redcapAPI Best Practices

Benjamin Nutter, Savannah Obregon, Shawn Garbett

2023-06-27

Contents

Introduction 2
API_KEY security 2
The Connection Object (caching) 4
`exportRecordsTyped` 6
  The algorithm 6
  NA 6
  Validation 8

Failed Validations from REDCap project 'TestRedcapAPI (Experimental)' 9
  Casting 10
  Labels and Units 11
  Forms 11
Post Processing 12
  `recastRecords` 12
  mChoice 12
  guessCast 13
  Split Data Into Forms 13
  Widen / Shorten a Repeating Instrument 14
  castForImport 14
Helper Functions 17
  Bulk Records Pull 17
  Branching Logic NA detection 18
  Cornucopia of Functions to explore 19
  Reproducing this Vignette 20
  Custom API Calls 20
Thanks 21

Introduction

Research Electronic Data Capture or REDCap puts a lot of power into the hands of those wishing to collect data, from surveys to running clinical trials. Once the data is collected the statistician or data scientist is responsible for summarizing the collected data into reports. R being a useful tool for this purpose, the department of Biostatistics at Vanderbilt University Medical Center has provided the community with the package `redcapAPI` to facilitate using REDCap from R.

`redcapAPI` has undergone significant growth over time, causing its previous code and interface to no longer align with the current state of the REDCap project. The original package started to directly expose the raw API in R and the common needs of users propagated via snippets of R code. To address the true needs of the user, a major refactor based on user feedback was undertaken to better address the challenges of a researcher in today's computing environments. This new interface began with version 2.7.0.

The primary change has been in the method of retrieving records, which has shifted from using `exportRecords` to `exportRecordsTyped`. The reason for renaming the function is to provide ample time for systems to transition to the new interface. It is important to read over this document and understand the changes if one is a current user of `exportRecords`. However, the modifications are considerably more extensive. This document will outline the best practices approach to using the library.

The ultimate goal is to minimize the number of calls a user needs to make to accomplish their task and have the data prepared for analysis. This can't happen without user involvement–if the library doesn't work easily for ones needs, please open an issue on GitHub and we will do our best to provide a solution.

If one wishes to reproduce these examples, see '*Reproducing this Vignette*' towards the end of this document.

API_KEY security

The first thing to consider is the API_KEY. This key is what enables the export of data 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 one's password into the systems that store one's 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.

Therefore, **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 juggling multiple API_KEYs will quickly become burdensome. Copying and pasting that API_KEY into code (plain text!) and then remembering to delete when finished is too easy to forget. A single git commit and simple push to share code and the API_KEY is exposed to the world. Making a mistake is highly probable, and the risk of exposing any plain text file in a directory is high. The problem is so rampant that at one point there was a website scanning `github.com` for API_KEYS and posting them on a rolling kiosk. Exposures were occurring every few seconds. Many of these were for other APIs, but the risk of exposure through an inadvertent commit cannot be understated.

The library provides a helper function that uses an encrypted local file to store API_KEYS for opening connections to REDCap. Using this function greatly reduces the risk of exposure. It has tools to facilitate a transfer of code to an automated environment as well. See `?unlockREDcap` for those details.

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.*

```r
library(redcapAPI)
library(Hmisc)
```

```
##
## Attaching package: 'Hmisc'

## The following objects are masked from 'package:base':
##
##     format.pval, units
```

```
# 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',
             envir        = globalenv(),
             url          = 'https://redcap.vanderbilt.edu/api/')
```

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

## <environment: R_GlobalEnv>
```

The first time this is called, it asks the user for a password that will be used to unlock the crypto locker `API_KEYs`. A keyring can contain multiple API_KEYs and hence the name we've given it here–one is free to use any naming they desire. The first time it is run it will prompt for each API_KEY by the name one has 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 one to enter it again. Subsequent calls will not prompt for an API_KEY, just the password one has given to unlock the remaining keys in the locker. It will stay open in an R session and not prompt again. *caveat: each knit button press from RStudio creates a new session, so it will prompt each call to knitr*. MacOS has a password prompt problem with `getPass`; this only works from RStudio at present on MacOS.

Specifying `envir=globalenv()` tells the function to create the connections in the global environment as variables. Without this the function returns a list of the connections.

In summary, the keyring is stored in an encrypted form accessible by a single password. If one's 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 a 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.

MacOS and Password Prompts Unfortunately, at present the package `getPass` is broken on MacOS when using RStudio and pressing the 'Knit' button, and the alternative package `askpass` doesn't work on any platform in this case. If one wishes to use RStudio and `knitr` on MacOS then the `rstudioapi` package should be installed and `passwordFUN=rstudioapi::askForPassword` should be specified in the call.

If one is not running from RStudio+MacOS the `passwordFUN` argument is not required and should be left off.

Other API_KEY Leakage Risks To prevent R from inadvertently saving API_KEYs to files, it is recommended to turn off any saving of workspace data. In RStudio this is under "Tools -> Global Options (General)", set 'Save Workspace to .RData on exit' to *Never*.

For command line users create an `.Rprofile` file in one's home directory containing the following base function override:

