A Practical and Consistent Mapping Between JSON Data and Common Object Classes in The R Language

Jeroen Ooms

November 26, 2013

1 Introduction

JSON (JavaScript Object Notation) is a lightweight data-interchange format, and one of the most common methods today for exchange of data structures in systems and applications. It is easy for humans to read and write, and easy for machines to parse and generate. The syntax is completely defined in a single page at `http://www.json.org/`: it specifies 5 primitive types (`string`, `number`, `true`, `false`, `null`) and two *universal data structures*:

- A collection of name-value pairs. In R, this is realized as a *named list*.
- An ordered set of values. In R, this is realized as an *unnamed list*.

Because the syntax is limited to these universal types and structures, `JSON` data is easily converted into a native object in almost any programming language. This is convenient for developers and has contributed to the popularity of the format. Several R packages already implement `toJSON` and `fromJSON` functions which convert between R objects and `JSON` structures. However, the exact mapping between the various R classes and the two universal data structures that are supported by `JSON` is not self evident. Currently, there are no formal guidelines, or even consensus between implementations on how exactly each class in R should be represented in `JSON`. Furthermore, upon closer inspection, even the most basic data structures in R actually do not perfectly map to their `JSON` counterparts, and leave some ambiguity for edge cases. These problems have resulted in different behavior between implementations, and can lead to unexpected output for special cases. To further complicate things, best practices of representing data in `JSON` have already been established outside the R community. Respecting these conventions where possible is important to maximize interoperability of data in R when using `JSON`.

When relying on `JSON` as the data interchange format, the mapping between R objects and `JSON` data must be consistent and unambiguous. Clients relying on `JSON` to get data in and out of R must know exactly what to expect in order to facilitate reliable communication, even if the data themselves are dynamic. Similarly, R code using dynamic `JSON` data from an external source is only reliable when the conversion from `JSON` to R is consistent. This document attempts to take away some of the ambiguity by explicitly describing the mapping between R classes and `JSON` data, highlighting problems and propose conventions that can generalize the mapping to cover all common classes and cases in R.

1.1 Reference implementation: the `jsonlite` package

The `jsonlite` package provides a reference implementation of the conventions proposed in this document. `jsonlite` is a fork of the `RJSONIO` package by Duncan Temple Lang, which again builds on `libjson C++` library from Jonathan Wallace. The `jsonlite` package uses the parser from `RJSONIO`, but the R code has been rewritten from scratch. Both packages implement `toJSON` and `fromJSON` functions, but their output is quite different. Finally, the `jsonlite` package contains a large set of unit tests to validate that R objects are correctly converted to `JSON` and vice versa. These unit tests cover all classes and edge cases mentioned in this document, and could be used to validate if other implementations follow the same convetions.

```
library(testthat)
test_package("jsonlite")
```

Note that even though `JSON` allows for inserting arbitrary white space and indentation, the unit tests assume that white space is trimmed.

1.2 Class-based versus type-based encoding

The `jsonlite` package actually implements two systems for translating between R objects and `JSON`. This document focuses on the `toJSON` and `fromJSON` functions which use R's class-based S3 method system. For all of the common classes in R, the `jsonlite` package implements `toJSON` methods as described in this document. Users in R can easily extend this system by implementing additional `toJSON` methods for other classes. However this also means that classes that do not have the `toJSON` method defined are not supported. Furthermore, the implementation of a specific `toJSON` method determines which data and metadata in the objects of this class gets encoded in its `JSON` representation, and how. In this respect, `toJSON` is similar to e.g. the `print` function, which also provides a certain *representation* of an object based on its class and optionally some print parameters. This representation does not necessarily reflect all information stored in the object, and there is no guaranteed one-to-one correspondence between R objects and `JSON`. I.e. calling `fromJSON(toJSON(object))` will return an object which only contains the data that was encoded by the `toJSON` method for this particular class, and which might even have a different class than the original.

The alternative to the S3 method system is to use type-based encoding, which `jsonlite` implements in the functions `serializeJSON` and `unserializeJSON`. All data structures in R get stored in memory using one of the internal `SEXP` storage types, and `serializeJSON` defines an encoding schema which captures the type, value, and attributes for each storage type. The result is `JSON` output which closely resembles the internal structure of the underlying C data types, and which can be perfectly restored to the original R object using `unserializeJSON`. This method is relatively straightforward to implement, however the disadvantage is that the resulting `JSON` is very verbose, hard to interpret, and cumbersome to generate in the context of another language or system. For most applications this is actually impractical because it requires the client/consumer to understand and manipulate R types, which is difficult and reduces interoperability. Instead we can make data in R more accessible to third parties by defining sensible `JSON` representations that are natural for the class of an object, rather than its internal storage type. This document does not discuss the `serializeJSON` system in any further detail, and solely treats the class based system implemented in `toJSON` and `fromJSON`. However the reader that is interested in full serialization of R objects into `JSON` is encouraged to have a look at the respective manual pages.

1.3 Scope and Limitations

Before continuing, we want to stress some limitations of encoding R data structures in `JSON`. Most importantly, there are the limitations to types of objects that can be represented. In general, temporary in-memory properties such as connections, file descriptors and (recursive) memory references are always difficult if not impossible to store in a sensible way, regardless of the language or serialization method. This document focuses on the common R classes that hold *data*, such as vectors, factors, lists, matrices and data frames. We do not treat language level constructs such as expressions, functions, promises, which hold little meaning outside the context of R. We also don't treat special compound classes such as linear models or custom classes defined in contributed packages. When designing systems or protocols that interact with R, it is highly recommended to stick with the standard data structures for the interface input/output.

