PharmaSUG China 2021 - Paper SR-031

JSON Data in R

Peng Li, Sanofi

ABSTRACT

This paper aims to explore the consistent mapping between JSON data and R objects, and how do certain special cases of JSON structures defined by the JSON packages in R map to the simpler R types. We will demonstrate the mapping rules from JSON to R and R to JSON. The jsonlite R package is used throughout the paper as a reference implementation.

INTRODUCTION

JavaScript Object Notation (JSON) is a text format for the serialization of structured data. Its design is simple and concise, the syntax is easy for humans to read and write, easy for machines to parse and generate. JSON has become hugely popular on the internet as a general-purpose data interchange format, and high-quality parsing libraries are available for almost any programming language.

The JSON format specifies 4 primitive types (string, number, boolean, null) and two universal structures (object, array). Both these structures are heterogeneous. Therefore, the native R realization of these structures is a named list for JSON objects and an unnamed list for JSON arrays. In practice, however, vectors, matrices, or data frames are widely used as a more efficient type of data storage and manipulation in R than lists. To give these data structures a JSON representation, some R packages, such as jsonlite, define certain special cases of JSON structures that get parsed into other, more specific R types.

NATIVE MAPPING RULE

The fromJSON and toJSON functions in the jsonlite package implement the conversion between R objects and JSON based on class method dispatch.

```
jsonlite::fromJSON("[1, 2, 3]")
## [1] 1 2 3

jsonlite::toJSON(c(1, 2, 3))
## [1,2,3]
```

As mentioned above, the mapping rule of native R is a named list for JSON objects and an unnamed list for JSON arrays.

```
jsonlite::toJSON(list(data = c(1, 2, 3)))
## {"data":[1,2,3]}

jsonlite::toJSON(list(c(1, 2, 3)))
## [[1,2,3]]
```

fromJSON is not a perfect inverse function of toJSON because there is no guaranteed one-to-one correspondence between R objects and JSON. The JSON representation of an empty vector, empty list, or empty data frame are all the same empty array, and a call from fromJSON (toJSON (Object)) to each of them will return the same empty list object. Therefore, the user should be fully aware of the data structure during the data transformation process, even if it is dynamic.

```
toJSON(vector()); toJSON(list()); toJSON(data.frame())
## []; ## []; ## []

fromJSON(toJSON(vector()));
fromJSON(toJSON(list()));
fromJSON(toJSON(data.frame()))
## list(); ## list(); ## list()
```

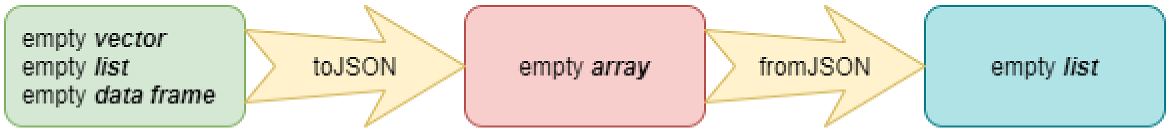

Figure 1. Examples of the Limitations of fromJSON and toJSON

CONVERTING BETWEEN JSON AND R CLASSES

This section lists examples of how the common R classes are represented in JSON. If an R object has multiple class values, R uses the first occurring class which has a toJSON method.

VECTORS

As a common data type in R, vectors map to JSON arrays.

```
jsonlite::toJSON(c(1, 2, 3))
## [1,2,3]
```

Missing values

jsonlite uses the following default mapping rules when there are missing values in the vector.