```
utils::assignInNamespace(
  "q",
  function(save="no", status=0, runLast=TRUE)
  {
    .Internal(quit(save, status, runLast))
  },
  "base"
)
```

More details on keyring management are in the `keyring` package. If one forgets their password, one helpful solution is to delete it and try again using the `keyring::keyring_delete("API_KEYs")` function.

Once again, per our design goals, our choices and recommendations do not limit the user. If one has their own system of API_KEY management, one can still open a connection directly using `redcapConnection`.

If the easiest path is the best path, it will become the common path. We've done our best to make best security practice the easiest path.

The Connection Object (caching)

The connection objects are a much richer object than the older version of the library. During many REDCap interactions the meta data is necessary to properly interpret the data and guide 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.

```r
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
```

```r
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 of the new connection object is the idea of retry. When developing, it's okay if the network hiccups, one can simply 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 in order to complete execution. 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** and by specifying `retries=10` it will try up to 30 minutes per call if necessary, allowing downtime to not affect report generation. This is important on automated systems that require reliability and can wait.

exportRecordsTyped

exportRecords, redcapFactor and redcapFlipFactor still exist in the library but are deprecated. These functions will no longer be updated. exportRecordsTyped is the preferred method moving forward.

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 need to be converted into a usable format that makes statistical modelling easy. Type theory is a very deep theoretical topic in mathematics and computer science and thus this topic is complex. redcapAPI has made a lot of default choices which we felt will satisfy 80% of use cases.

However, these choices are not a limitation. Care has been taken to allow user defined overrides for each of these choices and to 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 referring to the transformation of one data class in R to another (aka type casting).

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 ""–the empty string.
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 at the time 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 "2023-02-24" 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 redcap_repeat_instrument redcap_repeat_instance
## 1         1     event_1_arm_1                     <NA>                   <NA>
## 2         1     event_1_arm_1     repeating_instrument                      1
## 3         1     event_1_arm_1     repeating_instrument                      2
## 4         2     event_1_arm_1                     <NA>                   <NA>
## 5         3     event_1_arm_1                     <NA>                   <NA>
## 6        10     event_1_arm_1                     <NA>                   <NA>
##     date_dmy   date_mdy   date_ymd     datetime_dmy_hm     datetime_mdy_hm
## 1 2023-02-24 2023-02-24 2023-02-24 2023-02-24 12:04:00 2023-02-24 12:04:00
## 2       <NA>       <NA>       <NA>                <NA>                <NA>
## 3       <NA>       <NA>       <NA>                <NA>                <NA>
## 4       <NA>       <NA>       <NA>                <NA>                <NA>
## 5       <NA>       <NA>       <NA>                <NA>                <NA>
## 6       <NA>       <NA>       <NA>                <NA>                <NA>
##       datetime_ymd_hm
## 1 2023-02-24 12:04:00
## 2                <NA>
## 3                <NA>
## 4                <NA>
## 5                <NA>
## 6                <NA>
```

```
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 redcap_repeat_instrument redcap_repeat_instance
## 1         1     event_1_arm_1                     <NA>                   <NA>
## 2         1     event_1_arm_1     repeating_instrument                      1
## 3         1     event_1_arm_1     repeating_instrument                      2
## 4         2     event_1_arm_1                     <NA>                   <NA>
## 5         3     event_1_arm_1                     <NA>                   <NA>
## 6        10     event_1_arm_1                     <NA>                   <NA>
##   date_dmy date_mdy date_ymd     datetime_dmy_hm     datetime_mdy_hm
## 1     <NA>     <NA>     <NA> 2023-02-24 12:04:00 2023-02-24 12:04:00
## 2     <NA>     <NA>     <NA>                <NA>                <NA>
## 3     <NA>     <NA>     <NA>                <NA>                <NA>
## 4     <NA>     <NA>     <NA>                <NA>                <NA>
## 5     <NA>     <NA>     <NA>                <NA>                <NA>
## 6     <NA>     <NA>     <NA>                <NA>                <NA>
##       datetime_ymd_hm
## 1 2023-02-24 12:04:00
## 2                <NA>
## 3                <NA>
## 4                <NA>
## 5                <NA>
## 6                <NA>
```

