Paper TT01

How the R Programming Language Handles the CDISC Library API

Joseph Hinson, Orbis Clinical, Woburn, MA, USA

ABSTRACT

CDISC is currently ushering in a new era of automated standards-based clinical programming where software can directly access the CDISC standards. This is through a massive cloud-based metadata repository called “CDISC Library”, which also contains a REST-based Application Programming Interface (API) and which already has been shown to be accessible by SAS and JAVA programs. Meanwhile, the R programming language is gaining a substantial foothold in big pharma and CROs, due to the open-source language’s extensive suite of advanced statistical, modeling, and graphical tools, plus over 12,000 ancillary software packages. This paper aims to demonstrate that R is also well-suited for doing CDISC Library-based programming. Using just two R packages “httr” and “jsonlite”, the JSON outputs from the CDISC Library API are then parsed with the R function “strsplit“ and regular expression functions “regexec” and “regmatches”, to create a SAS dataset-like “data frame” containing SDTM or ADaM variable metadata.

INTRODUCTION

Until now, clinical programmers have had to rely on eyeballing and copying/pasting from CDISC implementation guides in PDF format. But that is all about to change. In creating CDISC Library, CDISC has adopted a very popular web technology -- REST API -- for making standards electronically accessible by software.

Until much recently, the World Wide Web was just a network for fetching HTML pages. A human user would utilize a browser to request a web page, and a server somewhere would return an HTML page to that browser for rendering. But lately, it has become evident that just as humans can request HTML pages from remote servers, software applications can also request raw data from distant servers using the web. This meant there had to be “translators” to allow different applications to communicate with different server software languages. The Application Programming Interface (API) is such a translator. It has often been likened to the “waiter in a restaurant”. A customer calls the waiter, issues an order for a meal, and the waiter disappears into the kitchen, have the order processed, and eventually returns with the desired meal. Supposing it is a Portuguese restaurant. The waiter can take the order in English and in the kitchen present the request in Portuguese for the non-English speaking kitchen staff to understand. In this metaphor, the “customer” is the browser, the “waiter” is the API, and the “kitchen” is the remote server. It is because of APIs that one can use an ATM of any bank to access one’s own bank, in spite of the fact that different banks might be using different computer systems.