Then there are limitations introduced by the format. Because `JSON` is a human readable, text-based format, it does not support binary data, and numbers are stored in their decimal notation. The latter leads to loss of precision for real numbers, depending on how many digits the user decides to print. Several dialects of `JSON` exists such as `BSON` or `MSGPACK`, which extend the format with various binary types. However, these formats are much less popular, less interoperable, and often impractical, precisely because they require binary parsing and abandon human readability. The simplicity of `JSON` is what makes it an accessible and widely applicable data interchange format. In cases where it is really needed to include some binary data in `JSON`, one can use something like `base64` to encode it as a string.

Finally, as mentioned earlier, `fromJSON` is not a perfect inverse function of `toJSON`, as would be the case for `serialializeJSON` and `unserializeJSON`. The class based S3 method system allows for concise and practical encoding of the various common data structures. In our design of `toJSON` and `fromJSON`, we have tried to approach a reversible mapping between R objects and `JSON` for the standard data classes, but there are always limitations and edge cases. For example, the `JSON` representation of an empty vector, empty list or empty data frame are all the same: `"[ ]"`. Also some special vector types such as factors, dates or timestamps get coersed to strings, as they would in for example `CSV`. This is a quite typical and expected behavior among text based formats, but it does require some additional interpretation on the side of the consumer.

1.4 Goals: Consistent and Practical

It can be helpful to see the problem from both sides. The R user needs to interface external `JSON` data from within R. This includes reading data from a public source/API, or posting a specific `JSON` structure to an online service. From perspective of the R user, `JSON` data should be realized in R using classes which are most natural in R for a particular structure. A proper mapping is one which allows the R user to read any incoming data or generate a specific `JSON` structures using the familiar methods and classes in R. Ideally, the R user would like to forget about the interchange format at all, and think about the external data interface in terms of its corresponding R structures rather than a `JSON` schema. The other perspective is that of an third party client or language, which needs to interface data in R using `JSON`. This actor wants to access and manipulate R objects via their `JSON` representation. A good mapping is one that allows a 3rd party client to get data in and out of R, without necessarily understanding the specifics of the underlying R classes. Ideally, the external client could forget about the R objects and classes at all, and think about input and output of data in terms of the `JSON` schema, or the corresponding realization in the language of the client.

Both sides come together in the context of an RPC service such as OpenCPU. OpenCPU exposes a

HTTP API to let 3rd party clients call R functions over HTTP. The function arguments are posed using `JSON` and OpenCPU automatically converts these into R objects to construct the R function call. The return value of the function is then converted to `JSON` and sent back to the client. To the client, the service works as a `JSON` API, but it is implemented as standard R function uses standard data structures for its arguments and return value. For this to work, the conversion between `JSON` data and R objects must be consistent and unambiguous. In the design of our mapping we have pursued the following requirements:

- Recognize and comply with existing conventions of encoding common data structures in `JSON`, in particular (relational) data sets.
- Consistently use a particular schema for a class of objects, including edge cases.
- Avoid R-specific peculiarities to minimize opportunities for misinterpretation.
- Mapping should optimally be reversible, but at least coercible for the standard classes.
- Robustness principle: be strict on output but tolerant on input.

2 Converting between `JSON` and R Classes

This section lists examples of how the common R classes are represented in `JSON`. As explained before, the `toJSON` method relies on the S3 method system, which means that objects get encoded according to their `class`. If an object has multiple `class` values, R uses the first occurring class which has a `toJSON` method. If none of the classes of an object has a `toJSON` method, an error is raised.

2.1 Atomic Vectors

The most basic data type in R is the atomic vector. The atomic vector holds an ordered set of homogeneous values of type `"logical"` (booleans), `character` (strings), `"raw"` (bytes), `integer`, `numeric` (doubles), `"complex"` (complex numbers with a real and imaginary part). Because R is fully vectorized, there is no user level notion of a primitive: a scalar value is considered a vector of length 1. Atomic vectors are encoded using a `JSON` array:

```
x <- c(1, 2, pi)
cat(toJSON(x))

[ 1, 2, 3.14 ]
```

The `JSON` array is the only appropriate structure to encode a vector, however note that vectors in R are homogeneous, whereas the `JSON` array is actually heterogeneous, but `JSON` does not make this distinction.

2.1.1 Missing Values

A typical domain specific problem when working with statistical data is presented by missing data: a concept foreign to many other languages. Besides regular values, all vector types except for `raw` can hold `NA` as a value. Vectors of type `double` and `complex` define three additional types of non finite values: `NaN`, `Inf` and

`-Inf`. `JSON` does not natively support any of these types; therefore such values values need to be encoded in some other way. There are two obvious approaches. The first one is to use the `JSON` `null` type. For example:

```
x <- c(TRUE, FALSE, NA)
cat(toJSON(x))

[ true, false, null ]
```

The other option is to encode missing values as strings by wrapping them in double quotes:

```
x <- c(1, 2, NA, NaN, Inf, 10)
cat(toJSON(x))

[ 1, 2, "NA", "NaN", "Inf", 10 ]
```

Both methods result in valid `JSON`, but both have a limitation: the problem with the `null` type is that there is no way to distinguish between different types of missing data, which could be a problem for numeric vectors. The values `Inf`, `-Inf`, `NA` and `NaN` have different meanings, and these should not get lost in the encoding. However, the problem with encoding missing values as strings is that this method can not be used for character vectors, because the consumer won't be able to distinguish the actual string `"NA"` and the missing value `NA`. This would create a likely source of bugs, where clients mistakenly interpret `"NA"` as an actual value, which is a common problem with text-based formats such as CSV. For this reason, `jsonlite` uses the following defaults:

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

We expect that these conventions are most likely to result in the correct interpretation of missing values. Some examples:

```
cat(toJSON(c(TRUE, NA, NA, FALSE)))

[ true, null, null, false ]

cat(toJSON(c("FOO", "BAR", NA, "NA")))

[ "FOO", "BAR", null, "NA" ]

cat(toJSON(c(3.14, NA, NaN, 21, Inf, -Inf)))

[ 3.14, "NA", "NaN", 21, "Inf", "-Inf" ]

cat(toJSON(c(3.14, NA, NaN, 21, Inf, -Inf), na = "null"))

[ 3.14, null, null, 21, null, null ]
```