```
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 redcap_repeat_instrument redcap_repeat_instance
## 1         1     event_1_arm_1                     <NA>                   <NA>
## 2         1     event_1_arm_1     repeating_instrument                      1
## 3         1     event_1_arm_1     repeating_instrument                      2
## 4         2     event_1_arm_1                     <NA>                   <NA>
## 5         3     event_1_arm_1                     <NA>                   <NA>
## 6        10     event_1_arm_1                     <NA>                   <NA>
##     date_dmy date_mdy   date_ymd     datetime_dmy_hm     datetime_mdy_hm
## 1 2023-02-24     <NA> 2023-02-24 2023-02-24 12:04:00 2023-02-24 12:04:00
## 2       <NA>     <NA>       <NA>                <NA>                <NA>
## 3       <NA>     <NA>       <NA>                <NA>                <NA>
## 4       <NA>     <NA>       <NA>                <NA>                <NA>
## 5       <NA>     <NA>       <NA>                <NA>                <NA>
## 6       <NA>     <NA>       <NA>                <NA>                <NA>
##       datetime_ymd_hm
## 1 2023-02-24 12:04:00
## 2                <NA>
## 3                <NA>
## 4                <NA>
## 5                <NA>
## 6                <NA>
```

It is hopefully a rare case when this is needed. The next step, validation, has an available report that should clarify when it is required.

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 unfamiliar. Let's look at a building a simple validation for `form_complete`: `valRx("^[012]$")`. The regex here starts with "^" for beginning of string, it's followed by a set in square brackets meaning to match one of those characters, then the "$" 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 .castRecords_attachInvalid(rcon = rcon, Records = Records, Raw =
## Raw, : 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      23
```

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

Failed Validations from REDCap project 'TestRedcapAPI (Experimental)'

Tue 27 Jun 2023 12:00:00 AM UTC
Package redcapAPI version 2.7.3
REDCap version 13.7.2

- Field[number] 'prereq_number' has 10 failures
  - Row 6, Record Id '10', Value '1'
  - Row 7, Record Id '11', Value '1'
  - Row 8, Record Id '12', Value '1'
  - Row 9, Record Id '13', Value '1'
  - Row 10, Record Id '14', Value '1'
  - Row 11, Record Id '15', Value '1'
  - Row 12, Record Id '16', Value '1'
  - Row 13, Record Id '17', Value '1'
  - Row 14, Record Id '18', Value '1'
  - Row 15, Record Id '19', Value '1'

This shows that the number records containing “1” did not pass the regex validation and these 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 the 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 to use the internal as.Date function instead of POSIXct for handling dates.

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 redcap_repeat_instrument redcap_repeat_instance
## 1         1     event_1_arm_1                     <NA>                   <NA>
## 2         1     event_1_arm_1     repeating_instrument                      1
## 3         1     event_1_arm_1     repeating_instrument                      2
## 4         2     event_1_arm_1                     <NA>                   <NA>
## 5         3     event_1_arm_1                     <NA>                   <NA>
## 6        10     event_1_arm_1                     <NA>                   <NA>
##     date_dmy   date_mdy   date_ymd     datetime_dmy_hm     datetime_mdy_hm
## 1 2023-02-24 2023-02-24 2023-02-24 2023-02-24 12:04:00 2023-02-24 12:04:00
## 2       <NA>       <NA>       <NA>                <NA>                <NA>
## 3       <NA>       <NA>       <NA>                <NA>                <NA>
## 4       <NA>       <NA>       <NA>                <NA>                <NA>
## 5       <NA>       <NA>       <NA>                <NA>                <NA>
## 6       <NA>       <NA>       <NA>                <NA>                <NA>
##      datetime_ymd_hm
## 1 2023-02-24 12:04:00
## 2                <NA>
## 3                <NA>
## 4                <NA>
## 5                <NA>
## 6                <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 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.