REST API

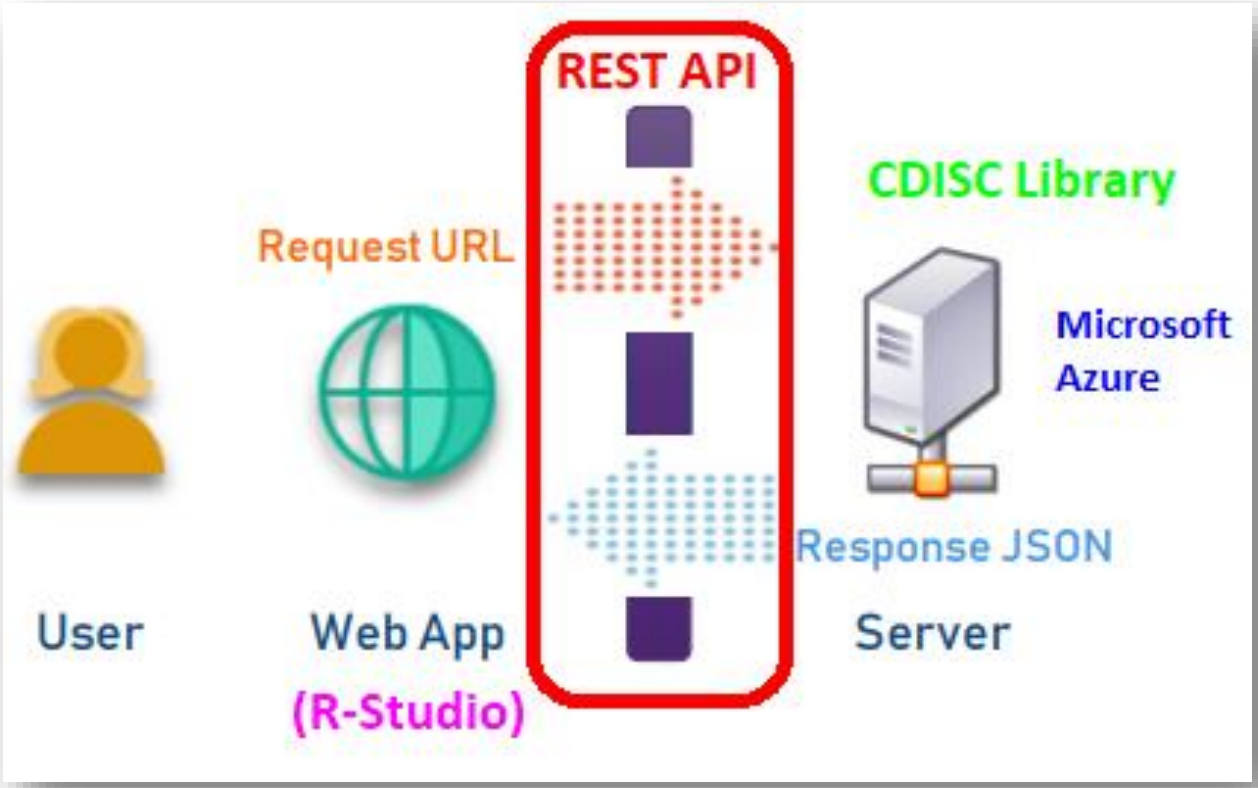
The most popular API to date is what is known as “REST API” (also called “RESTful API”), where “REST” is the abbreviation for REpresentational State Transfer. It is an implementation of a standard data exchange protocol that allows the use of HTTP verbs (e,g, GET, POST) to request data from a server and get a response in a standard format -- typically Javascript Object Notation (JSON) or XML. It is now believed that about 80% of web traffic are REST API requests and responses. Many of the familiar web applications – streaming services, social media, online stores – pull information from the web using REST APIs. Below are two of the popular public REST API services that require no login authorization:

(a) Which astronauts are in space right now
http://api.open-notify.org/astros.json

(b) What is the latest Clinical Trial
https://clinicaltrialsapi.cancer.gov/v1/clinical-trials

CDISC LIBRARY

CDISC Library is a cloud-based metadata repository of CDISC standards residing on a Microsoft Azure server, and containing a REST API. It uses linked data and the REST API to deliver CDISC standards metadata directly to software applications.



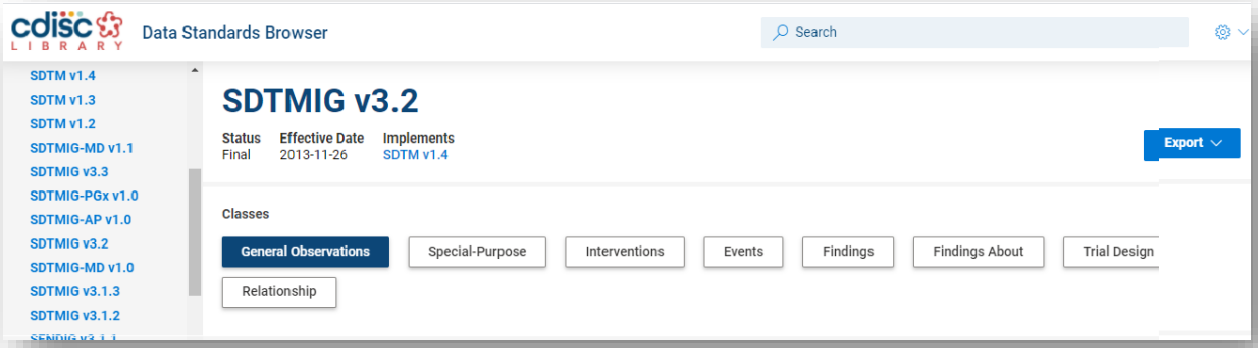
Use of CDISC Library requires a specific account, in addition to the CDISC membership account. The following steps lead to an application for a CDISC Library account:

1. Go to **cdisc.org** and log in with your CDISC membership credentials.
2. Navigate to “Members Only” and select “CDISC Library” to open.
3. Click on “Request an Account”.
4. Click on the box labeled “CDISC Library Account Request Form”.
5. Filling the form requires your organization’s contact information for:
   (a) Account Administrator
   (b) Account User
   (c) Technical Contact

THE DATA STANDARDS BROWSER

CDISC Library also comes with a portal called “Data Standards Browser”, with which one can manually explore and examine the standards. The portal is accessible to those with CDISC Library account at the location:

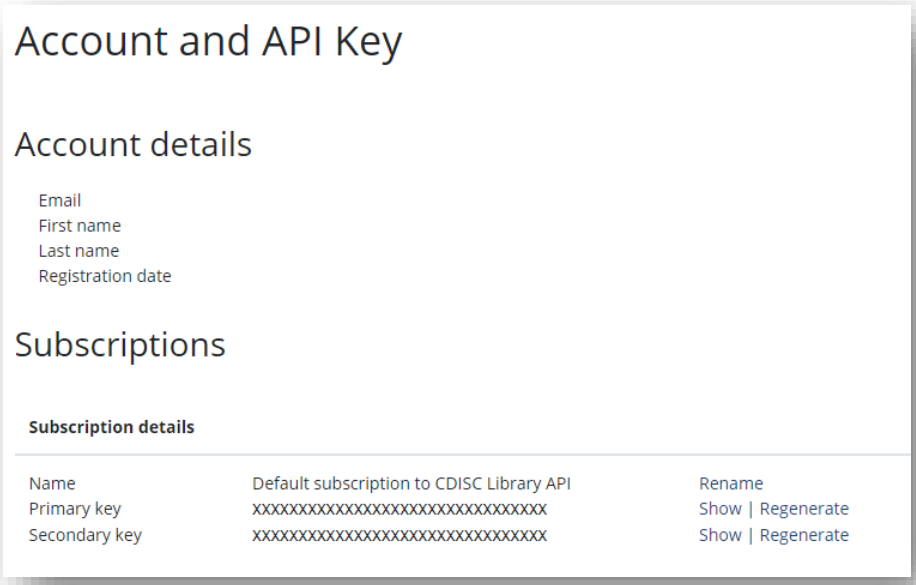
https://library.cdisc.org/browser



THE API MANAGEMENT DEVELOPMENT PORTAL

In order to use code to access the library, one would need an API Key. For CDISC Library account holders, a portal exists called “the API Management Development Portal”, for obtaining one’s API Key. The portal is accessed at:

https://api.developer.library.cdisc.org



THE R PROGRAMMING LANGUAGE

R is a free open-source programming language, possessing an extensive catalog of statistical and graphical methods. Traditionally, the language has been used in Pharma for mainly simulations, modeling, and exploratory work, but its use for analysis is steadily growing in many companies. GlaxoSmithKline has already embarked on a plan to establish R as a primary statistical analysis tool for clinical reporting.

A popular graphical user interface and integrated development environment available for R programming is RStudio. For this paper, RStudio version 1.3.1093 was used to program in R version 4.0.3.

REGULAR EXPRESSIONS IN R

This paper uses regular expressions for parsing the REST API responses. A regular expression is a string that contains special symbols and characters to find and extract information needed from a given piece of text. The simplest form of regular expressions are those that match a single character. Most characters, including all letters and digits, are regular expressions that match themselves. But there are some special characters that have reserved meaning and they are referred to as 'Metacharacters":

