Casting Data with `exportRecordsTyped`

and Frequently Asked Questions

2023-11-02

Contents

Introduction

The addition of `exportRecordsTyped` opened a great deal of flexibility and potential for customization when exporting data from REDCap and preparing them for analysis. The tasks of preparing data are broadly categorized into three phases

1. Missing Value Detection
2. Field Validation
3. Casting Data

This document will focus on casting data and customizing casting to fit the user's preferences.

```
library(redcapAPI)
url <- "https://redcap.vanderbilt.edu/api/" # Our institutions REDCap instance

unlockREDCap(c(rcon = "Sandbox"),
             envir = .GlobalEnv,
             keyring = "API_KEYs",
             url = url)
```

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

Casting Data

The default casting functions were chosen with consideration for what is believed to be the most frequently desired results (the default casting list is shown in the appendix). It is inevitable that the circumstances of a particular project will necessitate customization. Furthermore, the decisions regarding default casting are inherently opinionated, and some users will prefer different castings. This section will discuss how to customize casting for field types as well as how to customize the casting of a single field.

A full listing of the casting functions provided by `redcapAPI` are listed in the "Value" section of `?fieldValidationAndCasting`.

Customizing a Field Type Casting

Using the `cast` argument, the user may issue alternative casting instructions for any of the supported field types (listed in the appendix). In the following call, any fields having the type `date_` will be cast using the `as.Date()` function instead of `as.POSIXct()`. Meanwhile, all other field types will be cast using the default casting list.

```
########################################################################
# Demonstrate default casting behavior
Rec <- exportRecordsTyped(rcon,
                          fields = c("date_example"))
class(Rec$date_example)
```

```
## [1] "POSIXct" "POSIXt"
```

```
########################################################################
# Demonstrate casting with as.Date()
Rec <- exportRecordsTyped(rcon,
                          fields = c("date_example"),
                          cast = list(date_ = as.Date))
class(Rec$date_example)
```

```
## [1] "Date"
```

Radio button and drop down fields are field types where users frequently need a value different than the default. In most cases, the user desires that these fields be cast to their coded values instead of the labeled values. Compare the results of these three commands:

```
########################################################################
# Returns a factor with levels "Balalaika", "Ukulele", "Banjo", "Guitar"
Rec <- exportRecordsTyped(rcon,
                          fields = c("radio_example"))
Rec$radio_example
```

```
## [1] Balalaika Ukulele   Banjo
## attr(,"label")
## [1] Radio button example
## Levels: Balalaika Ukulele Banjo Guitar
```

```
class(Rec$radio_example)
```

```
## [1] "factor"
```

```
########################################################################
# Returns a factor with levels "3", "4", "5", "6"
Rec <- exportRecordsTyped(rcon,
                          fields = "radio_example",
```

```
                          cast = list(radio = castCode))
Rec$radio_example
```

```
## [1] 3 4 5
## attr(,"label")
## [1] Radio button example
## Levels: 3 4 5 6
```

```
class(Rec$radio_example)
```

```
## [1] "factor"
```

```
########################################################################
# Returns a character value of the labeled values
Rec <- exportRecordsTyped(rcon,
                          fields = "radio_example",
                          cast = list(radio = castLabelCharacter))
Rec$radio_example
```

```
## [1] "Balalaika" "Ukulele"   "Banjo"
## attr(,"label")
## [1] "Radio button example"
```

```
class(Rec$radio_example)
```

```
## [1] "character"
```

Customizing Field Casting with User-Made Functions

It is also permissible to use user-made functions in casting. Consider the scenario where it is necessary to multiply a numeric field by 3 when performing the export. This may be accomplished by first defining a function then passing it to the override for the `number` field type.