Forms

If the forms argument is specified, the return from `exportRecordsTyped` filters the data down to only rows and columns containing information for the specified forms. I.e., REDCap raw data is in “block sparse” format and what users really want is “long” format without extraneous empty rows.

```
exportRecordsTyped(test_conn, forms="repeating_instrument")
```

```
##   record_id redcap_event_name redcap_repeat_instrument redcap_repeat_instance
## 2         1     event_1_arm_1     repeating_instrument                      1
## 3         1     event_1_arm_1     repeating_instrument                      2
##   repeat_question_1     repeat_datetime repeating_instrument_complete
## 2              Good 2023-04-20 13:34:00                    Incomplete
## 3               Bad 2023-04-20 13:35:00                      Complete
```

There are only 2 rows from a test project with over 20 rows of data for this form.

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

Users 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.

```
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 redcap_repeat_instrument redcap_repeat_instance
## 1         1     event_1_arm_1                     <NA>                   <NA>
## 4        10     event_1_arm_1                     <NA>                   <NA>
## 5        11     event_1_arm_1                     <NA>                   <NA>
## 6        12     event_1_arm_1                     <NA>                   <NA>
## 7        13     event_1_arm_1                     <NA>                   <NA>
## 8        14     event_1_arm_1                     <NA>                   <NA>
##     date_dmy   date_mdy prereq_yesno
## 1 2023-02-24 2023-02-24           NA
## 4       <NA>       <NA>            0
## 5       <NA>       <NA>            0
## 6       <NA>       <NA>            0
## 7       <NA>       <NA>            0
## 8       <NA>       <NA>            0
```

Recasting may be performed using a character vector of field names; a numeric vector of field indices; or a logical vector (the logical vector must be the same length as the number of columns in the data frame).

mChoice

Users of Hmisc or rms might want multiple choice class fields added to their resulting Record data.frame.

```
x <- exportRecordsTyped(test_conn) |> mChoiceCast(test_conn)
x$checkbox_test
```

```
##  [1]
##  [5]                 Guitar                                Mandolin
##  [9] Ukulele         Mandolin         Ukulele;Mandolin Guitar;Mandolin
## [13] Ukulele         Guitar;Ukulele   Guitar;Mandolin  Guitar
## [17] Ukulele          Mandolin         Guitar           Guitar;Mandolin
## [21] Ukulele          Guitar;Ukulele   Ukulele          Ukulele
## [25] Guitar;Ukulele
## attr(,"label")
## [1] checkbox_test
## Levels: Guitar Ukulele Mandolin
```

guessCast

What if validations were never added to the project and one would like to take a guess at casting, i.e. not rely on the meta data? Any field that remains character can be subject to a guess based on passing validation.

This is kept as a separate function to ensure that the user makes a clear choice in using guesswork.

```
exportRecordsTyped(test_conn, fields="date_dmy", cast=raw_cast) |>
  guessCast(
    test_conn,
    validation=valRx("^[0-9]{1,4}-(0?[1-9]|1[012])-(0?[1-9]|[12][0-9]|3[01])$"),
    cast=as.Date,
    threshold=0.1)
```

```
## guessCast triggered on date_dmy for 2 of 2 records.

##    record_id redcap_event_name redcap_repeat_instrument redcap_repeat_instance
## 1          1     event_1_arm_1                     <NA>                   <NA>
## 25        35     event_1_arm_1                     <NA>                   <NA>
##      date_dmy
## 1  2023-02-24
## 25 2023-02-24
```

Since dates are common, a helper specifically for this guess is provided.

```
exportRecordsTyped(test_conn, fields="date_dmy", cast=raw_cast) |>
  guessDate(test_conn, threshold=0.1)
```

```
## guessCast triggered on date_dmy for 2 of 2 records.

##    record_id redcap_event_name redcap_repeat_instrument redcap_repeat_instance
## 1          1     event_1_arm_1                     <NA>                   <NA>
## 25        35     event_1_arm_1                     <NA>                   <NA>
##      date_dmy
## 1  2023-02-24
## 25 2023-02-24
```

Split Data Into Forms

There are times when it is desirable to separate a data set into its forms/instruments. Most notably, this may be necessary to work with repeating instruments in projects that have complex repeating structures.

Separating forms can be done via multiple calls to the API, or it can be done in post-processing via `splitForms`.