METACHARACTERS

(1) **.** = Matches Any Character
(2) **\d** = Digit (0–9)
(3) **\D** = Not a digit (0–9)
(4) **\w** = Word Character (a-z, A-Z, 0–9, _)
(5) **\W** = Not a word character
(6) **\s** = Whitespace (space, tab, newline)
(7) **\S** = Not whitespace (space, tab, newline)
(8) **\b** = Word Boundary
(9) **\B** = Not a word boundary
(10) **^** = Beginning of a string
(11) **$** = End of a String
(12) **[ ]** = matches characters in brackets
(13) **[^]** = matches characters Not in backets
(14) **|** = Either Or
(15) **( )** = Group
(16) ***** = 0 or more
(17) **+** = 1 or more
(18) **?** = Yes or No
(19) **{x}** = Exact Number
(20) **{x, y}** = Range of Numbers (Maximum, Minimum)

To use the metacharacters to represent their original meaning, they are escaped with "\\".

Literal meaning	Escape in R
the period or dot	\\.
the dollar sign	\\$
the asterisk or star	*
the plus sign	\\+
the question mark	\\?
the vertical bar or pipe symbol	\\|
the backslash	\\\\
the caret	\\^
the opening square bracket	\\[
the closing square bracket	\\]
the opening curly bracket	\\{
the closing curly bracket	\\}
the opening round bracket	\\(
the closing round bracket	\\)

POSIX Character Classes:
In R, POSIX character classes are represented with expressions inside double brackets [[ ]]. The following table show how they are used in R:

POSIX Character Classes in R

Class	Description
[[:lower:]]	Lower-case letters
[[:upper:]]	Upper-case letters
[[:alpha:]]	Alphabetic characters ([[:lower:]] and [[:upper:]])
[[:digit:]]	Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
[[:alnum:]]	Alphanumeric characters ([[:alpha:]] and [[:digit:]])
[[:blank:]]	Blank characters: space and tab
[[:cntrl:]]	Control characters
[[:punct:]]	Punctuation characters: ! " # % & ' ( ) * + , - . / : ;
[[:space:]]	Space characters: tab, newline, vertical tab, form feed, carriage return, and space
[[:xdigit:]]	Hexadecimal digits: 0-9 A B C D E F a b c d e f
[[:print:]]	Printable characters ([[:alpha:]], [[:punct:]] and space)
[[:graph:]]	Graphical characters ([[:alpha:]] and [[:punct:]])

PROGRAMMING STRATEGY FOR EXTRACTING METADATA FROM CDISC LIBRARY

The four steps used in R:
1. Make a REST API call to CDISC Library API
2. Split response into string segments with each segment for an SDTM variable
3. Use Regular Expressions to extract variable metadata from each segment
4. Assemble all the extracted info from all the segments into a Data Frame

1. MAKING THE API CALL

The R package ***httr*** is used (***jsonlite*** too can be used).
The ***httr*** functions **GET ()**, **add_headers ()**, and **content ()** are involved
An API-Key supplied by CDISC is required (a CDISC account is mandatory)
The **GET ()** function requires two parameters: Endpoint and Configuration Options
The Endpoint is made up of the Primary URL and the CDISC sections of interest
The Configuration Options are provided with the "**add_headers ()**" function.

The R statement for the API call is therefore written as:

```
Response <- GET (Endpoint, add_headers ("API-Key"=, "accept"=))
```

An example R code for calling CDISC Library API to get AE metadata of SDTMIG 3.2:

```
myAPIkey <- "12345abcde67890fghij"
PrimaryURL<- "https://library.cdisc.org/api/mdr/sdtmig/"
SDTMversion <- "3-2"
DOMAIN <- "AE"
endpoint <- paste0 (PrimaryURL, SDTMversion, "/datasets/", DOMAIN)
Response <- GET (endpoint, add_headers ("api-key" = myAPIkey, "accept" =
"application/json"))
responseBody <- httr::content (response, as = "text", encoding = "UTF-8")
```