Custom functions should have the arguments `x`, `field_name`, and `coding`. These arguments are necessary, even if they will not be used by the function.

```
multiply3 <- function(x, field_name, coding) as.numeric(x) * 3

########################################################################
# Return the actual values from the project
Rec <- exportRecordsTyped(rcon,
                          fields = c("radio_example",
                                     "number_example"))
Rec
```

```
##   record_id      redcap_event_name number_example radio_example
## 1         1 Event 1 (Arm 1: Arm 1)        72.0404     Balalaika
## 2         2 Event 1 (Arm 1: Arm 1)        18.9252       Ukulele
## 3         3 Event 1 (Arm 1: Arm 1)        17.8558         Banjo
```

```
########################################################################
# Return the values with the custom casting function
Rec <- exportRecordsTyped(rcon,
                          fields = c("radio_example",
                                     "number_example"),
                          cast = list(number = multiply3))
Rec
```

```
##   record_id      redcap_event_name number_example radio_example
```

```
## 1         1 Event 1 (Arm 1: Arm 1)       216.1212     Balalaika
## 2         2 Event 1 (Arm 1: Arm 1)        56.7756       Ukulele
## 3         3 Event 1 (Arm 1: Arm 1)        53.5674         Banjo
```

It should be noted that applying a custom function in this way would impact all of the fields of type "number". It would be rare that such an outcome is desirable. These custom functions can also be written in a manner that impacts only one specific field.

Customizing a Casting for a Single Field

User-written functions used in casting overrides must contain the arguments `x`, `field_name`, and `coding`, even if these arguments are not intended to be used by the function. Their inclusion, however, makes it possible to write casting overrides that target only a specific field. In this example, a function is written that rounds `number_example` to two decimal places, but other "number" fields are cast using the default function. By adding an `if` statement, a test can be performed against the field name and modifications can be applied only to the targeted field.

```
round2_one_field <- function(x, field_name, coding){
  x <- as.numeric(x)
  if (field_name == "number_example") round(x, 2) # round to two decimal places
  else x                                          # return other fields unaltered
}

########################################################################
# Default casting behavior
Rec <- exportRecordsTyped(rcon,
                          fields = c("number_example",
                                     "number_example_duplicate"),
                          drop_fields = "redcap_event_name")
Rec
```

```
##   record_id      redcap_event_name number_example number_example_duplicate
## 1         1 Event 1 (Arm 1: Arm 1)        72.0404                  72.0404
## 2         2 Event 1 (Arm 1: Arm 1)        18.9252                  18.9252
## 3         3 Event 1 (Arm 1: Arm 1)        17.8558                  17.8558
```

```
########################################################################
# Use the user-defined function for casting
Rec <- exportRecordsTyped(rcon,
                          fields = c("number_example",
                                     "number_example_duplicate"),
                          drop_fields = "redcap_event_name",
                          cast = list(number = round2_one_field))
Rec
```

```
##   record_id      redcap_event_name number_example number_example_duplicate
## 1         1 Event 1 (Arm 1: Arm 1)          72.04                  72.0404
## 2         2 Event 1 (Arm 1: Arm 1)          18.93                  18.9252
## 3         3 Event 1 (Arm 1: Arm 1)          17.86                  17.8558
```

Radio buttons and drop down fields are, again, field types where such customization is frequently needed. Consider the case of a radio button field where the coded values have special meaning. However, other radio button fields in the project are desired to return the labeled values for categorical analysis. A user-defined function can be written to accommodate this scenario.

In this example, the `radio_example` labels identify an stringed instrument, and the coding indicates the number of strings on that instrument. The user is able to single out `radio_example` to return numeric values

in the following manner:

```
special_cast_radio <- function(x, field_name, coding){
  if (field_name %in% "radio_example"){
    as.numeric(x)                    # Cast target field as numeric
  } else {
    castLabel(x, field_name, coding) # still uses the default for
                                     # the non-targeted fields
  }
}

######################################################################
# Using the default casting
Rec <- exportRecordsTyped(rcon,
                          fields = c("radio_example",
                                     "radio_example_duplicate"),
                          drop_fields = "redcap_event_name")
Rec
```

```
##   record_id      redcap_event_name radio_example radio_example_duplicate
## 1         1 Event 1 (Arm 1: Arm 1)     Balalaika               Balalaika
## 2         2 Event 1 (Arm 1: Arm 1)       Ukulele                 Ukulele
## 3         3 Event 1 (Arm 1: Arm 1)         Banjo                   Banjo
```

```
######################################################################
# Use the user-defined function to change casting of one field
Rec <- exportRecordsTyped(rcon,
                          fields = c("radio_example",
                                     "radio_example_duplicate"),
                          drop_fields = "redcap_event_name",
                          cast = list(radio = special_cast_radio))
Rec
```

```
##   record_id      redcap_event_name radio_example radio_example_duplicate
## 1         1 Event 1 (Arm 1: Arm 1)             3               Balalaika
## 2         2 Event 1 (Arm 1: Arm 1)             4                 Ukulele
## 3         3 Event 1 (Arm 1: Arm 1)             5                   Banjo
```

Frequently Asked Questions

How do I stop casting fields to factors?