FIXME: Make an example with splitForms after we finish the function.

NOTE: The later section on `exportBulkRecords` might be a better option than using splitForms.

Widen / Shorten a Repeating Instrument

When working with repeating instruments in REDCap the default export is a tall and thin data frame where repeat instances are split into separate rows. The widerRepeated function converts this data frame into a short and wide one and ignores any data.frame that is not a repeated_instrument (this allows for post processing pipelining). Instead of multiple rows for one record, this function will transform all the data for each record into one row using reshape. The function accepts a single form and returns the reshaped data frame.

```
Records <- exportRecordsTyped(test_conn,
                              forms = "repeating_instrument")

Records
```

```
##   record_id redcap_event_name redcap_repeat_instrument redcap_repeat_instance
## 2         1     event_1_arm_1     repeating_instrument                      1
## 3         1     event_1_arm_1     repeating_instrument                      2
##   repeat_question_1     repeat_datetime repeating_instrument_complete
## 2              Good 2023-04-20 13:34:00                    Incomplete
## 3               Bad 2023-04-20 13:35:00                      Complete
```

```
widerRepeated(Records, test_conn)
```

```
##   record_id redcap_event_name redcap_repeat_instrument repeat_question_1.1
## 2         1     event_1_arm_1     repeating_instrument                Good
##     repeat_datetime.1 repeating_instrument_complete.1 repeat_question_1.2
## 2 2023-04-20 13:34:00                      Incomplete                 Bad
##     repeat_datetime.2 repeating_instrument_complete.2
## 2 2023-04-20 13:35:00                        Complete
```

The widerRepeated function will not widen forms passed into it without repeating instruments. It will return these records in the original format. This function expects that all values in the redcap_repeat_instrument column are the same. If this is not the case it will return and error.

castForImport

While it is true that importRecords will convert most data types into a format that can be imported, it has proven to be overly rigid with blind spots that cannot be easily overcome. In order to provide better support for importing data, we have provided the stand-alone function castForImport.

castForImport follows the same strategy of validation and casting used in exportRecordsTyped. It returns a data frame where the fields are cast in a format (usually character) that can be passed into importRecords.

```
Records <- exportRecordsTyped(test_conn,
                              records = 10:29,
                              forms = "data_import_export")

Records$checkbox_test___x
```

```
##  [1] Checked   Unchecked Unchecked Unchecked Unchecked Unchecked Checked
##  [8] Unchecked Checked   Checked   Checked   Unchecked Unchecked Checked
## [15] Checked   Unchecked Checked   Unchecked Unchecked Checked
## attr(,"label")
## [1] Checkbox Example
## Levels: Unchecked Checked
```

```
Records$dropdown_test
```

```
##  [1] Lavender Blue     Lavender Green    Lavender Blue     Blue     Lavender
##  [9] Green    Green    Green    Blue     Blue     Green    Lavender Lavender
## [17] Lavender Lavender Green    Blue    
## attr(,"label")
## [1] Drop Down Field
## Levels: Green Blue Lavender
```

The default settings of `castForImport` are arranged so that most cases of data will be recast for import as desired.

```
ForImport <- castForImport(Records, 
                           test_conn)

ForImport$checkbox_test___x
```

```
##  [1] 1 0 0 0 0 0 1 0 1 1 1 0 0 1 1 0 1 0 0 1
```

```
ForImport$dropdown_test
```

```
##  [1] 3 2 3 1 3 2 2 3 1 1 1 2 2 1 3 3 3 3 1 2
```

The actual default casting list for `castForImport` is

```
list(
  date_                    = as.character, 
  datetime_                = as.character,
  datetime_seconds_        = as.character,
  time_mm_ss               = castTimeMMSS,
  time_hh_mm_ss            = as.character,
  time                     = castTimeHHMM,
  alpha_only               = as.character,
  float                    = as.character,
  number                   = as.character,
  number_1dp               = castDpCharacter(1, dec_symbol = "."),
  number_1dp_comma_decimal = castDpCharacter(1),
  number_2dp               = castDpCharacter(2, dec_symbol = "."),
  number_2dp_comma_decimal = castDpCharacter(2),
  calc                     = as.character,
  int                      = function(x, ...) as.character(as.integer(x)),
  integer                  = function(x, ...) as.character(as.integer(x)),
  yesno                    = castRaw,
  truefalse                = function(x, ...) (x=='1' | tolower(x) =='true') + 0L,
  checkbox                 = castRaw,
  form_complete            = castRaw,
  select                   = castRaw,
  radio                    = castRaw,
  dropdown                 = castRaw,
  email                    = as.character,
  phone                    = as.character,
  zipcode                  = as.character,
  slider                   = as.numeric,
  sql                      = NA
)
```

At this time, we have not changed anything within `importRecords`. Doing so would require making a breaking change and we aren't prepared to do that on short notice. However, we are considering this change in the future. For that reason, we advise changing one's processes to utilize `castForImport` prior to importing data.