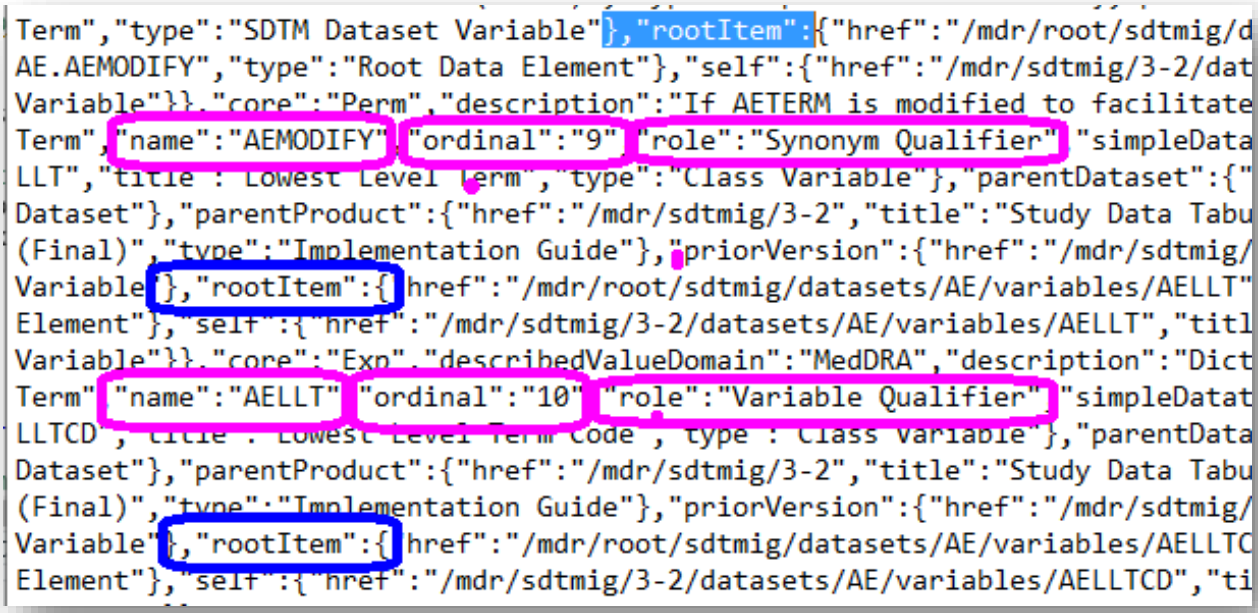
- ***paste0()*** *function concatenates strings without gaps*
- ***httr*** *function "**content ()**" converts the response to a JSON format.*

2. SPLITTING THE RESPONSE TEXT WITH THE STRSPLIT() FUNCTION AND THE "ROOTITEM" STRING PATTERN AS SEPARATOR

The CDISC Library API responds with a single large chunk of text, "responseBody", in JSON format.

The JSON is a popular data-exchange format made up of item-value pairs in double quotes separated by a single colon (as shown in the figure below in purple).

As shown below, the single large text has several instances of the characters **"},"rootItem":{.** These landmarks are partitions for each block of SDTM variable metadata.



USING THE STRSPLIT() FUNCTION:
The function is used for splitting an entire string or a vector of strings into multiple substrings using a literal or a regular expression. This function takes a vector of strings or a string as input and returns a list of all elements that are repeated at a specific character.

The syntax of strsplit() is:

strsplit(x, split, fixed = FALSE)

x – specifies a vector of character strings or a string to be splitted.
split – specifies the delimiter (separator) at which the split will happen.
fixed – if set TRUE will allow text to be split at a fixed width.

For the current processing of respBody:

```
jsplits <- strsplit (paste0 (respBody), split='},"rootItem":{',
fixed=TRUE, useBytes=FALSE)
```

convert back to vector with the UNLIST function:

```
jsx < - unlist (jsplits)
```

obtain number of split segments (determined by number of variables)

```
jsn <- length (jsx)
```

will use for iteration:

```
maxrows <- jsn[[1]]
```

3. EXTRACTING INFORMATION FROM THE SPLIT TEXT FRAGMENTS WITH REGULAR EXPRESSION FUNCTIONS

The R regular expression functions **regexec()** and **regmatches()** are used for data extraction.

