# R + REDCap Example Data Cleaning Workflow

 $Jennifer\ Thompson,\ MPH$   $June\ 4,\ 2018$ 

### Introduction

#### **Summary and Goals**

This document demonstrates the ongoing process of data cleaning used by the Vanderbilt CIBS Center. Most of our data is stored in REDCap databases and is cleaned at multiple points throughout data collection, with the goal of the highest quality data possible in the least amount of time once enrollment is complete. This example will demonstrate the R code we use to accomplish this goal, and briefly describe the rest of our process.

#### Notes

• All code assumes that the user has rights to use the REDCap API for data export, and that a working API token is stored in the .Renviron file in the working directory, in the format

#### RCTOKEN=manylettersandnumbers

For more information on the REDCap API, please see Project Setup -> Other Functionality within an existing REDCap project. For general information on working with the API, the Github wiki of the redcapAPI package has a good overview. (This example includes basic API usage and will not use the package, but if you are interested in using more of the API's functionality, it would be a great one to investigate.)

- The code will use several helper functions, sourced from dataclean\_helpers.R in the same working directory/Github repository. The code will also be copied in an appendix to this document.
- This script intentionally sticks to base R with very few exceptions, to maximize adoption and minimize
  dependencies. Should you care to refactor with the tidyverse or other packages, alternate versions are
  welcome!

### **Motivating Example**

This example uses a sample REDCap database for a three-month longitudinal study of adult patients taking a dietary supplement and measuring creatinine, HDL and LDL cholesterol, and weight over time. (Sample database is adapted with thanks from REDCap's project templates.) The study codebook is available here.

# Contents

| 1 | Introduction Summary and Goals Notes Motivating Example  |                            |
|---|--|----------------------------|
| 2 | Process Overview   | 3                          |
| 3 | Step 1: Use REDCap API to Export Raw Data Requirements:  1. Working API token  2. R's httr package, built for working with APIs  Approaches  Approach 1: Example  Approach 2 | 4<br>4<br>4<br>4<br>5<br>6 |
| 4 | Step 2: Create a data.frame of All Issues Workflow   | <b>9</b><br>9<br>9         |
| 5 | Step 3: Remove Unfixable Queries  Documentation Project Structure  | 11<br>12<br>12             |
| 6 | Step 4: Disseminate Issues to Study Staff  | 14                         |
| 7 | Step 5: Iterate! Repeat This Process, Early and Often  | 15<br>15<br>15             |

# Process Overview

# Step 1: Use REDCap API to Export Raw Data

We use REDCap's API capabilities to export the data automatically every time the script is run, reducing the potential for error and saving time compared to manually exporting every time data is cleaned.

#### Requirements:

#### 1. Working API token

You must have appropriate user rights for your database in order to request an API token. Once you have the correct user rights, log into the REDCap project. On the lefthand side under Applications, you will see a line for API and API Playground. Click here, then on the button titled Generate API token.

Once your token is generated, **never share it with anyone**. It gives you permission and ability to access research data, and should be kept protected at all times. If you share code with other people, one way to do this safely is to store your API token in a hidden .Renviron file in the appropriate working directory, like this:

#### RCTOKEN=manylettersandnumbers

You can then access the token using the function Sys.getenv().

#### 2. R's httr package, built for working with APIs

More information on httr can be found in the documentation and vignettes, linked from CRAN.

### Approaches

There are (at least) two approaches to exporting REDCap data:

- 1. Read in the entire database in a single httr::POST call, then create subsets in R as needed
- 2. Read in specific subsets in separate calls

Approach #1 is fine if your database is not complex or very large, and/or if you are not yet comfortable working with the API. Approach #2 is valuable in more complicated situations.