Customizing Checkbox Casting We have encountered a special case of a checkbox function that `importRecords` is entirely incapable of handling. In this case, the checkbox variable was defined with the options

```
0, 0
1, 1
2, 2
```

In this case, the field `checkbox_example___0` could actually be cast in a way that "0" indicates the checkbox was "checked". This is a scenario that is problematic, as the API would determine that any value of "0" is unchecked.

In order to handle this scenario, we have provided a special casting function, `castCheckForImport`, that allows the user to designate what values are to represent a checked value.

```
Records <- data.frame(checkbox_test___x = c("0", "", "", "0"),
                      checkbox_test___y = c("y", "y", "", "y"))

ForImport <- castForImport(Records,
                           test_conn,
                           fields = "checkbox_test___x",
                           cast = list(checkbox = castCheckForImport(checked = "0")))
ForImport
```

```
##   checkbox_test___x checkbox_test___y
## 1                 1                 y
## 2                 0                 y
## 3                 0
## 4                 1                 y
```

To complete an import in this scenario may require two passes with `castForImport` before calling `importRecords`.

```
ForImport <-
  castForImport(Records,
                test_conn,
                fields = "checkbox_test___x",
                cast = list(checkbox = castCheckForImport(checked = "0"))) |>
  castForImport(test_conn)
```

```r
importRecords(test_conn,
              data = ForImport)
```

Helper Functions

Bulk Records Pull

A helper function is provided that will pull a set of projects and/or forms from projects into memory and apply a post processor set in bulk.

```r
exportBulkRecords(
 rcon  = list(test = test_conn,
              sand = sandbox_conn),
 forms = list(test = c('repeating_instrument', 'branching_logic')),
 envir = globalenv(),
 post  = function(Records, rcon)
         {
           Records               |>
           mChoiceCast(rcon)     |>
           guessDate(rcon)       |>
           widerRepeated(rcon)
         }
)
```

```
## <environment: R_GlobalEnv>
```

```r
test_repeating_instrument
```

```
##   record_id redcap_event_name redcap_repeat_instrument repeat_question_1.1
## 2         1     event_1_arm_1     repeating_instrument                Good
##     repeat_datetime.1 repeating_instrument_complete.1 repeat_question_1.2
## 2 2023-04-20 13:34:00                      Incomplete                 Bad
##     repeat_datetime.2 repeating_instrument_complete.2
## 2 2023-04-20 13:35:00                        Complete
```

```r
head(test_branching_logic)
```

```
##   record_id redcap_event_name redcap_repeat_instrument redcap_repeat_instance
## 1         1     event_1_arm_1                     <NA>                   <NA>
## 2         2     event_1_arm_1                     <NA>                   <NA>
## 3         3     event_1_arm_1                     <NA>                   <NA>
## 4        10     event_1_arm_1                     <NA>                   <NA>
## 5        11     event_1_arm_1                     <NA>                   <NA>
## 6        12     event_1_arm_1                     <NA>                   <NA>
##
## 1
## 2
## 3
## 4                                    Pre-requisite field: None  Target field: no_prereq_number    Des
## 5          Pre-requisite field: None  Target field: no_prereq_number    Desired outcome: With a valu
## 6 Pre-requisite field: None  Target field: no_prereq_checkbox    Desired outcome: with no values in t
##                    prereq_radio prereq_number prereq_date prereq_yesno
## 1                          <NA>            NA        <NA>         <NA>
## 2                          <NA>            NA        <NA>         <NA>
## 3                          <NA>            NA        <NA>         <NA>
## 4 Do not use in branching logic             1  2023-03-08           No
## 5 Do not use in branching logic             1  2023-03-08           No
## 6 Do not use in branching logic             1  2023-03-08           No
##   no_prereq_number one_prereq_non_checkbox one_prereq_checkbox two_prereq_and
## 1               NA                    <NA>                <NA>           <NA>
## 2               NA                    <NA>                <NA>           <NA>
## 3               NA                    <NA>                <NA>           <NA>
## 4               NA                    <NA>                <NA>           <NA>
## 5           123456                    <NA>                <NA>           <NA>
## 6           123456                    <NA>                <NA>           <NA>
##   two_prereq_or two_prereq_and_one_check three_prereq_andor
## 1            NA                     <NA>               <NA>
## 2            NA                     <NA>               <NA>
## 3            NA                     <NA>               <NA>
## 4            NA                     <NA>               <NA>
## 5            NA                     <NA>               <NA>
## 6            NA                     <NA>               <NA>
##   one_prereq_inequality branching_logic_complete               prereq_checkbox
## 1                  <NA>               Incomplete
## 2                  <NA>               Incomplete
## 3                  <NA>               Incomplete
## 4                  <NA>               Incomplete Do not use in branching logic
## 5                  <NA>               Incomplete Do not use in branching logic
## 6                  <NA>               Incomplete Do not use in branching logic
##   no_prereq_checkbox
## 1
## 2
## 3
## 4          Chocolate
## 5
## 6
```