- Missing values in non-numeric vectors (logical, character) are encoded as null.
- Missing values in numeric vectors (double, integer, complex) are encoded as strings.

```
jsonlite::toJSON(c(TRUE, FALSE, NA))
## [true,false,null]

jsonlite::toJSON(c("foo", "bar", "NA", NA))
## ["foo","bar","NA",null]

jsonlite::toJSON(c(1, 2, NA, NaN))
## [1,2,"NA","NaN"]
```

Special vector types: dates, times, factor, complex

jsonlite uses as.character to coerce a vector of Date, POSIXt, factors, and complex types to a string type. When parsing such JSON strings, these values will appear as character vectors. To obtain the original types, the user needs to manually coerce them back to the desired type using the corresponding “as” function, e.g. as.POSIXct, as.Date, as.factor or as.complex.

```
jsonlite::toJSON(Sys.time() + 1:3)
## ["2021-07-29 14:20:13","2021-07-29 14:20:14","2021-07-29 14:20:15"]

jsonlite::toJSON(as.Date(Sys.time()) + 1:3)
## ["2021-07-30","2021-07-31","2021-08-01"]

jsonlite::toJSON(factor(c("foo", "bar", "foo")))
## ["foo","bar","foo"]

jsonlite::toJSON(complex(real = runif(3), imaginary = rnorm(3)))
## ["0.8174+0.4338i","0.277+2.018i","0.243-1.38i"]
```

Special cases: vectors of length 0 or 1

Two edge cases deserve special attention: vectors of length 0 and vectors of length 1. In jsonlite, these are encoded respectively as an empty array, and an array of length 1.

```
jsonlite::toJSON(vector())
## []

jsonlite::toJSON(1)
## [1]
```

MATRICES

In jsonlite, a matrix maps to an array of equal-length subarrays. Note that even though R stores matrices in column-major order, jsonlite encodes matrices in row-major order. The default JSON map for the matrix does not contain "dimnames" information. When row or column names of a matrix seem to contain vital information, we might want to transform the data into a more appropriate structure.

```
jsonlite::toJSON(matrix(1:12, nrow = 3, ncol = 4))
## [[1,4,7,10],[2,5,8,11],[3,6,9,12]]
```

LISTS

Two types of lists are distinguished: named lists and unnamed lists. A list is considered a named list if it has an attribute called "names". We can access elements of named lists by name, while unnamed lists can only be accessed by index number.

Unnamed lists

Just like vectors, an unnamed list maps to a JSON array.

```
jsonlite::toJSON(list(c(1, 2), "foo", TRUE, list(c(1, 2))))
## [[1,2],["foo"],[true],[[1,2]]]
```

Named lists

A named list in R maps to a JSON object.

```
jsonlite::toJSON(list(foo = c(1, 2), bar = TRUE))
## {"foo":[1,2],"bar":[true]}

jsonlite::toJSON(list(foo = c(1, 2), TRUE))
## {"foo":[1,2],"2":[true]}

jsonlite::toJSON(list(foo=list(bar=list(baz=pi))))
## {"foo":{"bar":{"baz":[3.1416]}}}
```

DATA FRAME

jsonlite encodes data frames based on rows and maps them to an array of records by default.

```
jsonlite::toJSON(iris[1:2,], pretty = TRUE)
## [
##   {
##     "Sepal.Length": 5.1,
##     "Sepal.Width": 3.5,
##     "Petal.Length": 1.4,
##     "Petal.Width": 0.2,
##     "Species": "setosa"
##   },
##   {
##     "Sepal.Length": 4.9,
##     "Sepal.Width": 3,
##     "Petal.Length": 1.4,
##     "Petal.Width": 0.2,
##     "Species": "setosa"
##   }
## ]
```

Missing values in data frames

When there are missing values in the data frame, jsonlite simply does not include these fields in the JSON record by default.

```
toJSON(data.frame(foo=c(TRUE,FALSE,NA,NA),bar=c(1,NA,2,NA)),pretty=TRUE)
## [
##   {
##     "foo": true,
##     "bar": 1
##   },
##   {
##     "foo": false
##   },
##   {
##     "bar": 2
##   },
##   {}
## ]
```

Relational data: nested records

R doesn’t have native support for relational data, but nested data structures are often encountered in JSON, and jsonlite uses nested data frames to achieve this mapping. When a certain field in each record in a dataset contains a subrecord with additional fields, a directly nested subrecord represents a one-to-one or many-to-one relationship between the parent table and child tables, while in R the nested relationship can be represented by a data frame which column is a list or another data frame with matching dimension.