2.1.2 Special vector types: dates, times, factor, complex

Besides missing values, `JSON` also lacks native support for some of the basic vector types in R that frequently appear in data sets. These include vectors of class `Date`, `POSIXt` (timestamps), `factors` and `complex` vectors. By default, the `jsonlite` package coerces these types to strings (using `as.character`):

```
cat(toJSON(Sys.time() + 1:3))

[ "2013-11-26 16:21:53", "2013-11-26 16:21:54", "2013-11-26 16:21:55" ]

cat(toJSON(as.Date(Sys.time()) + 1:3))

[ "2013-11-28", "2013-11-29", "2013-11-30" ]

cat(toJSON(factor(c("foo", "bar", "foo"))))

[ "foo", "bar", "foo" ]

cat(toJSON(complex(real = runif(3), imaginary = rnorm(3))))

[ "0.79-0.02i", "0.7+0.49i", "0.27-0.71i" ]
```

When parsing such `JSON` strings, these values will appear as character vectors. In order to obtain the original types, the user needs to manually coerce them back to the desired type using the corresponding `as` funtion, e.g. `as.POSIXct`, `as.Date`, `as.factor` or `as.complex`. In this respect, `JSON` is subject to the same limitations as text based formats such as CSV.

2.1.3 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:

```
# vectors of length 0 and 1
cat(toJSON(vector()))

[]

cat(toJSON(pi))

[ 3.14 ]

# vectors of length 0 and 1 in a named list
cat(toJSON(list(foo = vector())))

{ "foo" : [] }

cat(toJSON(list(foo = pi)))

{ "foo" : [ 3.14 ] }
```

```
# vectors of length 0 and 1 in an unnamed list
cat(toJSON(list(vector())))

[ [] ]

cat(toJSON(list(pi)))

[ [ 3.14 ] ]
```

This might seem obvious but these cases result in very different behavior between different `JSON` packages. This is probably caused by the fact that R does not have a scalar type, and some package authors decided to treat vectors of length 1 as if they were a scalar. For example, in the current implementations, both `RJSONIO` and `rjson` encode a vector of length one as a `JSON` primitive when it appears within a list:

```
# Other packages make different choices:
cat(rjson::toJSON(list(n = c(1))))

## {"n":1}

cat(rjson::toJSON(list(n = c(1, 2))))

## {"n":[1,2]}
```

When encoding a single data set this might seem harmless, but in the context of dynamic data this inconsistency is almost guaranteed to cause bugs. For example, imagine an R web service which lets the user fit a linear model and sends back the fitted parameter estimates. In this application, the R server returns the vector with coefficients encoded as a `JSON` array. The client code then parses the `JSON`, and then iterates over the array of coefficients to display them in a GUI. All goes well, until the user decides to fit a model with only one predictor. If the `JSON` encoder returns a primitive value where the client is assuming an array, the application will likely break. Any consumer or client would need to be aware of the special case where the vector becomes a primitive, and explicitly take this exception into account when processing the result. When the client fails to do so and proceeds as usual, it will probably call an iterator or loop method on a primitive value, resulting in the obvious errors. For this reason `jsonlite` uses consistent encoding schemes which do not depend on variable object properties such as its length. Hence, a vector is always encoded as an array, even when it is of length 0 or 1.

2.2 Matrices

Arguably one of the strongest sides of R is its ability to interface libraries for basic linear algebra subroutines (BLAS) such as LAPACK, OpenBLAS, etc. These libraries provide well tuned, high performance implementations of important linear algebra operations to calculate anything from inner products and eigen values to singular value decompositions. These are in turn the building blocks of many higher level statistical methods such as linear regression or principal component analysis. Linear algebra methods operate on *matrices*, making the matrix one of the most central data classes in R. Conceptually, a matrix consists of a 2 dimensional structure of homogeneous values. It is indexed using 2 numbers (or vectors), representing the row and column number of the matrix respectively.

```
x <- matrix(1:12, nrow = 3, ncol = 4)
print(x)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

```
print(x[2, 4])
```

```
[1] 11
```

A matrix is stored in memory as a single atomic vector with an attribute called `"dim"` defining the dimensions of the matrix. The product of the dimensions is equal to the length of the vector.

```
attributes(volcano)
```

```
$dim
[1] 87 61
```

```
length(volcano)
```

```
[1] 5307
```

Even though the matrix is stored as a single vector, it the way it is printed, indexed and treated otherwise makes it conceptually a 2 dimensional structure. `jsonlite` treats a matrix as a set of equal-length homogeneous vectors, and therefore encodes it using an array of sub arrays:

```
x <- matrix(1:12, nrow = 3, ncol = 4)
cat(toJSON(x))
```

```
[ [ 1, 4, 7, 10 ], [ 2, 5, 8, 11 ], [ 3, 6, 9, 12 ] ]
```

We hope that this represention will be the most intuitive to interpret, even within languages that do not have a native notion of a matrix. When the JSON string is properly indented (recall that whitespace and linebreaks are optional in JSON), it looks very similar to the way R prints matrices:

```
[ [ 1, 4, 7, 10 ],
  [ 2, 5, 8, 11 ],
  [ 3, 6, 9, 12 ] ]
```

Because the matrix is implemented in R as an atomic vector, it automatically inherits the conventions mentioned earlier with respect to edge cases and missing values:

```
x <- matrix(c(1, 2, 4, NA), nrow = 2)
cat(toJSON(x))
```

```
[ [ 1, 4 ], [ 2, "NA" ] ]

cat(toJSON(x, na = "null"))

[ [ 1, 4 ], [ 2, null ] ]

cat(toJSON(matrix(pi)))

