redcapAPI Best Practices

Shawn Garbett, Benjamin Nutter

2023-04-19

Introduction

REsearch Data Collection or REDCap puts a lot of power into the hands of folks wishing to collect data, from surveys to running clinical trials. Once the data is collected the task of summarization into reports falls upon the statistician or data scientist. R being a useful tool, the department of Biostatistics at Vanderbilt University Medical Center has provided the community with `redcapAPI` to facilitate using REDCap from R.

`redcapAPI` has grown a lot over time, and recently what once worked inside the code and interface for `redcapAPI` no longer aligned with where the REDCap project was today. To this end, a major refactor based on user feedback was undertaken to better address the challenges of a researcher in todays computing environments. This new interface began with version 2.7.0.

Primarily the means of retrieving records has changed from `exportRecords` to `exportRecordsTyped`. The rename of the function is due to the change of the interface to allow time for systems to switch over to the new function. It is important to read over this document and understand the changes if one is a current user of `exportRecords`. However, the changes go much deeper and this document will outline what is a best practices approach to using the library.

The real goal is that a user make the fewest calls to do accomplish their job and have data ready for analysis. This can't happen without user involvement–if the library doesn't work easily for your needs open an issue on github and we'll do our best to work with you.

API_KEY security

The first thing to consider is the API_KEY. This key is what allows for data export from a REDCap project. It is the equivalent of a user name, password and project identifier in a single character string. As such it should be protected as strongly as your password into the systems storing ones data. In the United States, the HIPAA law has a minimum violation of $100 per private health record exposed. In a large clinical trial setting this can easily run into millions of dollars of potential risk.

As such, **the API_KEY should never be stored in a plain text file** unless it's on a tightly monitored and secured production system that cannot work without it.

Logging into REDCap every time one wants to work, and then having to juggle multiple API_KEYs quickly becomes burdensome. Copy and pasting that API_KEY into code (plain text!) and then remembering to delete when finished is too easy to forget. Then a single git commit and simple push to share code and the API_KEY is exposed to the world. It's a very easy mistake to make.

Note: **This functionality was originally in the package `rccola`, but this library is no longer needed. The functionality is built into `redcapAPI` and only requesting connections is supported. This is the preferred long term solution.

```
library(redcapAPI)

# Cuts down on password requests on MAC
options(keyring_backend=keyring::backend_file)

unlockREDCap(c(test_conn    = 'TestRedcapAPI', # REDCap project 1
               sandbox_conn = 'SandboxAPI'),   # REDCap project 2
               keyring      = 'API_KEYs',
               url          = 'https://redcap.vanderbilt.edu/api/')
```

```
## Please enter password in TK window (Alt+Tab)
```

The first time this is called, it asks the user for a password that will be used to unlock the crypto locker `MyKeyring`. A keyring can contain multiple keys, or API_KEY. It will prompt for each API_KEY by the name you've given it, e.g. 'TestRedcapAPI'. If an API_KEY does not connect the call will fail and halt execution in R and it will be deleted from the key_ring to prompt you again. The function creates variables in the current environment of the given names. In future runs it only asks for the password and all the API_KEYs are retrieved and connected.

The keyring is stored in an encrypted form accessible by a single password. If ones laptop were stolen or compromised it is far more difficult for a hacker to gain further access due to the encryption.

This library also cooperates with our production environments by looking for these things in an plain text file `yml` in the directory above execution. This functionality is *only* recommended for system admins and should **never** be used on a work desktop or laptop.

If the easiest path is the best path, it will become the common path.

The Connection Object (caching)

The connection objects are a much richer object. During a lot of REDCap interaction the meta data is necessary to properly interpret the data and guides data transformation. Instead of calling multiple times with each call for this data, the meta data is now cached in the connection object.