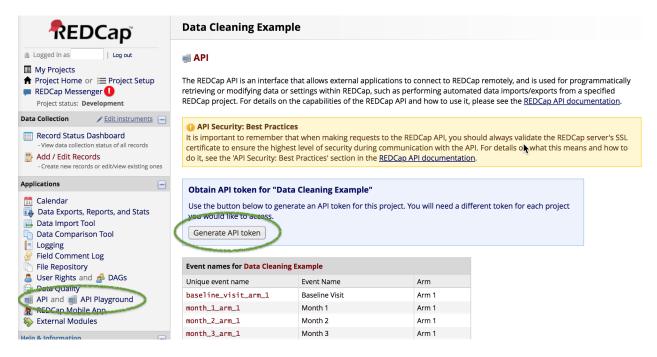


Figure 3.1: Getting an API token

For example, here, our database is longitudinal, and different data is collected at different time points (for example, date of birth is only collected at baseline). If we read in the entire database at once, we will have a lot of missing values (for example, date of birth will be missing at each monthly visit and study completion, unless we proactively subset the data).

I will show an example of Approach 1 for reference, but will primarily use Approach 2. Either way, we want to create the following datasets:

- Baseline data
- · Monthly data
- Study completion data

For both approaches, we need this setup:

```
## Load httr
library(httr)

## Source helper functions (script should be stored in this working directory)
source("dataclean_helpers.R")

## Set URL for REDCap instance (yours may be different)
rc_url <- "https://redcap.vanderbilt.edu/api/"</pre>
```

#### Approach 1: Example

This approach uses the simplest API call, but needs some R work after exporting the data.

```
## Use API + httr::POST to get all data at once
main_post <- httr::POST(
   url = rc_url,
   body = list(</pre>
```

```
token = Sys.getenv("RCTOKEN"),
      ## API token gives you permission to get data
    content = "record",
                               ## export *records*
    format = "csv",
                               ## export as *CSV*
    rawOrLabel = "label", ## export factor *labels* vs numeric codes
    exportCheckboxLabel = TRUE ## export ckbox *labels* vs Unchecked/Checked
  )
)
## main_post has class "response"; read it as a CSV to create a data.frame
main_df <- post_to_df(main_post)</pre>
## Create subsets with data collected at various time points
baseline_df <- subset(main_df, redcap_event_name == "Baseline Visit")</pre>
monthly_df <- subset(main_df, redcap_event_name %in% paste("Month", 1:3))</pre>
completion_df <- subset(main_df, redcap_event_name == "Study Completion")</pre>
```

Note that unless we spend the time to manually subset them, each of those three data.frames will have many columns with blank values. For example, baseline\_df will have a column for compliance, even though compliance is only collected at monthly visits. This is not a major problem if your project is small, but can be a problem if you have a large project.

#### Approach 2

This approach uses three separate httr::POST calls to create separate datasets that are exactly what we need

REDCap's API Playground can be useful in figuring out which options to include in the body argument of httr::POST. (Do note that as of the time of this writing, the example R code from the Playground uses RCurl; httr is currently more commonly used and thus it is easier to find documentation and assistance for it.)

(The rawOrLabel = "label" and exportCheckboxLabel = TRUE elements in the body argument of POST are personal preference. I set these to export labels because I usually find that it is more clear - ie, it is easier to figure out what sex == Male is doing than sex == 1. However, depending on your database and your purposes, you may want to change these to use the raw numeric codes - for example, if you have fields with very long labels.)

Differences from Approach 1 in the body of httr::POST:

- 1. We specify forms, using their raw names (eg, baseline\_data instead of Baseline Data).
- 2. We specify events, again using their raw names (eg, baseline\_visit\_arm\_1 instead of Baseline Visit).
- 3. We always specify study\_id as a field in addition to the other forms. REDCap does not always export the ID by default.

If you are exporting >1 form or event, separate them with commas. You can find raw event names by going to Project Setup -> Define My Events within your REDCap project, and raw form names by looking at the data dictionary. (They can also be exported as project metadata using the API; there is an example in dataclean\_helpers.R.)

**Note:** The function post\_to\_df(), which creates a data.frame from the result of httr::POST, is created in dataclean\_helpers.R.