[ [ 3.14 ] ]
```

2.2.1 Matrix row and column names

Besides the `"dim"` attribute, the matrix class has an additional, optional attribute: `"dimnames"`. This attribute holds names for the rows and columns in the matrix. However, we decided not to include this information in the default `JSON` encoding for matrices for several reasons. First of all, because this attribute is optional, often either row or column names or both are `NULL`. This makes it difficult to define a practical encoding that covers all cases with and without row and/or column names. Secondly, the names in matrices are mostly there for annotation only; they are not actually used in calculations. For example, the linear algebra subroutines mentioned before completely ignore them, and do not include any names in their output. So there is often little purpose of setting names in the first place, other annotation.

When the row names or column names of a matrix actually contain vital information, chances are they actually are not names but variable values. It might be needed to transform the data into a more appropriate structure. Wickham (2013) calls this tidying the data and outlines best practices on storing statistical data in its most appropriate form. He lists the issue where *"column headers are values, not variable names"* the most common source of *"messy"* data. It happens when the structure is optimized for presenentation (e.g. printing), rather than computation. In the following example, taken from Wickham (2013), the column headers do not list names, but actually contain values of the main variable (treatment). As a result, these are not represented in the `JSON` encoding:

```
x <- matrix(c(NA, 1, 2, 5, NA, 3), nrow = 3)
row.names(x) <- c("Joe", "Jane", "Mary")
colnames(x) <- c("Treatment A", "Treatment B")
print(x)

     Treatment A Treatment B
Joe           NA           5
Jane           1          NA
Mary           2           3

cat(toJSON(x))

[ [ "NA", 5 ], [ 1, "NA" ], [ 2, 3 ] ]
```

Wikcham recommends that the data be *melted* into its *tidy* form. Once the data is tidy, the `JSON` encoding will naturally contain the treatment values:

```
library(reshape2)
y <- melt(x, varnames = c("Subject", "Treatment"))
print(y)
```

```
  Subject   Treatment value
1     Joe Treatment A    NA
2    Jane Treatment A     1
3    Mary Treatment A     2
4     Joe Treatment B     5
5    Jane Treatment B    NA
6    Mary Treatment B     3
```

```
cat(toJSON(y, pretty = TRUE))
```

```
[
  {
    "Subject" : "Joe",
    "Treatment" : "Treatment A"
  },
  {
    "Subject" : "Jane",
    "Treatment" : "Treatment A",
    "value" : 1
  },
  {
    "Subject" : "Mary",
    "Treatment" : "Treatment A",
    "value" : 2
  },
  {
    "Subject" : "Joe",
    "Treatment" : "Treatment B",
    "value" : 5
  },
  {
    "Subject" : "Jane",
    "Treatment" : "Treatment B"
  },
  {
    "Subject" : "Mary",
    "Treatment" : "Treatment B",
    "value" : 3
  }
]
```

In some other cases, the column headers actually do contain variable names, and melting is inappropriate. For data sets with records consisting of a set of named columns (fields), R has more natural and flexible class: the data-frame. The `toJSON` method for data frames is described in a later section, and is more suitable when we want to refer to rows or fields by a particular name. Any matrix can easily be converted to a dataframe using the `as.data.frame` function:

```
cat(toJSON(as.data.frame(x), pretty = TRUE))
```

```
[
  {
    "$row" : "Joe",
    "Treatment B" : 5
  },
  {
    "$row" : "Jane",
    "Treatment A" : 1
  },
  {
    "$row" : "Mary",
    "Treatment A" : 2,
    "Treatment B" : 3
  }
]
```

For some cases this results in the desired encoding, but in this example melting seems more appropriate.

2.3 Lists

The `list` is the most general purpose data structure in R. It holds an ordered set of elements, including other lists, each of arbitrary type and size. 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"`. In practice, a named list is any list for which we can access an element by its name, whereas elements of an unnamed lists can only be accessed using their index number:

```
mylist1 <- list(foo = 123, bar = 456)
print(mylist1$bar)
```

```
[1] 456
```

```
mylist2 <- list(123, 456)
print(mylist2[[2]])
```

```
[1] 456
```

2.3.1 Unnamed lists

Just like vectors, an unnamed list is encoded using a JSON array:

```
cat(toJSON(list(c(1, 2), "test", TRUE, list(c(1, 2)))))
```

```
[ [ 1, 2 ],[ "test" ],[ true ],[ [ 1, 2 ] ] ]
```

Note that even though both vectors and lists are encoded using JSON arrays, they can be distinguished by looking at their contents: an R vector results in a JSON array which contains only primitives, whereas a list results in a JSON array which contains only objects and arrays. This allows the JSON parser to reconstruct the original type from encoded vectors and arrays:

```
x <- list(c(1, 2, NA), "test", FALSE, list(foo = "bar"))
identical(fromJSON(toJSON(x)), x)
```

```
[1] TRUE
```

The only exception is the empty list and empty vector, which are both encoded as `[ ]` and therefore indistinguishable, but this is rarely a problem in practice.

2.3.2 Named lists

A named list in R gets encoded as a JSON *object*:

```
cat(toJSON(list(foo = c(1, 2), bar = "test")))
```

```
{ "foo" : [ 1, 2 ], "bar" : [ "test" ] }
```

Because a list can contain other lists, this works recursively:

```
cat(toJSON(list(foo=list(bar=list(baz=pi)))))
```

```
{ "foo" : { "bar" : { "baz" : [ 3.14 ] } } }
```

Named lists map almost perfectly to JSON objects with one exception: list elements can have empty names:

```
x <- list(foo = 123, "test", TRUE)
attr(x, "names")
```

```
[1] "foo" ""    ""
```

```
x$foo
```

```
[1] 123
```

```
x[[2]]
```

```
[1] "test"
```

In a `JSON` object, each element in an object must have a valid name. To ensure this property, `jsonlite` uses the same solution as the `print` method, which is to fall back on indices for elements that do not have a proper name:

```
x <- list(foo = 123, "test", TRUE)
print(x)

$foo
[1] 123

[[2]]
[1] "test"

[[3]]
[1] TRUE

cat(toJSON(x))

{ "foo" : [ 123 ], "2" : [ "test" ], "3" : [ true ] }
```

This behavior ensures that all generated `JSON` is valid, however named lists with empty names should be avoided where possible. When actually designing R objects that should be interoperable, it is recommended that each list element has a proper name.

2.4 Data frame

The `data frame` is perhaps the most central data structure in R from the user point of view. This class holds tabular data in which each column is named and (usually) homogeneous. Conceptually it is very similar to a table in relational data bases such as `MySQL`, where *fields* are referred to as *column names*, and *records* are called *row names*. Like a matrix, a data frame can be subsetted with two indices, to extract certain rows and columns of the data:

```
is(iris)

[1] "data.frame" "list"       "oldClass"   "vector"

names(iris)

[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
[5] "Species"

print(iris[1:3, c(1, 5)])

  Sepal.Length Species
1          5.1  setosa
2          4.9  setosa
3          4.7  setosa
```

```
print(iris[1:3, c("Sepal.Width", "Species")])
```

```
  Sepal.Width Species
1         3.5  setosa
2         3.0  setosa
3         3.2  setosa
```

For the previously discussed classes such as vectors and matrices, behavior of jsonlite is quite similar to the other available packages that implement `toJSON` and `toJSON` functions, with only minor differences for missing values and edge cases. But when it comes to data frames, jsonlite takes a completely different approach. The behavior of jsonlite is designed to be compatible with conventional ways of encoding table-like structures outside the R community. The implementation is more complex, but results in a powerful and more natural way of interfacing data frames through `JSON` and vice versa.

2.4.1 Column based versus row based tables

Generally speaking, tabular data structures can be implemented in two different ways: in a column based, or row based fasion. A column based structure consists of a named collection of equal-length, homogenous arrays representing the table columns. In a row-based structure on the other hand, the table is implemented as a set of heterogeneous associative arrays representing table rows with field values for each particular record. Even though most languages provide flexible and abstracted interfaces that hide such implementation details from the user, they can have huge implications for performance. A column based structure is efficient for inserting or extracting certain columns of the data, but it is inefficient for manipulating individual rows. For example to insert a single row somewhere in the middle, each of the columns has to be sliced and stiched back together. For row-based implementations, it is the extact other way around: we can easily manipulate a particular record, but to insert/extract a whole column we would need to iteriate over all records in the table and read/modify the appropriate field in each of them.

The data frame in R is implemented in a column based fasion: it constitutes of a `named list` of equal-length vectors. Thereby the columns in the data frame naturally inherit the properties from atomic vectors discussed before, such as homogeneity, missing values, etc. Another argument for column-based implementation is that statistical methods generally operate on columns. For example, the `lm` function fits a *linear regression* by extracting the columns from a data frame as specified by the `formula` argument. R simply binds the specified columns together into a matrix `X` and calls out to a highly optimized fortran subroutine to calculate the OLS estimates $\hat{\beta} = (X^T X)X^T y$ using the QR factorization of `X`. Many other statistical modeling functions follow similar steps, and are computationally efficient because of the column-based data storage in R. However, unfortunately R is an exception in its preference for column-based storage: most languages, systems, databases, APIs, etc, are optimized for record based operations. For this reason, the conventional way to store and communicate tabular data in `JSON` seems to almost exclusively row based. This discrepancy presents various complications when converting between data frames and `JSON`. The remaining of this section discusses details and challenges of consistently mapping record based `JSON` data as frequently encountered in the wild, into column-based data frames which are convenient for statistical computing.

2.4.2 Row based data frame encodingin jsonlite

The encoding of data frames is one of the major differences between jsonlite and implementations from other currently available packages. Instead of using the column-based encoding also used for lists, jsonlite encodes data frames by default as an array of records:

```
cat(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"
  }
]
```

This output looks a bit like a list of named lists. However, there is one major difference: the individual records contain primitives values, which will never appear in lists:

```
cat(toJSON(list(list(Species = "Foo", Width = 21))))
```

```
[ { "Species" : [ "Foo" ], "Width" : [ 21 ] } ]
```

This leads to the following convention: when encoding R objects, `JSON` primitives only appear in vectors and data-frame rows. Primitive values within a `JSON` array indicate a vector, and `JSON` primitives appearing inside a `JSON` object indicate a data-frame row. A `JSON` encoded `list`, (named or unnamed) will never contain `JSON` primitives. This is a subtle but important convention that helps to distinguish between R classes from their `JSON` resprestation, without explicitly encoding any metadata.

2.4.3 Missing values in data frames

The section on atomic vectors discussed two methods of encoding missing data appearing in a vector: either using strings or using the `JSON` `null` type. When a missing value appears in a data frame, there is a third option: simply not include this field in `JSON` record:

```
x <- data.frame(foo = c(FALSE, TRUE, NA, NA), bar = c("Aladdin", NA, NA, "Mario"))
print(x)
```

```
    foo     bar
1 FALSE Aladdin
2  TRUE    <NA>
3    NA    <NA>
4    NA   Mario
```

```
cat(toJSON(x, pretty = TRUE))
```

```
[
  {
    "foo" : false,
    "bar" : "Aladdin"
  },
  {
    "foo" : true
  },
  {},
  {
    "bar" : "Mario"
  }
]
```

The default behavior of jsonlite is to omit missing data from records in a data frame. This seems to be the most conventional way of doing things, and we expect this encoding will most likely lead to the correct interpretation of *missingness*, even in languages which have no explicit notion of `NA`.

2.4.4 Relational data: nested records

Nested datasets are somewhat unusual in R, but frequently encountered in `JSON`. Such structures do not really fit the vector based paradigm which makes them harder to manipulate in R. However, nested structures are too common in `JSON` to ignore, and with a little work most cases still map to a data frame quite nicely. The most common scenario is a dataset in which a certain field within each record contains a *subrecord* with additional fields. jsonlite interprets these subrecords as a nested data frame. Whereas the data frame class usually consists of vectors, technically a column can also be list or another data frame with matching dimension (this stretches the meaning of the word "column" a bit):