Caching saves a lot of round trip calls, but brings with it the burden that sometimes it needs to be refreshed. For example, one is developing in a REDCap object and has an R environment interacting with it. After a call, it's noted that something needs changed in the project proper. Using the REDCap GUI the project's definition is changed. This requires flushing the cache so the next call will retrieve and cached the new data.

```
head(test_conn$fieldnames())
```

```
##   original_field_name choice_value export_field_name
## 1           record_id         <NA>         record_id
## 2            date_dmy         <NA>          date_dmy
## 3            date_mdy         <NA>          date_mdy
## 4            date_ymd         <NA>          date_ymd
## 5     datetime_dmy_hm         <NA>   datetime_dmy_hm
## 6     datetime_mdy_hm         <NA>   datetime_mdy_hm
```

```
test_conn$flush_fieldnames()

head(test_conn$metadata())
```

```
##        field_name            form_name section_header field_type
## 1       record_id fieldtovar_datetimes           <NA>       text
## 2        date_dmy fieldtovar_datetimes           <NA>       text
## 3        date_mdy fieldtovar_datetimes           <NA>       text
## 4        date_ymd fieldtovar_datetimes           <NA>       text
## 5 datetime_dmy_hm fieldtovar_datetimes           <NA>       text
## 6 datetime_mdy_hm fieldtovar_datetimes           <NA>       text
##            field_label select_choices_or_calculations field_note
## 1            Record ID                           <NA>         NA
## 2         Date (D-M-Y)                           <NA>         NA
## 3         Date (M-D-Y)                           <NA>         NA
## 4         Date (Y-M-D)                           <NA>         NA
## 5 Datetime (D-M-Y H:M)                           <NA>         NA
## 6 Datetime (M-D-Y H:M)                           <NA>         NA
##   text_validation_type_or_show_slider_number text_validation_min
## 1                                       <NA>                  NA
## 2                                   date_dmy                  NA
## 3                                   date_mdy                  NA
## 4                                   date_ymd                  NA
## 5                               datetime_dmy                  NA
## 6                               datetime_mdy                  NA
##   text_validation_max identifier branching_logic required_field
## 1                  NA         NA            <NA>             NA
## 2                  NA         NA            <NA>             NA
## 3                  NA         NA            <NA>             NA
## 4                  NA         NA            <NA>             NA
## 5                  NA         NA            <NA>             NA
## 6                  NA         NA            <NA>             NA
##   custom_alignment question_number matrix_group_name matrix_ranking
## 1             <NA>              NA                NA             NA
## 2             <NA>              NA                NA             NA
## 3             <NA>              NA                NA             NA
## 4             <NA>              NA                NA             NA
## 5             <NA>              NA                NA             NA
## 6             <NA>              NA                NA             NA
##   field_annotation
## 1             <NA>
## 2   units={"time"}
## 3             <NA>
## 4             <NA>
## 5             <NA>
## 6             <NA>
```

```
test_conn$flush_metadata()

test_conn$flush_all()
```

Tip: Remember to flush cache after updating project meta data in the GUI.

Another benefit the new connection object brings is the idea of retry. When developing, it's okay if the network hiccups, one can simple rerun the report or command and try again. In a production environment, a report that makes a lot of API calls is assuming that all of those calls are successful to be successful. This is not that case 100% of the time, so a mitigation strategy is needed on the connection object. This is implemented via the `retries`, `retry_interval` and `retry_quietly` parameters when calling to build the connection objects. These are passed to `redcapAPI::redcapConnection` as additional parameters. The default is to quietly make 5 retries on a call, with an interval of 2, 4, 8, 16, and 32 seconds between retries. This greatly improves the odds of building a complex report involving a lot of REDCap calls. The user of the package gets this for **free**.

`exportRecordsTyped`

`exportRecords`, `redcapFactor` and `redcapFlipFactor` still exist in the library but are deprecated. `exportRecordsTyped` is the preferred way forward.

Once armed with a connection from a secured API_KEY in one's R session. The usual goal is to get the data into R, properly typed for use in an R model. Dates and Factors converted into a usable format that makes statistical modelling easy. Type theory is a very deep theoretical topic in mathematics and computer science. `redcapAPI` has made a lot of default choices which are felt to hopefully satisfy 80% of requests.

However, these choices are not a limitation. Care has been taken to allow each of these choices a user defined override and be extensible to handle just about anything the user would prefer. The strategy chosen is called inversion of control.

