, manipulate the resulting data.frames appropriately and turn them back into FHIR resources in the end. We'll go through an example of this use case now.

To be able to follow this vignette, you should have a thorough understanding of how `fhircrackr` deals with multiple entries in resources. If necessary, please revisit the vignette *fhircrackr: Flattening resources*.

Flatten and manipulate original resources

We'll use a very simple and particular example bundle for demonstration to show that the respective functions are able to recover the correct resources even in difficult cases. The following example contains 3 Patient resources that we want to recreate in exactly the same structure:

```r
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)
```

Flatten resources

First we'll flatten and melt this bundle so that we'll be able to anonymize the addresses of the patients. It is important to set the `cols` element of the design to `NULL`, because we need the default column names of `fhir_crack()`, as those reflect the structure of the resource and will be needed in the process of building the resources later. It is also important to set `rm_indices = FALSE` in `fhir_melt_all()` as we need the indices to build the resources later on:

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

dfs <- fhir_crack(bundles = bundle_list, design = design, verbose = 0)

#have a look at data.frame
dfs$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

#melt data.frame completely
data <- fhir_melt_all(dfs$Patients, sep = " | ", brackets  = c("[", "]"),
                      rm_indices = FALSE)

#have a look at molten data
data
#       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
```

Manipulate data

The information in the indices is crucial, because we need this information to put every element of `address` back into the correct place later on. Without this information, there would be no way of knowing which of those elements belong to the same `address`.

Nevertheless, those in indices are in our way when we want to manipulate the data. To circumvent this problem, we can use `fhir_extract_indices()` to save the indices, then remove the indices from our data.frame and restore them later on with `fhir_restore_indices()`:

```
#save indices
indices <- fhir_extract_indices(data, brackets = c("[", "]"))
```

```
#remove indices from data
new_data <- fhir_rm_indices(data, brackets = c("[", "]"))

#anonymize address.city and name.given
new_data$address.city <- gsub("[[:alpha:]]", "x", new_data$address.city)
new_data$name.given <- gsub("[[:alpha:]]", "x", new_data$name.given)

#View
new_data
#    id address.use address.city address.type address.country name.given
# 1 id1        home    xxxxxxxxx     physical     Netherlands      xxxxx
# 2 id2        home         xxxx     physical           Italy      xxxxx
# 3 id2        work    xxxxxxxxx       postal          Sweden      xxxxx
# 4 id3        home       xxxxxx         <NA>            <NA>      xxxxx
# 5 id3        home       xxxxxx         <NA>            <NA>        xxx
# 6 id3        work       xxxxxx       postal         England      xxxxx
# 7 id3        work       xxxxxx       postal         England        xxx
# 8 id3        <NA>         <NA>       postal          France      xxxxx
# 9 id3        <NA>         <NA>       postal          France        xxx

#restore indices
new_data <- fhir_restore_indices(new_data, indices)

#View
new_data
#        id address.use   address.city  address.type  address.country name.given
# 1 [1]id1    [1.1]home [1.1]xxxxxxxxx [1.1]physical [1.1]Netherlands [1.1]xxxxx
# 2 [1]id2    [1.1]home      [1.1]xxxx [1.1]physical       [1.1]Italy [1.1]xxxxx
# 3 [1]id2    [2.1]work [2.1]xxxxxxxxx   [2.1]postal      [2.1]Sweden [1.1]xxxxx
# 4 [1]id3    [1.1]home    [1.1]xxxxxx          <NA>             <NA> [1.1]xxxxx
# 5 [1]id3    [1.1]home    [1.1]xxxxxx          <NA>             <NA>   [2.1]xxx
# 6 [1]id3    [3.1]work    [3.1]xxxxxx   [2.1]postal     [2.1]England [1.1]xxxxx
# 7 [1]id3    [3.1]work    [3.1]xxxxxx   [2.1]postal     [2.1]England   [2.1]xxx
# 8 [1]id3         <NA>           <NA>   [3.1]postal      [3.1]France [1.1]xxxxx
# 9 [1]id3         <NA>           <NA>   [3.1]postal      [3.1]France   [2.1]xxx
```

When you manipulate the data you have to make sure that the dimensions as well as row and column order and position of `NA` in the data.frame stay exactly the same, otherwise `fhir_restore_indices()` won't be able to fit the indices correctly.

Recreate resources

After manipulating the data you can recreate the FHIR resources either one by one or altogether in a bundle.

Create single resource

To create a single resource you can use `fhir_create_resource()` on the subset of rows in `new_data` that represents one resource. Rows 4-9 for example represent the third patient resource. Apart from the data rows you need to specify the resource type you want to create and the brackets that are used for the indices in you data:

```
new_data[4:9,]
#        id address.use address.city address.type address.country name.given
# 4 [1]id3    [1.1]home  [1.1]xxxxxx         <NA>            <NA> [1.1]xxxxx
# 5 [1]id3    [1.1]home  [1.1]xxxxxx         <NA>            <NA>   [2.1]xxx
# 6 [1]id3   [3.1]work  [3.1]xxxxxx  [2.1]postal    [2.1]England [1.1]xxxxx
# 7 [1]id3   [3.1]work  [3.1]xxxxxx  [2.1]postal    [2.1]England   [2.1]xxx
# 8 [1]id3        <NA>          <NA>  [3.1]postal     [3.1]France [1.1]xxxxx
# 9 [1]id3        <NA>          <NA>  [3.1]postal     [3.1]France   [2.1]xxx

patient3 <- fhir_create_resource(resourceType = "Patient",
                                 data = new_data[4:9,], brackets = c("[", "]"))
```

You can use functions from the `xml2` package to have a look at the resource. `xml_structure()` will show you the structure of the xml-Object (but not its values), so you can see that the three address entries were restored correctly. As it complicated to display the entire xml nicely within R, we recommend that you save the resource to file with `write_xml` and have a look at it in your preferred text editor.

```
library(xml2)
xml_structure(patient3)
# <resource>
#   {text}
#   <Patient>
#     <id [value]>
#     <address>
#       <use [value]>
#       <city [value]>
#     <address>
#       <type [value]>
#       <country [value]>
#     <address>
#       <use [value]>
#       <city [value]>
#       <type [value]>
#       <country [value]>
#     <name>
#       <given [value]>
#     <name>
#       <given [value]>
```

```
write_xml(patient3, file="patient3.xml")
```

Create a bundle

In most cases you are not interested in single resources. Instead you'll want the resources in a bundle that can be uploaded to a FHIR server again. We'll show you how to create the bundle first and than have a look at how to post it.

To create a bundle, you use the function `fhir_create_bundle()` like this:

```
new_bundle <- fhir_create_bundle(resourceType = "Patient",
                                 data=new_data, brackets = c("[", "]"))
```

`new_bundle` is an xml object representing a bundle of the type `transaction`. You have to save this file to be able look at it or `POST` it to a server:

```r
write_xml(new_bundle, file="new_bundle.xml")
```

Upload Bundle to server

First of all: Uploading resources is not trivial, especially when you want to upload resources that reference each other. It is beyond the scope of this vignette to explain the details of how the upload of resources works but the general concept is described at https://www.hl7.org/fhir/http.html. If you have complex resource networks that need to be created and uploaded correctly, the `fhircrackr` might not be the right tool for you.

Most of the time, you'll want to upload your bundle either as a transaction bundle or as a batch bundle. Both types contain the resources themselves together with a request method for each resource.

While the transaction bundle will fail or pass as a whole, the batch bundle will be processed resource for resource. That means that if 1 out 10 resources in your bundle is invalid, in a transaction bundle none of the resources are uploaded to the server, while in the batch bundle the 9 valid resources are uploaded. You can adjust the bundle type with the argument `bundleType` of `fhir_create_bundle()`.

The request method for each of the resources on the other hand determines how the specific resource is uploaded to the server. The argument `requestMethod` of `fhir_create_bundle()` controls this and should be set to either `"PUT"` or `"POST"`.

With PUT, the resource IDs from the data.frame are preserved in the new resources. When they are sent to the server, it will check whether a resource with this ID already exists. If it does, the existing resource is updated with the new version. If it doesn't exist, the resource is created under the specified ID (this is called *update as create*).

With POST, the server creates the resource under a newly assigned resource ID. This means that the resource IDs from the data.frame are not used in the resources that are sent to the server.

If you want to send your bundle to the server, you should use the `POST()` function from the `httr()` package. Posting the bundle means that the resources are sent to server, where they are processed and either created or updated depending on their request method:

```r
library(httr)
POST(url = "http://fhir.hl7.de:8080/baseDstu3",
     body = upload_file("new_bundle.xml"))
```

A note on referential integrity

Referential integrity is one of the most challenging aspects when you want to upload resources to a server. When you download linked resources of different types (e.g. Observations and the Patients they refer to) from a server, those links are found in the elements of the type `reference` in your resources, which will contain the link with a resource ID.

Cracking the resources, you get two data.frames, one for Patients and one for Observations. If you want to keep the links working, you have to make sure the resource IDs stay exactly the same.

This means references will always break when you upload bundles using `requestMethod = "POST"` as the `requestMethod` (at least if as each bundle contains only one resource type as is currently the only possibility with `fhircrackr`). The correct way to resolve this would be bundling up all resources linking to the same Patient resource (e.g. Observations, Encounters, Conditions, MedicationAdministrations) in one transaction bundle. However, this is not yet implemented in `fhircrackr`.

For now, the only way of keeping links intact is by using the `requestMethod = "PUT"` and carefully considering in which order the resources have to be uploaded. You should start by uploading resources that don't link to any other resources and then upload the resources that link to those you already uploaded. In the simple example of Patient and Observation resources, e.g., you'd have to upload the Patient resources first and then the Observations that link to them.