```
options(stringsAsFactors=FALSE)
x <- data.frame(driver = c("Bowser", "Peach"), occupation = c("Koopa", "Princess"))
x$vehicle <- data.frame(model = c("Piranha Prowler", "Royal Racer"))
x$vehicle$stats <- data.frame(speed = c(55, 34), weight = c(67, 24), drift = c(35, 32))
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"
  ..$ stats:'data.frame': 2 obs. of  3 variables:
  .. ..$ speed : num  55 34
  .. ..$ weight: num  67 24
  .. ..$ drift : num  35 32
```

```
cat(toJSON(x, pretty=TRUE))
```

```
[
  {
    "driver" : "Bowser",
    "occupation" : "Koopa",
    "vehicle" : {
      "model" : "Piranha Prowler",
      "stats" : {
        "speed" : 55,
        "weight" : 67,
        "drift" : 35
      }
    }
  },
  {
    "driver" : "Peach",
    "occupation" : "Princess",
    "vehicle" : {
      "model" : "Royal Racer",
      "stats" : {
        "speed" : 34,
        "weight" : 24,
        "drift" : 32
      }
    }
  }
]
```

```
myjson <- toJSON(x)
y <- fromJSON(myjson)
identical(x,y)
```

```
[1] TRUE
```

When encountering `JSON` data containing nested records in the wild, chances are that these data were generated from *relational* database. The `JSON` field containing a subrecord represents a *foreign key* pointing to a record in an external table. For the purpose of encoding these into a single `JSON` structure, the tables were joined into a nested structure. The directly nested subrecord represents a *one-to-one* or *many-to-one* relation between the parent and child table, and is most naturally stored in R using a nested data frame. In the example above, the `vehicle` field points to a table of verhicles, which in turn contains a `stats` field pointing to a table of stats. When there is no more than one subrecord for each record, we easily *flatten* the structure into a single non-nested data frame.

```
z <- cbind(x[c("driver", "occupation")], x$vehicle["model"], x$vehicle$stats)
str(z)

'data.frame': 2 obs. of  6 variables:
 $ driver    : chr  "Bowser" "Peach"
 $ occupation: chr  "Koopa" "Princess"
 $ model     : chr  "Piranha Prowler" "Royal Racer"
 $ speed     : num  55 34
 $ weight    : num  67 24
 $ drift     : num  35 32
```

2.4.5 Relational data: nested tables

The one-to-one relation discussed above is relatively easy to store in R, because each record contains at most one subrecord. Therefore we can use either a nested dataframe, or flatten the data frame. However, things get more difficult when `JSON` records contain a field with a nested array. Such a structure appears in relational data in case of a *one-to-many* relation. A standard textbook illustration is the relation between authors and titles. For example, a field can contain an array of values:

```
x <- data.frame(author = c("Homer", "Virgil", "Jeroen"))
x$poems <- list(c("Iliad", "Odyssey"), c("Eclogues", "Georgics", "Aeneid"), vector());
names(x)

[1] "author" "poems"

cat(toJSON(x, pretty = TRUE))

[
  {
    "author" : "Homer",
    "poems" : [
      "Iliad",
      "Odyssey"
    ]
  },
  {
```

```
    "author" : "Virgil",
    "poems" : [
      "Eclogues",
      "Georgics",
      "Aeneid"
    ]
  },
  {
    "author" : "Jeroen",
    "poems" : []
  }
]
```

As can be seen from the example, the way to store this in a data frame is using a list of character vectors. This works, and although unconventional, we can still create and read such structures in R relatively easily. However, in practice the one-to-many relation is often more complex. It results in fields containing a *set of records*. In R, the only way to model this is as a column containing a list of data frames, one seperate data frame for each row:

```
x <- data.frame(author = c("Homer", "Virgil", "Jeroen"))
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)),
  data.frame()
)
cat(toJSON(x, pretty=TRUE))
```

```
[
  {
    "author" : "Homer",
    "poems" : [
      {
        "title" : "Iliad",
        "year" : -1194
      },
      {
        "title" : "Odyssey",
        "year" : -800
      }
    ]
  },
  {
    "author" : "Virgil",
    "poems" : [
```