Understanding the type 'casting' algorithm is important if the default choices are not satisfactory. Casting refers to the transformation of one data class in R to another.

The algorithm

REDCap stores all data as character strings. A validation on input may be specified as a `field_type` in the REDCap project. However, these might be added later, changed or raw data from a different system pushed up. The declared `field_type` from the REDCap meta data has no guarantee to describe the data format of the actual data. This divergence can be a source of frustration and difficulty, thus we've designed the following steps of the process to cast a column of data from a project:

1. Detect fields that are NA. This defaults to `""` or `"NA"`.
2. Fields that are not NA, are passed through a validation for the `field_type`.
3. Fields that are not NA, that pass validation are then cast to the desired class.

The choice of which routine to call is a defined by `field_type`. The current version of REDCap as of this writing is: `date_`, `datetime_`, `datetime_seconds_`, `time_mm_ss`, `time_hh_mm_ss`, `time`, `float`, `number`, `calc`, `int`, `integer`, `yesno`, `truefalse`, `checkbox`, `form_complete`, `select`, `radio`, `dropdown`, and `sql`.

The field_type for `date_`, `datetime_` and `datetime_seconds` are all truncated from the original as all of these are reported in the API as ymd.

NA

The definition of NA may vary. An example is someone uploaded external data that says "-5" is an NA due to a code book. These values are not desired to be treated as anything but NA. In this case the user needs to specify an override.

The expected function signature is `function(x, field_name, coding)`. The following demonstrates some test data. It follows with a declaration that date "2023-03-24" is to be treated as NA. Then, "2023-03-24" is only to be treated as NA for the field `date_mdy`. Coding is only provided if there is a defined code book for the variable.

```r
head(exportRecordsTyped(test_conn)[,1:10])
```

```
##   record_id redcap_event_name   date_dmy   date_mdy   date_ymd
## 1         1     event_1_arm_1 2023-02-24 2023-02-24 2023-02-24
## 2         2     event_1_arm_1       <NA>       <NA>       <NA>
## 3         3     event_1_arm_1       <NA>       <NA>       <NA>
## 4        10     event_1_arm_1       <NA>       <NA>       <NA>
## 5        11     event_1_arm_1       <NA>       <NA>       <NA>
## 6        12     event_1_arm_1       <NA>       <NA>       <NA>
##       datetime_dmy_hm     datetime_mdy_hm     datetime_ymd_hm
## 1 2023-02-24 12:04:00 2023-02-24 12:04:00 2023-02-24 12:04:00
## 2                <NA>                <NA>                <NA>
## 3                <NA>                <NA>                <NA>
## 4                <NA>                <NA>                <NA>
## 5                <NA>                <NA>                <NA>
## 6                <NA>                <NA>                <NA>
##      datetime_dmy_hms    datetime_mdy_hms
## 1 2023-02-24 12:40:50 2023-02-24 12:40:50
## 2                <NA>                <NA>
## 3                <NA>                <NA>
## 4                <NA>                <NA>
## 5                <NA>                <NA>
## 6                <NA>                <NA>
```

```r
my_na_detector <- function(x, field_name, coding) is.na(x) | x=="" | x == "2023-02-24"

head(exportRecordsTyped(test_conn, na=list(date_=my_na_detector))[,1:10])
```

```
##   record_id redcap_event_name date_dmy date_mdy date_ymd     datetime_dmy_hm
## 1         1     event_1_arm_1     <NA>     <NA>     <NA> 2023-02-24 12:04:00
## 2         2     event_1_arm_1     <NA>     <NA>     <NA>                <NA>
## 3         3     event_1_arm_1     <NA>     <NA>     <NA>                <NA>
## 4        10     event_1_arm_1     <NA>     <NA>     <NA>                <NA>
## 5        11     event_1_arm_1     <NA>     <NA>     <NA>                <NA>
## 6        12     event_1_arm_1     <NA>     <NA>     <NA>                <NA>
##       datetime_mdy_hm     datetime_ymd_hm    datetime_dmy_hms
## 1 2023-02-24 12:04:00 2023-02-24 12:04:00 2023-02-24 12:40:50
## 2                <NA>                <NA>                <NA>
## 3                <NA>                <NA>                <NA>
## 4                <NA>                <NA>                <NA>
## 5                <NA>                <NA>                <NA>
## 6                <NA>                <NA>                <NA>
##      datetime_mdy_hms
## 1 2023-02-24 12:40:50
## 2                <NA>
## 3                <NA>
## 4                <NA>
## 5                <NA>
## 6                <NA>
```

```r
my_limited_na_detector <- function(x, field_name, coding)
  is.na(x) |
  x==""    |
  field_name=='date_mdy'

head(exportRecordsTyped(test_conn, na=list(date_=my_limited_na_detector))[,1:10])
```

```
##   record_id redcap_event_name   date_dmy date_mdy   date_ymd
## 1         1     event_1_arm_1 2023-02-24     <NA> 2023-02-24
## 2         2     event_1_arm_1       <NA>     <NA>       <NA>
## 3         3     event_1_arm_1       <NA>     <NA>       <NA>
## 4        10     event_1_arm_1       <NA>     <NA>       <NA>
## 5        11     event_1_arm_1       <NA>     <NA>       <NA>
## 6        12     event_1_arm_1       <NA>     <NA>       <NA>
##       datetime_dmy_hm     datetime_mdy_hm     datetime_ymd_hm
## 1 2023-02-24 12:04:00 2023-02-24 12:04:00 2023-02-24 12:04:00
## 2                <NA>                <NA>                <NA>
## 3                <NA>                <NA>                <NA>
## 4                <NA>                <NA>                <NA>
## 5                <NA>                <NA>                <NA>
## 6                <NA>                <NA>                <NA>
##      datetime_dmy_hms    datetime_mdy_hms
## 1 2023-02-24 12:40:50 2023-02-24 12:40:50
## 2                <NA>                <NA>
## 3                <NA>                <NA>
## 4                <NA>                <NA>
## 5                <NA>                <NA>
## 6                <NA>                <NA>
```