#### Baseline and Demographic Data

This code chunk exports all the fields collected at the baseline visit (the Demographic and Baseline Visit forms), as well as study ID. It only exports the Baseline Visit event, because all other events would have NA values for these fields. Therefore, each study ID will have at most one record in this dataset.

```
## Data from baseline visit only: Demographics and Baseline Data forms
baseline_post <- httr::POST(</pre>
  url = rc_url,
  body = list(
   token = Sys.getenv("RCTOKEN"), ## API token gives you permission
   content = "record",
format = "csv".
                                   ## export *records*
                                   ## export as *CSV*
   forms = "demographics,baseline_data", ## forms
   fields = c("study id"),
                                          ## additional fields
   events = "baseline visit arm 1",
                                         ## baseline visit event only
   rawOrLabel = "label", ## export factor *labels* vs numeric codes
    exportCheckboxLabel = TRUE ## export ckbox *labels* vs Unchecked/Checked
  )
)
## baseline_post has class "response"; read it as a CSV to create a data.frame
baseline_df <- post_to_df(baseline_post)</pre>
## Double-check if you like! Commented out to save space
## baseline_df
```

Beautiful! Keep going for the monthly and study completion forms.

#### Monthly Visit Data

This code chunk exports all the fields collected at each monthly visit (the Monthly Visit form), as well as study ID. It exports all three monthly visit events; all other events will have NA values for these fields. Each study ID will have up to three records in this dataset.

```
## Data from all monthly visits
monthly_post <- httr::POST(</pre>
  url = rc_url,
  body = list(
   token = Sys.getenv("RCTOKEN"),
                                            ## API token gives you permission
   content = "record",
                                              ## export *records*
   format = "csv",
                                              ## export as *CSV*
   forms = "monthly_data",
                                              ## forms
   fields = c("study_id"),
                                              ## additional fields
   events = paste(sprintf("month_%s_arm_1", 1:3), collapse = ","),
      ## all 3 monthly visit events
   rawOrLabel = "label",
                               ## export factor *labels* vs numeric codes
    exportCheckboxLabel = TRUE ## export ckbox *labels* vs Unchecked/Checked
  )
monthly_df <- post_to_df(monthly_post)</pre>
## Double-check if you like! Commented out to save space
## monthly df
```

#### Study Completion Data

This code chunk exports all the fields collected at study completion, as well as study ID. It exports only the study completion event; therefore, each study ID will have at most one record.

```
## Data from study completion visits
completion_post <- httr::POST(</pre>
 url = rc_url,
  body = list(
   token = Sys.getenv("RCTOKEN"),
                                     ## API token gives you permission
   content = "record",
                                      ## export *records*
   format = "csv",
                                      ## export as *CSV*
   forms = "completion_data",
                                      ## form
   fields = c("study_id"),
                                       ## additional fields
   events = "study_completion_arm_1", ## study completion event
   rawOrLabel = "label", ## export factor *labels* vs numeric codes
   exportCheckboxLabel = TRUE ## export ckbox *labels* vs Unchecked/Checked
 )
completion_df <- post_to_df(completion_post)</pre>
## Double-check if you like! Commented out to save space
## completion_df
```

# Step 2: Create a data.frame of All Issues

This sounds deceptively straightforward, but is the most involved part of this process. Our goal is to create a single data frame of all potential problems in our REDCap project as of the date we run the script, which can then be imported into our documentation database(s) (more on that later) so that the issues can be resolved.

#### Workflow

Typically, I work form by form: here, we'll clean the entire demographic form, then the entire baseline data collection form, then... This keeps the code in manageable chunks and makes it easier to both write initially and debug and maintain as the study progresses.

We often have groups of very similar potential problems: There are many fields that should be present no matter what, for example, and there are several fields which should be between specific limits. We can write custom functions to check these types of issues.

One note: REDCap has many data validation capabilities built in - use them! If your database already checks fields like email for formatting, or gives live warning if you enter unlikely lab values, it will save you and the study staff time and headaches later. The same applies to using branching logic often and well. The added value for this data cleaning script comes from the more complex data checks that are possible here and not within REDCap itself. But the more of REDCap's powerful features you can use from the beginning, the better your data will be throughout the study.