I used to be able to set `factors = FALSE` to prevent categorical values from being returned as factors. How do I do that with `exportRecordsTyped`?

Users may substitute an alternate casting list specification within the call to `exportRecordsTyped`. `redcapAPI` provides two lists for this purpose: `default_cast_character` and `default_cast_no_factor`. These two lists are identical and may be used interchangeably.

```
exportRecordsTyped(rcon,
                   cast = default_cast_character)

exportRecordsTyped(rcon,
                   cast = default_cast_no_factor)
```

Aside from not casting factors, all other settings in this list are identical to the default casting.

How do I control the casting of `redcap_event_name`?

In earlier versions of `redcapAPI`, the `redcap_event_name` field commonly returned the values such as `event_1_arm_1`, `event_2_arm_1`, etc. It now returns "fancy" values. How do I get the original behavior?

The `redcap_event_name` field is one of the fields referred to as a "system" field. These fields are not part of the project's data dictionary, and are automatically returned by the API based on the configuration of the project.

By default, `exportRecordsTyped` returns the "labeled" values of the event names.

```
exportRecordsTyped(rcon,
                   fields = "redcap_event_name",
                   records = 1:3)
```

```
##        redcap_event_name
## 1 Event 1 (Arm 1: Arm 1)
## 2 Event 1 (Arm 1: Arm 1)
## 3 Event 1 (Arm 1: Arm 1)
```

This behavior can be changed using the `system` casting override (this will also affect the casting of other system fields).

```
exportRecordsTyped(rcon,
                   fields = "redcap_event_name",
                   records = 1:3,
                   cast = list(system = castRaw))
```

```
##   redcap_event_name
## 1     event_1_arm_1
## 2     event_1_arm_1
## 3     event_1_arm_1
```

Appendix

Casting Field Types

- `calc`: Calculated fields.
- `checkbox`: Checkbox fields.
- `date_`: Text fields with the "Date" validation type.
- `datetime_`: Text fields with the "Datetime" validation type.
- `datetime_seconds_`: Text fields with the "Datetime with seconds" validation type.
- `dropdown`: Drop down multiple choice fields.
- `float`: Text fields with the "Number" validation type.
- `form_complete`: Fields automatically added by REDCap indicating the completion status of the form.
- `int`: Text fields with the "Integer" validation type. This appears to be a legacy type, and integer appears to be used by more recent version of REDCap.
- `integer`: Text fields with the "Integer" validation type.
- `number`: Text fields with the "Number" validation type.
- `number_1dp`: Text fields with the "number (1 decimal place)" validation type.
- `number_1dp_comma_decimal`: Text fields with the "number (1 decimal place - comma as decimal)" validation type.
- `number_2dp`: Text fields with the "number (2 decimal place)" validation type.
- `number_2dp_comma_decimal`: Text fields with the "number (2 decimal place - comma as decimal)" validation type.
- `radio`: Radio button fields.
- `select`: Possible alias for `dropdown` or `radio`.
- `sql`: Fields that use a SQL query to make a drop down tools from another project.
- `system`: Fields automatically provided by REDCap for the project. These include `redcap_event_name`, `redcap_data_access_group`, `redcap_repeat_instrument`, and `redcap_repeat_instance`.
- `time_mm_ss`: Text fields with the "Time (MM:SS)" validation type.
- `time_hh_mm_ss`: Text fields with the "Time (HH:MM:SS)" validation type.
- `truefalse`: True - False fields.
- `yesno`: Yes - No fields.

Default Casting List

```
.default_cast <- 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="h:m:s")),
  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,
  number_1dp              = as.numeric,
  number_1dp_comma_decimal = castDpNumeric(),
  number_2dp              = as.numeric,
  number_2dp_comma_decimal = castDpNumeric(),
  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,
  system                   = castLabel
)
```