It is hopefully a rare case when this is needed. The next step, validation, has an available report that will hopefully make it clear when it is.

Validation

This step based on `field_type` calls a function that returns a vector of logical specifying what is valid or not. The simplest of these is via a regular expression or regex. Detailing construction of a regex for validation of a field is outside the scope of this document, good tutorials are available online such as https://regextutorial.org/. It's helpful to have an interactive environment to develop one, we used https://regex101.com/ frequently in developing the regexs provided by default.

The function signature once again is `function(x, field_name, coding)`.

The default set of validations is:

```
list(
  date_              = valRx("^[0-9]{1,4}-(0?[1-9]|1[012])-(0?[1-9]|[12][0-9]|3[01])$"),
  datetime_          = valRx("^[0-9]{1,4}-(0?[1-9]|1[012])-(0?[1-9]|[12][0-9]|3[01])\\s([0-9]|0[0-9]|1[
  datetime_seconds_  = valRx("^[0-9]{1,4}-(0?[1-9]|1[012])-(0?[1-9]|[12][0-9]|3[01])\\s([0-9]|0[0-9]|1[
  time_mm_ss         = valRx("^[0-5][0-9]:[0-5][0-9]$"),
  time_hh_mm_ss      = valRx("^([0-9]|0[0-9]|1[0-9]|2[0-3]):[0-5][0-9]:[0-5][0-9]$"),
  time               = valRx("^([0-9]|0[0-9]|1[0-9]|2[0-3]):[0-5][0-9]$"),
  float              = valRx("^[-+]?(([0-9]+\\.?[0-9]*)|(\\.[0-9]+))([Ee][+-]?[0-9]+)?$"),
  number             = valRx("^[-+]?(([0-9]+\\.?[0-9]*)|(\\.[0-9]+))([Ee][+-]?[0-9]+)?$"),
  calc               = valRx("^[-+]?(([0-9]+\\.?[0-9]*)|(\\.[0-9]+))([Ee][+-]?[0-9]+)?$"),
  int                = valRx("^[-+]?[0-9]+(|\\.|\\.[0]+)$"),
  integer             = valRx("^[-+]?[0-9]+$"),
  yesno               = valRx("^(?i)(0|1|yes|no)$"),
  truefalse           = valRx("^(0|1|true|false)$"),
  checkbox            = valRx("^(?i)(0|1|yes|no)$"),
  form_complete       = valRx("^[012]$"),
  select              = valChoice,
  radio               = valChoice,
  dropdown            = valChoice,
  sql                 = NA # Incomplete at present
)
```