(a) **regexec( )** - returns the first location of the matched pattern in a given string.
(Both matched string position and the position of parenthesized matched string are returned).

Syntax : **regexec(pattern, text, ignore.case=FALSE, perl=FALSE, fixed=FALSE, useBytes=FALSE)**

ARGUMENTS:
Pattern – regular expression for pattern matching.
Text – target string
Ignore.case – If FALSE then pattern matching is case insensitive. If TRUE then pattern matching is case sensitive.
Perl – If TRUE then Perl compatible regular expressions are used
UseBytes – If TRUE then matching is done byte-by-byte instead of character-by-character.

(b) **regmatches()** -- uses the locations obtained by regexec() to extract the matched values.
Syntax: **regmatches(text, locations)**

ARGUMENTS::
Text – target string
Locations – results of regexec()

In the current exercise, the metadata of each AE variable would be extracted from each split fragment of respBody. Even though each split segment contains the seven SDTM variable attributes ***description, name, ordinal, label, core, role,*** and ***simpleDatatype,*** just NAME and DESCRIPTION will be extracted into a table for this paper, for the sake of simplicity.

52 Variables are obtained from AE.

The first variable (STUDYID) is processed first into a single row table.
This is because the first segment has JSON header information in addition to the variable attributes and requires additional processing.
The remaining 51 variables are then processed by iteration.

EXTRACTING METADATA FOR THE FIRST VARIABLE:

Obtain locations of column names:

```
jscolsx <-
regexec('[[:print:]]+,\\"(description|describedValueDomain)\\":\\".+\\",[[:print:]]+,\
\"(name)\\":\\".+\\","[[:print:]]*',
jsx[[2]], ignore.case = FALSE, perl = TRUE, useBytes = FALSE)
```

Obtain locations of matched values, as contained in the parenthesized regular expressions:

```
jsrowsx <-
regexec('[[:print:]]+,\\"description|describedValueDomain\\":\\"(.+)\\",[[:print:]]+,\
\"name\\":\\"(.+)\\"[[:print:]]*',
jsx[[2]], ignore.case = FALSE, perl = TRUE, useBytes = FALSE)
```

Obtain matched values with regmatches() using the locations obtained by regexec() :

```
colex <- regmatches(jsx[[2]], jscolsx)
rowex <- regmatches(jsx[[2]], jsrowsx)
```

ASSEMBLE THE EXTRACTED VALUES INTO A SINGLE ROW (FIRST ROW OF TABLE):

- *matrix() function converts the extracted values into a 1x2 matrix*
- *data.frame() converts the matrix to a dataset table*
- *colnames() assigns the extracted column headers to the columns of the data frame*

```
jsmat <- matrix(data=rowex[[1]][2:3], nrow=1, ncol=2, byrow=TRUE)
jsmatrix <- data.frame(jsmat)
colnames(jsmatrix) <- unlist(colex[[1]][2:3])
```

EXTRACTING METADATA FOR THE REMAINING VARIABLES:

Rows 3 to maxrows are processed into a table called "jsframe"

```
for (n in 3:maxrows){
  jsrows <-
regexec('[[:print:]]+,\\"descri\\w{5,14}\\":\\"(.+)\\",[[:print:]]+,\\"name\\":\\"(\\w
{3,8})\\",[[:print:]]+', jsx[[n]], ignore.case = FALSE, perl = TRUE, useBytes = FALSE)

  rowextrax <- regmatches(jsx[[n]],jsrows)
  jsmax <- matrix(data=rowextrax[[1]][2:3], nrow=1, ncol=2, byrow=TRUE)
  jsframe <- data.frame(jsmax)
  colnames(jsframe) <- unlist(colex[[1]][2:3])if mix(var1, var2) > 0 then do;
```

The assembled table "jsframe" is appended to the previously created single-row table: "jsmatrix"

```
  jsmatrix <- rbind.data.frame(jsmatrix,jsframe)
}
```