When the value of a field in each record in JSON is an object, that is, each record contains at most one subrecord, the data frame can be represented as a column for nesting.

```
x <-
data.frame(driver=c("Bowser","Peach"),occupation=c("Koopa","Princess"))
x$vehicle <-
data.frame(model=c("Piranha Prowler","Royal Racer"),speed=c(55, 34))
str(x)

## 'data.frame':    2 obs. of  3 variables:
##  $ driver    : chr  "Bowser" "Peach"
##  $ occupation: chr  "Koopa" "Princess"
##  $ vehicle   :'data.frame':  2 obs. of  2 variables:
##   ..$ model: chr  "Piranha Prowler" "Royal Racer"
##   ..$ speed: num  55 34

jsonlite::toJSON(x, pretty = TRUE)

## [
##   {
##     "driver": "Bowser",
##     "occupation": "Koopa",
##     "vehicle": {
##       "model": "Piranha Prowler",
##       "speed": 55
##     }
##   },
##   {
##     "driver": "Peach",
##     "occupation": "Princess",
##     "vehicle": {
##       "model": "Royal Racer",
##       "speed": 34
##     }
##   }
## ]
```

Relational data: nested tables

When JSON records contain a field with a nested array, you can use a column that contains a list of vectors to represent an array of values, and you can also use a column that contains a list of data frames to represent an array of objects.

```
x <- data.frame(author = c("Homer", "Virgil"))
x$foo <- list(c(1:2), c(1:3))
x$poems <- list(
data.frame(title=c("Iliad", "Odyssey"),year=c(-1194, -800)),
data.frame(title=c("Eclogues", "Georgics", "Aeneid"),year=c(-44, -29, -19))
)
str(x)
## 'data.frame':    2 obs. of  3 variables:
##  $ author: chr  "Homer" "Virgil"
##  $ foo   :List of 2
##   ..$ : int  1 2
##   ..$ : int  1 2 3
##  $ poems :List of 2
##   ..$ :'data.frame': 2 obs. of  2 variables:
##   .. ..$ title: chr  "Iliad" "Odyssey"
##   .. ..$ year : num  -1194 -800
##   ..$ :'data.frame': 3 obs. of  2 variables:
##   .. ..$ title: chr  "Eclogues" "Georgics" "Aeneid"
##   .. ..$ year : num  -44 -29 -19

jsonlite::toJSON(x, pretty = TRUE)
## [
##   {
##     "author": "Homer",
##     "foo": [1, 2],
##     "poems": [
##       {
##         "title": "Iliad",
##         "year": -1194
##       },
##       {
##         "title": "Odyssey",
##         "year": -800
##       }
##     ]
##   },
##   {
##     "author": "Virgil",
##     "foo": [1, 2, 3],
##     "poems": [
##       {
##         "title": "Eclogues",
##         "year": -44
##       },
##       {
##         "title": "Georgics",
##         "year": -29
##       },
##       {
##         "title": "Aeneid",
##         "year": -19
##       }
##     ]
##   }
## ]
```

CONCLUSION

There are two important conventions to follow when generating JSON data in R. When we return a different set of keys every time we call the API, it's very difficult to write software to process that data. Hence, the first rule is to limit JSON interfaces to a finite set of keys that are known *a* priory by all parties. Fields or collections with ambiguous object types are difficult to describe, interpret and process in the context of inter-system communication. Therefore, the second rule is to keep that each property and array is type consistent. In summary, it is highly recommended to build a schema with consistent field names/types and homogeneous JSON arrays before generating JSON data.

REFERENCES

Ooms, J. 2014. “The jsonlite Package: A Practical and Consistent Mapping Between JSON Data and R Objects”. *ArXiv, abs/1403.2805.*

RECOMMENDED READING

- *“A Simple and Robust JSON Parser and Generator for R”. Available at https://cran.r-project.org/web/packages/jsonlite/jsonlite.pdf*

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Peng Li
Sanofi
Chengdu Yintai Centre Tower 3, Tianfu Dadao Beiduan 1199, Wuhou District
Chengdu, Sichuan 610041
marco.li@sanofi.com