Ignore the complex regular expressions above if you're not familiar. Let's look at a building a simple validation for `form_complete`: `valRx("^[012]$")`. The regex here starts with "^" for beginning of string, it's followed by a set in square brakets meaning to match one of those characters, then the "$" meaning end of string. Thus, it asks to build a validation function of the right signature that will return a vector that is TRUE for input that is a single character "0", "1" or "2" and FALSE otherwise.

All characters that fail a validation are returned as an attribute "invalid" on the resulting data.frame. The default print method will format this into Markdown, and all records that are not NA that fail validation will be called out.

We will use a RegEx to make a lot of numbers fail in this example, and use the [1:10,] selector to limit the output for this example.

```
Records <- exportRecordsTyped(test_conn,
             validation=list(number=valRx("^5$|^-100$")))
```

```
## Warning in .exportRecordsTyped_attachInvalid(conn = rcon, Records = Records, :
## Some records failed validation. See 'invalid' attr.
```

```
summary(Records$prereq_number)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##  -100.0  -100.0   -47.5   -47.5     5.0     5.0      21
```

```
knitr::asis_output(format(attr(Records, "invalid")[1:10,]))
```

Failed Validations from REDCap project 'TestRedcapAPI'

Wed 19 Apr 2023 12:00:00 AM CST
Package redcapAPI version 2.7.0
REDCap version 13.4.9

- Field[number] 'prereq_number' has 10 failures
    - Row 4, Record Id '10', Value '1'
    - Row 5, Record Id '11', Value '1'
    - Row 6, Record Id '12', Value '1'
    - Row 7, Record Id '13', Value '1'
    - Row 8, Record Id '14', Value '1'
    - Row 9, Record Id '15', Value '1'
    - Row 10, Record Id '16', Value '1'

  - Row 11, Record Id ‘17’, Value ‘1’
  - Row 12, Record Id ‘18’, Value ‘1’
  - Row 13, Record Id ‘19’, Value ‘1’

This shows that the number records containing “1” did not pass the regex validation and they will become NA in the final output. The field name, type, row number and record id all help the user to quickly diagnose what is not validating.

Once again, overriding these default is expected to be a rare need, but the option is available should it arise. Casting variables to the desired class is up next.

Casting

The na and validation callback list serve to exclude what should not be attempted to cast into a class. This prevents the library from crashing when the input does not match the expected format. This is particularly troublesome with date and time casting, and excluding these failed validations ensures the cast will be successful.

The function signature for these callbacks is the familiar `function(x, field_name, coding)`.

```
list(
  date_               = function(x, ...) as.POSIXct(x, format = "%Y-%m-%d"),
  datetime_           = function(x, ...) as.POSIXct(x, format = "%Y-%m-%d %H:%M"),
  datetime_seconds_  = function(x, ...) as.POSIXct(x, format = "%Y-%m-%d %H:%M:%S"),
  time_mm_ss          = function(x, ...) chron::times(ifelse(is.na(x),NA,paste0("00:",x)), format=c(times
  time_hh_mm_ss       = function(x, ...) chron::times(x, format=c(times="h:m:s")),
  time                = function(x, ...) chron::times(gsub("(^\\d{2}:\\d{2}$)", "\\1:00", x),
                                                     format=c(times="h:m:s")),
  float               = as.numeric,
  number              = as.numeric,
  calc                = as.numeric,
  int                 = as.integer,
  integer             = as.numeric,
  yesno               = castLabel,
  truefalse           = function(x, ...) x=='1' | tolower(x) =='true',
  checkbox            = castChecked,
  form_complete       = castLabel,
  select              = castLabel,
  radio               = castLabel,
  dropdown            = castLabel,
  sql                 = NA
)
```

A common request is instead of using POSIXct for the dates, to use the internal as.Date function.