### **Demographics Form**

The Demographics form is one of the two collected at the baseline visit. In Step 1, we exported it as part of baseline\_df; this data.frame has one record per patient. This is important because it will keep us from finding many missing fields where, in actuality, no data should be.

#### 1. Decide what to clean

The first step is to determine what issues need to be looked for. Some of these are straightforward: for example, every patient should have a date of birth. Some are more complex or not as obvious, however. This is where communication and detail are key. Typically, our project manager will spend a good

deal of time creating a list of things that need to be checked, based on the study protocols and goals. It is enormously important for the clinical data team and the statistician/database manager to overcommunicate at this stage!

For our demographics form, we want to check the following:

- Study ID should always be an integer (no letters or special characters)
- These fields should always be present:
  - Date of consent
  - Consent form
  - All contact information
  - Phone
  - Mood
  - Statins
- Postal code should be properly formatted (so should email, but REDCap will validate this field automatically take advantage of REDCap's capabilities when designing your study!)
- Date of birth should be between 18 and 110 years before consent
- If the patient is female, whether she has ever given birth should be entered
- If the patient has given birth, the number of births should be entered
- If no activity questions are marked, study staff should confirm this (it might be OK, but it is unusual and should be checked)
- If the patient is marked as being on statins, at least one specific statin should be checked; if the patient is marked as *not* being on statins, *no* statins should be checked
- Height and weight should both be present and within soft limits set in the database

Our first step is to create a data.frame where one column is an error code, and the second column is the corresponding error message. For example, our error code might be id\_format, and the corresponding error message might be Study ID should be an integer with at most four digits.

```
## -- Create error codes + corresponding messages for all issues *except* -----
## -- fields that are simply missing or should fall within specified limits ----
## Codes: Short, like variable names
## Messages: As clear as possible to the human reader
demog_codes <- data.frame(</pre>
  code = c(
    "id_format", "postcode_format", "dob_limits", "birth_yn", "birth_num",
    "no_activity", "which_statin", "no_statins"
  ),
  msg = c(
    "Study ID should be an integer with at most four digits",
    "Postal code should be formatted properly",
    "Date of birth should be within 18 and 110 years prior to consent",
   "If patient is female, whether she has given birth should be marked",
    "If patient has given birth, number of births should be present",
    "This patient has no activities marked; please confirm or correct",
    "Patient is marked as taking statins, but no specific statins checked",
    "Patient is marked as not taking statins, but at least one statin is checked"
  )
)
```

# Step 3: Remove Unfixable Queries

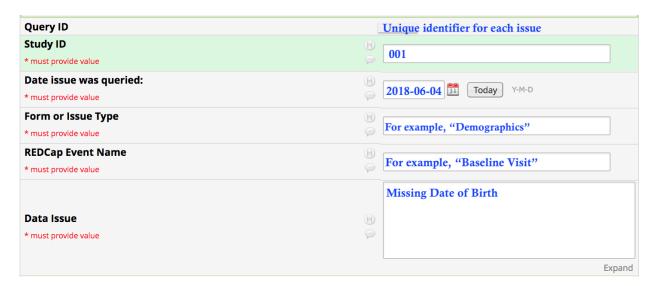
Sometimes, issues with the data are known but cannot be resolved. For example, if a patient was not weighed at a monthly visit, that value can never be entered. Repeating these queries forces study staff to re-investigate problems they have already documented, which is both irritating and a waste of time and effort.

Therefore, an important step in our process is the **documentation** of each issue. This both serves as a record of why the original data changed and enables us to **remove** issues which are unfixable or correct from future data cleans.