```
      {
        "title" : "Eclogues",
        "year" : -44
      },
      {
        "title" : "Georgics",
        "year" : -29
      },
      {
        "title" : "Aeneid",
        "year" : -19
      }
    ]
  },
  {
    "author" : "Jeroen",
    "poems" : []
  }
]
```

Because R doesn't have native support for relational data, there is no natural class to store such structures. The best we can do is a column containing a list of sub-dataframes. This does the job, and allows the R user to access or generate nested `JSON` structures. However, a data frame like this cannot be flattened, and the class does not guarantee that each of the individual sub-dataframe contain the same fields, as would be the case in an actual relational data base.

3 Structural Consistency of Dynamic Data

Systems that automatically exchange information over some interface, protocol or API require well defined and unambiguous meaning and arrangement of data. In order to process and interpret input and output, contents must obey a steady structure. Such structures are usually described either informally in documentation or more formally in a schema language. The previous section emphasized the importance of consistency in the mapping between `JSON` data and R classes. This section takes a higher level view and explains the importance of structure consistency for dynamic data. This topic can be a bit subtle because it refers to consistency among different instantiations of a `JSON` structure, rather than a single case. We try to clarify by breaking down the concept into two important parts, and illustrate with analogies and examples from R.

3.1 Classes, Types and Data

Most object-oriented languages are designed with the idea that all objects of a certain class implement the same fields and methods. In strong-typed languages such as S4 or Java, names and types of the fields are formally declared in a class defintion. In other languages such as S3 or JavaScript, the fields are not enforced by the language, and the responsibility of the programmer. But one way or another they all assume that

all members of a certain class use identical field names and types, so that the same methods can be applied to any object of a particular class. This basic principle holds for dynamic data in the same way as for class objects. Software that process dynamic data can only work reliably if the various elements of the data have consistent names and structure. Consensus must exist between the different parties on data that is exchanged as part an interface or protocol. This requires the structure to follow some sort of schema that specifies which attributes can appear in the data, what they mean and how they are composed. Thereby each possible scenario can be accounted for in the software so that data gets interpreted/processed appropriately and no exceptions arise during runtime.

Some other data interchange formats, such as `XML` or `Protocol Buffers` take a formal approach to this matter, and have well established *schema languages* and *interface description languages*. Using such a meta language it is possible to define the exact structure, properties and actions of data interchange in a formal arrangement. However, in `JSON`, such formal definitions are relativily uncommon. Some initiatives for `JSON` schema languages exist, but they aren't very well established and rarely seen in practice. One reason for this might be that defining and implementing formal schemas is complicated and a lot of work which defeats the purpose of using an lightweight format such as `JSON` in the first place. But another reason is that it is often simply not necessary to be overly formal. The `JSON` format is simple and intuitive, and under some general conventions, a well chosen example can suffice to characterize the structure. This section describes two important rules that are required to ensure that data exchange using `JSON` is consistent.

3.2 Rule 1: Consistent Keys

When using `JSON` without a schema, there are no restrictions on the keys (field names) that can appear in a particular object. However, an API that returns a different set of keys for every time it is called makes it very difficult to write software to process these data. Hence, the first rule is to limit `JSON` interfaces to a finite set of keys that are known *a priory* by all parties. It can be helpful to think about this in analogy with for example a relational database. Here, the database model seperates the data from metadata. At runtime, records can be inserted or deleted, and a certain query might return different data each time it is executed. But for a given query, each execution will return exactly the same *field names*; hence as long as the table definitions are unchanged, the *structure* of the output consistent. Client software needs this structure to validate input, optimize implementation, and process each part of the data appropriately. In `JSON`, data and metadata are not formally seperated as in a database, but similar principles that hold for fields in a database, apply to keys in dynamic `JSON` data.

A beautiful example of this going wrong in practice was given by Mike Dewar (Data Scientist at bit.ly) in at the New York Open Statistical Programming Meetup on Jan. 12, 2012. In his talk he emphasizes to use `JSON` keys only for *names*, and not for *data*. He refers to this principle as the "golden rule", and explains how he learned his lesson the hard way. In one of his early applications, timeseries data was encoded by using the epoch timestamp as the `JSON` key. Therefore the keys are different each time the query is executed:

```
[
  {"1325344443" : 4},
  {"1325344456" : 5},
  {"1325344478" : 6},
]
```

Even though being valid `JSON`, dynamic keys as in the example above are likely to introduce trouble down the line. For example, when documenting the API, either informally or formally using a schema language, we want to describe for each property that might appear in the data what it's value means and is compososed of. The client or consumer of the data needs to implement code that interprets and process each element in the data in an appropriate manner. Both the documentation and interpretation of `JSON` data rely on fixed keys with well defined meaning. Also note that the structure cannot be extended in the future. If we want to add an additional property to each observation, the entire structure needs to change. In his talk, Dewar explains that life gets much easier when we switch to the following encoding:

```
[
  {"time": "1325344443" : "price": 4},
  {"time": "1325344456" : "price": 5},
  {"time": "1325344478" : "price": 6}
]
```

This structure will play much nicer with existing software that assumes fixed keys. Moreover, the structure can easily be described in documentation, or captured in a schema. Even when we have no intention of writing documentation or a schema for a dynamic `JSON` source, it is still wise to design the structure in such away that it *could* be described by a schema. When the keys are fixed, a well chosen example can provide all the information required for the consumer to implement client code. Also note that the new structure is extensible: additional peroperties can be added to each observation without breaking backward compatibility.

In the context of R, consitency of keys is closely related to Wikcham's concept of *tidy data* discussed earlier. Wickham states that the most common reason for messy data are column headers containing values instead of variable names. Column headers in tabular datasets become keys when converted to `JSON`. Therefore, when headers are actually values, the `JSON` keys contain in fact data, and can become unpredictable. The cure to inconsistent keys is almost always to tidy the data according to recommendations given by Wickham.

3.3 Rule 2: Consistent Types

In a strong typed language, fields declare their class before any values are assigned. Thereby the type of a given field is identical in all objects of a particular class, and arrays only contain objects of a single type. The S3 system in R is weakly typed and puts no formal restrictions on the class of a certain properties, or the types of objects that can be combined into a collection. For example, the list below contains a character vector, a numeric vector and a list:

```
# Heterogeneous lists are bad!
x <- list("FOO", 1:3, list(bar = pi))
cat(toJSON(x))

[ [ "FOO" ],[ 1, 2, 3 ],{ "bar" : [ 3.14 ] } ]
```

Also we can assign any type to a any field:

```
x <- list()

# first its numberic
x$test <- 5:2
cat(toJSON(x))

{ "test" : [ 5, 4, 3, 2 ] }

# and then its a list
x$test <- list(bar = TRUE)
cat(toJSON(x))

{ "test" : { "bar" : [ true ] } }
```

But just because this is possible, doesn't mean you should use it. On the contrary: it puts the responsibility of using consistent types in the hands of the developer. Fields or collections with ambiguous object types are very diffucult to describe, interpret and process in the context of inter-system communication. When using `JSON` to exchange dynamic data, it is important that each property and array is *type consistent*. This means that the developer needs to make sure that properties are of the correct type before encoding to `JSON`. Moreover, the `unnamed list` object in R should usually be avoided all together when designing interoperable structures. However, note that consitency is somewhat subjective as it refers to the *meaning* of the elements; they don't necessarily need to have precisely the same structure. What is important is to keep in mind that the consumer of the data can treat each element identically, e.g. iterate over the elements in the collection and apply the same method to each of them. To illustrate this, lets take the example of the data frame:

```
# conceptually homogenous array
x <- data.frame(name = c("Jay", "Mary", NA, NA), gender = c("M", NA, NA, "F"))
cat(toJSON(x, pretty = TRUE))

[
  {
    "name" : "Jay",
    "gender" : "M"
  },
  {
    "name" : "Mary"
  },
  {},
  {
    "gender" : "F"
  }
]
```

The `JSON` array above has 4 elements, each of which a `JSON` object. However, due to the `NA` values, not all elements have an identical structure: some records have more fields than others. But as long as they

are conceptually the same type (e.g. a person), the consumer can iterate over the elements to process each "person" in the set according to a predefined action. For example each element could be used to construct a Person object. When combining objects of different classes, a `named list` can be used to distinguish between the different types:

```
x <- list(
  humans = data.frame(name = c("Jay", "Mary"), married = c(TRUE, FALSE)),
  horses = data.frame(name = c("Star", "Dakota"), price = c(5000, 30000))
)
cat(toJSON(x, pretty=TRUE))
```

```
{
  "humans" : [
    {
      "name" : "Jay",
      "married" : true
    },
    {
      "name" : "Mary",
      "married" : false
    }
  ],
  "horses" : [
    {
      "name" : "Star",
      "price" : 5000
    },
    {
      "name" : "Dakota",
      "price" : 30000
    }
  ]
}
```

When looking at the public APIs, limiting `JSON` arrays to homogeneous sets is a very strong convention and we have not been able to find a single exception.

4 Public `JSON` APIs

Some examples of popular public APIs that use `JSON` to server data. These are great to get a sense of the common structures that we encounter with real world `JSON` data. All examples use HTTP GET, some require registration/authentication.

4.1 Github

Github is an active online code repository and has APIs to get live data on almost anything that is happening. Some examples related to activity from a well known R package and author:

```
hadley_orgs <- fromJSON("https://api.github.com/users/hadley/orgs")
hadley_repos <- fromJSON("https://api.github.com/users/hadley/repos")
gg_issues <- fromJSON("https://api.github.com/repos/hadley/ggplot2/issues")
gg_commits <- fromJSON("https://api.github.com/repos/hadley/ggplot2/commits")
```

4.2 Citi Bike NYC

A single public API that shows location, status and current availability for all stations in the New York City bike sharing initative.

```
citibike <- fromJSON("http://citibikenyc.com/stations/json")
```

4.3 Sunlight Foundation

The Sunlight Foundation is a non-profit that helps to make government transparent and accountable through data, tools, policy and journalism. The API can be accessed for free, but requires registration to obtain an API key (which is very easy). Examples:

```
#register key at http://sunlightfoundation.com/api/accounts/register/
key <- "&apikey=6e021ea1c1264a658e259708076f04a1"

#some queries
drones <- fromJSON(paste0("http://openstates.org/api/v1/bills/?q=drone", key))
word <- fromJSON(paste0("http://capitolwords.org/api/1/dates.json?phrase=obamacare", key))
legislators <- fromJSON(paste0("http://congress.api.sunlightfoundation.com/",
  "legislators/locate?latitude=42.96&longitude=-108.09", key))
```

4.4 New York Times

The New York Times has serveral free APIs that are part of its "developer network". They interface to different parts of the organization, such as news articles, book reviews, real estate, etc. Registration is required to obtain a API keys.

```
# Register keys at http://developer.nytimes.com/docs/reference/keys

# search for articles
article_key = "&api-key=12b3ff5decbd664d4d9edbd4ad450e54:8:68414090"
url = "http://api.nytimes.com/svc/search/v2/articlesearch.json?q=obamacare+socialism"
```

```
articles <- fromJSON(paste0(url, article_key))

# search for best sellers
bestseller_key = "&api-key=c900c6eae827adb52a373a976c4dcd33:16:68414090"
url = "http://api.nytimes.com/svc/books/v2/lists/overview.json?published_date=2013-01-01"
bestsellers <- fromJSON(paste0(url, bestseller_key))

# movie reviews
movie_key = "&api-key=a2bf433b68cc54d04a5b5d9fd44a66d6:8:68414090"
url = "http://api.nytimes.com/svc/movies/v2/reviews/dvd-picks.json?order=by-date"
reviews <- fromJSON(paste0(url, movie_key))
```

4.5 Twitter

The Twitter API provides access to almost anything happening on Twitter. Unfortunately recent changes in the policies have made the API much more complex and require a multi phase authentication procedure. Below some example code in R:

```
#Create your own appication key at https://dev.twitter.com/apps
consumer_key = "1XxSHU2XTchttIJapwwCQ";
consumer_secret = "TWFqJK66La96OgF12aWJn3kxq2fE6iNDTKrdzVeukg";

#basic auth
library(httr)
secret <- RCurl::base64(paste(consumer_key, consumer_secret, sep=":"));
req <- POST("https://api.twitter.com/oauth2/token",
  config(httpheader = c(
    "Authorization" = paste("Basic", secret),
    "Content-Type" = "application/x-www-form-urlencoded;charset=UTF-8"
  )),
  body = "grant_type=client_credentials",
  multipart = FALSE
);

res <- fromJSON(rawToChar(req$content))
token <- paste("Bearer", res$access_token);

#Actual API call
url = "https://api.twitter.com/1.1/statuses/user_timeline.json?count=10&screen_name=UCLA"
call1 <- GET(url, config(httpheader = c("Authorization" = token)))
obj1 <- fromJSON(rawToChar(call1$content))
```