4. THE ASSEMBLED INFORMATION IN A DATA FRAME

The 52 AE variable names and descriptions are displayed in a dataframe (first 29 shown in the table):

	description	name
1	Unique identifier for a study.	STUDYID
2	Two-character abbreviation for the domain.	DOMAIN
3	Identifier used to uniquely identify a subject across all studies for all applications or submissions involving the product.	USUBJID
4	Sequence Number given to ensure uniqueness of subject records within a domain. May be any valid number.	AESEQ
5	Used to tie together a block of related records in a single domain for a subject.	AEGRPID
6	Internal or external identifier such as a serial number on an SAE reporting form.	AEREFID
7	Sponsor-defined identifier. It may be pre-printed on the CRF as an explicit line identifier or defined in the sponsor's operational database. Example: Line number on an Adverse Events page.	AESPID
8	Verbatim name of the event.	AETERM
9	If AETERM is modified to facilitate coding, then AEMODIFY will contain the modified text.	AEMODIFY
10	Dictionary-derived text description of the Lowest Level Term.	AELLT
11	Dictionary-derived code for the Lowest Level Term.	AELLTCD
12	Dictionary-derived text description of AETERM or AEMODIFY. Equivalent to the Preferred Term (PT in MedDRA). The sponsor is expected to provide the dictionary name and version used to map the terms utilizing the ...	AEDECOD
13	Dictionary-derived code for the Preferred Term.	AEPTCD
14	Dictionary-derived text description of the High Level Term for the primary System Organ Class.	AEHLT
15	Dictionary-derived code for the High Level Term for the primary System Organ Class.	AEHLTCD
16	Dictionary-derived text description of the High Level Group Term for the primary System Organ Class.	AEHLGT
17	Dictionary-derived code for the High Level Group Term for the primary System Organ Class.	AEHLGTCD
18	Used to define a category of related records. Example: BLEEDING, NEUROPSYCHIATRIC.	AECAT
19	A further categorization of adverse event. Example: NEUROLOGIC.	AESCAT
20	A value of \"Y\" indicates that this adverse event was pre-specified on the CRF. Values are null for spontaneously reported events (i.e., those collected as free-text verbatim terms)	AEPRESP
21	Dictionary derived. Body system or organ class used by the sponsor from the coding dictionary (e.g., MedDRA). When using a multi-axial dictionary such as MedDRA, this should contain the SOC used for the sponsor's...	AEBODSYS
22	Dictionary derived. Code for the body system or organ class used by the sponsor. When using a multi-axial dictionary such as MedDRA, this should contain the SOC used for the sponsor's analyses and summary tables,...	AEBDSYCD
23	Dictionary-derived text description of the primary System Organ Class. Will be the same as AEBODSYS if the primary SOC was used for analysis.	AESOC
24	Dictionary-derived code for the primary System Organ Class. Will be the same as AEBDSYCD if the primary SOC was used for analysis.	AESOCCD
25	Describes anatomical location relevant for the event (e.g., ARM for skin rash).	AELOC
26	The severity or intensity of the event. Examples: MILD, MODERATE, SEVERE.	AESEV
27	Is this a serious event?	AESER
28	Describes changes to the study treatment as a result of the event. AEACN is specifically for the relationship to study treatment. AEACNOTH is for actions unrelated to dose adjustments of study treatment. Examples of ...	AEACN
29	Describes other actions taken as a result of the event that are unrelated to dose adjustments of study treatment. Usually reported as free text. Example: \"TREATMENT UNBLINDED. PRIMARY CARE PHYSICIAN NOTIFIED...	AEACNOTH

SUMMARY AND CONCLUSION

The R programming language has functions that allow making REST API calls to CDISC Library, and permit the parsing of the JSON response, leading to the creation of a data frame.
The key steps in the process can be summarized as follows:

a. To make the API call, use the R functions of the ***httr*** package: **GET()**, **add_headers()**, and **content()**.
b. To split the resulting JSON response, use **strsplit()** with the characters },"rootItem":{ as a separator.
c. To extract metadata from the split segments, use the regular expression functions **regexec()** and **regmatches()**.
d. To assemble all the extracted metadata into a data frame, use **matrix()**, **data.frame()**, and **rbind.data.frame()**.