#### **Documentation Project Structure**

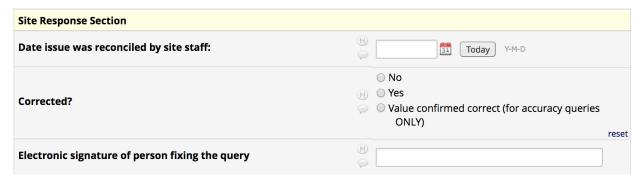
We document all issues and record their resolution in a separate REDCap project, which contains at minimum the following fields (example values in blue):

#### Values created by R script



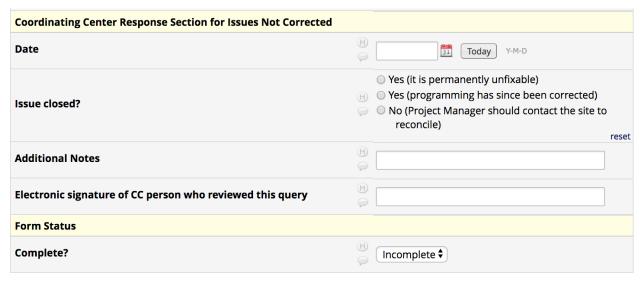
Query ID is typically a combination of patient ID, the date of the data clean, and a number between 1 and the total number of queries found during that data clean.

#### Values filled out by study staff



<sup>&</sup>quot;Accuracy queries" are those that ask staff to confirm extreme values - for example, "Height is less than recommended minimum of xxx cm; please correct or confirm."

#### Values filled out by coordinating center/manager



"Permanently unfixable" queries are those that relate to data that can never be recovered; for example, weight is missing at a monthly assessment because the patient wasn't weighed at all.

Some queries are due to errors in the script or miscommunications between the coordinating center and the statistician/database manager; as the study progresses, you will always find new, fun problems! Thus, we allow for this category of "not fixed."

Depending on the study, it might also be helpful to have additional fields, such as a reason the query was not corrected or whether a Note to File was recorded.

The example documentation codebook for this project can be found here.

#### **Exporting Documentation Data**

We download the data in our documentation database in the same way as the raw data. However, note that since it is a separate REDCap project, you will need a separate API token. Mine is saved in my .Renviron file as the object DOCTOKEN.

```
## Documentation of queries already checked
doc_post <- httr::POST(
    url = rc_url,
    body = list(
        token = Sys.getenv("DOCTOKEN"),
        content = "record",
        format = "csv",
        rawOrLabel = "label",
        exportCheckboxLabel = TRUE
    )
)

## This won't work till I enter some data!

# doc_df <- post_to_df(doc_post)

## Double-check if you like! Commented out to save space
## doc_df</pre>
```

Step 4: Disseminate Issues to Study Staff

## Step 5: Iterate!

I mean iteration in two ways.

#### Repeat This Process, Early and Often

The more frequently you clean your data, the more prepared you will be for things like interim analyses and DSMB or progress reports, and the less time you'll have to spend at the end of the study (when everyone is very excited about getting the final results!). This is especially important for multicenter studies or studies that enroll over years, where sites or staff members may join and leave the group; once a site is closed or a coordinator has retired, it is (understandably!) a challenge to get effort from that site to clean data. Besides, no one wants to get a large, overwhelming number of queries at the end of the study!

How frequently you choose to clean your data will depend on enrollment rates and how many staff members are available to do the cleaning, but we recommend repeating this process as often as is reasonable.

### Always Be Improving

Much like your study protocols, your cleaning script will rarely be perfect on the first try. As the study goes on, you will always find more ways that data can be "wrong," or have more questions that are inspired by unexpected data or discussions with study staff. The vast majority of time spent on this data cleaning script is during its initial development, but there will always be things that need to be changed or added.

This is one reason that **communication** between the statistician/database manager and study staff is hugely important! We work together not only at the beginning of this process, to design the database and come up with lists of data points that need to be checked, but throughout study enrollment and data collection to make sure that protocol changes are adequately accounted for, misunderstandings are cleared up quickly, etc. Typically, as study staff are working through one round of data cleaning, I keep a list of things that need to be investigated - queries they believe shouldn't be there or aren't clear, queries that need to be added due to a protocol change, etc. Then when it's time for the next round of data cleaning, I block off some time to investigate anything that has come up, fix or add what needs attention, and then rerun the next round.

In addition to improving the data itself and the data cleaning script, we use this process as a way to improve our study documents and staff education: If there is a piece of data that is systematically showing up as an issue, perhaps it is due to something that was not clearly addressed at the study startup visit and needs to be revisited, or should be written out fully in an SOP.