NOTE: An exported object `cast_raw` consists of NA for each of these keys. If one desires raw data the cast function is NA.

```
head(exportRecordsTyped(test_conn, cast=list(date_=as.Date))[,1:10])
```

```
##   record_id redcap_event_name   date_dmy   date_mdy   date_ymd
## 1         1     event_1_arm_1 2023-02-24 2023-02-24 2023-02-24
## 2         2     event_1_arm_1       <NA>       <NA>       <NA>
## 3         3     event_1_arm_1       <NA>       <NA>       <NA>
## 4        10     event_1_arm_1       <NA>       <NA>       <NA>
## 5        11     event_1_arm_1       <NA>       <NA>       <NA>
## 6        12     event_1_arm_1       <NA>       <NA>       <NA>
##       datetime_dmy_hm     datetime_mdy_hm     datetime_ymd_hm
## 1 2023-02-24 12:04:00 2023-02-24 12:04:00 2023-02-24 12:04:00
## 2                <NA>                <NA>                <NA>
## 3                <NA>                <NA>                <NA>
## 4                <NA>                <NA>                <NA>
## 5                <NA>                <NA>                <NA>
## 6                <NA>                <NA>                <NA>
##      datetime_dmy_hms    datetime_mdy_hms
## 1 2023-02-24 12:40:50 2023-02-24 12:40:50
## 2                <NA>                <NA>
## 3                <NA>                <NA>
## 4                <NA>                <NA>
## 5                <NA>                <NA>
## 6                <NA>                <NA>
```

The date columns are now of the internal base R `date` class. Various helper routines are available on the `?fieldValidationAndCasting` help page. One of note is `castCode` which when used instead of `castLabel` it will cast to the coded value and not the labelled value.

With na, validation and cast covered a large amount of new functionality and control is in the hands of the user.

Labels and Units

Inversion of control is available for the assignment of attributes to columns as well. There exists an assignment argument which will is a list of functions that will assign their output to the attribute using the name of the list key.

The defaults add labels and units.

```
assignment=list(label=stripHTMLandUnicode, units=unitsFieldAnnotation)
```

The function signature for these is `function(field_name, field_label, field_annotation)`.

The label for a column is created by stripping HTML and Unicode characters from the REDCap field label. The units are done by searching the field annotation for something of the following form: `units={"meters"}` (using a regex).

If one desired custom attributes on columns based on this information it can be done with an override.

Post Processing

The scope and purpose of `exportRecordsTyped` was to extract the data frame in the desired classes for analysis. Sometimes post processing of the frame for further cleanup is desired and casting cannot do all that is required. Several useful helper routines for post processing are provided. The first we'll cover is `recastRecords`.

`recastRecords`

User have reported that `redcapFactorFlip` has been very useful for them to switch the way the data was cast in a back and forth manner. The current library has deprecated `redcapFactorFlip` and the new method to replace it is `recastRecords`.

```r
exportRecordsTyped(test_conn,
  fields=c("record_id", "date_dmy",
           "date_mdy", "prereq_yesno")) |>
  recastRecords(test_conn,
                fields = c("date_dmy", "date_mdy", "prereq_yesno"),
                cast   = list(date_  = as.Date,
                              yesno = castRaw)) |>
head()
```

```
##   record_id redcap_event_name   date_dmy   date_mdy prereq_yesno
## 1         1     event_1_arm_1 2023-02-24 2023-02-24           NA
## 2         2     event_1_arm_1       <NA>       <NA>           NA
## 3         3     event_1_arm_1       <NA>       <NA>           NA
## 4        10     event_1_arm_1       <NA>       <NA>            0
## 5        11     event_1_arm_1       <NA>       <NA>            0
## 6        12     event_1_arm_1       <NA>       <NA>            0
```

BONUS

Branching Logic NA detection

Cornacopia of Functions to explore