This references the connections we opened in the `unlockREDCap` section at the beginning and provides the names we want for the resulting records. The environment post execution contains the data.frames: `test.repeating_instrument`, `test.branching_logic`, `sand`. Each of these were retrieved, possibly using the forms argument and all were post processed in the same manner as specified by `post`. Any additional arguments are passed on to the `exportRecordsTyped` call.

Branching Logic NA detection

The `missingSummary` function provides a utility to look for missing values within a dataset. The results account for branching logic in the instrument; fields that are missing because branching logic did not expose them do not get counted as missing. One may restrict the summary to fields, forms, and/or records as well.

```
missingSummary(test_conn,
               exportRecordsArgs = list(records = 10:29,
                                        fields = "record_id",
                                         forms = "branching_logic")) |>
  head()
```

```
## Please use exportRecordsTyped instead. exportRecords is DEPRECATED.
```

```
## Registered S3 methods overwritten by 'labelVector':
##   method         from
##   [.labelled     Hmisc
##   print.labelled Hmisc
```

```
##   record_id redcap_event_name redcap_repeat_instrument redcap_repeat_instance
## 1        10     event_1_arm_1                       NA                     NA
## 2        11     event_1_arm_1                       NA                     NA
## 3        12     event_1_arm_1                       NA                     NA
## 4        13     event_1_arm_1                       NA                     NA
## 5        14     event_1_arm_1                       NA                     NA
## 6        15     event_1_arm_1                       NA                     NA
##   n_missing                 missing
## 1         1        no_prereq_number
## 2         0
## 3         0
## 4         0
## 5         1 one_prereq_non_checkbox
## 6         0
```

One limitation of `missingSummary`, however, is that the summary operates exclusively within each row of the data. Thus, if one's branching logic utilizes values from previous events, this summary will not correctly identify non-missing values.

Cornucopia of Functions to explore

The functions offered by `redcapAPI` have expanded significantly in recent versions. The table below names all of the methods provided by the REDCap API and indicates which are supported by `redcapAPI`.

System	Export	Import	Delete	Other Method
Arms	Yes	Yes	Yes	N/A
DAGs	No	No	No	SwitchDag (No) exportDagAssigment (No) importDagAssigment (No)
Events	Yes	Yes	Yes	N/A
Field Names	Yes	N/A	N/A	N/A
Files	Yes	Yes	Yes	N/A
File Repository	Yes	Yes	Yes	createFileRepositoryFolder (Yes) exportFileRepositoryListing (Yes)
Instruments	Yes	N/A	N/A	exportMappings (Yes) importMappings (Yes)
Logging	Yes	N/A	N/A	N/A
Meta Data	Yes	Yes	N/A	N/A
Project Info	Yes	Yes	N/A	createProject (No) exportProjectXML (No)
Records	Yes	Yes	Yes	renameRecord (No)

System	Export	Import	Delete	Other Method
				exportNextRecordName (Yes)
Reports	Yes	N/A	N/A	N/A
Version	Yes	N/A	N/A	N/A
Surveys	N/A	N/A	N/A	exportSurveyLink (No) exportSurveyParticipants (Yes) exportSurveyQueueLink (No) exportSurveyReturnCode (No)
Users	Yes	No	No	N/A
UserRoles	No	No	No	exportUserRoleAssigments (No) importUserRoleAssigments (No)

Reproducing this Vignette

To reproduce this vignette one must build a copy of the `redcapAPI` test database. The first step is creating a new project via the REDCap web interface and generating an API Key. Using this key create a connection as usual, then `restoreProject` will install all forms and data used in the construction of this vignette.