With such R functions, the creation of an SDTM or ADaM specifications workbook can be fully automated, thereby preventing typographical errors due to eyeballing and cutting/pasting from the SDTM or ADaM Implementation Guide.

RECOMMENDED READING

Sally Cassells, "How Will You Use CDISC Library?"
PHUSE EU Connect 2019, Paper SI01.
https://www.lexjansen.com/phuse/2019/si/SI01.pdf

Jozef Aerts, "Implementing the CDISC Library API in software applications:first experiences"
PHUSE EU Connect 2019, Paper SI02.
https://www.lexjansen.com/phuse/2019/si/SI02.pdf

Roman Radelicki, “CDISC Library try-out: from implementation to evaluation of the API”
PHUSE US Connect 2020, Paper SD05
https://www.lexjansen.com/phuse-us/2020/sd/SD05_ppt.pdf

CONTACT INFORMATION

(In case a reader wants to get in touch with you, please put your contact information at the end of the paper.)
Your comments and questions are valued and encouraged. Contact the author at:

Joseph Hinson, PhD
Orbis Clinical
joehinson@outlook.com

APPENDIX: Full Code

```
library (httr)

myAPIkey<-"(enter your API key here)"
endpoint<-"https://library.cdisc.org/api/mdr/products/Terminology"
SDTMversion="3-2"
DOMAIN="AE"
endpoint2<-paste0("https://library.cdisc.org/api/mdr/sdtmig/",
SDTMversion, "/datasets/", DOMAIN)
resp<-GET(endpoint2, add_headers("api-key" = myAPIkey, "accept" =
"application/json"))
respBody <- httr::content(resp, as = "text", encoding = "UTF-8")

jsplits <- strsplit(paste0(respBody), split='},"rootItem":{', fixed=TRUE,
useBytes=FALSE)

jsx<-unlist(jsplits)
jsn<-length(jsx)

typeof(jsx)
length(jsx)

maxrows<-jsn[[1]]

jsrowsx<-
regexec('[[:print:]]+,\\"descri\\w{5,14}\\":\\"(.+)\\",[[:print:]]+,\\"nam
e\\":\\"(\\w{3,8})\\",[[:print:]]+', jsx[[2]], ignore.case = FALSE, perl =
TRUE, useBytes = FALSE)
jscolsx<-
regexec('[[:print:]]+,\\"(description|describedValueDomain)\\":\\".+\\",[[
:print:]]+,\\"(name)\\":\\".+\\","[[:print:]]*', jsx[[2]], ignore.case =
FALSE, perl = TRUE, useBytes = FALSE)

rowex<-regmatches(jsx[[2]],jsrowsx)
colex<-regmatches(jsx[[2]],jscolsx)

jsmat<-matrix(data=rowex[[1]][2:3], nrow=1, ncol=2, byrow=TRUE)
```

```
jsmatrix<-data.frame(jsmat)
colnames(jsmatrix)<-unlist(colex[[1]][2:3])

for (n in 3:maxrows){

  jsrows<-
regexec('[[:print:]]+,\\"descri\\w{5,14}\\":\\"(.+)\\",[[:print:]]+,\\"nam
e\\":\\"(\\w{3,8})\\",[[:print:]]+', jsx[[n]], ignore.case = FALSE, perl =
TRUE, useBytes = FALSE)

  rowextrax<-regmatches(jsx[[n]],jsrows)

  jsmax<-matrix(data=rowextrax[[1]][2:3], nrow=1, ncol=2, byrow=TRUE)
  jsframe<-data.frame(jsmax)
  colnames(jsframe)<-unlist(colex[[1]][2:3])
  jsmatrix<-rbind.data.frame(jsmatrix,jsframe)

}

#-----------------(run this on command prompt)--------------------------

View(jsframe)
```