In addition, there are other project helper functions available. `purgeProject` will purge all data and metadata from a REDCap project. Once can create their own archive of a REDCap project using `preserveProject`.

```r
purgeProject(rcon, records = TRUE) # Delete everything in the project

# Find path to project definition in package
load(file.path(path.package('redcapAPI'),
     "extdata",
     "testingBranchingLogic",
     "TestingBranchingLogic.Rdata"))

# Rebuild the Project
restoreProject(TestingBranchingLogic, rcon = rcon)
```

Custom API Calls

`redcapAPI` calls are very specific in how they access the REDCap API and leave very little flexibility to the user in the choice of arguments to pass to the API. This lack of flexibility is deliberate, as it helps limit the potential for errors and frustration to the typical user. Advanced users may, at times, find our decisions limiting. Or there may be a need to use an API method that `redcapAPI` does not yet offer.

Users wishing to customize their API calls may use `makeApiCall`. This is a flexible function that utilizes the `redcapConnection` object, permitting customized calls within the same code style of the rest of the package. It has an added benefit of the retry strategy for API call failures as mentioned in the '*Connection*' section above.

For example, using `makeApiCall`, we can get User-Roles, even though `redcapAPI` does not have a dedicated method to retrieve those.

```r
response <- makeApiCall(sandbox_conn,
                        body = list(content = 'userRole',
                                    format = 'csv',
                                    returnFormat = 'csv'))
read.csv(text = as.character(response),
         na.strings = '')
```

```
##   unique_role_name          role_label design alerts user_rights
## 1     U-646TTM48XJ redcapApiTestEditor      1      0           1
##   data_access_groups reports stats_and_charts manage_survey_participants
## 1                  0       1                1                          1
##   calendar data_import_tool data_comparison_tool logging file_repository
## 1        1                1                    0       0               1
##   data_quality_create data_quality_execute api_export api_import mobile_app
## 1                   0                    0          0          0          0
##   mobile_app_download_data record_create record_rename record_delete
## 1                        0             1             0             0
##   lock_records_customization lock_records lock_records_all_forms
## 1                          0            0                      0
##                                                                                  forms
## 1 date_time_tests:1,multiple_choice_tests:1,numeric_import_tests:1,import_meta_data:1
##
## 1 date_time_tests:1,multiple_choice_tests:1,numeric_import_tests:1,import_meta_data:1,date_time_tests
```

When constructing custom calls, the user should read the REDCap API documentation carefully. Any time a parameter calls for an *array* of values, the values from R must be passed in a very specific format–even if there is only a single value to pass. `redcapAPI` uses a function to format R vectors into the proper format to be accepted by the API. The user should also adopt this strategy in order to make custom API calls.

The `vectorToApiBodyList` function returns a list that can be appended to the list passed to the `body` argument.

```
vectorToApiBodyList(c(1, 3, 4), "arms")
```

```
## $`arms[1]`
## [1] 1
##
## $`arms[2]`
## [1] 3
##
## $`arms[3]`
## [1] 4
```

A call to export only a selection of arms from a project would look like this:

```
makeApiCall(rcon,
            body = c(list(content = 'arm',
                          format = 'csv',
                          returnFormat = 'csv'),
                     vectorToApiBodyList(c(1, 3, 4), "arms")))
```

Thanks

Thanks to all those that have made this effort possible for `redcapAPI` as an R package, and striven to make it better.

- Will Beasley
- Cole Beck
- Lynne Berry

- Caroline Birdrow
- Thomas Dupont
- Shawn Garbett
- Will Gray
- Frank Harrell
- **Jeffrey Horner**
- Omair Khan
- Stephen Lane
- Marcus Lehr
- Dandan Liu
- **Benjamin Nutter**
- Savannah Obregon
- Jeremy Stephens
- The Biostatistics department at Vanderbilt University Medical Center
- The R-Project and CRAN team
- The REDCap Team at Vanderbilt University Medical Center
- National Institutes of Health (NIH/NCATS UL1 TR000445)